Applying and Evaluating "The Agree Predictor: A Mechanism of Reducing Negative Branch History Interference"

Based on an academic paper regarding the agree predictor, we created a simulation and evaluated it based on a benchmark of our choosing.

Allen Uta
Department of Computer Science
The University of North Carolina at Charlotte
Charlotte, NC
auta@uncc.edu

Aryan Gupta
Department of Computer Engineering
The University of North Carolina at Charlotte
Charlotte, NC
agupta40@uncc.edu

Christopher Evans
Department of Computer Science
The University of North Carolina at Charlotte
Charlotte, NC
cevans60@uncc.edu

George Pollard
Department of Computer Engineering
The University of North Carolina at Charlotte
Charlotte, NC
gpollard@uncc.edu

Sani Patel
Department of Computer Engineering
The University of North Carolina at Charlotte
Charlotte, NC
spate205@uncc.edu

Abstract - **This report will further explore a research paper that is based on the agree branch predictor. A simulator and a large number of traces were developed to observe the effects of the agree branch predictor with different variations of parameters. The simulator was then used for testing and the results were analyzed for further interpretation. Lessons are learned towards approaching this topic.**

Keywords - agree, predictor, interference, agree predictor, negative interference, neutral interference, positive interference

I. Introduction And Motivation

Our idea for this research paper was obtained from the research paper, "The Agree Predictor: A Mechanism of Reducing Negative Branch History Interference", by Eric Sprangle, Robert S. Chappell, Mitch Alsup, and Yale N. Patt. The main idea revolving around their research paper is that with many predictors, there can be multiple branches that map to the same entry in a pattern history table, this is called interference [1]. This interference can be positive, neutral, or negative [1].

When a branch's impact on a saturating counter negatively impacts the decision making of the saturating counter in regard to a different branch, this is called negative interference [1]. This negative interference can otherwise be worded as that one branch would have had the correct prediction, but due to them using the same entry in the table, it was a misprediction [1]. When a branch would have had the same prediction regardless if the interference of another branch was present or not, this is considered neutral interference [1]. When a branch would have had a misprediction but the end result was a correct prediction due to another branch's interference, this is considered positive interference [1].

In most cases, the interference is neutral, but it can Abstract - This report will further explore a research paper that is based on the agree branch predictor. A simulator and a large number of traces were developed to observe the effects of the agree branch predictor with different variations of parameters. The

simulator was then used for testing and the results are analyzed for further interpretation. Lessons are learned towards approaching this topic.

In most cases, the interference is neutral, but it can become a big contributor towards branch mispredictions [1]. Many researchers have previously tried to increase the size of the PHT, utilize an different indexing or hashing function to index the PHT, or use separate different branch classes, with these methods aiming to reduce total interference [1]. In their original paper, they performed calculations to confirm that negative interference has a impact on misprediction rates that should be attempted to be resolved [1]. Their calculations were that 74.5% of the time there will be a negative interference when two branches collide in a PHT. Their research is on a new method to change this negative interference to positive or neutral interference, with their new method being a branch predictor called the agree predictor [1]. Their experimentation results show that their implementation of the agree predictor decreased the interference misprediction rate to low as 25.5% [1].

We noticed that there were some items of possible importance left out of their paper during their experimentations. The first of which being that the agree predictor can be added on top of any basic branch predictor; however, they only used a GShare predictor [1]. The second being the lack of statements regarding the bit locations obtained from the program counter.

The Agree Predictor transforms instances of negative interferences into more neutral and positive instances through modifying the purpose of the PHT [1]. A biasing bit is attached to each branch, now each saturation counters will predict if the branch will go in the direction indicated by the biasing bit [1]. The predictor will see if the base branch's predictions will agree with the biasing bit, so when the branch is resolved, the counter will increment or decrement depending on the biasing bit [1]. A well-chosen biasing bit can have two branches using the same PHT entry go towards the correct state, which increases prediction accuracy [1]. The increase in accuracy can happen during a new branch's warm up period since it's PHT entries are already warmed-up [1].

The agree predictor changes the use of the saturation counter. Rather than the counter predicting if the branch should be taken or not, such as in most Two-Level predictors, the counter predicts if we agree or disagree with the biasing bit [1, 2]. For example, if the biasing bit is 0, and the saturation counter tells us disagree (aka not taken), we will take the branch because we disagreed with the biasing bit of 0. The biasing bit can be obtained in many ways, however is should be in the direction the branch most likely takes [1]. We used 3 methods to gain the biasing bit: from the first time the branch was evaluated, by ISA Extensions (most likely), and backwards biased forward not biased (BTFNT). The biasing bit should not change throughout the execution. The saturation counter is incremented if our branch evaluation is in the direction of the basing bit.

II. Approach

The original paper used SoSS which provided the ability to simulate the Sun 4m workstation and it's user-machine relationship [1]. We decided to not proceed with using a full system simulator, but a standalone branch prediction simulator. The reasoning for this is because we felt that the system used is outdated and it would be inefficient to repeat the experiments with such an old system, but instead, to focus solely on the core of the experiments.

Each simulation being a different variant, there would have to be many differences in between. When considering what other differences we wanted each simulation to have, we found that there were many options. Due to the presence of so many options, it was infeasible to use them all, therefore, we had to choose and focus on the ones that mattered the most. The first option was to change the bit length of the saturating counter in the PHT, however, we decided against this as in the original paper, they had used only 2-bit saturating counters [1].

The second option was to change the base predictors used, which we decided to do as using different types of base predictors could should a large change in results, in contrast to their original paper mainly focusing on the GShare base predictor [1]. The base predictors chosen to be used are One-Level, Two-Level Global, Two-Level Local, and GShare. Simple descriptions of these are as follows. One-Level is where only the address of the instruction is taken into consideration when obtaining the appropriate PHT entry [3]. Two-Level Global is where a shift register is used to keep track of the most recent branches that were used, whether they were taken or not taken [3]. Two-Level Local is where the address is used alongside shift registers that keep track of the recent values of that branch, if it was taken or not taken, and then this shift register is used to obtain the location of the PHT entry [3]. GShare is where a shift register with the same functionality of the Two-Level Global predictor is XORed with the address, using the output of this calculation to obtain the location of the PHT entry [3].

The third option was to vary the bit length of entries in the PHT table, which in their original paper, they used lengths varying from 10 to 16, or in other words, 1K to 64K entries in the PHT [1]. We decided to use the lengths 10 to 16

as well as we thought this could be a variant that could affect results substantially, and that using original values would be beneficial. The fourth option we used is to vary the starting location for the bits obtained from the address. We decided to use 0, 7, and 14 as this provides a decently spaced gap in the starting locations to see if this will provided any changes in results.

III. Simulation Setup

The trace was created by running rsync to copy about 568MB of data to a server over an ssh connection. The data was created by a python script that creates random folders and files of random size. The trace was collected using Pintool[1] by Intel using an extension. The branch extractor extension made by Reze Baharani[2], was used to extract branch information required and create a trace. The trace was then filtered using another python script to remove the unconditional branches and superfluous data. The final trace had approximately 30 million branches and contained the branch address, the target address, whether it was taken or not, and a bias bit to emulate the ISA Extensions. The biasing in the trace file was chosen from the trace file. A python script went through the trace and counted the number of times a branch was taken or not taken. If the branch was taken more times than not taken, the biasing bit is set to 1, otherwise 0. Once the trace file was created, the program was run with each branch predictor base and agree predictor types and analyzed in section IV. After our presentation, we decided to rerun our simulator using a larger trace. Another trace was created while rsync copied 3GB of files similar to the previous trace. The resulting trace, after filtering, had 156 million branches. The code to run the simulation is open source and can be accessed on github[3].

IV. Results And Analysis

The agree predictor was used on top of various branch predictors which consists of the Two-Level Global branch predictor, the Two-Level GShare predictor, the Two-Level Local predictor, and the One-Level predictor.

In (Appendix A, Figure 4), a graph can be seen that acts as a baseline for the base predictors running the approximately 30 million branch trace, without any agree predictors. With the exception of the Two-Level Global Branch Predictor, a trend can be seen that the misprediction rates go down as the number of entries in the PHT gets larger. In the Two-Level Global Branch Predictor results, an odd trend occurs where there are consistent spikes in the misprediction rates as the number of entries changes. Since this branch predictor only uses a global shift register as previously mentioned, it is possible that it is due to the benchmark's trace of branches [3]. For example, it is possible that every 11, 13, 14, and 15 branch, there is a lot of negative interference, as shown in the graph.

The first set of tests set the bias bit to when the branch was first reached. These tests measured the misprediction rate against the type of branch predictor and the and the location of the starting address bit used to determine PHT entries (Appendix A, Figure 1). For all of the Two-Level branch predictors tested, all of them were drastically optimized as the number of PHT entries was increased. All of their misprediction rates fell drastically. Overall, the GShare branch predictor implementation had the least number of mispredictions. The One-Level branch predictors as a whole had a high misprediction rate. In terms of using a One-Level predictor with a starting address bit of 0, its' misprediction rate and effects of optimization can be most comparable to the Two-Level Local branch prediction. When the starting address bit grows, the One-Level branch predictor misprediction rate skyrockets from over 5 percent to over 15 percent showing very little effect when the number of PHT entries is incremented. This is made more apparent when starting its' address bit at 14 with about a 10 percent increase in misprediction rate and almost no optimization occuring.

From this first set of testing, we have learned that One-Level predictors are less than optimal for use as a base predictor for this agree predictor implementation. Aside from One-Level predictors, a larger number of PHT entries generally provides better performance. GShare prediction as a base predictor performs the best in this set of tests. Experimenting with starting address bits is a good idea, because overall trends for each predictor may vary. In the Two-Level Global branch prediction, the starting address bit had very little effect. In the GShare branch predictor, the maximum mispredictions became erratic from the increase of the starting address bit. In the Two-Level Local branch predictor the maximum mispredictions lowered from the increase of the starting address bit. When we compare the First Time Agree Predictor to the baseline predictors, there are improvements present with some exceptions such as Two-Level Local with a Starting Address Bit of 0, and any of the One-Level predictors.

---

1 https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

2 https://github.com/mbaharan/branchExtractor

3 https://github.com/aryan-gupta/ECGR4181/tree/master/FP

When we later tested this configuration of the First Time Agree Predictor with an approximately 155 million branch trace, which is five times bigger than the initial test, the final results did not change as seen on its' chart (Appendix A, Figure 5).

The second set of results were achieved after modifying the ISA extension to set a biasing bit on the branch to note whether it was taken or not taken. The results we achieved were likely incorrect, this was noted because while varying the bit size for the Two-Level Global and the GShare the results were exactly the same and this cannot be. To verify the error and note the results for the Two-Level Local and One-Level refer to (Appendix A, Figure 2). While the results are inconclusive, if there is no error present in our code, this would mean that Two-Level Global and GShare performed the best with this agree predictor version.

From the graph for "Most Likely Agree Predictor For Rsync Benchmark (Approximately 30 million branch trace)" located in (Appendix A, Figure 2), results we see when we varied the parameters between 1K and 64K PHT entries, it had not effected the branch prediction rate of the Two-Level Global and GShare. While this is true there was a slight variation in the prediction rate of the Two-Level Local and One-Level. The variation of starting address bit between 0, 7 and 14 had the same effect on the predictors this lead us to believe there is a error in either our code. The branch prediction rate for Two-Level Global and GShare were about 95.32 percent of correct branch prediction. These were incorrect due to them all having the same percentage. For the Two-Level Local and One-Level there was slight variation in the results achieved while varying the bit size and starting address bit, the results seemed to predict about 56-58 percent for both the predictors. With these percentage and the bar graph in (Appendix A, Figure 2) we can conclude overall that the Two-Level Global and GShare work best as a base predictor to go along with the Agree Predictor. When we compare the Most Likely Agree Predictor to the baseline predictor, the baseline misprediction rates were better than the Most Likely Agree Predictor.

From the graph for "Most Likely Agree Predictor For Rsync Benchmark (Approximately 155 million branch trace)", located in (Appendix A, Figure 6), results we see that the varied parameters between 1K and 64K PHT entries had the same effect as the approximately 30 million branch trace. The resulting graphs looked just about the same, so varying the length of the trace had no effect. This leads us to conclude that the issue present is not in the length of the trace, but either there is an issue in the code, or that the predictor is working as it should.

The third way to receive the bias bit is Backward Taken Forward Not Taken. Or more accurately Backward Biased Forward Not Biased, however we will refer it as Taken/Not Taken for the sake of simplicity. Similar to our other testing we varied the size of the Pattern History Table from 1K to 64K. The change in PHT is coupled with varying the starting address bit. This section of testing corresponds with (Appendix A, Figure 3). The data immediately shows that there is some fault in our simulator or method of getting the bias bit. For the first round of testing we see positive results when increasing the PHT entries. When using BTFNT we see an anomaly where increasing the PHT entries yields no increase or decrease in performance for the Two-Level Global and GShare base branch predictors. In (Appendix A, Figure 3) the Two-Level and One-Level Local base predictors decrease in performance and have a higher misprediction rate. In theory increasing the PHT entries should have a positive effect. Increasing the PHT size could lead to more bias bits and consequently increase the start-up period for branches. More simulations should have been ran to see if this trend continues or if it was an issue with BTFNT method of determining the biasing bit. With more data we could pinpoint where we may have went wrong with building the simulator. When we compare BTFNT to the baseline predictor we saw immediately that the results were off. The PHT entries are increased in the baseline predictor (Appendix A, Figure 4) and notable performance gains can be seen by a much lower misprediction rate. The data helps us compare the differences between the baseline and various methods of getting the bias bit. The baseline would be useful in reassessing our Agree Predictor and making the necessary revisions. When comparing the graphs in (Appendix A, Figure 3), and (Appendix A, Figure 7), we can see that the trends stay the same regardless of the lines present in the traces used, with these being approximately 30 million and approximately 155 million. After going over the results of using BTFNT we concluded there was no performance gain and we must use another method to perform similarly to the baseline.

V. Lessons Learned

The results we got for the initial testing proved to be inconclusive and there were no notable trends for BTFNT and Most Likely Agree Predictor. When verifying the applications of the Agree Predictor every possible scenario must be covered to ensure performance gains. It can be very difficult to

diagnose what is going wrong with a simulator. The size of the trace file was increased to determine if that was a potential solution. When testing a new computer architecture it is imperative to cover all the basics and provide a wide variety of data. During our presentation it become apparent we should have included a baseline for our testing. Without a baseline simulation we were not able to properly illustrate the performance of our Agree Predictor. We wanted to explore performance gains of different variations of the Agree Predictor. In addition, we realized that we would have produced a wider variety of results if we were to have run more benchmarks to compare several result sets, however, due to time constraints and technical issues this was not accomplished. From this we learned that this could have been combated with better planning and time management skills.

Computer architects have to be prepared to be under fire when presenting an idea. Designing a new architecture is akin to proving a mathematical formula. The problem must be approached from many angles, because there will be people poking holes in the theory. When presenting you have to be prepared for every question. The computing world evolves every day and it would not be possible without computer architects striving for perfection.

We would have liked to use the branch predictors they use in today's processors. Many companies however guard their branch predictors as trade secrets. It would have been simpler to troubleshoot what initially went wrong with our Agree Predictor. The Agree Predictor is dated, but it was still similar to the homework we completed.

This project highlighted the importance of collaboration. Every person had their individual strengths and worked efficiently. In a group the work is delegated so that the process is similar to a pipeline. We were able to have multiple “instructions” running concurrently and solve many problems together. The project was a very good way to bring the semester to a close. The incorporation of professional writings and coding applications adequately prepared us to present our own paper and ideas.

References

[1] E. Sprangle, R.S. Chappell, M. Alsup, and Y.N. Pat, “The Agree Predictor: A Mechanism For Reducing Negative Branch History Interference,” IEEE,,, June 1997.

[2] “Branch prediction,” danluu. [Online]. Available: https://danluu.com/branch-prediction/. [Accessed: 13-Dec-2018].

[3] H. Tabkhi, “Lecture-08-Branch Prediction.” UNCC, Charlotte, 03-Nov-2018.

Appendix A

Figure 1 - First Time Agree Predictor Graph For Rsync Benchmark (Approximately 30 million branch trace)

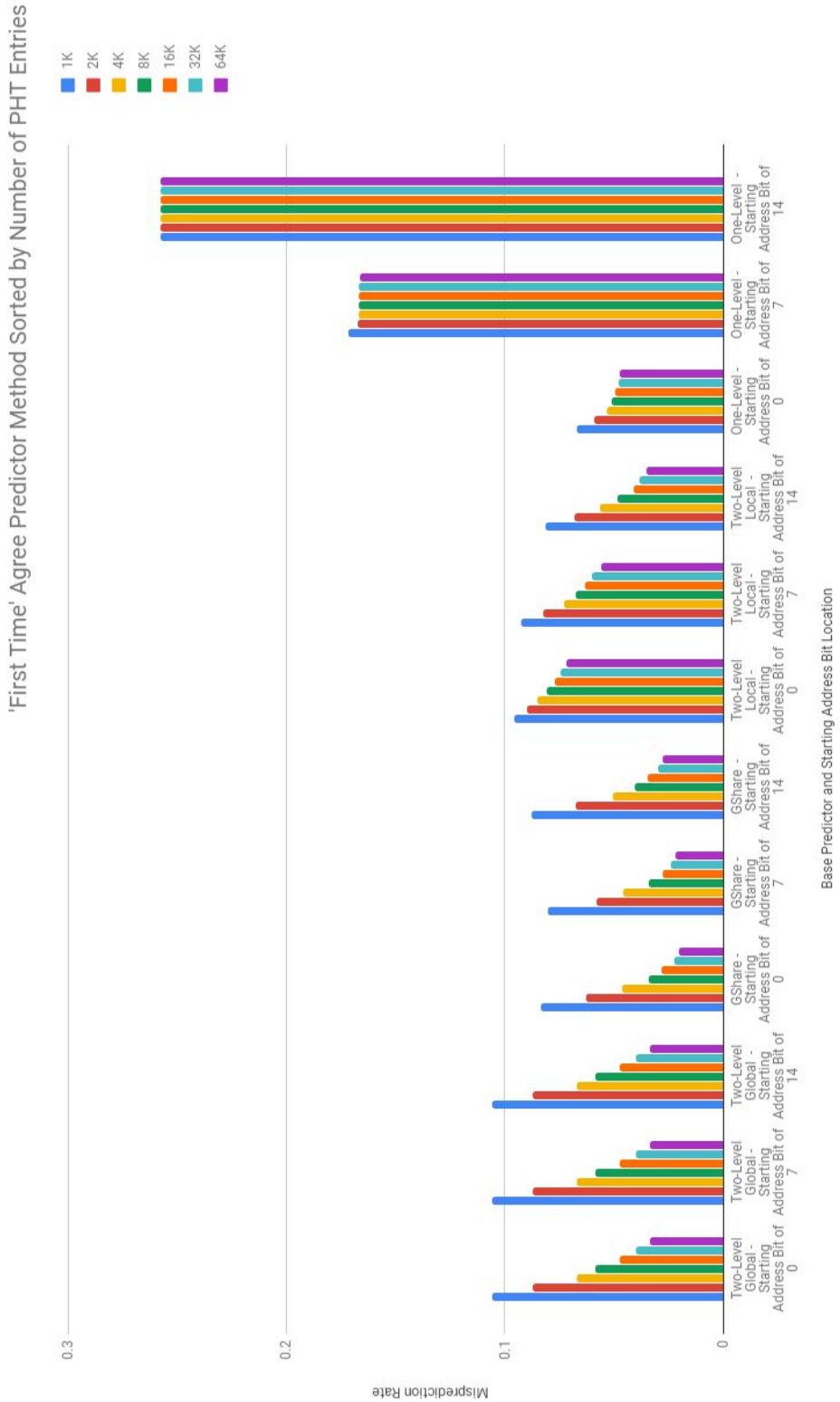

Figure 2 - Most Likely Agree Predictor For Rsync Benchmark (Approximately 30 million branch trace)

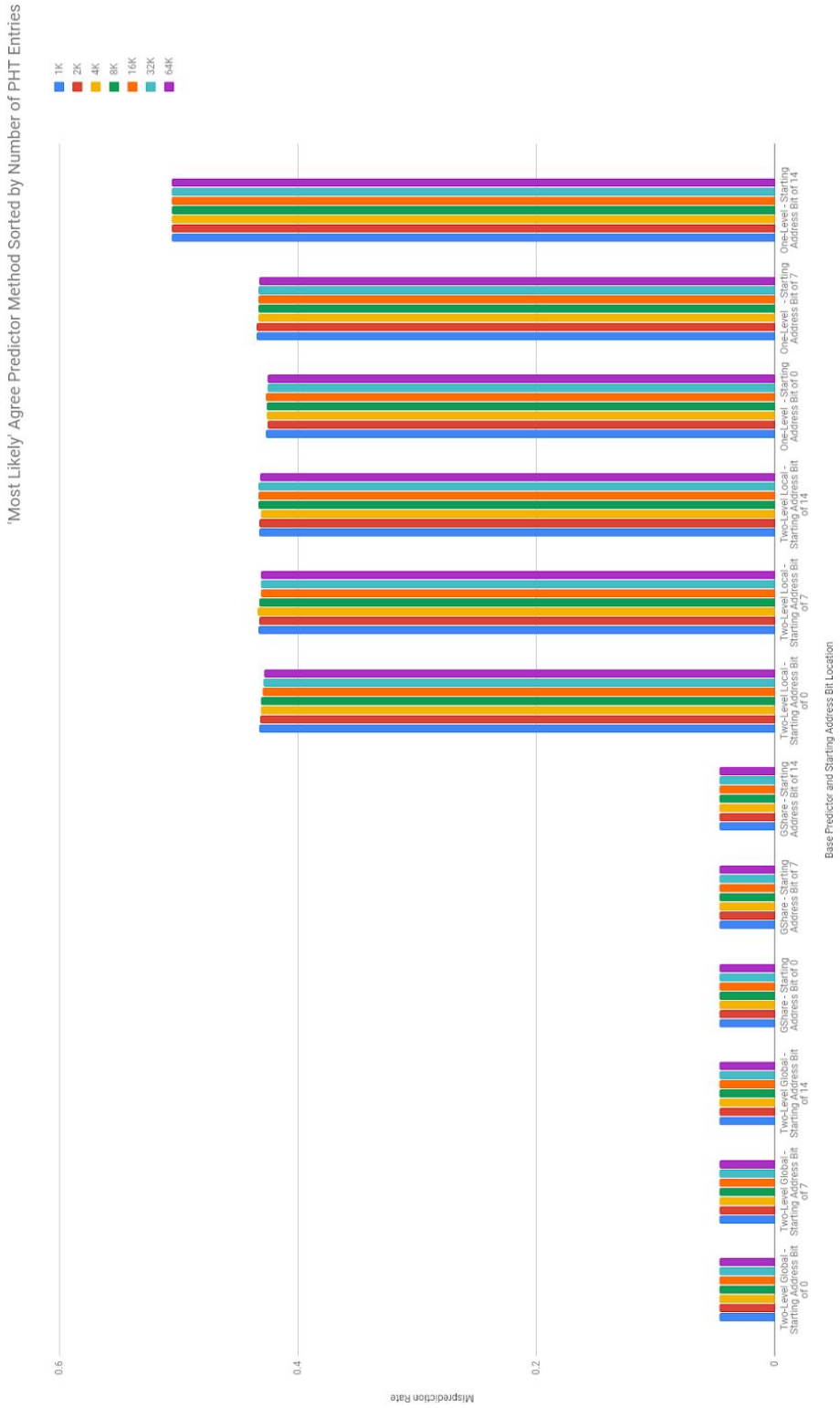

Figure 3 - Backward Taken Forward Not Taken For Rsync Benchmark (Approximately 30 million branch trace)

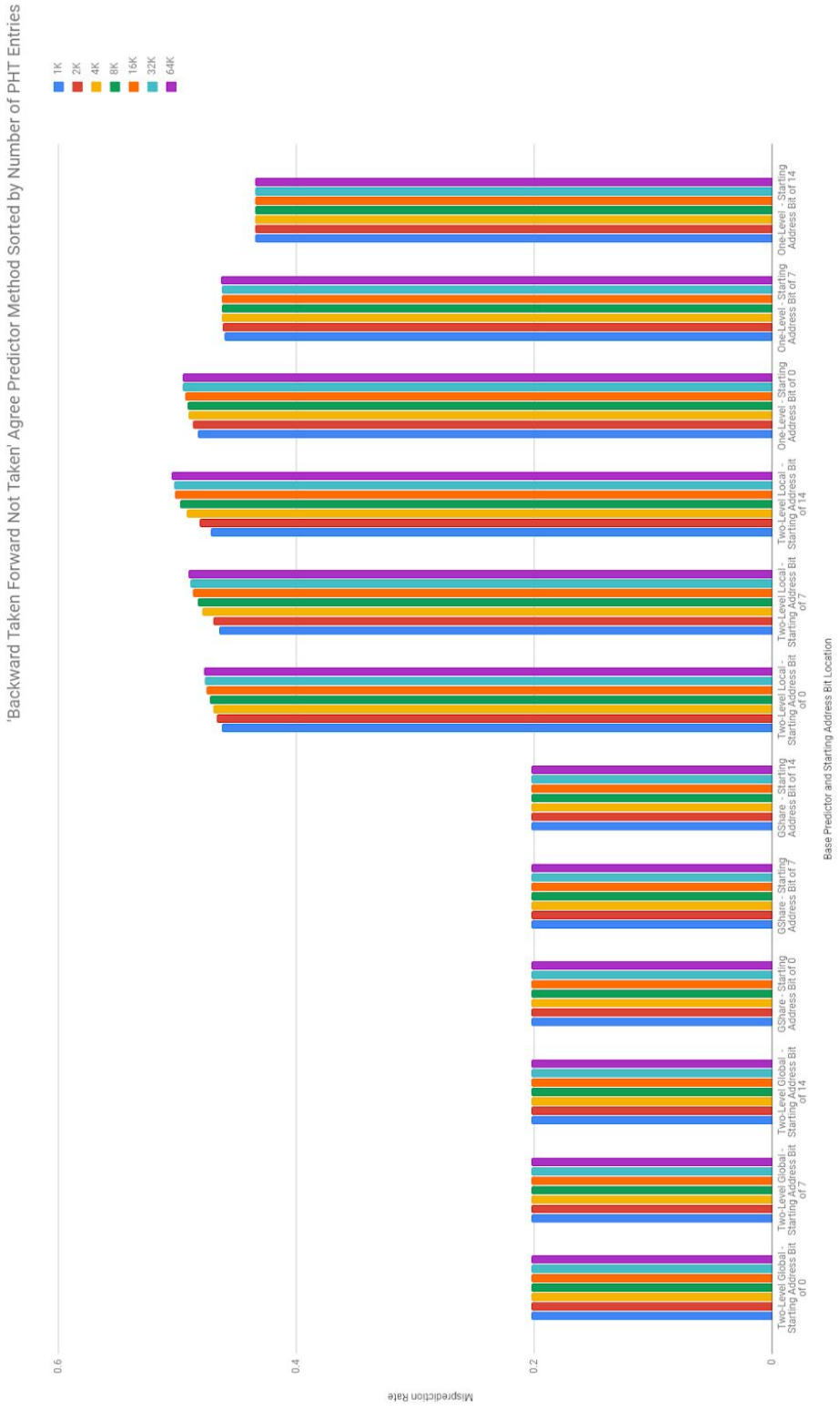

Figure 4 - Base Predictors with no Agree Predictors (Approximately 30 million branch trace)

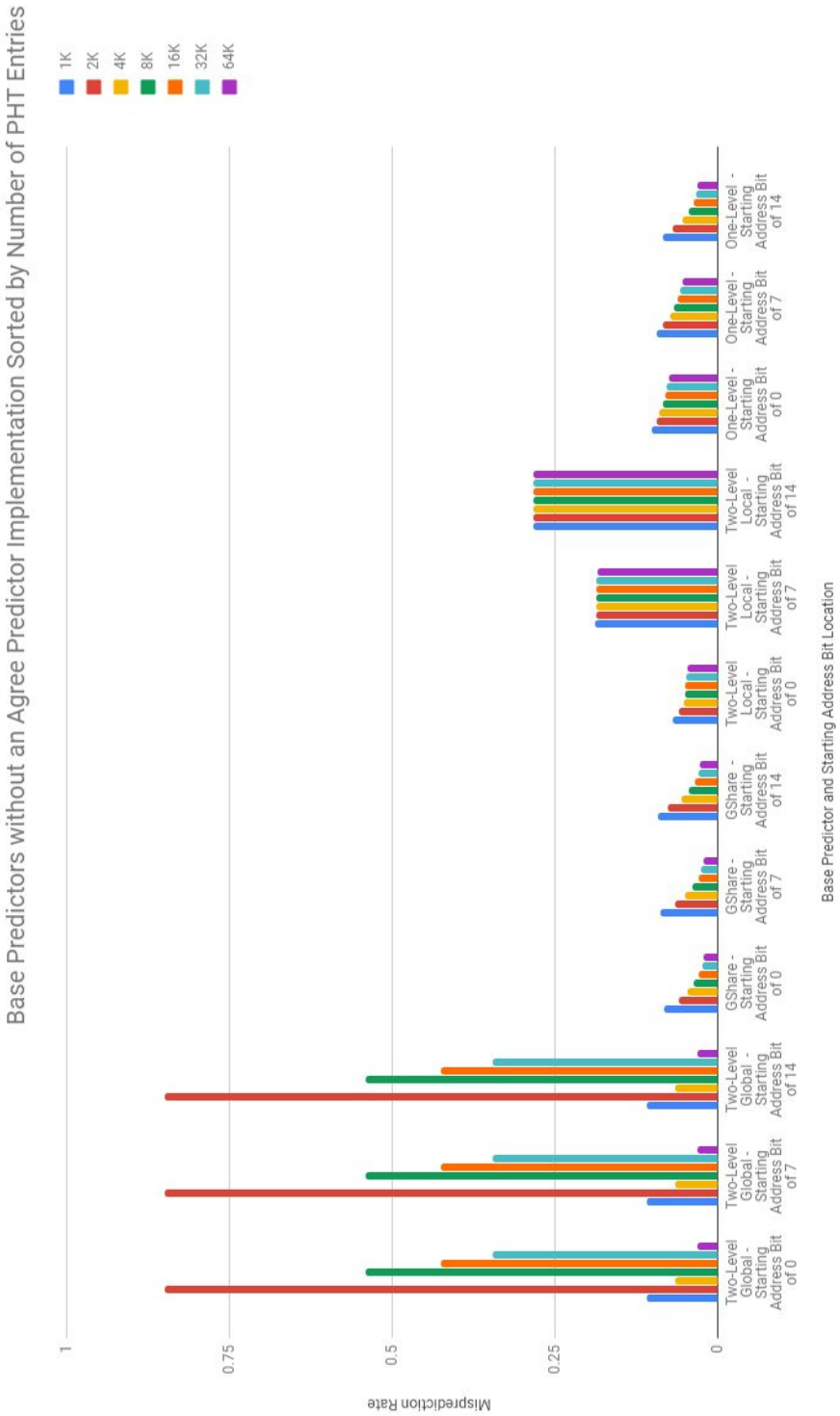

Figure 5 - First Time Agree Predictor Graph For Rsync Benchmark (Approximately 155 million branch trace)

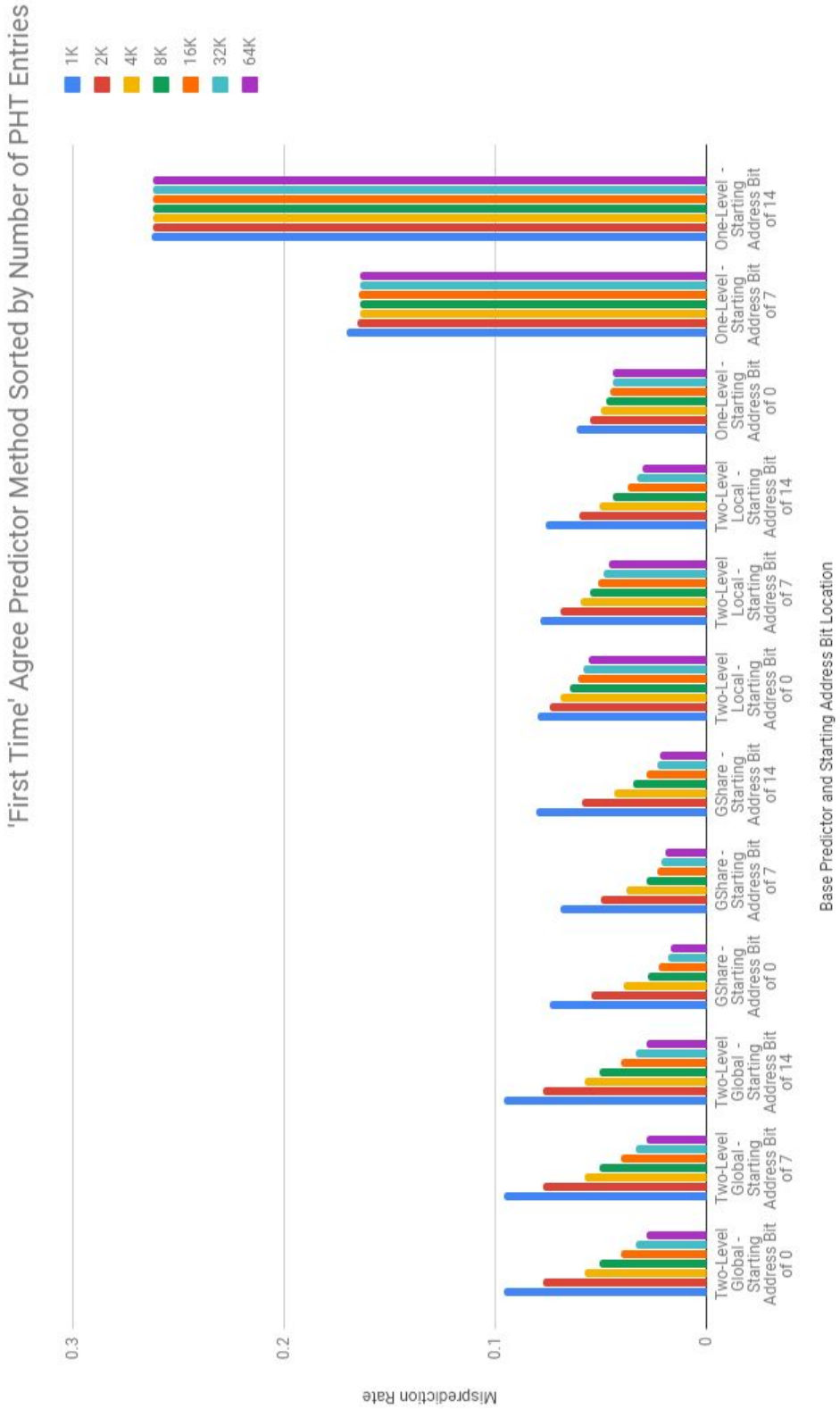

Figure 6 - Most Likely Agree Predictor For Rsync Benchmark (Approximately 155 million branch trace)

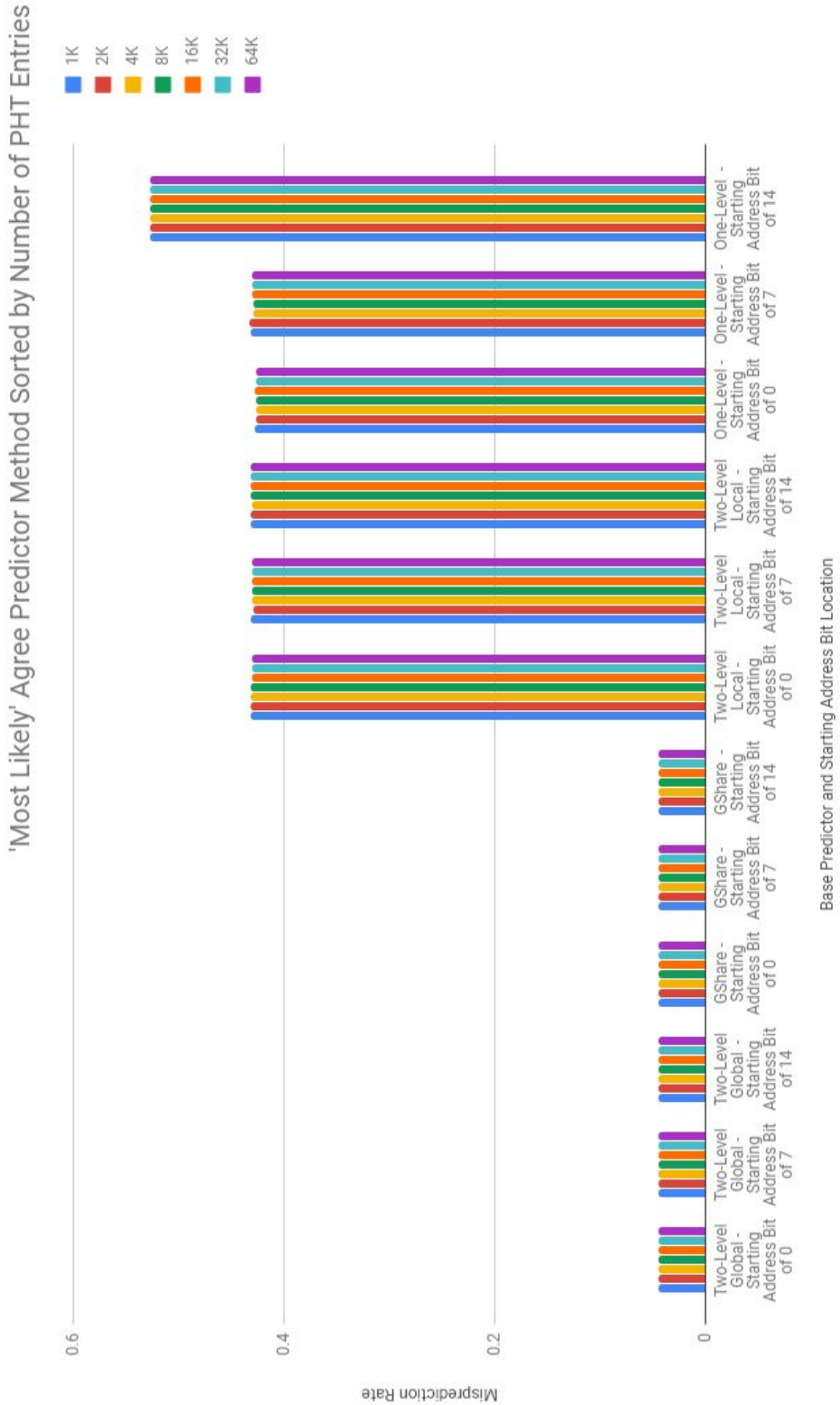

Figure 7 - Backward Taken Forward Not Taken For Rsync Benchmark (Approximately 155 million branch trace)