CPRE 581 Final Report

Jake Hafele, Gregory Ling, Thomas Gaul
Department of Electrical and Computer Engineering
Iowa State University, Ames, IA 50014

ABSTRACT

Branch predictor designs have often sought out higher prediction accuracy's to achieve a higher IPC value and better throughput for instruction flows. As new designs and architectures are proposed, the main center of focus in terms of improvement is prediction accuracy, with little room to include variations in terms of area, power, or timing constraints. By comparing different branch predictor designs such as TAGE, tournament, global, and more, we are able to analyze the power, area, and timing implications in the context of a soft-core processor design on a synthesized FPGA application. Utilizing Chipyard, we can generate RTL designs and synthesize multiple variations of a softcore (BOOM) processor to compare Vivado synthesis results with a small sample of SPEC benchmarks in parallel on reconfigured hardware.

I. INTRODUCTION

From as early as 1981, researchers have investigated the topic of branch prediction in the context of high level architecture designs. J. E. Smith [1] proposed multiple strategies including static predict if taken designs and dynamic designs which maintain a table of most recent branches or a 2-bit counter to predict strongly taken or not taken. For each of these compared designs, the main determination of branch predictor performance was in terms of prediction accuracy, where a higher prediction accuracy would lead to less discarded instructions or latency on a branch misprediction. In following works, such as Yeh and Patt's [2] two level predictor analysis, multiple varying prediction schemes are compared between global, per-address, and per-set schemes for each branch prediction level, with again prediction accuracy as the only factor for performance. While Yeh and Patt did only analyze branch predictor tables with less than 512K bits, that was the only consideration to hardware complexity. For many of these papers proposing new architectures, analyzing the impacts of branch predictors in terms of area, power, or logical delay fall to the side in terms of performance comparison.

In terms of complexity, branch predictors have become increasingly advanced to satisfy the high demand of issuing multiple instructions per cycle in modern processors. In newer technology nodes, it can be seen that as technology nodes decrease in size, shortened clock cycles and larger wire delays will become a much more prevalent issue [3]. As the clock period decreases, this can lead to more architectural units impacting the critical path and delay of the design, which can lead to the hardware complexity and delay impact of branch predictors becoming more significant. It can also be seen that increasing the delay of a branch predictor to improve prediction accuracy is never a good trade off [4], leading to the motivation that the comparison of delay between modern branch predictor implementations should be further investigated. While the referenced paper does use Gshare as a baseline prediction model, three new architectures are proposed, instead of following up work on previously implemented prediction designs.

When considering options for improvement for branch prediction or many other areas of improvement throughout a processor, bigger correlates to at least a marginal improvement [5]. Thus, when making the optimal theorized processor, many sections would be of infinite size. However, when it comes to implementing a design in hardware, certain decisions need to be made due to hardware limitations. Branch predictors with a BTB that consists of a size of a small cache can take up to 10% of CPU power [6]. Now, different branch prediction strategies have varying power consumption and size requirements. Each of these prediction strategies has a different size where their benefits per size increase fall off. With the limited power capabilities of a processor, power to improvement must be considered as increasing the power consumption of

different segments of the processor yields different levels of improvement. In the work we explored, they investigate the miss rate and performance of many branch prediction algorithms, but rarely do they compare the power cost to performance and not across multiple approaches.

Similar to power, the area required by different strategies differs and increases as the size of each strategy increases [7]. With the decline in Moore's law, relying on being able to pack more into the same area is no longer an option either, so architecture designers need to be wise about how they spend their area to maximize their performance benefits. When looking at other area-intensive performance improvements, it may be found that the area is better spent increasing cache sizes or instruction window size [5]. Like power, the concern of area required by branch prediction strategies is not investigated by many papers but is a crucial consideration when implementing these in a design.

For this work, our motivation was to combine existing branch predictor configurations and utilize an open-source platform to generate area, power, and timing results in an adequate time frame, to further focus on the comparison and effects of varying predictor configurations. By utilizing Chipyard, we would be able to apply different premade branch predictor configurations such as TAGE, Tournament, and GSHare, and apply our own custom configurations such as Local, Global, and Null. This will enable us to determine area, power, and timing results from Vivado synthesis by utilizing generated RTL with Chipyard, and to generate a reconfigurable bitstream for an application FPGA which can run benchmark results. Then, we would be able to compare synthesis results with the SPEC benchmark performance metrics for a sample set of ASTER, BZIP2, and MCF benchmarks.

II. Related Work

To recontextualize branch predictors in terms of area, power, and timing constraints, we must analyze previously published branch predictor designs, as well as analyzing other existing works that consider other factors of performance besides just prediction accuracy. By analyzing previous prediction schemes, we will get a better idea of how to model them in Chipyard, and gain further knowledge in how to present our results. By using more modern works related to other constraints such as area, power, and delay, we will also learn more about what is done and how we can recontextualize prediction accuracy between each of the three additional factors for performance.

A. Branch Predictor History

One of the earliest works in branch prediction was presented by J. E. Smith [1], which presented multiple static and dynamic branch prediction schemes within the context of an improved prediction accuracy. The complexity of these designs varied between static predictions which would always assume branch taken, or more complex schemes where a branch would prediction would be determined on the previous result of the branch, based on the program counter address for the instruction memory. While each of these models were driven based on their prediction accuracy, multiple different FORTRAN benchmarks were used, which shows that it is important to run test programs with varying instruction sets, to test the full capabilities of each branch prediction scheme. Alongside this, multiple table sizes were considered in terms of size for dynamic branch prediction history results, which could benefit out own Chipyard results.

As authors sought out higher prediction accuracy's, more works were published, such as Yeh and Patt's [2] comparison of dynamic predictors which used multiple variations of two-level schemes. In this paper, three types of predictor schemes were applied, including using a history of the last N branches used (Global), the last N branches of the same branch PC address (per-address), or the last N branches in the same set (per-set). Each of these variations were applied for both the first and second level of the dynamic predictors, leading to 9 total variations to analyze for prediction accuracy, for multiple SPEC benchmarks. One useful piece from this paper that could be examined is the comparison of hardware cost for each of the 9 configurations, based on the number of branch history table entries and number of branch sets. This work also modeled many varying branch prediction schemes based on varying branch history lengths, and was cut off after 512K bits, showing that some size considerations were used when comparing prediction accuracy.

Around a similar time that the above work was published, S. McFarling [8] presented a novel idea of a tournament predictor. The core idea of this design is to

use both a local and global branch prediction scheme for the second level predictor, and select between which predictor to use with a 2-bit prediction counter. A 2-bit counter would act as a saturated counter to select the best predictor to use, either the global or local predictor, based on the performance history of each predictor, which would drive the choice counter to either increment or decrement. This work presents its results in terms of branch accuracy, and models multiple configurations of tournament predictors in the same plot, including bimodal, bimodal/Gshare, and local/gshare combinations. Overall, a prediction accuracy improvement improved to 98.1%, from the previously known best prediction scheme of 97.1%. This work could be re-contextualized in terms of area, power, and delay constraints to determine if the added 1% in accuracy improvement can be justified in a real-world application.

More recent research within branch prediction "SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine" [9] has found created the fastest open source processor iterating on the Berkeley Out-of-Order Machine which is the processor design we are basing our research on. This new processor design uses a Tage branch predictor instead of the previous iteration of the BOOM processor, which uses a Gshare branch predictor. With this change, among others, they found significant performance improvement.

B. Branch Predictor Motivation

The overall inspiration for our project is similar to that explored by the paper "Branch Prediction, Instruction-Window Size, and Cache Size: Performance Trade-offs and Simulation Techniques" [5]. In the paper, the authors look at the performance benefit of changing the sizes of their branch prediction unit, instruction window size, and cache size due to the die constraints of a design. In it, they specifically explore a hybrid branch predictor with a global sector, a GAg predictor, and a PAg predictor. In the paper, they find that incorrect branch prediction always ends up being a major bottleneck and tends to be an area where resources are required.

One of the major aspects of our project goal is investigating the delays associated with the different predictor designs, and this idea was explored by "The Impact of Delay on the Design of Branch Predictors" [4]. In their paper, they look at the delays caused by increasing branch predictor size because, up to the writing of their paper, the only variable explored was the predictor's accuracy. The paper specifically looks at the gshare predictor and a hybrid predictor with varying sizes. They find that if they just optimize of accuracy of prediction it has a negative effect overall due to the heavy delays incurred by their size. They found their hybrid predictor functioned well until its delay reached a full cycle, and then everything fell apart. Due to their findings, they suggest all papers investigating branch prediction should also report delays due to the implications they found.

The next major aspects we want to explore in our project is power and area. Power is discussed well in the paper "The Impact of Delay on the Design of Branch Predictors" [6]. They explored the trade-off between the power and accuracy of four hybrid predictors and found that the application is an important consideration due to the varying power consumption. Another paper, "Low Power/Area Branch Prediction Using Complementary Branch Predictors," explores power in addition to area, another aspect we want to explore [7]. They cover all topics we want to explore and as a result, propose a complementary branch predictor. It is a delay, area, and power-efficient branch prediction algorithm that only completes complicated predictions for branches that are typically challenging for the processor to guess which yields good prediction accuracy without as much of the costs.

A more modern area outside our project's scope, which would be an interesting continuation of our work, would be the security vulnerability of branch prediction covered in "Spectre Attacks: Exploiting Speculative Execution" [10]. The paper explored different ways branch prediction can be exploited to reach memory sections that are otherwise unreachable. They explore options to mitigate the risk of these attacks to the detriment of the processor's performance.

III. Main Idea

The main idea of our project is to test different branch predictors to compare their area, power, and performance. Our main approach is threefold. First, we wish to use Chipyard to generate an out-of-order core. Chipyard provides several base branch predictors we can configure the soft CPU to use. This will provide us with the base TAGE, Tournament, and GShare predictors, then we can modify those to create global,

local, and null predictors. Second, run these out-of-order cores on an FPGA with a small Linux distribution. This portion was mostly plug-and-play, as a prior research group had already set up running Linux on the ZCU106 FPGA. Third, run SPEC benchmarks inside Linux. This is not a simple task as the SPEC benchmarks are primarily intended to be compiled and run on the same device, so this will require cross-compiling and following the alternative SPEC instructions[1] as runspec will not run on the base Linux installed on the FPGA due to missing Perl and other dependencies.

IV. Methodology

Our workflow is centered around using the open-source tool Chipyard [11], which can provide generated RTL designs for soft-core processors and synthesized to FPGAs. Based on the available hardware and resources, we decided to use the ZCU106 FPGA development board provided in Durham 310 at Iowa State University. Another graduate student, Jordan McGhee, was able to provide previous work with Chipyard and an interface to using the ZCU board, which varied from the provided VCU configuration in the default Chipyard repository.

Both Chipyard and Vivado were used on the provided ECpE Linux instances, in which we could utilize Conda to source the required dependencies to run Chipyard. To generate the FPGA instance, a make command pointing to the ZCU106 configuration was ran under the /fpga/ folder from the Chipyard repository. In the BOOM submodule directory of the Chipyard repository, a Scala source file titled config-mixins contained multiple preconfigured branch predictors, including TAGE (WithTAGELBPD), Tournament (WithBoom2BPD), and GShare (WithAlpha21264BPD). By editing the branch predictor class inherited under the WithNSmallBooms class under config-mixins.scala, we were able to update the included branch predictor scheme to be used for RTL generation and the following synthesis.

Alongside utilizing existing branch prediction schemes, we opted to create our own predictor schemes for more variation in power, area, and timing results. To modify the branch predictor configurations, we were able to modify the global history length, local history length, and number of local history sets to configure local, global, and null prediction schemes. Using the same config-mixins Scala source code under the BOOM git submodule, we created three new branch predictor classes, Global, Local, and Null, to be inserted into the WithNSmallBooms class to generate RTL for the BOOM core. These branch predictor classes were referenced the same way as the predictors above.

[1] https://www.spec.org/cpu2006/Docs/runspec-avoidance.html

Vivado was able to be sourced on the Linux machines, which could then be used to synthesize the generated results from Chipyard. After running the provided FPGA make command provided by Jordan McGhee, we were able to cancel the Vivado synthesis and rerun it inside of the Vivado GUI, to step through synthesis, implementation, and bitstream generation. After implementation, we were able to generate a power, timing, and utilization report for each of our branch configurations, which enabled us to compare timing, area, and power for each branch predictor. By generating the bitstream, we would then be able to program the ZCU106 which is wired over USB to a Linux machine in Durham 310. A tarball of each Vivado project was saved via a Git repository to ensure that each Vivado implementation of the BOOM core and respective branch configuration could be re-used at any time.

We were able to use the FireMarshal build provided by Jordan's research group, so we did not have to compile that ourselves. We downloaded the SPEC benchmarks from the prior CprE 581 homework assignment and picked three to cross compiled to riscv for the FPGA. After cross compiling, we copied the ASTAR, MCF, and BZIP2 benchmarks to the micro SD card (benchmarks picked arbitrarily from the SPEC2006 suite), and used runspec to retrieve the test data and commands to run on the fpga [12]. We used the test data for these three benchmarks (four commands in total as BZIP2 has two commands in the test suite). We used the test suite as the test suite took about 20 minutes for ASTAR on the FPGA, and running a full reference suite would not be practical. These are not a valid SPEC benchmark run, but this will give us usable data for comparison. For timing the benchmarks, the time command was called with each of the benchmark commands.

We discovered an issue running the benchmarks from the micro SD card directly as the micro SD card is connected via SPI and is extremely slow and unreliable. To work around this, a RAM-only tmpfs is mounted

in /tmp, so the test data is copied to /tmp before running so all data and executables are in RAM for the duration of the test and the micro SD card can be disconnected and not affect the test results.

This process was not automated, so for each processor you had to open Vivado, program the bitstream to the FPGA, wait for Linux to boot, copy the tests from the SD card to a RAM filesystem, then run the tests. A future goal would be to write a script to automate this setup as the FPGA can be accessed via a serial port and Vivado can be scripted to program the FPGA.

V. Results

Method	Global Len	Local Len	Local Sets
TAGE	64	1	0
Tournament	32	32	128
GShare	16	16	1
Local	0	32	128
Global	32	0	0
Null	0	0	0

Table I: Branch Predictor History and set sizes

Figure 1, 2, and 3 represent the synthesized results of all six branch configurations for the Small BOOM Core ZCU106 synthesis. As expected, the null predictor scheme had the least amount of logic cells utilized. Surprisingly, the global predictor had a lower power consumption. The TAGE prediction scheme had a much larger utilization count than most predictors, specifically including many more registers utilized, leading to a higher power consumption.

Half of our results were derived from the Vivado synthesis using the generated RTL for each BOOM core and branch predictor using Chipyard. Figure 1 demonstrates how many Look-up Tables and registers were used for each branch predictor configuration, to allow for comparisons of area for each predictor design. As expected, the Null predictor (or no predictor), had the least amount of utilization with both LUTs and registers, coming in with 87,616 LUTs and 61,583 registers. The Global, Local, GShare, and Tournament predictor schemes all had very similar utilization amounts, which all had near 5% additional utilization for both LUTs and registers. The standout prediction scheme with the most utilization was TAGE, with a 13% increase in LUT utilization and 22% increase in register utilization, relative to the Null predictor.

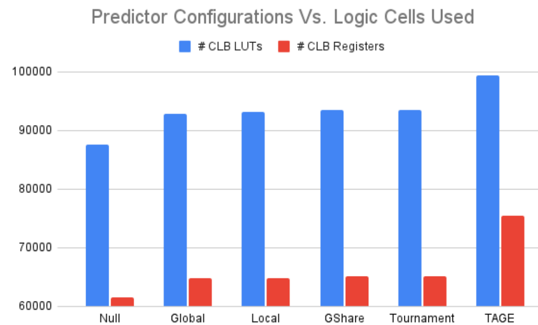

Figure 1: Branch Prediction Utilization Results

Synthesis results for Power consumption was also generated, as seen in Figure 2. The power consumption stayed relatively similar, with some variation due to the different amounts of utilization per the different predictor schemes. The Null predictor reported a power rating of 2.326W, which was surprisingly not the lowest reported. Despite the Global predictor scheme having a larger utilization count than the Null predictor, it had a slightly lower power consumption, rating at 2.319W. Every other prediction scheme followed with increasing power consumption relative to the Null predictor, as expected with the higher utilization in cells.

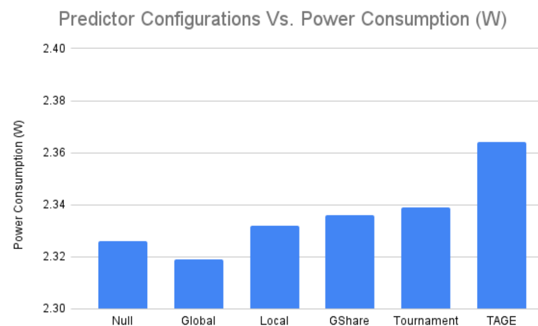

Figure 2: Branch Prediction Power Results

For the final Vivado results, we generated reports for the Worst Negative Slack (WNS) for each prediction scheme, in Figure 3. The timing results are heavily dependent on the constraints placed on the design, so these could vary by a wide margin. This can also vary based on the placement and utilization of the design. The most noteworthy portion of these results were

that both the Local and GShare predictors came in with around 0.2ns of slack, compared to the average of around 0.4ns for every other design. The main benefit of these results is to ensure that each design met its timing requirements and would not face any setup or hold time violations.

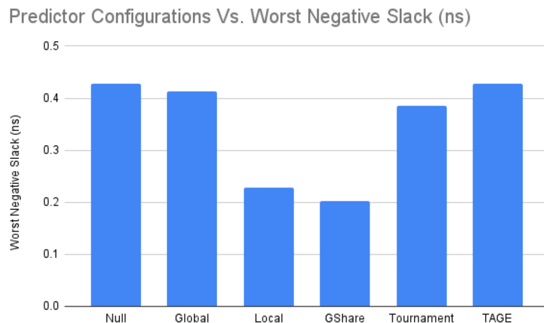

Figure 3: Branch Prediction Slack Results

Once we synthesised the the processor for each branch prediction scheme we ran each of them on the ZCU106 with on FireMarshal Figure 4 displays the user execution time for each branch predictor SPEC benchmark pair as reported from the system. In each case the Null predictor executed the slowest and made a baseline for future graphics. It should be noted when running the ASTAR benchmark verifying the results returns and error which is consistent throughout all the branch predictor configurations.

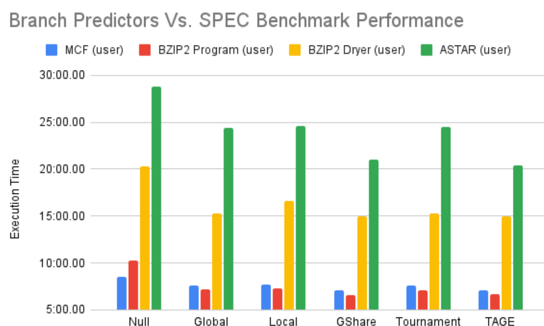

Figure 4: Branch Prediction Benchmark Execution Time

Comparing the relative performance of each of the branch predictors is best demonstrated by using the null predictor as a baseline and then comparing the speedup of each as shown in Figure 5. Every branch prediction scheme saw significant speedup over the Null branch predictor with the largest benefit occurring in higher level branch prediction schemes on BZIP2.

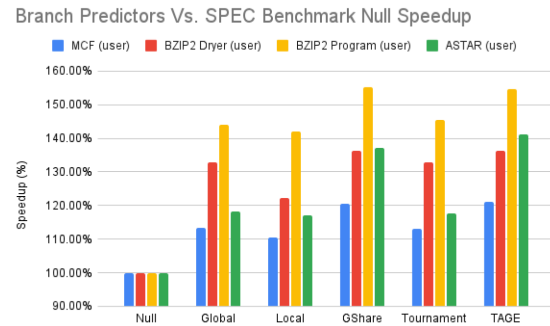

Figure 5: Branch Prediction Benchmark Speedup

VI. Analysis

The relationship between predictor power and speedup is shown in Figure 6. From this comparison, it appears that GShare has a comparable performance to the TAGE predictor, but also has a lower power usage. This is surprising as TAGE is generally considered a more advanced branch predictor than GShare [9].

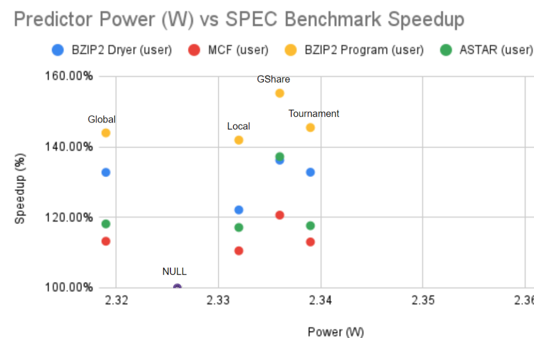

Figure 6: Branch Prediction Power Speedup Comparison

A similar comparison for utilization is shown in Figure 7. This shows the utilization of the NULL predictor was low and TAGE was the highest as expected, but it also shows that Global, Local, and Tournament all used very similar utilization and provided very similar performance. This also reinforces the benefits of GShare over TAGE in these benchmarks in the lower

utilization of GShare compared to TAGE for a similar speedup.

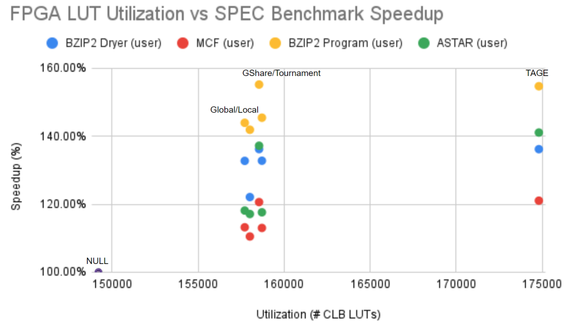

Figure 7: Branch Prediction Utilization Speedup Comparison

This figure also shows again the unusually low power consumption of the Global predictor, taking less power than the NULL predictor, but also using more LUTs than the NULL predictor. This might be a side affect of measuring power consumption in Vivado synthesis rather than a real measurement while running the benchmark.

We found that in our tests that Tournament performed marginally better than Global and Local, but has the power (Figure 6) and utilization costs (Figure 7) more akin to Tournament or GShare. As Tournament is effectively made up of those two branch predictors, the performance being similar to that Global and Local seems reasonable. We believe with the relatively small benchmarks we ran, it did not have enough time to set the selection between the two of them properly to achieve a greater benefit as we were expecting.

When running the ASTAR benchmark, there was a consistent error where every run of the ASTAR benchmark produced slightly incorrect results:

```
--- lake.out
+++ lake.out.cmp
@@ -10,7 +10,7 @@

 Create reg ways time : 0
 Reg ways quantity : 21248
-Total reg way length : 1005921
+Total reg way length : 1005873

 Create river ways time : 0
 River ways quantity : 250
@@ -28,8 +28,8 @@
 Total way length : 612234

 Create reg ways time : 0
-Reg ways quantity : 29222
-Total reg way length : 1009555
+Reg ways quantity : 29220
+Total reg way length : 1009318

 Create river ways time : 0
 River ways quantity : 0
```

Every ASTAR benchmark run produced the same output which was not the same as the SPEC-provided correct output. This indicates that there might be an issue in the chipyard core itself, but as the error was consistent across all branch predictors, it is likely not relevant to our results. All other benchmarks output the correct value.

VII. Future Work

In terms of future work, much could be done to expand our project. To begin, an extended set of SPEC benchmarks and test suites could be compiled and utilized to test our current generated BOOM core and branch predictor configurations. By testing other SPEC benchmarks, such as GCC, BWAVES, or HMMER, we could evaluate other corner cases for branch prediction, with more variation in the types of branching and addresses. This would be useful in analyzing the performance results of each predictor scheme, and ensuring that the three SPEC benchmarks that we ran for this experiment were not outliers. Additionally it would be valuable to run longer versions of the benchmarks we did use to give them a longer section of time to see benefit from warming up the predictors.

To expand and evaluate more on the branch prediction schemes themselves, more work could also be done in analyzing different parameters for the same prediction scheme, such as the global history length, local history length, and number of local history sets for a single branch prediction scheme, such as a Tournament branch predictor. We could then analyze the area, power, and performance tradeoffs for the same architectural design, which would help motivate an ideal configuration for one single scheme based on area, power, or performance constraints. Given that this would only require slight tweaks using Chipyard, this would be very feasible.

To investigate further critical variables with hardware design, our project could be extended to look into hardware security. Many works explore the security hazards of branch prediction [10], but our work could be expanded to explore how secure each branch predictor design is, how they can be altered to provide security, and how each of those secured branch predictors performs compared to each other keeping in mind the physical attributes we have investigated within this project such as power, area, and delay. Security may be a hard metric to quantify and would have many more combinations to try, but any extended investigation into security would provide very interesting and practical information.

VIII. Conclusion

In this project, we were able to successfully model six different branch prediction schemes utilizing Chipyard, Vivado, and a ZCU106 FPGA Development board to generate synthesis results. We also ran multiple portions of SPEC benchmarks, including MCF, BZIP2, and ASTAR, in which we could compare the execution time and speedup comparisons for each design, with the context of the synthesized Vivado results. Throughout this process, we overcame multiple challenges in utilizing Chipyard, Linux with Firemarshal, and successfully running benchmarks. With this project and toolflow, we have set up the potential for future work to be investigated with branch predictor schemes on real hardware, to further compare more SPEC benchmarks or branch predictor configurations using the same toolflow and methodology.

P.S. hope you are having fun grading Dr. Duwe

References

[1] J. E. Smith, "A study of branch prediction strategies," in *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA '81, (Washington, DC, USA), p. 135–148, IEEE Computer Society Press, 1981.

[2] T.-Y. Yeh and Y. N. Patt, "A comparison of dynamic branch predictors that use two levels of branch history," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, (New York, NY, USA), p. 257–266, Association for Computing Machinery, 1993.

[3] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger, "Clock rate versus ipc: the end of the road for conventional microarchitectures," in *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*, pp. 248–259, 2000.

[4] D. Jimenez, S. Keckler, and C. Lin, "The impact of delay on the design of branch predictors," in *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000*, pp. 67–76, 2000.

[5] K. Skadron, P. Ahuja, M. Martonosi, and D. Clark, "Branch prediction, instruction-window size, and cache size: performance trade-offs and simulation techniques," *IEEE Transactions on Computers*, vol. 48, no. 11, pp. 1260–1281, 1999.

[6] D. Parikh, K. Skadron, Y. Zhang, M. Barcella, and M. Stan, "Power issues related to branch prediction," in *Proceedings Eighth International Symposium on High Performance Computer Architecture*, pp. 233–244, 2002.

[7] R. Sendag, J. J. Yi, P.-f. Chuang, and D. J. Lilja, "Low power/area branch prediction using complementary branch predictors," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–12, 2008.

[8] S. McFarling, "Combining branch predictors," Tech. Rep. Technical Note 36, Western Research Laboratory, June 1993.

[9] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "Sonicboom: The 3rd generation berkeley out-of-order machine," in *Fourth Workshop on Computer Architecture Research with RISC-V*, vol. 5, 2020.

[10] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *Commun. ACM*, vol. 63, p. 93–101, jun 2020.

[11] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanović, and B. Nikolić, "Chipyard: Integrated design, simulation, and implementation framework for custom socs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020.

[12] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, p. 1–17, sep 2006.

APPENDIX

Timing Results				
Configuration	**MCF (user)**	**BZIP2 Dryer (user)**	**BZIP2 Program (user)**	**ASTAR (user)**
Null	8:33.60	20:20.86	10:17.17	28:49.85
Global	7:33.63	15:19.68	7:08.77	24:24.27
Local	7:44.73	16:39.85	7:14.92	24:36.94
GShare	7:05.77	14:56.32	6:37.67	21:00.78
Tournament	7:34.42	15:19.38	7:04.32	24:30.64
TAGE	7:04.29	14:56.42	6:38.94	20:26.17

Table II: Benchmark Timing Results

Speedup				
Configuration	**MCF (user)**	**BZIP2 Dryer (user)**	**BZIP2 Program (user)**	**ASTAR (user)**
Null	100.00%	100.00%	100.00%	100.00%
Global	113.22%	132.75%	143.94%	118.14%
Local	110.52%	122.10%	141.90%	117.12%
GShare	120.63%	136.21%	155.20%	137.20%
Tournament	113.02%	132.79%	145.45%	117.63%
TAGE	121.05%	136.19%	154.70%	141.08%

Table III: Benchmark Timing Results

Synthesis							
Predictor type	**Predictor Config**	**# CLB LUTs**	**# CLB Regs**	**Power (W)**	**WNS (ns)**	**WHS (ns)**	**# LUTs + Regs**
Null	BPD Max Meta Length = 0 Global History Length = 0 Local History Length = 0 Local History Sets = 0	87616	61583	2.326	0.427	0.011	149199
Global	BPD Max Meta Length = 64 Global History Length = 32 Local History Length = 0 Local History Sets = 0	92870	64842	2.319	0.413	0.01	157712
Local	BPD Max Meta Length = 64 Global History Length = 0 Local History Length = 32 Local History Sets = 128	93129	64881	2.332	0.229	0.01	158010
GShare	BPD Max Meta Length = 45 Global History Length = 16 Local History Length = 1 Local History Sets = 0	93413	65132	2.336	0.203	0.01	158545
Tournament	BPD Max Meta Length = 64 Global History Length = 32 Local History Length = 32 Local History Sets = 128	93483	65228	2.339	0.385	0.01	158711
TAGE	BPD Max Meta Length = 120 Global History Length = 64 Local History Length = 1 Local History Sets = 0	99375	75435	2.364	0.427	0.01	174810

Table IV: Benchmark Timing Results