Development and Selection of Optimal Branch Prediction Methods

A. Davis
Dept. of Computer Science
Syracuse University
adavis@syr.edu

***Abstract*—Modern computing often uses instruction level parallelism (ILP) to improve processor performance. Oftentimes when instructions are executed in parallel a branch condition cannot be evaluated immediately, thus triggering a control hazard. Branch Prediction is a computer architecture technique that attempts to minimize processing time by predicting the path that will be taken at a branch. Over time many varieties of static and dynamic branch predictors have been proposed, analyzed, and improved upon. This paper seeks to examine how previous computer architects have answered the question: *what is the best method for predicting the outcome of a branch in parallel processing?* Based on a review of previously implemented techniques, the optimal choice of branch predictor depends on a variety of factors. Some of the most versatile options for both accuracy and efficiency focused on the combination of both static and dynamic prediction techniques.**

I. Introduction

One of the most effective methods for achieving speedup in modern computing is through parallel processing [1]. Moore's Law states that transistor density-—and by extension processor speeds-—will double nearly every two years. While not indefinitely sustainable, this prediction has held true for the past several decades since it was made. These consistent increases in chip density have made it possible for architectural improvements and superior processing methods to be implemented that have further improved processor speeds such as computer pipelining [2].

Pipelining considers a model in functional units and, similar to an assembly line, means that the processor can start performing different operations on the same functional unit prior to finishing with a previous one. Pipelining allows parallelism of steps even within a single processor [3]. Programs are ordinarily written with the intention of their execution occuring in sequence. Instruction-Level Parallelism (ILP) is a method that allows several instructions to be executed simultaneously and may even change the order in which the instructions are executed [3].

However, executing instructions simultaneously and/or out of order can become hazardous due to *dependencies*. Sometimes information produced as a result of an earlier instruction is needed for use in a later instruction—for example the definition of a condition that determines the path taken at a branch. This creates what is known as a *control dependency*, where the choice of the subsequent instruction is conditional on information evaluated in later pipeline stages. Waiting for the required information to determine the path taken at a branch can lead to additional stall cycles, thus decreasing the performance benefits of ILP. When a stall occurs, the processor does not perform any useful work and instructions are delayed by the length of the stall time. One way to minimize branch stalls is by predicting the branch the path will take and continuing execution in the predicted direction. Correctly predicting branches can improve processor performance, decrease execution time, and ultimately improve throughput [4].

Thus, branch prediction techniques are critical to maximizing the benefits of ILP [5]. Approximately one in every eight instructions is a branch. Thus, it is a significant deficit to performance to stall the pipeline each time a branch is encountered. Branch prediction uses information at compile-time or from past iterations of the branch to predict which path the branch will take next, prior to having the necessary information. This allows the pipeline flow to continue execution in parallel. Unfortunately, these predictions are not always correct. Mispredictions lead to a need for recovery logic to stop the current flow, flush the pipeline of incorrect instructions, and proceed down the correct path. These mispredictions come at a high cost and should be avoided if at all possible [6].

In addition to methods for increasing accuracy, power performance and cache performance are also factors that may be impacted by a computer architect's choice of branch predictor. Branch prediction represents a significant power expense on a system and may also way into the choice of an optimal predictor [7]. Similarly, cache performance may be reduced by increasing code size or the amount of branch history stored. If miss penalties in a cache are high enough, this could also reduce the utility of a branch predictor in increasing performance [5]. Thus, there are many factors to consider in the choice of branch prediction method.

II. Background

Prior to 1990, the most prevalent method of branch prediction relied solely on most recent behavior of one branch to predict what its behavior will be next. Popular methods can be divided into two varieties: static and dynamic [5].

Dynamic Branch Prediction methods utilize processor hardware to decide during program execution which direction should be taken at each branch to fetch the next instruction.

These schemes use information from previous encounters with branches in the execution history to change the prediction during run-time of the program [6]. Amongst dynamic schemes, the most popular is 2-bit, saturating, up/down counters. These used a branch history table (BHT) which logs the paths taken on previous runs of the program [5].

Static Branch Prediction methods are simpler in that they utilize software and are gathered at compilation of a program, prior to execution [8]. A single static prediction is made for each branch instruction. As it's name implies, this prediction does not change during execution [6]. However, it is possible in the implementation of static branch prediction to actually run the program with a *profiler* and training data to obtain a log of the branch directions that were most frequently taken. This knowledge can later be put to use in branch prediction using *hint bits* in the branch opcode.

In general, dynamic predictors have been shown to achieve considerably higher accuracy than static predictors. They are able to adapt to the different datasets that can be used by a single program [6].

III. Dynamic Branch Prediction Schemes

A dynamic branch predictor is separated into two units: a direction predictor and a target predictor. The direction predictor determines if the branch is taken or not-taken, meanwhile the target predictor determines the target address for a taken branch. Target prediction is simpler; once a branch instruction has been encountered once it usually becomes very accurate to re-predict as the target often remains constant for a long period of time. The real difficulty is in direction prediction. Thus, many approaches have been developed to try to better accomplish this task.

Most of these approaches can be further classified into two categories: local and global predictors. Local predictors use the past behavior of the singular branch being predicted to formulate the next prediction. Global predictors use the past behavior of some pre-determined number of preceding branches to dynamically predict for the current branch [6]. Some common dynamic prediction schemes include:

A. 1-Bit Branch Predictor

A 1-bit branch prediction scheme is the simplest dynamic method. A singular bit is used to store the most recent behavior of a branch (i.e. 0 if not-taken, 1 if taken). The prediction for the next behavior of this branch is then based entirely off of the branch's most recent behavior [9].

B. 2-Bit Saturating Counter Branch Predictor

The 2-bit predictor scheme is a dynamic scheme that relies upon the recent behavior of a single branch to make future predictions [9]. It takes two mispredictions for a simple 2-bit, saturating counter to flip its prediction. The 2-bit predictor transitions between four possible states: strongly taken (ST), weakly taken(WT), weakly not-taken(WN), and strongly not-taken(SN)—incrementing and decrementing between the four with each actual outcome of the branch. This method also uses a Branch History Table (BHT) to track the history of each individual branch instruction. Each entry of the BHT consists of a 2-bit value which represents one of the four possible states. Referencing the BHT requires a fixed number of the least significant bits of the branch instruction's address (i.e. Program Counter) [4]. The simple increase from one to two bits shows a significant increase in prediction accuracy, especially in the case of loops. When a loop exits, there inevitably is one misprediction, however when the loop branch is reached again it most likely will not start by not being taken. Thus, needing two misses to change the prediction can lead to an improvement in the prediction accuracy for loop behavior [8].

C. Bimodal Prediction

Bimodal predictors consist of two types of counters: direction and choice counters. Direction counters are either taken or not-taken predictor counters: consisting of a 2-bit value that increments and decrements like the 2-bit saturating counter. The choice predictor is indexed by the branch instruction address. This choice predictor is then used to select out of the two options for the direction predictor. The value in the table at the corresponding index location is considered the final prediction. Finally, the corresponding branch counter is updated with the predicted branch outcome. Choice counters are always updated with an instruction outcome, except for when the choice counter is opposite to the instruction's outcome but the selected predictor counter has made a correct final prediction. Bimodal branch predictors utilize a Global History Register (GHR) whose width, m corresponds to the history considered from the last m branches [4].

Large bimodal predictors have been shown to see accuracy up to 93.5% [6]. The bimodal branch predictor can also help eliminate the issue of destructive aliasing. This is a data conflict that occurs when multiple branches hash to the same entry in a dynamic predictor table [10]. Aliasing is discussed in greater detail in Section VI.

D. Local Branch Prediction Methods

Local branch predictors use two tables. The first is indexed by low-order bits of the program counter to store the taken/not-taken history for the current branch. The second is similar to the bimodal history table but is indexed using the content from the first of the two bimodal tables. Thus, the prediction produced for the given branch is based on its own previous behavior. This method has seen up to 97.1% accuracy [6].

E. Global Branch Prediction Methods

Global predictors use the past behavior of some pre-determined number of preceding branches (n) to dynamically predict for the current branch. Most global predictors keep a shift register which holds history of the taken/not-taken behavior for some number n previous branches. This is used to index a table of bimodal counters. The global predictor is never as good as the local predictor but can be almost as accurate and is faster and easier to implement. It has seen accuracy up to 96.6% [6].

F. Combined Branch Prediction

Combined prediction uses three predictors: bimodal, gshare, and another bimodal-like predictor. The bimodal-like predictor is used to determine which of the previous two predictors to use data from. This operation is performed uniquely for each branch and achieves similar accuracy to local predictors, yet speeds nearing that of global prediction [6].

G. Neural Branch Prediction

Neural branch predictors utilize neural networks to formulate directional predictions. They incur high latency in the formaion of the prediction, however increase quickly in accuracy during program execution [6].

H. Correlating Prediction

Correlating prediction observes that the behavior of some branches may impact the behavior of other branches. In fact, the direction taken at a conditional branch is frequently determined by the branch's historical pattern of outcomes; or the historical outcome pattern of neighboring branches. In order to utilize this behavior in prediction methods, the predictor needs to consider more branch behaviors than just that of a single branch. A general example of this method would be for an (m, n) predictor. In this case, m would capture the number of previous branches captured and the model could choose from 2^m branch predictors. Each of these m branch predictors would be an n-bit predictor of a singular branch. The benefits of this method is a higher accuracy than a simple 2-bit scheme. The downside is the additional hardware required to implement it [11].

Thus, an optimization concern arises regarding the trade-off between increasing performance at the cost of increasing complexity and required storage. The number of bits in an (m, n) predictor can be represented by expression:

$$\#\,bits\; =\; 2^m\; *\; n\; *\; \#\,prediction\,entries \qquad (1)$$

A simple 2-bit predictor is the equivalent of a correlating predictor where $(m, n) =$ (0,2). [9].

The origin of the correlating predictor stemmed from the observation that the standard 2-bit predictor which uses only local information missed predictions that could have been successful had the predictor also had global information about the program. The success of correlating predictors led to the origin of another dynamic predictor: the tournament predictor [12].

I. Tournament Predictors

Tournament predictors utilize predictors at multiple levels: both local and global. Information from both of these predictors is combined using a selector. Tournament predictors further improve accuracy. Tournament predictors use the 2-bit saturating counter scheme for each branch. At run-time, the program can then choose between two different predictors depending on which type of predictor—local, global, or a mix—has had a better track record of accuracy in recent predictions.

The primary benefit of a tournament predictor is this ability to select from local, global, or a mixture of predictors which is optimal for a particular branch. More specifically the different types of predictor counters are Local History, Local Prediction, Global Prediction, and Choice Prediction counters. Local History counters are referenced by the branch instruction address and the corresponding value is used to index the local predictor table. The Global Prediction branch history resides in the GHR and updated on each branch outcome. The Choice counter entry is used to make the choice between the local and global counters. The final prediction relies on the selected counter value from one of the two branch predictors. Based on the Local Prediction and Global Prediction counters and the actual outcome, the Choice Prediction counter is updated accordingly. The final step is to update the Local History counters, Local Prediction counters, Global Prediction counters, and GHR according to the actual outcome [4].

It has been seen that in tournament predictors, the global predictor is selected approximately 40% of the time for integer operations, however it is used a mere 15% of the time with floating point operations. Alpha processors debuted tournament predictors, however they have also been used in AMD processors including Opteron and Phenom [12].

IV. Static Branch Prediction Techniques

While often less accurate, static branch prediction can be an energy and resource efficient alternative to dynamic prediction. Static branch prediction is a software method that optimizes branch prediction at compile-time and does not involve any changes within a singular run-time. However, running the program may still be necessary for certain static techniques such as profiling. Static schemes have been used to answer the question of how to predict branch direction in the following ways:

A. Predict Not-Taken

The most simplistic static branch prediction is to assume a branch is never taken. This is the same as never making the prediction at all and just assuming the next instruction will always be consecutively executed. This least sophisticated method is also the least effective [5]. A slightly more accurate variation is to assume that the branch is always taken instead [6].

B. Predict Backward Pointing Branches as Taken

This prediction method increases the accuracy for loops, as they are a series of backward pointing branches that are almost always taken until the last iteration [6].

C. Profile-Based Branch Prediction

Fisher and Freudenberger showed that profile-based branch prediction is often accurate since the majority of branches take a singular direction more often and also because this more probable direction is often the same across different program executions. This phenomenon is commonly referred to as a *bias* [9].

However, program profiling also involves instrumenting a program with counting code using either the compiler or an additional external tool. Next, the program with the counting code must be executed—usually over several iterations—to produce a profile. Lastly, the program requires recompilation with the additional help of the profile. Therefore, at a minimum profiling involves two compilations and an execution. Additionally, the whole process must be repeated given any changes to the code as the profile changes [8]. Optimization and importance of the algorithms involved in profiling is examined more closely in Section VII.

D. Program-Based Branch Prediction

Program-based prediction oftentimes performs at a factor of two worse than profile-based prediction, however it still reaches enough accuracy to have a reasonable amount of utility. Ball and Larus attempt to optimize on this method using various heuristics, as is further discussed in Section VII [8].

E. Perfect Static Predictor

The perfect static predictor is ideal-case scenario for performance of a static branch predictor. It strictly predicts in the more frequently executed direction for every singular branch. In the case that most branches show strong preference to one direction over the other, a perfect static predictor will perform very well. However, on the contrary, in the case where the majority of branches take either direction approximately half of the time (programs with branches that have little bias), a perfect static predictor could be anticipated to perform no better than a 50% miss rate. This theoretical predictor provides the upper bound on any static predictor. Ball and Larus evaluate static branch predictor performance for a given predictor, P, as: C/D, where C is the percentage of dynamic branches that are mispredicted by P, and D is the miss rate for the perfect predictor [8].

F. Static Correlated Branch Prediction (SCBP) Method

SCBP is a static method which takes the information obtained by dynamic branch correlation statistics collected at run-time and utilizes them in the software of the program itself to improve static branch prediction accuracy. Inspired by a dynamic scheme known as GAs, SCBP was developed to benefit from the correlation data obtained in a branch profile to improve global branch prediction accuracy, but using only compiler-specified branch prediction bits. It operates by encoding branch history into the PC, meanwhile it also generates extra copies of the blocks with unique branch histories which [11].

Thus, SCBP is a static software branch prediction scheme that improves performance of a singular branch by generating several versions of the same block. However, as a consequence, SCBP also increases code size and by extension the burden the program places on the cache. As a further consequence, this increases the number of compulsory and capacity misses. Frequency of these misses increase since SCBP increases the amount of code required to complete a process. Increased code also leads to decreased spatial locality as code is spread across a larger amount of space. Thus, by extension, SCBP also leads to more conflict misses [5].

Gloy looks beyond simple prediction accuracy for selection of a branch predictor and considers the point at which the negative impacts of increased code volume on memory from SCBP outweigh its ability to increase processor performance. Previous studies show that in order to reap the benefits of SCBP it can take up to a 30-110% code increase. Gloy considers the combined effects that both cache behavior and branch prediction have on processor performance. In Gloy's analysis he saw that the increase in blocks fetched to the cache was actually even greater than the overall code expansion ratio. This increase was even worse for larger values of k (where k is the history depth saved per conditional branch). Fortunately, it was also seen that SCBP misprediction rates do not necessarily decrease monotonically with increasing k, but rather can actually see a worsening prediction accuracy with too large of a value of k. Thus, there is likely an optimal value of k which can help limit the severity of the code increase [5].

SCBP has been shown to achieve the greatest performance boosts when used in conjunction with profiling and branch alignment techniques. Branch alignment techniques control and limit the negative impacts of code expansion on instruction cache performance. SCBP has the greatest improvement on performance on machines with high mispredict/misfetch penalties and low miss rate/miss penalties in the cache. This is the case because this method can cause occurrences of these to decrease and increase, respectively [5]. Thus, as the implementation of future processors leads to larger instruction caches and incur larger misprediction penalties, SCBP will continue to grow in relevance [11].

Following his analysis on SCBP, Gloy concluded that static prediction methods that maximize prediction accuracy do not necessarily maximize application performance. Placing limitations on the degree of branch correlation performed by SCBP may actually have the potential to improve overall application performance [5].

V. Impacts of Branch Prediction on Energy Efficiency

As a byproduct of decreasing CPU time, branch prediction accuracy also has the benefit of increasing energy efficiency. Decreasing strain on the processor decreases overall static power dissipation. Stalling is a poor option and should be avoided at all costs. However, dynamic branch techniques also come at a cost. Even after years of improvements, dynamic branch prediction is still responsible for up to 10% of the total power dissipation from a processor. Thus, in addition to a desire to reduce computation time—improving branch prediction accuracy is also motivated by a desire to optimize energy. The power consumed by a dynamic branch predictor is correlated to the size of the rest of the processor [6].

Hicks proposes a novel approach to reducing the power draw of dynamic branch prediction using a combination of local delay region, scheduling, profiling, and hint bits. Loss

of any prediction accuracy would only incur additional power waste, thus Hicks sought to propose a new branch prediction method that sacrificed minimal accuracy while also decreasing power usage. He proposed a combinational algorithm that reduced the dynamic branch predictor activity factor by reducing the number of accesses to dynamic branch predictors. His proposal involved bypassing the dynamic predictor for certain branch instructions such as unconditional or strongly biased branches and by combining dynamic techniques with static techniques such as profiling. The decision between static and dynamic prediction methods were made by comparing the dynamic bias to the dynamic branch predictor accuracy for a given branch. When a branch had more bias than the dynamic prediction, the static prediction was assigned instead. Hicks' research showed that approximately 63% of dynamic branch predictor accesses could be eliminated, thus reducing global power draw by a significant margin. His proposed solution was estimated to reduce the power consumption of the global system by 6.2%. This estimate would likely be even higher in cases where the dynamic predictor is less accurate as more branches will be hinted.

In conclusion, Hicks stated that the topic of decreasing power consumption in dynamic branch predictions is far from saturated. There is a definite potential for improvements on this front by combining static scheduling/hinting with dynamic branch prediction methods. In doing so, a significant amount of dynamic branch predictor accesses can be avoided along with their corresponding power consumption [6].

Another study of branch predictor impacts on power/energy performance supported that it is worthwhile to spend *more* power on a branch predictor if it results in more accurate predictions that ultimately improve running time. In fact, for integer programs, larger but more accurate programs actually reduce energy consumption. A few specific techniques were seen to have no negative impacts on branch prediction accuracy while improving energy efficiency including: banking, implementation of a prediction probe detector (PPD), and use of bias information. Banking leads to a reduction in energy usage by decreasing the activity of the branch predictor by using only a fraction of the full predictor structure. A PPD utilizes predecode bits to avoid unnecessary accesses to a Branch Target Buffer (BTB)—causing as much as 3% of the total system energy. Parikh et. al. also encouraged the use of data regarding strong branch biases. Branches with a strong affinity to a certain direction should not waste energy accessing a BTB, but rather should predict in favor of the bias, similar to the suggestion made by Hicks in his thesis [7].

VI. Combining Techniques

As mentioned in Section V, Hicks proposed the combination of static and dynamic methods to achieve one cohesive prediction technique with adequate accuracy and improved energy performance [6]. However, traditional branch prediction is segregated into either static or dynamic prediction with little combination of the two. Predictors are usually either static and compiler-assigned or dynamically implemented via hardware. If dynamic prediction is going to be implemented, static is then usually deemed as obsolete.

Hicks proposes the implementation of Adaptive Branch Bias Measurement (ABBM). This is a technique that uses static profiling data in combination with the dynamic branch predictor to allow static hints to be utilized in an instruction, but only in instructions that do not require dynamic prediction. Combining this technique with static scheduling has the potential to significantly reduce the number of accesses made to the dynamic predictor unit. ABBM not only saved energy, it was also shown by Hicks to increase overall prediction accuracy instead of sacrificing it [6].

A combination of static and dynamic branch prediction has also been shown to have the benefit of reducing destructive aliasing [10]. Aliasing occurs in branch prediction when multiple branches hash to the same entry in a dynamic predictor. Thus, these different branches are then stuck using the same predictor entry. This can result in several possibilities: destructive interference (i.e. aliasing), neutral interference which does not change the prediction, or positive interference which corrects the prediction as a result. Aliasing is the most common outcome and it results in a misprediction [9]. The aliasing issue was previously combated by increasing predictor size to hopefully reduce identical mappings, selecting a different indexing scheme, or separating different classes of branches into different prediction schemes [10].

A new option proposed by Patil and Emer involved implementing a dynamic scheme but also using static predictions wherever possible to hopefully reduce these interferences. However, it then becomes critical to choose the correct set of branches to predict statically instead of dynamically [10].This makes profiling data critical. The two targeted varieties of branches to statically predict were ones with a strong bias making them easy, or, ones whose dynamic bias is also so poor that they are difficult to predict anyways. These ones might as well also be predicted statically. Using this technique saw improvements in branch prediction accuracy, however it also emphasized the importance of accurate profiling methods [10].

VII. Optimal Profiling

Algorithms exist to profile and trace programs. Effective profiling algorithms can greatly reduce the expense of inserting code into each basic block. A program profile counts the number of executions for each basic block, whereas a program trace reports on the sequence of basic block execution for a program. Profiling can be performed by either a *vertex profile* or and *edge profile*, depending on where in the execution of the code counts are made for the frequency of execution. Ball and Larus show that when profiling, there is a benefit to placing the instrumentation code on edges rather than vertices. This is in large-part because there are more edges than vertices. Using edges provides more opportunities to place profiling code in areas of low execution frequency. More accurately profiling code could increase the accuracy of static branch prediction, and vice versa; more accurate static branch prediction can allow for better placement of instrumentation

code for program profiling, thus decreasing instrumentation overhead and making profiling a more desireable option as a branch prediction method [13].

Another nuance on profiling is the difference between path profiling and point profiling. Path frequencies report the frequency in which a sequence of program blocks are executed in that specific order. Paths are useful to consider as then can be useful for reducing conditional-branch overheads in modern processors [11].

VIII. Utilizing Profile Information

Identifying frequently executed regions makes it possible to aid in scheduling instructions. Profile-based predictors are a static method, as discussed in Section IV, that can greatly improve the accuracy of branch prediction. Unfortunately, they are also time-hungry and tedious: requiring a repetitive compile-profile-compile cycle to obtain accurate predictions. Furthermore, due to their static nature, any changes to the code necessitate the repetition of the entire profiling process. [8].

Ball and Larus proposed a *program-based* branch predictor. Since many programs execute non-loop branches more frequently than loop branches, Ball and Larus sought to propose a solution that considers beyond just loops. According to their work, a perfect static predictor would have the capability to predict all branches (both loop and non-loop) with a mere 10% miss rate. Their actual predictions following their study saw a miss rate of 26% for non-loop branches and 20% for loop branches.

Most static compiler optimizations rely upon the identification of heavily-executed paths. The static predictor they worked with was binary: meaning either the target or fall-thru successor must be predicted. The target successor gains control of the branch condition evaluates to true, whereas the fall-thru is taken in the case that the branch condition is false [8].

A. Loop vs. Non-Loop Branches

The ability to predict non-loop branches is critical to the creation of a better predictor. Ball and Larus show that a static predictor can be very capable of doing so. As previously mentioned in Section III, most loop branches can be identified as backwards pointing branches, however this excludes many other loop branches such as iterations that exit. In some of the benchmarks, loops had branches that pointed forwards more than 40% of the time. Ball and Larus strictly redefine loop branches for their problem as a branch where either of its outgoing edges is an exit edge or a loop backedge. They redefine a non-loop branch as a branch where neither of its outgoing edges is an exit edge or a backedge. For loop prediction, if either one of its directions is a backedge, that is predicted. Otherwise, the non-exit direction is predicted. This process, called natural loop analysis, sees an extremely high degree of accuracy.

In the trickier case of non-loop branches, simply predicting the target or fall-thru successor is a poor option—and in some cases even inferior to an entirely random prediction scheme. The best approach static prediction can take to these non-loop branches is to use knowledge of the program to choose the direction that executes with higher probability [8].

B. Additional Considerations for Non-Loop Branches

In the study of predicting non-loop branches there are some additional factors available at compile-time that may be useful to consider in the static predictor model. One of these factors is instruction opcodes. Certain opcodes for different branches have different ratios of likelihoods for being taken/not-taken. Accounting for these improves the static approach. Another nuance to consider is whether the instruction is a call, return, guard, or store. Ball and Larus developed a system of what they called *heuristics*. In their system, each heuristic consisted of two parts: a selection property and a predictor. In this case the selection property dominates the choice. If neither of the possible successor blocks after the conditional branch had the selection property, or both had the property—the predictor then made the choice either with or without the property depending on the *heuristic* for that conditional branch.

For calls, in the case where the next block contains a call or unconditionally passes to a block with a call—predict *away* from the direction with the call. This is because call branches that decide between executing or avoiding a call are actually are highly more likely to not make the call. In the case of returns, if a successor block contains a return or unconditionally passes to a block with a return, predict away from the return. This is because returns are usually an exception or base case, like in recursion [8].

Ball and Larus then had the task of combining the different local heuristics, and acknowledged that more than one heuristic may apply to a single branch. To remedy this, they ordered the heuristics and checked for their presence in order and used their prediction for the first one that matched. Finally, Ball and Larus combined their heuristics into a single program-based static branch predictor that performed well for a variety of programs [8].

IX. Opinion and Analysis

Following my review of the many techniques of branch prediction: it is clear that there is a divide between static and dynamic techniques with limited overlap between the two. Within each of these domains, there are still many different prediction scheme options. Thus there is still the question of: *Which prediction scheme is the best for my application?* This question has the same frustrating answer as many other questions in the realm of computer architecture: *it depends*.

Based on my review of past work on this topic, the answer to this question could depend upon a wide variety of factors. When employing static methods, I would be concerned with my cache and the miss penalties involved as I now know that these methods often increase code size. Knowing these values for the system you are looking to design your branch prediction technique for should certainly play a role in the decision. If cache miss penalties are low, SCBP as a static method—or even more memory-strenuous dynamic methods such as correlating predictors that use more bits to increase

accuracy—may be good options. I would also be concerned with the methods and algorithms that I use for my profiling information. Accurate profiling information is critical to static predictions, but it can also lead to additional challenges and overheads.

The answer to this question may also come down to the amount of specificity you want it to have to an individual program. Based on the research provided by Ball and Larus in their papers *Optimally profiling and tracing programs* and *Branch prediction for free*, it seems that static methods can be tuned more to a specific program, for example using profiling methods that involve a cycle of profiling and compiling. While this makes the prediction scheme less versatile, I would infer that spending a large amount of time building an optimal profile for a singular program could ultimately yield highly accurate results. On the other hand, this is an undesirable option if the program is still subject to change or if the developer of the branch predictor does not have a lot of time to devote to this process, therefore the choice can also depend on that.

Gloy, Smith, and Young bring up yet another valid point in their paper *Performance Issues in Correlated Branch Prediction Schemes* with their concern regarding static methods such as SCBP (also discussed in *Static Correlating Branch Prediction*), resulting in negative repercussions for the system as a whole in terms of increased cache misses. For this reason and due to the fact that static prediction methods do not see as high of accuracy as dynamic methods, it seems undesirable to employ a purely static method.

Another point about the SCBP method brought up by Young in his paper, *Static correlated branch prediction*, is the importance of accurate profiling and tracing. The high success of the SCBP amongst static methods, and the importance of good profiling and tracing to the accuracy of this method also lead me to think that development of the best branch prediction method cannot be considered independently from improvements to these technologies. Thus, the discussion of improving profiling and tracing methods presented in Ball and Larus's *Optimally profiling and Tracing* and *Branch prediction for free* could also hold key positions in the improvements of branch prediction methods.

Overall, I stand with the argument made by Hicks in his paper *Energy Efficient Branch Prediction* that there must be an optimal way to combine the best of the existing static and dynamic branch prediction methods. His proposed method could use more research than his singular analysis to verify the validity of his claim that no accuracy was sacrificed in his combination of these methods. If the proposed ABBM method truly sees no loss of accuracy, while increasing system energy efficiency then this could often be a worthwhile method to implement. Especially as modern technologies decrease in size and increase in portability. ABBM would be a friendlier option than some to the cache as it is not a purely static method like SCBP, yet could also play nice with a system's need to conserve power. I think that moving into the future, further examination of hybrid static/dynamic schemes could be worthwhile. It also appears that the research topic of determining which branches are better for static prediction rather than dynamic prediction in a static/dynamic hybrid scheme is not yet a saturated area of study.

X. Conclusion

With the ever increasing enhancements being made to modern processors, the emphasis on instruction level parallelism (ILP) is also increasing. Unfortunately, ILP comes with the consequence of many potential hazards: data hazards, structural hazards, and control hazards. Control hazards may arise at points where the program may proceed to one of two different locations depending on a condition that cannot yet be evaluated. To avoid expensive and wasteful stall cycles, branch prediction methods are used to predict the outcome of the condition and retrieve the next instruction. Unfortunately, mispredictions can also be very expensive. Thus the questions arises: *what is the best method for predicting a branch in parallel processing?*

This question has been answered in a variety of ways that can be segregated into two main categories: dynamic prediction schemes and static prediction schemes. Dynamic prediction schemes utilize computer hardware to make and improve branch prediction at run-time. Static prediction schemes utilize software techniques to make branch predictions at compile-time. Dynamic schemes are in large-part more accurate, but also have a tendency to be more complex. Dynamic schemes range in complexity, from the simplest schemes which consider only 1-bit local preictions, meaning whether or not a singular branch was last taken or not taken, all the way up to tournament predictors which combine 2 or more other prediction methods which can consider global predictions into the past for many different branches.

Static schemes also cover a range of complexity and detail as well. The simplest schemes innvolve predicting always taken/not-taken. Static schemes grow in difficulty and accuracy when profiling and tracing become involved. These are techniques that involve running a program with training data to improve branch prediction for a specific program. Finally, one of the more complex and accurate static schemes examined was SCBP which uses profiling data to create correlating predictions at compile-time. Unfortunately, SCBP is also seen to have negative repercussions in the form of increased cache misses due to increasing code size. Static techniques can also incorporate additional *heuristics*. These are rules-of-thumb that can be implemented into static predictions based on the nature of the instructions and have been shown to add additional accuracy to predictions.

Finally, accuracy is not the only factor to consider when choosing the best branch prediction scheme. More intensive schemes have also been shown to have negative repercussions on energy efficiency for a system. This was shown to be alleviated using a combination of static and dynamic techniques—using static predictions where they should be mostly accurate to reduce energy expense. Ultimately, selection of the best branch prediction technique is an optimization problem which

must balance accuracy, energy efficiency, and complexity. Fortunately, there are many highly accurate options available depending on the needs of the system and the nature of the program.

REFERENCES

[1] Dubey, P.K., Flynn, M.J., Optimal pipelining, *Journal of Parallel and Distributed Computing*. 8 (1990) pp. 10–19.

[2] B. Rau and J. Fisher. "Instruction-level parallel processing: History Overview and Perspective." The Journal of Supercomputing, 7:9-50, 1993.

[3] J. Fisher, "Instruction-Level Parallel Processing," Science, vol. 253, (5025), pp. 1233, 1991.

[4] B. Kalla, N. Santhi, A. A. Badawy, G. Chennupati and S. Eidenbenz, "Probabilistic Monte Carlo simulations for static branch prediction," 2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC), San Diego, CA, 2017, pp. 1-4.

[5] C. Young, N. Gloy, and M. Smith, "A Comparative Analysis of Schemes for Correlated Branch Prediction," *Proc. 22nd Ann. Int'l Symp. Computer Architecture*, May 1995.

[6] M. A. Hicks, "Energy Efficient Branch Prediction." Order No. U236628, University of Hertfordshire (United Kingdom), Ann Arbor, 2010.

[7] Parikh D, Skadron K, Zhang Y, Stan M. "Power-aware branch prediction: characterization and design." IEEE Trans Comput. 2004; 53(2): 168-186. High-Performance Computer Architecture, Jan. 2000.

[8] T. Ball and J. R. Larus, "Branch prediction for free," Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation (published in SIGPLAN Notices) 28(6) pp. 300-13 ACM, (June 1993).

[9] Mittal S. "A survey of techniques for dynamic branch prediction." Concurrency Computat Pract Exper. 2018; 31(1):e4666.

[10] H. Patil and J. Emer, "Combining Static and Dynamic Branch Prediction to Reduce Destructive Aliasing," Proc. Sixth Int'l Symp.

[11] C. Young, M. D. Smith, "Static correlated branch prediction", ACM Trans. Program Languages and Systems, vol. 21, no. 5, pp. 1028-1075, Sep. 1999.

[12] John L. Hennessy and David A. Patterson. 2011. Computer Architecture, Fifth Edition: A Quantitative Approach (5th. ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[13] Optimally T. Ball, J. R. Larus, "Optimally profiling and tracing programs", ACM Trans. Program. Languages Syst., vol. 16, no. 4, Jul. 1994, pp. 1319-1360.