Confidence Estimation for Speculation Control

Dirk Grunwald and Artur Klauser

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309
{grunwald,klauser}@cs.colorado.edu

Srilatha Manne and Andrew Pleszkun

Department of Electrical and Computer Engineering
Campus Box 425
University of Colorado
Boulder, CO 80309
{bobbie,arp}@cs.colorado.edu

Abstract

Modern processors improve instruction level parallelism by speculation. The outcome of data and control decisions is predicted, and the operations are speculatively executed and only committed if the original predictions were correct. There are a number of other ways that processor resources could be used, such as threading or eager execution. As the use of speculation increases, we believe more processors will need some form of *speculation control* to balance the benefits of speculation against other possible activities.

Confidence estimation is one technique that can be exploited by architects for speculation control. In this paper, we introduce performance metrics to compare confidence estimation mechanisms, and argue that these metrics are appropriate for speculation control. We compare a number of confidence estimation mechanisms, focusing on mechanisms that have a small implementation cost and gain benefit by exploiting characteristics of branch predictors, such as clustering of mispredicted branches.

We compare the performance of the different confidence estimation methods using detailed pipeline simulations. Using these simulations, we show how to improve some confidence estimators, providing better insight for future investigations comparing and applying confidence estimators.

1 Introduction

Speculation is a fundamental tool in computer architecture. It allows an architectural implementation to achieve higher instruction level parallelism, and thus performance, by predicting the outcome of specific events. Most processors currently implement branch prediction to permit speculative control-flow; more recent work has focused on predicting data values to reduce data dependencies [10].

Confidence estimation is a technique for assessing the quality of a particular prediction. Confidence estimation has usually been studied in the context of branch prediction. Jacobsen *et al* [7] described a number of uses for confidence estimation: they suggested that it may be used to improve the branch prediction rate, control resource use in a dual-path execution pipeline or control context switching in a multithreaded processor.

In this paper, we study the design of confidence estimators and make the several contributions. First, we feel that confidence estimators will usually be used for some form of *speculation control*. Previous metrics used to compare confidence estimators would result in inappropriate design decisions. We introduce standard, consistent metrics to compare the performance of confidence estimators, and argue that different applications of confidence estimators require different metrics. Second, we compare hardware intensive confidence estimators against several less complex estimators that use existing branch prediction or processor state information. While the complex implementation has uniformly better performance, the less complex methods have similar performance and a significantly reduced implementation cost, making them appealing for many of the practical cases where confidence estimation would be used. Lastly, our pipeline-level simulations indicate ways to improve the hardware-intensive confidence estimator in an actual implementation.

In the next section, we describe *screening* or *diagnostic tests*, and adopt their terminology for branch prediction and confidence estimation. In §2, we apply this terminology to confidence estimation, and conduct a series of measurements to compare different confidence estimators. We close with a discussion of temporal aspects of branch predictors and how they can be exploited to improve confidence estimation.

1.1 Diagnostic Tests

The following description is adapted from a paper by Gastwirth [4], as described in [1]. A *diagnostic test* is used to determine if an individual belongs to a class D of people that have a particular disease, or to the class of people who do not have the disease, $\bar{D}$. The result of a test places a person either into the class S, those who are suspected of having the disease, or class $\bar{S}$. The accuracy of the diagnostic test is indicated by two parameters: *sensitivity* and *specificity*. The sensitivity is defined to be $\text{SENS} = P[S|D]$, or the probability that a person with the disease is properly diagnosed. The specificity is $\text{SPEC} = P[\bar{S}|\bar{D}]$, or the probability that a person who does not have the disease is correctly diagnosed. For good tests, both SPEC and SENS are close to one.

The problem with all diagnostic tests is that if a disease occurs infrequently, there will be a large number of "false positives" – the diagnostic test will indicate that a person has the disease when in fact they do not. This can be expressed as $P[S|\bar{D}] = 1-P[\bar{S}|\bar{D}] = 1-\text{SPEC}$. The last metric of interest is the probability that someone has a disease, $p = P[D]$. In most tests, we are interested in the *predictive value of a positive test* (PVP), which is $P[D|S]$. The PVP is the probability that a person has the disease given that a test indicates they might.

Gastwirth cites a study of the ELISA test for AIDS used to

screen donated blood, where the sensitivity was SENS $= 0.977$, indicating that the test should find samples with the disease, and the specificity was SPEC $= 0.926$, indicating that most tests that come back positive would really have the disease. The large values for SENS and SPEC can be misleading for large populations or for very rare diseases. For example, assume that only 0.01% of the population actually has AIDS($p = 0.0001$). Then, using the above equation, we compute PVP $= P[D|S] = 0.001319$. In other words, *even if the diagnostic test indicates you have the disease, there is only a* 0.13% *probability that you actually have the disease, simply because the disease is so rare.*

So far, we have described parameters of diagnostic tests independently of the *cost* of different outcomes. For example, in the ELISA test for AIDS, it is very important to have a high sensitivity – tainted blood samples shouldn't be accepted. However, it's acceptable to have a lower specificity, because you may be able to use a series of (more expensive) tests to determine if the person really has the disease.

2 Confidence Estimation as a Diagnostic Test

It is more difficult to compare two confidence estimators than two branch predictors in part because confidence estimators can be used for a number of purposes while branch predictors are typically only used to predict the outcome of control-dependent instructions. Most architectures are designed to use speculation and the general assumption is that "you might as well be doing something", and thus each branch is predicted.

By comparison, we think that confidence estimators will normally be used for *speculation control*. For example, if a particular branch in a Simultaneous Multithreading [14] processor is of low confidence, it may be more cost effective to switch threads than speculatively evaluate the branch. A confidence predictor attempts to corroborate or assess the prediction made by a branch predictor. Each branch is eventually determined to have been predicted correctly or incorrectly. For each prediction, the confidence estimator assigns a "high confidence" or "low confidence" to the prediction. In addition to the standard terminology of diagnostic tests, we have found that another notation simplifies the comparison of different confidence estimators. We draw a 2×2 matrix listing the frequency for each outcome of a test. When we apply this framework to architectural simulation, each of the quadrants can be directly measured during simulation or analysis. Typically, we normalize the values to insure that the sum equals one. Thus, our quadrant table for confidence estimation is:

		Prediction Outcome C	Prediction Outcome I
Confidence	HC	C_{HC}	I_{HC}
	LC	C_{LC}	I_{LC}

In this table, "C" and "I" refer to "correct" and "incorrect" predictions, respectively, and "HC" refers to "high confidence" and "LC" to "low confidence". During a simulation, we can measure C_{HC}, I_{HC}, C_{LC} and I_{LC} using a branch predictor for each branch and concurrently estimate the confidence in that branch predictor using a specific confidence estimator. When the branch is actually resolved, we classify the branch as belonging to class C_{HC}, I_{HC}, C_{LC} or I_{LC}.

2.1 Metrics for Comparing Confidence Estimators

There are many possible designs for confidence estimators, and we need a consistent method to compare the effectiveness of two confidence estimators. To date, only Jacobsen *et al* [7] have published comparisons of confidence estimators, and their paper considered only two designs. When converted to our terminology, Jacobsen *et al* defined the "confidence misprediction rate" as $I_{HC} + C_{LC}/C_{HC} + I_{HC} + C_{LC} + I_{LC}$. This represents the fraction when the confidence estimator was wrong or disagreed with the eventual branch outcome. Jacobsen *et al* also defined the "coverage" of a confidence predictor as $C_{LC} + I_{LC}/C_{HC} + I_{HC} + C_{LC} + I_{LC}$.

We believe that when a confidence estimator is applied, the architectural feature using that confidence estimation will either be used for "high confidence" *or* "low confidence" branches, but not both. Since the "confidence misprediction rate" includes both outcomes, we felt more effective metrics needed to be designed. For example, consider a simultaneous multithreading (SMT) processor that uses a confidence estimator to determine if a predicted branch is likely to be mispredicted. If the branch prediction is of "low confidence", the processor may switch to another available thread rather than fetch additional instructions from the current thread. The performance of such a processor is very sensitive to $P[I|LC] = I_{LC}/C_{LC} + I_{LC}$, the probability that the branch is incorrectly predicted if it was low confidence. A high value for $P[I|LC]$ indicates that the processor can switch contexts only when the following instructions will not commit. A low value of $P[I|LC]$ indicates that the SMT processor may needlessly switch threads, reducing the performance of the primary thread. A low value of the SPEC ($P[LC|I]$) means that the processor will miss some opportunities to improve aggregate performance by switching threads.

Not all uses of confidence estimators will make the same kind of decisions, but we feel it is most useful to compare confidence estimators using metrics that reflect how the confidence estimators are used. For example, SMT processors want a confidence estimator with a large $P[I|LC]$ and a large $P[LC|I]$. We have found in our own discussion that terms such as "accuracy" and "coverage" tend to cause confusion, because accuracy has an inherit implication about the application of a technique. Thus, we use neutral terms that also have the benefit of being standard terms in statistics. Each of these metrics is easy to compute, and each is a "higher is better" metric. To simplify discussion, we assign the following names to these conditions.

Sensitivity: The SENS is $P[HC|C] = C_{HC}/C_{HC} + C_{LC}$, and represents the fraction of correct predictions identified as "high confidence".

Predictive value of a Positive Test: The PVP is $P[C|HC] = C_{HC}/C_{HC} + I_{HC}$ and represents the probability that a high-confidence estimate is correct.

Specificity: The SPEC is $P[LC|I] = I_{LC}/I_{HC} + I_{LC}$, and represents the fraction of incorrect predictions identified as "low confidence".

Predictive value of a Negative Test: The PVN is $P[I|LC] = I_{LC}/C_{LC} + I_{LC}$ and represents the probability that the a *low-confidence* estimate is correct.

There is a natural relation between the SPEC and PVN and the SENS and PVP that can be clarified by an example. Assume a program executed 100 conditional branches. Of those, 20 are mispredicted. The confidence estimator indicates "high confidence" for 61 of the 80 correctly predicted branches and 2 of the incorrectly

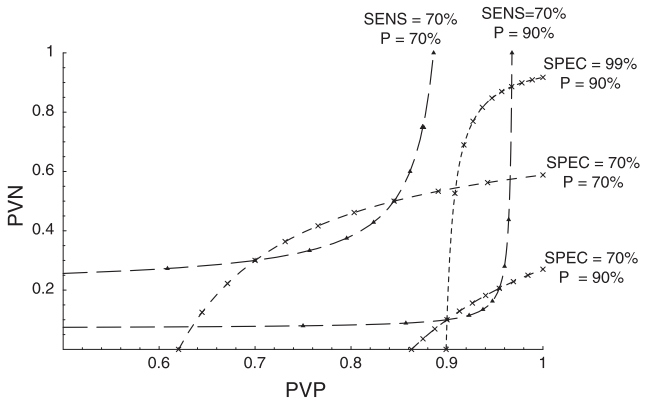

Figure 1: Parametric plots showing how the sensitivity (SENS), specificity (SPEC) and branch prediction accuracy (p) influence the values of PVP and PVN. Each line shows the value of PVP and PVN when we hold two values constant and vary the third value. For example, in the right-most curve, the specificity and branch prediction accuracy are held constant, and we vary the value of the sensitivity. The markers on each line indicate the decile values of the parameter being varied.

predicted branches. It indicates "low confidence" for 19 of the 80 correctly predicted branches and 18 of the 20 incorrectly predicted branches.

		Prediction Outcome	
		C	I
Confidence	HC	61	2
	LC	19	18

The SENS would be $\frac{61}{61+19} = 76\%$, and the PVP would be $\frac{61}{61+2} = 97\%$. A larger SENS indicates more of the correctly predicted branches are correctly estimated, and a larger PVP indicates that the confidence estimator doesn't designate incorrect predictions as "high confidence". The SPEC would be $\frac{18}{18+2} = 90\%$, indicating that the confidence estimator is good at finding most of the incorrectly predicted branches. The PVN would be $\frac{18}{18+19} = 49\%$, indicating that the confidence estimator is reasonably able to exclude correctly predicted branches. Since branch predictor accuracy is $C_{HC} + C_{LC}$, the SENS and SPEC are independent of the branch predictor accuracy. In other words, SENS is only a property of correctly predicted branches, and SPEC is only a property of incorrectly predicted branches.

Figure 1 provides some insight into the relation between the SENS, SPEC, prediction accuracy, PVP and PVN. The curves are plotted for values of SENS, SPEC and prediction accuracy (p) that are representative of the measured values that will be discussed in §3. For a given sensitivity and prediction accuracy (e.g., $[\text{SENS} = 70\%, p = 70\%]$ and $[\text{SENS} = 70\%, p = 90\%]$), increasing the sensitivity will greatly improve the PVP until it reaches an asymptotic limit and then improves the PVN. Likewise, for a given SPEC and prediction accuracy (e.g., $[\text{SPEC} = 70\%, p = 70\%]$, $[\text{SPEC} = 70\%, p = 90\%]$ and $[\text{SPEC} = 99\%, p = 90\%]$), increasing the SENS improves the PVN. This improvement is faster if the SENS is high or the branch prediction accuracy is low.

When designing a confidence estimator, we need to understand whether the final application will be using the PVP or PVN and the importance of the SENS and SPEC to that application. Typically, we would not want to change the branch prediction accuracy; although we can increase the PVN by decreasing the prediction accuracy, this would be counter-productive for most applications of confidence estimation.

2.2 Using confidence estimators

Although the particulars of any given application are beyond the scope of this paper, there are a number of obvious uses of confidence estimation with associated costs that can illustrate the importance of the relative values of these metrics. We have described one such application (speculation control for simultaneous multithreading), and list four others.

Bandwidth multithreading: In a multithreading CPU designed to assume a large number of threads, the architectural model would be more willing to switch threads if there is any uncertainty in the outcome of a branch. Unless the confidence estimator returned a "high confidence" estimate, the architecture would switch threads. Thus, we want a confidence estimator with a high SENS, meaning that most correct branches are identified as high-confidence, and a high PVP, meaning that most branches designated as high confidence are predicted correctly.

SMT: As mentioned, in this architecture, you could use a confidence estimator to control the number of instructions issued by individual threads. Since this architecture would err on the side of speculatively issuing instructions, confidence estimators with a high PVN are very important, while PVP would be less important. A higher SPEC means that more opportunities for avoiding wasteful speculation are identified.

Power conservation: In related work [11], we are investigating how to use confidence estimators to reduce power usage in a processor by suppressing instruction issue following low-confidence branches. The goals in the power conservation architecture are similar to those of the SMT design, and we want a confidence estimator with large PVN and SPEC.

Eager Execution: Some proposed architectures evaluate instructions on both paths of a conditional branch [16, 9, 15, 6, 8]. These architectures might use a confidence estimator to determine when to diverge and evaluate both paths. A confidence estimator with high PVN would indicate that a low-confidence estimate for a given conditional branch has a high chance of being a mispredicted branch and may benefit from eager execution. A higher SPEC would mean that more opportunities for applying eager execution are found.

Improving Branch Predictors: Jacobsen *et al* [7] suggested that a confidence estimator could be used to improve the accuracy of a branch predictor. If the PVN $> 50\%$, then the confidence estimator can improve the branch prediction accuracy by inverting the outcome of a low-confident branch. Conversely, if PVP $< 50\%$, then the branch prediction for high-confident branches should be inverted. We have examined many confidence estimators in many configurations, but have not found a situation where these conditions hold across a range of programs.

To summarize, in most of these applications, a higher PVN would improve the underlying architecture, but none of the applications needing a higher PVN would sacrifice prediction accuracy

to increase the PVN. A higher SPEC would indicate that the architectural optimization (multithreading, eager execution, power conservation) might have greater impact because more of the opportunities where it can be applied are exposed. Our own immediate applications for confidence estimation (power conservation and eager execution) biased our investigation towards confidence estimators with a high PVN and SPEC.

3 Comparison of Confidence Estimators

We have implemented four confidence estimators either discussed or implied in existing literature, and used our performance metrics to compare their performance. Later, we examine the temporal characteristics of branch predictors and show how those properties can be used to design another inexpensive confidence estimator.

JRS Estimator: The first method we implemented is one-level resetting counter mechanism proposed by Jacobsen, Rotenberg, and Smith (JRS) [7]. This predictor uses a miss distance counter (which we call an MDC) table in addition to the branch predictor. The structure of the confidence estimator is similar to that of the Gshare predictor. An index is computed using an exclusive-or of the program address and the branch history register. This index is used to read a value from a table of MDCs. The width of these counters can vary in size, but we used 4-bit counters as suggested in [7]. We used a large table containing 4096 4-bit counters. Each time a branch is predicted, the value of the MDC is compared to a specific threshold. If the value is above that threshold, then the branch is considered to have high confidence, otherwise it has low confidence. When a branch resolves, the corresponding confidence counter is incremented if the branch was correct; otherwise, it is reset to zero. We tried all different threshold levels, and show detailed results for a threshold of 15 and show the trend for other thresholds. We called this the *JRS* confidence estimator.

Pattern History Estimator: Lick *et al* [9, 15] proposed a confidence estimator for dual-path execution. the confidence estimator was used to determine when dual-path execution should be used. Although neither of the available papers focused on the confidence estimator itself, the basic design is described. Lick *et al* observed that a small number of branch history patterns typically lead to correct predictions in a branch architecture using a PAs predictor (i.e., a BTB with a branch history stored for each branch site). The confidence estimator assigned high confidence to a fixed set of patterns and treated all other patterns as low confidence. Essentially, the patterns were always taken, almost always taken (once not-taken), always not-taken, almost always not-taken and alternating taken and not-taken. We called this the *pattern history* confidence estimator.

Saturating Counters Estimator: The third method we implemented was originally proposed in an early paper by Smith [13]. Here, we use the state of the *saturating counters* used in many branch prediction mechanisms to determine the confidence estimate. For example, in a simple gshare predictor, branch outcomes are determined by the state of a two-bit counter. We called this the *saturating counters* method.

Static Estimator: The last technique uses a *static confidence hint*. Here, we executed the program and simulated the underlying branch predictor (e.g., a gshare predictor). We record the number of correct outcomes for each branch instruction, and then use a "threshold" to determine confident branches. In our examples, we used a threshold of 90%, meaning that a branch with $\geq 90\%$ branch prediction accuracy was considered to have high confidence, and all other branches had low confidence. The results we report are from self-profiled executions where the same input was used to train and evaluate the confidence predictor. Thus, these results present a best-case evaluation of this confidence method. We mainly include this technique to indicate its potential.[1]

3.1 Experimental Methodology

Each of the confidence estimation techniques makes assumptions concerning the underlying branch predictor. Later, we compare these methods when using a gshare and a McFarling branch predictor [12]. In each case, the structure of the confidence estimator may change due to the branch predictor, and we indicate those changes there. We use the SimpleScalar [2] execution-driven simulation infrastructure to compare the different confidence estimators. Our simulator is an extension of the *sim-outorder* simulator, with a 5-stage pipeline and an additional 3 cycle misprediction recovery penalty.

We use a 64 kB L1 Dcache and a 128 kB L1 Icache[2], both with 2 cycle access latency. Our simulator knows the outcome of all branches at the point of instruction decode, even for branches that do not actually commit. This includes branches following a mispredicted branch. We essentially recorded a "speculative trace" for the processor, recording the prediction and eventual outcome of committed and uncommitted branches. We did this to compare the difference in branch prediction and confidence estimation for committed and uncommitted branches. When the processor is executing a conditional branch, it does not know if a branch will commit or not, so it is important to understand how *all* branches are predicted and estimated. It may be that some pattern arises in the uncommitted branches that would impact confidence estimation. We will always restrict our discussion to committed instructions unless we indicate otherwise. For example, when we report the SPEC and PVN for different confidence estimators, we only report these values for the committed instructions.

We used the SPECint95 benchmarks for our performance evaluation and did not simulate the SPECfp95 since those programs typically pose few difficulties for branch predictors. The benchmarks and important measurements from our simulations are listed in Table 1.

We used three underlying branch predictors to compare the different confidence estimators: a speculative gshare predictor, a speculative McFarling combining predictor [12] and a non-speculative SAg [17] predictor. Figure 2 gives a schematic illustration of each branch predictor. The gshare branch predictor (Figure 2a) combines a global branch history with the program counter to select a two-bit counter. The SAg predictor (Figure 2b) uses the program counter to index into an untagged table of branch history registers that are used to select a two-bit counter. The bimodal predictor (Figure 2c) is used in the combining predictor and uses the program counter to index a table of two-bit counters. The combining predictor (Figure 2d) uses both a gshare (Figure 2a) and bimodal (Figure 2c) predictor. A table of two-bit counters is used to select a component branch predictor for each prediction.

Only the gshare and combining predictors are speculatively updated. Non-speculative update would slightly increase the branch

[1] It is important to note that the "profile" technique cannot use a simple program profile, since the decisions depend on outcome and state of the branch predictor. Thus, the "profile" technique requires a branch predictor simulation (which is much slower than a simple profile) or hardware that reports performance information for the underlying branch predictor, such as the Profile-Me mechanism [3].

[2] The Icache is equivalent to a 64 kB cache, since SimpleScalar has a 64-bit instruction encoding, but we only use 32 bits for each instruction, so half the space is wasted.

	committed instructions						all instructions						ratio all/committed	
		conditional branches						conditional branches						
	inst. (million)	number (million)	taken	misprediction rate			inst. (million)	number (million)	taken	misprediction rate				
application				gshare	McF.	SAg	gshare			gshare	McF.	SAg	inst.	cond. bra.
compress	80.4	14.4	54.6%	10.1%	9.9%	10.1%	108.6	19.4	50.0%	16.9%	17.2%	19.9%	1.35	1.35
gcc	250.9	50.4	49.0%	23.9%	12.2%	12.8%	455.9	91.0	49.5%	33.5%	20.9%	21.6%	1.82	1.81
perl	228.2	43.8	52.6%	25.9%	11.4%	9.2%	402.7	76.5	52.9%	34.3%	18.8%	16.7%	1.76	1.74
go	548.1	80.3	54.5%	34.4%	24.1%	25.6%	1116.3	165.0	51.4%	41.1%	31.8%	33.5%	2.04	2.06
m88ksim	416.5	89.8	71.7%	8.6%	4.7%	4.7%	563.3	118.0	68.8%	14.9%	9.1%	11.8%	1.35	1.31
xlisp	183.3	41.8	39.5%	10.2%	6.8%	10.3%	263.6	59.2	39.8%	17.8%	14.4%	22.3%	1.44	1.42
vortex	180.9	29.1	50.1%	8.3%	1.7%	2.0%	225.6	37.4	48.2%	15.7%	4.0%	4.1%	1.25	1.29
jpeg	252.0	20.0	70.0%	12.5%	10.4%	10.3%	301.6	28.4	67.9%	20.1%	18.8%	17.8%	1.20	1.42
mean	267.6	46.2	54.3%	14.5%	8.1%	8.6%	429.7	74.4	52.8%	22.5%	14.6%	16.2%	1.61	1.61

Table 1: Program characteristics, differentiating between committed instructions and both committed and uncommitted instructions. The processor will typically issue 20-100% more instructions than actually commit, due to speculative execution. The values for speculative execution were measured when using the gshare branch predictor.

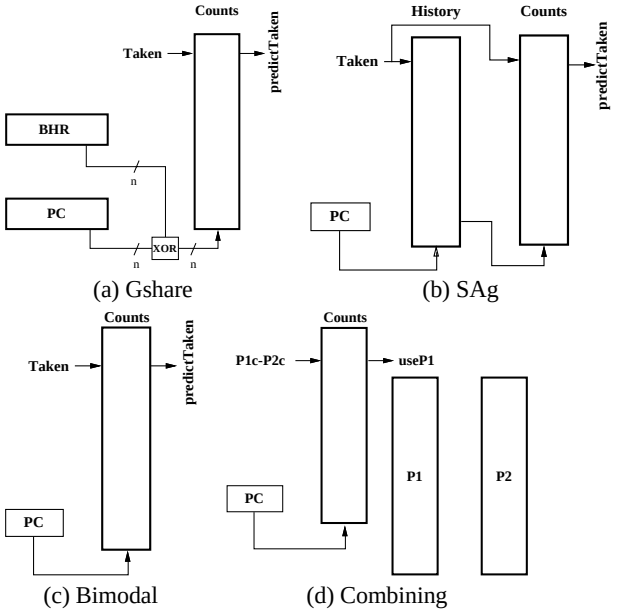

Figure 2: Schematic illustration of the different branch predictors

misprediction rate, since information from recent branches is not immediately available to succeeding branches. The SAg model is similar to the PAs, which is usually implemented with a branch target buffer, but the SAg is "tagless" and may alias branch histories. It is difficult to roll back from speculative history updates in a PAs or SAg predictor, and we did not implement speculative update for that reason. Restoring the table at a branch misprediction requires multiple cycles as each non-committed predicted branch restores its old history state in the branch history table (BHT). Alternatively, the whole BHT could be checkpointed for each predicted branch, and restored on misprediction. This scheme requires space to store multiple copies of the BHT. The SAg is much more expensive to implement than Gshare or McFarling, and only offers similar performance (see Table 1).

Throughout our analysis and comparison, it is important to remember that the JRS estimator is significantly more expensive to implement than either the saturating counters, the history pattern or the profile method, since extra tables and state are needed by the JRS estimator.

3.2 Comparison of Confidence Estimators When Using a Gshare Branch Predictor

In our first configuration, we used a 4096-entry gshare branch predictor. The JRS confidence estimator was implemented as described above. We implemented the history pattern confidence estimator using both the values determined by Lick *et al* and by repeating their measurements for the gshare predictor, selecting new "highly confident" patterns. In our presentation, we only show results using the patterns specified by Lick *et al* since there appear to be no dominant patterns in the global history register when using a gshare predictor. The saturating counters method used the heuristic described above - strongly taken or strongly not-taken branches were considered confident and all others were not confident. We used a 90% threshold for the static, profile-based technique.

The first column of Table 2 shows the performance of the different confidence estimators when using the gshare predictor. We report the geometric mean of the sensitivity, specificity, PVP and PVN for each confidence estimator; detailed information on each application can be found in [5]. The averages are computed from the averages of the original data. In other words, when computing the average for the PVP, we take the mean for C_{HC} and C_{LC} and compute $\bar{C_{HC}}/\bar{C_{HC}}+\bar{C_{LC}}$, rather than averaging the existing PVP's.

Unless we consider a specific application for the confidence estimators, it is difficult to select one estimator over another. In general, the JRS estimator has the highest PVP and an acceptable PVN, and the profile-based estimator is roughly similar. The saturating counter method has a better PVN than the JRS or profile method, but at the expense of a lower PVP. This occurs because the saturating counter method is more sensitive (i.e., reduces the relative value of low-confidence predictions for correct branches). However, the test is not very specific, and incorrectly classifies many incorrectly predicted branches as "high confidence" branches. The history pattern method fares poorly when using this and the McFarling predictors because no dominant patterns emerge. Since those patterns don't occur, the history pattern method will classify most branches as "low confidence", leading to a low sensitivity. Since most branches are marked "low confidence", most of the incorrectly predicted branches will be correctly diagnosed as low confidence.

	Gshare Predictor				McFarling Predictor				SAg Predictor			
Confidence Estimator	**sens**	**spec**	**pvp**	**pvn**	**sens**	**spec**	**pvp**	**pvn**	**sens**	**spec**	**pvp**	**pvn**
JRS, Threshold >= 15	56%	96%	98%	30%	64%	93%	99%	23%	64%	94%	99%	24%
Saturated Counters	88%	42%	88%	41%	67%	78%	96%	21%	90%	48%	94%	36%
History Pattern	17%	94%	93%	19%	18%	89%	94%	11%	73%	81%	97%	26%
Static, Threshold > 90%	55%	89%	96%	28%	72%	88%	98%	26%	66%	93%	98%	30%

Table 2: Comparison of Confidence Estimators when using a Gshare, McFarling and SAg branch predictors

3.2.1 Enhancing the JRS Estimator

We use an enhanced implementation of the JRS confidence estimator that improves performance. Rather than use the same branch history to index the branch prediction *and* MDC table, we first predict the branch and include that prediction when we index the MDC table. Figure 3 shows the noticeable performance difference. Each point on the lines indicates the performance when changing the "threshold" value. This improvement requires reading out both alternative MDC counters and then selecting the appropriate result when the branch prediction completes. We use this implementation throughout the remainder of the paper.

Figure 4 shows the PVP and PVN for the JRS estimator for different possible configurations of the hardware. As before, each line shows the results when we vary the number of the four-bit MDC entries, and each point on a line indicates the performance when changing the "threshold" value. The right-most point uses a threshold of 16; since this cannot be reached by a four-bit MDC, all branches are marked "low confidence", and the PVN is equal to the misprediction rate.

More branches are marked "low confidence" at a higher threshold. This increases the SPEC, but also decreases the PVN since more correctly predicted branches are marked as "low confidence". Lowering the threshold has the opposite effect: the SENS will increase, but the PVP will decrease. Selecting the appropriate configuration of the JRS estimator, as with selecting the appropriate configuration of *any* estimator, depends very much on the intended application.

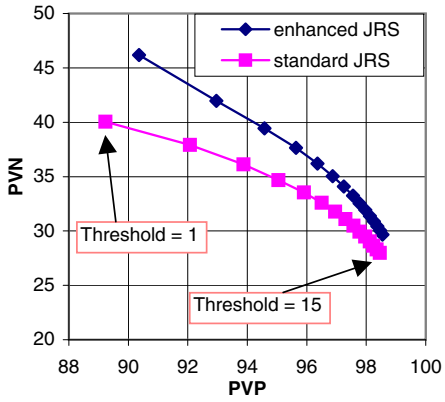

Figure 3: Performance of the JRS enhanced confidence estimator.

3.3 Comparison of Confidence Estimators When Using a McFarling Branch Predictor

In the second comparison, we used a McFarling combining predictor that combines the results from a gshare predictor and a table of two-bit saturating counters indexed only by the program counter. As indicated in [12], this configuration offers the best performance for the predictor sizes we are using in this evaluation. The JRS, static and history pattern confidence estimators were implemented as before. The "saturating counters" method was modified to use information from both prediction mechanisms in the combining predictor.

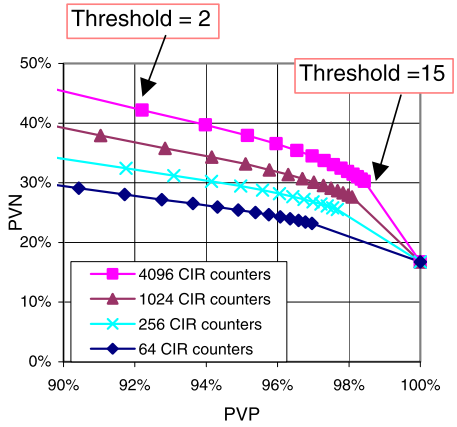

Figure 4: Performance of the JRS confidence estimator when using the Gshare predictor, as the design parameters are varied.

3.3.1 Saturating Counters Estimator for McFarling Predictors

In the McFarling predictor, two different two-bit counters provide branch predictions, and a "meta-predictor" chooses between the two predictions. Each component, the gshare or bimodal predictors, uses a two-bit counter to provide hysteresis in the branch prediction. In the McFarling predictor, both component predictors are queried for each branch prediction. A third table, the meta-predictor information, is used to determine which predictor should be used. When the branch actually commits, both branch predictors

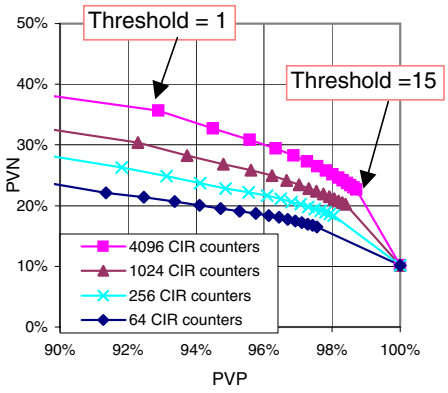

Figure 5: Performance of the Smith confidence estimator when using the McFarling predictor, as the design parameters are varied.

are updated. If the component predictor results were different, the meta predictor moves to re-enforce the use of the correct component predictor. Otherwise the meta predictor is unchanged.

There are a number of sources of information for the "saturating counters" mechanism. We found that two techniques work well, and that each has a benefit depending on the desired performance metric (PVP or PVN). We are not interested in the *direction* of a branch prediction, just the likelihood that the prediction will be correct. Thus, we categorize each branch component predictor as offering a "strong" or "weak" prediction, where the transitional states in the state machine are considered "weak" predictions. Ignoring the information from the meta predictor, there are now four states: (Strong, Strong), (Strong, Weak), (Weak, Strong), (Weak, Weak).

In the "Both Strong" variant, we signal "high confidence" only when both predictors are strongly biased in the same direction, and "low confidence" otherwise. In the "Either Strong" variant, we signal "low confidence" only when both branch predictors are in the "weak" state, and high confidence otherwise. Table 2 shows only the "Both Strong" variant to simplify the data presentation. Table 3 compares the "Both Strong" and "Either Strong" variants. The "Both Strong" method has a higher SPEC and PVP since only "strongly" predicted branches will be marked as high confidence, reducing the total number of correctly estimated low-confidence branches. Conversely, the "Either Strong" method will have a high SENS, lower PVN and higher PVP, since more branches will be considered "low confidence".

We also looked at a number of variations on these techniques which use the saturation state of only the selected counter to determine the confidence, information from the meta-predictor, or different combinations of the state information. However, these methods generally had a lower SPEC and PVN. Since we were mainly interested in applications of confidence estimation that emphasize the SPEC and PVN, we do not include those results in the paper.

The relative merits of the different estimators change when considering the McFarling branch predictor, as shown in the middle column of Table 2. In this configuration, the JRS, saturating counter and profile-based techniques are roughly similar. The JRS mechanism is more specific than the other methods, meaning it will identify more incorrectly predicted branches, but the PVN is about the same for each of those estimators.

	Saturated Counters							
	Both Strong				Either Strong			
application	sens	spec	pvp	pvn	sens	spec	pvp	pvn
compress	68%	77%	96%	21%	97%	18%	91%	38%
gcc	54%	80%	95%	20%	96%	15%	89%	36%
perl	52%	83%	96%	18%	96%	17%	90%	36%
go	36%	84%	88%	29%	91%	18%	78%	39%
m88ksim	79%	52%	97%	11%	99%	12%	96%	33%
xlisp	78%	68%	97%	18%	98%	15%	94%	34%
vortex	85%	76%	100%	8%	99%	17%	99%	33%
jpeg	77%	75%	96%	28%	97%	18%	91%	42%
Mean	67%	78%	96%	21%	97%	17%	91%	37%

Table 3: Performance of Low-Confidence *vs.* High-Confidence thresholds with the McFarling branch predictor

The SPEC of the JRS method decreases when we switch to the McFarling predictor. We believe this happens because the prediction accuracy is higher, and there are fewer incorrect predictions to identify. Identifying those few remaining incorrect predictions is more difficult. Essentially, the branch predictor is finding the easier mispredictions and thus improving the misprediction rate. The SPEC for the saturating counter estimator improves greatly when compared to the Gshare predictor, in part because the two-bit predictor in the Gshare has such a low specificity to begin with. The PVN of all the branch estimators is significantly lower when using the McFarling branch predictor. In part, this occurs because the underlying branch predictor is more accurate and the confidence estimator has to work harder to find mispredictions.

Figure 5 shows the performance of the JRS estimator as the hardware configuration is varied. The trends are similar to that explained in §3.2, but the overall PVN is lower.

3.4 Comparison of Confidence Estimators When Using a SAg Branch Predictor

The third comparison, shown in column three of Table 2, uses a SAg predictor with 2048 branch history entries and an 8192-entry counter table. Each branch history register was 13 bits long.

Since the counter entries are only two bits, the saturating counters estimation method performs poorly in this configuration, just as it did when using the Gshare predictor. Similarly, the JRS and static estimators have similar performance to that seen when using the gshare predictor. The performance of the history pattern estimator improves dramatically for SAg, where it performs roughly equivalent to the static and JRS methods. In addition, it has a much lower implementation cost than JRS and does not require profiling like the static method. Therefore, the history pattern estimator is very competitive for a SAg branch predictor.

3.5 Summary of Comparisons

Several observations arise from our comparison of confidence estimation techniques. First, the performance of a confidence estimator appears to be very dependent on the branch predictor and confidence estimator having a similar design or indexing method. For example, the JRS estimator has better performance for the gshare mechanism (to which it is similar) than for the McFarling predictor, and the History Pattern technique has excellent performance when using a SAg, but poor performance when using a global history, as in Gshare or McFarling. This indicates that we may be able to design a better variant of JRS for the McFarling predictor. Second, our improvement to the JRS method indicates the value of including more recent information in the confidence estimation process.

Our comparison also shows the value of inexpensive confidence estimators such as static profiling, the "saturating counters"

method, and the History Pattern technique. These methods performed almost as well as the JRS technique when using different branch predictors, but they require very little additional hardware to implement. It also shows that it is unlikely, albeit not impossible, that confidence estimation may be used to directly improve branch prediction, since none of the confidence estimators we examined had a PVN consistently greater than 50%.

4 Temporal Aspects of Branch Prediction and Confidence Estimation

We originally began studying confidence estimators because we are using them for a number of applications, including some of those mentioned in §2.2. We wanted to focus on confidence estimators with a low implementation cost. During our investigation, we made a number of observations concerning the temporal aspects of branch prediction and we have used these observations to design alternative confidence estimators.

4.1 Branch Misprediction Clustering

If branch mispredictions are *clustered*, then we may be able to use the distance since the last mispredicted branch as a confidence estimation mechanism. Our measurements confirm the observation of Heil and Smith [6] that mispredictions in a trace were clustered. However, we have found the degree of clustering is different when you look at all branches (e.g., during a pipeline-level simulation) or only at the committed branches (e.g., branches in a normal program trace). We use the information from all branches because that is what is actually of interest to an architect in a real pipeline or a pipeline level simulation.

Our data shows that mispredictions are tightly clustered, with few branches between mispredicted branches. Heil and Smith [6] plotted the probability distribution function of the branch misprediction distance. If branches are independent (and not clustered), that graph has a geometric distribution with a parameter equal to the misprediction rate. We found that presentation difficult to understand, and found it easier to understand if we plot the data as in Figure 6. In this figure, we graph the *misprediction rate* vs. the distance to the previous mispredicted branch. If mispredictions were not clustered, we would expect the misprediction rates to all be the same, as indicated by the average lines. Instead, we find that branches immediately following a misprediction are more likely to be mispredicted. In Figure 6, we plot two views of the data from our simulations. The data marked "all branches" includes both committed and uncommitted branches, whereas the "committed branches" includes only committed branches. Heil and Smith used a trace for their analysis, and only report the data for committed branches. We used a gshare branch predictor to generate the data in Figure 6, but we also used a precise value for the distance to the previous mispredicted branch – the processor model has complete knowledge of the pipeline state. Again, this corresponds to the information that would be recorded by a trace when we consider the committed branches without a pipeline-level simulator. Figure 7 shows a similar plot using the McFarling branch predictor.

A real architecture determines mispredictions when a branch is resolved, and not when a misprediction is actually made, as in our "precise" model. This will lengthen the time, and thus the number of branches executed, until the misprediction is actually detected, and should skew the branch clustering such that it appears to occur over a larger branch distance. Figure 8 shows the corresponding misprediction rate *vs.* misprediction distance when we only use information from resolved branches, using the same gshare branch predictor. Figure 9 shows similar information for the McFarling

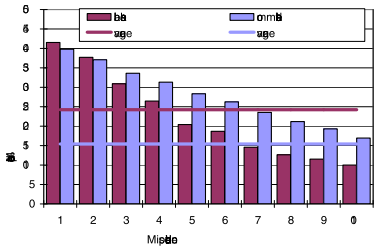

Figure 6: Misprediction distance using a gshare branch predictor and precise misprediction information. The vertical axis shows the misprediction rate of predictions that are made a specific number of branches after a previously mispredicted branch.

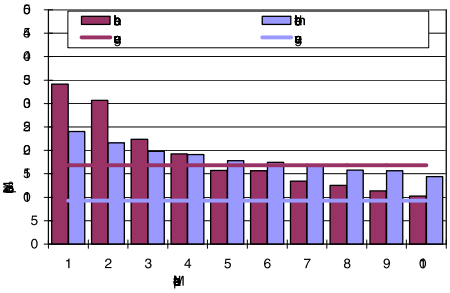

Figure 7: Misprediction distance using a McFarling branch predictor and precise misprediction information.

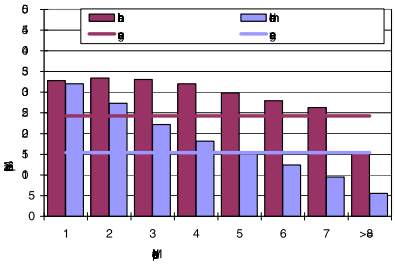

Figure 8: Perceived misprediction distance for Gshare predictor. This shows the misprediction rate of branches a specified number of branches after the most recent misprediction detected by the processor.

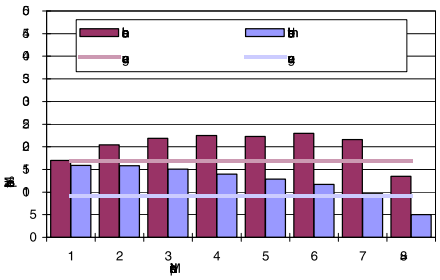

Figure 9: Perceived misprediction distance for McFarling predictor.

Table 4: Using misprediction distance as confidence estimator

branch predictor. As expected, both Figure 8 and Figure 9 still show clustering, but the results are skewed to higher misprediction distances. Interestingly, the distribution for all branches using McFarling predictor has a different shape than when using the gshare predictor; however, the committed branches have a very similar distribution. This occurs because of the variable time needed to determine if a branch misprediction has occurred.

Precise pipeline information is unavailable to a processor during execution, but it illustrates why the JRS estimator works. The JRS miss distance counters (MDC) are reset every time a branch misprediction is detected, and branches are not marked as "high confidence" until several branches mapping to that MDC register have been correctly predicted. Since branches are clustered, the "reset and count" insures that enough branches have executed to bypass the cluster of poorly predictable branches. You can use this same behavior to design a *misprediction distance confidence estimator*, which is essentially a JRS confidence estimator with a single MDC register. If more than a specific number of branches have been fetched since the last resolved (but not necessarily committed) misprediction, we consider the branch to have "high confidence". Table 4 shows the average performance of this technique *vs.* other confidence estimators, using a range of distance thresholds. We can vary the distance threshold to achieve different values of SPEC and PVN. Jacobsen *et al* [7] examined a related configuration, where a global MDC was used to index into a table of correct-incorrect registers. This solution still has a large MDC table, and [7] primarily investigated using the global indexing MDC as a way to improve accuracy - they were not looking for inexpensive confidence estimators. The variation used in [7] probably did not work well for the reasons illustrated in our earlier data – unless the indexing structure of a table-based confidence estimator matches that of the underlying branch predictor, the performance will suffer. By comparison, the misprediction distance confidence estimator uses the property that mispredicted branches are clustered to achieve its performance.

We conducted a similar set of experiments to see if *confidence estimators* also cluster their "correct" confidence estimates. We measured the JRS estimator with the gshare and McFarling predictors and the saturating counters estimator with McFarling, and recorded a "mis-estimation distance" similar to the misprediction distance previously discussed. In each of these configurations, we found that correct confidence estimations are slightly clustered, but only over large distances - e.g., the confidence estimations ranged from being correct 45% of the time immediately following a misestimated branch, decaying to a 41% misestimation rate at a distance of four branches and a 33% misestimation rate for a branch distance greater than 8.

4.2 Using Clustering to Improve Confidence Estimation

Since confidence mis-estimations are only slightly clustered, we can loosely approximate confidence estimation as a Bernoulli trial, particularly over the small number of branches actually resident in a pipeline. Doing this, we can boost specific metrics, such as the PVN, by waiting for several low (or high) confidence events to occur. Recall that $\text{PVN} = P[I|LC]$, the probability of an incorrect prediction given a low-confidence estimation. Now, assume we only consider low confidence estimates - if we see two low-confidence estimates, the probability of both of those estimates being wrong is $1 - (1 - \text{PVN})^2$, since the PVN is effectively the probability of being incorrect. In certain applications, we can use this to "boost" our confidence estimates. For example, two low confidence estimates from an estimator with a PVN of 30% would have an overall PVN $\approx 50\%$.

Not all applications can benefit from this boosting, because boosting doesn't identify which of the two low-confident branches are incorrect. Boosting only indicates the probability that one of the two branches is incorrect, and thus describes the state of the pipeline rather than the state of a particular branch. An eager-execution architecture that evaluates multiple paths following a low-confidence estimate would need to start evaluation down the alternate paths of both of the low-confidence branches. An SMT processor could use the two low-confidence estimates as evidence that the instructions from the current thread are unlikely to commit, and switch to an alternate thread. Likewise, a bandwidth multithreading processor can use boosting with the PVP.

5 Conclusions and Future Work

In this paper, we have focused on developing metrics that can be used to compare confidence estimators, and then used those metrics to evaluate different confidence estimators. We have also improved variants of specialized confidence estimators and shown how existing branch prediction resources can be used for confidence estimation. Equally important, we have shown that confidence estimators appear to work best if their structure mimics that of the underlying branch predictor. Furthermore, our pipeline-level simulations have shown that branch predictors exhibit characteristics, such as clustering, that can be exploited to provide better confidence estimators. This points out the importance of using pipeline level simulations for this kind of work.

Our motivation for this work is a broad study into *speculation control*, where we hope to control how a superscalar processor uses speculative execution. Two applications are described at this conference. One application involves controlling instruction fetch and issue based on confidence estimators to reduce power demands in speculative processors [11]. The second involves controlling variants of eager execution [8]. We are also working on adaptive control of multithreaded processors to better utilize processor resources. Each of these applications emphasizes the PVN

and SPEC metrics, and is very sensitive to the branch prediction accuracy. This study has shown that as prediction accuracy increases, the PVN decreases in every confidence estimator we examined, in a large part because there are fewer incorrectly predicted branches to discover. We think most applications of confidence estimation are going to be similar to our work in speculation control, and that confidence estimation will be useful even in the presence of highly accurate branch predictors. We have focused on inexpensive mechanisms such as the "saturating counters" method, and methods to improve those estimates in particular problem domains, such as applying the boosting techniques to multithreading.

There is considerable work to be done in speculation control, particularly when applied to eager execution, control of multithreaded processors, control of the memory resources and power conservation. Speculation control will require better and more precise confidence estimators, and we look forward to progress in this area. In particular, we are working on an algorithm to "tune" static confidence estimation to achieve a particular goal for PVN or SPEC. We are also working on a confidence estimator similar to the JRS mechanism designed to better exploit the structure of the McFarling two-level branch predictor.

Acknowledgements

We would like to thank Todd Austin and Doug Burger for developing and supporting SimpleScalar, Digital Equipment Corporation for an equipment grant that provide the simulation cycles, a grant from Hewlett-Packard, and the anonymous referees for providing helpful comments. This work was partially supported by NSF grants No. CCR-9401689, No. MIP-9706286 and in part by ARPA contract ARMY DABT63-94-C-0029.

References

[1] Arnold Allen. *Probability, Statistics, and Queuing Theory*, pages 24–34. Academic Press, 1990.

[2] Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating future microprocessors: The simplescalar tool set. Technical Report TR#1308, University of Wisconsin, July 1996.

[3] Jeffrey Dean, James E. Hicks, Carl A. Waldspurger, William E. Weihl, and George Chrysos. ProfileMe: Hardware support for instruction-level profiling on out-of-order processors. In *30th Annual International Symposium on Microarchitecture*. IEEE, December 1997.

[4] Joseph Gastwirth. The Statistical Precision of Medical Screening Procedures: Application to Polygraph and AIDS Antibodies Test Data. *Statistical Science*, 2(3), August 1987.

[5] Dirk Grunwald, Artur Klauser, Srilatha Manne, and Andrew Pleszkun. Confidence estimation for speculation control. Technical Report CU-CS-854-98, University of Colorado, Dept. of Computer Science, Campus Box 430, Boulder, CO 80309-0430, Mar 1998.

[6] Timothy Heil and James Smith. Selective Dual Path Execution, November 1996. University of Wisconsin-Madison, http://www.ece.wisc.edu/ jes/papers/sdpe.ps.

[7] Erik Jacobsen, Eric Rotenberg, and J. E. Smith. Assigning confidence to conditional branch predictions. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 142–152, Paris, France, December 2–4, 1996.

[8] Artur Klauser, Abhijit Paithankar, and Dirk Grunwald. Selective eager execution on the polypath architecture. In *Proceedings 25th Annual Annual International Symposium on Computer Architecture*, Barcelona, Spain, June 1998. ACM.

[9] Kelsey Lick. Limited Dual Path Execution. Master's thesis, University of California, Riverside, 1996.

[10] Mikko Lipasti. *Value Locality and Speculative Execution*. PhD thesis, Carnegie Mellon University, April 1997.

[11] Srilatha Manne, Artur Klauser, and Dirk Grunwald. Pipeline gating: Speculation control for energy reduction. In *Proceedings 25th Annual Annual International Symposium on Computer Architecture*, Barcelona, Spain, June 1998. ACM.

[12] Scott McFarling. Combining branch predictors. TN 36, DEC-WRL, June 1993.

[13] J. E. Smith. A study of branch prediction strategies. In *Proceedings 8th Annual International Symposium on Computer Architecture*. ACM, 1981.

[14] Dean M. Tullsen, Susan Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings 22th Annual International Symposium on Computer Architecture*, Jun 1995.

[15] Gary Tyson, Kelsey Lick, and Matthew Farrens. Limited Dual Path Execution. CSE-TR 346-97, University of Michigan, 1997.

[16] A. K. Uht, V. Sindagi, and K. Hall. Disjoint Eager Execution: An Optimal Form of Speculative Execution. In *28th International Conference on Microarchitecture*, pages 313–325, December 1995.

[17] Tse-Yu Yeh and Yale N. Patt. Alternative implementations of two-level adaptive branch predictions. In *Proceedings 19th Annual Annual International Symposium on Computer Architecture*, pages 124–134, Gold Coast, Australia, May 1992. ACM.