Limited Dual Path Execution

Gary Tyson
Electrical Engineering and
ˆComputer Science Department
The University of Michigan
Ann Arbor, MI
(tyson@eecs.umich.edu)

Kelsey Lick
Department of
Computer Science
University of California, Davis
Riverside, CA
(klick@cs.ucr.edu)

Matthew Farrens
Department of
Computer Science
University of California, Riverside
Davis, CA 95616
(farrens@cs.ucdavis.edu)

Abstract

This work presents a hybrid branch predictor scheme that uses a limited form of dual path execution along with dynamic branch prediction to improve execution times. The ability to execute down both paths of a conditional branch enables the branch penalty to be minimized; however, relying exclusively on dual path execution is infeasible due because instruction fetch rates far exceed the capability of the pipeline to retire a single branch before others must be processed. By using confidence information, available in the dynamic branch prediction state tables, a limited form of dual path execution becomes feasible. This reduces the burden on the branch predictor by allowing predictions of low confidence to be avoided.

In this study we present a new approach to gather branch prediction confidence with little or no overhead, and use this confidence mechanism to determine whether dual path execution or branch prediction should be used. Comparing this hybrid predictor model to the dynamic branch predictor shows a dramatic decrease in misprediction rate, which translates to an reduction in runtime of over 20%. These results imply that dual path execution, which often is thought to be an excessively resource consuming method, may be a worthy approach if restricted with an appropriate predicting set.

1. Introduction

Changes in control flow (branches) are a well-known impediment to high performance. And as pipelines deepen and issue widths increase, this problem is increasing in importance. The penalty for fetching down the wrong path (the branch mispredict penalty) is becoming a substantial fraction of overall performance loss.

One way to eliminate the branch mispredict penalty is to avoid fetching down the wrong path. This fact has led to extensive work on improving the quality of branch predictors in order to allow the processor to speculatively execute past branches with a high degree of confidence. Another way to avoid paying the branch mispredict penalty is to fetch instructions from both the fall-through and target locations, guaranteeing that the machine is executing the correct stream of instructions. This approach is not new - the IBM 3168 and 3033 mainframes used a similar approach back in the late 1970's (instructions were fetched from both paths, but the instructions from only one path could be decoded and executed) [1].

Unfortunately, given the known distribution of branch instructions and the number of instructions that are often "in flight" in a current processor, it is inevitable that another branch instruction will be encountered by one (or both) of the streams before the initial branch condition is resolved. Providing sufficient resources to continue to split and fetch on each of these new branches can clearly lead to a cascading effect, requiring an exponentially increasing amount of hardware and potentially significantly impeding the progress of the correct stream.

In order to limit the cost necessary to support multiple path execution, Uht [2] proposed a technique called Disjoint Eager Execution (DEE) which limits the number of concurrent executing paths by using a selector to decide which paths to execute. The selectors are based on statically calculated probabilities.

However, studies of the characteristics of branch behavior [3] indicate that certain branches are much more difficult to predict than others. The fact that different branches can be predicted with different accuracies leads to the possibility of a new approach, "Limited Dual Path Execution" (LDPE), in which hardware support is provided to allow the processor to split and fetch down two (but only two) paths. The decision on whether or not to fork will be based on the prediction accuracy of the branch - if the accuracy of the predictor for a given branch is high enough, then no forking will be done.

In this paper we will describe LDPE in more detail, and our work on finding an optimal set of patterns, based on reference count and misprediction rate, to use as a limiting factor for LDPE. Finally, we will present some simulation results and our future plans.[1]

2. Limited Dual-Path Execution

Limited Dual-Path Execution works by providing sufficient hardware to support the simultaneous fetching and execution of 2 (and only 2) distinct instruction streams. A box diagram of the technique is presented in Figure 1. This figure shows that the interesting parts of LDPE lie between the BHR and Current Predictor of a 2-level branch predictor (a 2-level branch predictor is assumed since they have been shown to provide the highest prediction accuracies).

The technique works as follows: The PC of a branch instruction is presented to the Branch History Register (the first level of the branch predictor) as usual. However, the pattern that comes out of the BHR is not sent directly to the predictor - instead, hardware checks to see if the pattern is in our predicting set, which consists of those patterns which are anticipated to have good prediction accuracy. If it is, the branch is considered highly predictable, and the pattern is forwarded on to the second level of the

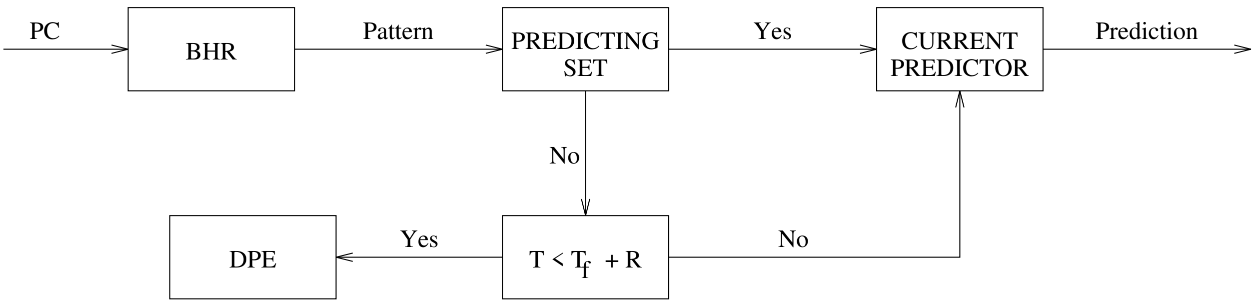

Figure 1: Limited Dual-Path Execution

---

[1] Another significant problem with dual path execution is that the front-end instruction fetch stage of the pipeline must be duplicated; given the heavy demands on the ICache, duplication of this resources seems prohibitive. Fortunately, recent improvements in the design of processor fetch mechanisms [4] imply that a more restrictive form of dual path execution is feasible. We will not, however, explore front end issues in this paper.

predictor. However, if the pattern is not in the predicting set, then it is a branch that is considered difficult to predict accurately, and an attempt will be made to begin fetching down both paths at this point. If the hardware is not currently fetching down both paths, then it will begin doing so. If dual-path fetching *is* in progress, then the pattern is sent on to the second level predictor and a regular prediction will occur.[2]

The benefits of this scheme come from reducing (or limiting) the occurrence of DPE. Those patterns that are predicted well continue to be predicted rather than using DPE resources, and the patterns that are believed to mispredict often are allowed to execute down both paths eliminating the prediction (and frequently a misprediction).

Clearly this approach relies heavily on the ability to distinguish between branches that are highly accurately predicted (have a *high confidence*) and branches that are poorly predicted (have a *low confidence*). In the following section we will explore this question in more detail.

3. Assigning Confidence to Branch Prediction

Hybrid branch predictors [5] use selectors to determine which component is more confident in its prediction. In the technique described in the previous section, confidences are assigned to the branch patterns and a selecting set was be used to determine whether to predict or perform dual path execution. This technique is substantially different than that used by Jacobsen et al [6]. In their work, extra hardware was added to keep track of the performance of the branch predictor. In LDPE we are proposing extracting *more* information from the *existing* hardware.

3.1. Experimental System

The initial experiments for this research were conducted using UltraSPARC model 170E systems. The data was gathered using a tracing tool developed at SUN Microsystems Laboratories called Shade, which provides exhaustive dynamic information at the instruction level by dynamically translating the code of the application into host machine code while incorporating analysis code to generate the traces. This new code is then directly executed to emulate and trace the application.

3.2. Implemented Branch Predictors

Four different types of branch predictors were implemented in order to evaluate how the accuracy of different programs were affected by differences in a variety of current prediction schemes. The first and simplest predictor implements a per-address scheme in which every static branch is given its own two-bit saturating up/down counter. The second scheme simulates gshare(12), with twelve bits used to keep track of the first level of history. This requires a pattern history table of size 4096 indexed by the lower 12 bits of the branch address XORed with the pattern. Gshare has been found to achieve high accuracies without the high cost needed for many other predictors. [5].

The last two predictors studied are variations of the two-level adaptive scheme. Both PAg and PAs are implemented. PAg uses a per-address branch history register table, and the depth of the history bits was varied between 8, 12, and 16 bits which then indexed into a global PHT. PAs(6,16) keeps 6 bits of history for each branch address. Each BHR hashes to one of 16 PHTs. There were assumed to be no contention among the BHRs with any of the predictors.

---

[2] Note that this is a conceptual diagram - if the branch prediction logic is limiting the machine cycle time, then the Current Predictor and the Predicting Set lookup could occur in parallel and the signal generated could simply be an "accept/reject" of the output of the Current Predictor).

3.3. Description of Benchmarks

The benchmarks analyzed were those from the SPEC95 suite. Six of the integer benchmarks and three of the floating point benchmarks were used. Although *compress* is included in the Spec95 suite, Berkeley compress version 5.9 was used due to problems with running the SPEC95 version. Each program was run to completion.

Table 1 shows many basic statistics of each of the programs used. The table lists the input files used, the total number of instructions executed, the number of conditional branch instructions, the total number of branches executed, and the percent of branches that are TAKEN. Only the branches that are executed at least once will be examined, since these are the branches that affect the accuracy of a branch predictor.

Table 2 shows the misprediction rate for each program when using the predictors: 2-bit, PAg(8), PAg(12), PAg(16), PAs(6,16), and gshare(12). The mispredictions given for *gcc* are the average for the five separate inputs. It can be seen that as the number of history bits increases in the PAg scheme, the accuracy increases. Increasing the depth of history that is maintained allows more patterns to be differentiated, enabling a predictor to find the best prediction per pattern.

3.4. Branch Characteristics

The benchmark programs exhibit many interesting characteristics that often depend upon the nature of the program. The integer benchmarks typically use more branch instructions than the floating point benchmarks, with *gcc* having a very large working set of active branch instructions. On the other hand, the floating point programs tend to execute a small number of instructions a large number of times. For example, in *perl*, only 80 branch instructions (less than 3% of the total branch instructions) are responsible for over 80% of all the executed branches. This phenomena indicates that it might be beneficial if a branch predictor could target the few branches that are responsible for the majority of the executions.

Table 1. Detailed Benchmark Information

Program	Input File	Total # of Instruction Executions (in millions)	Branch Instructions (executed at least once)	Total Branch Executions (in millions)	% Taken
go	9stone21.in	34423	7263	4296	59.24
compress	in	81	1364	12	71.12
li	*	57924	1880	9830	50.07
ijpeg	penguin.ppm	42619	2407	3534	67.73
perl	primes.pl	15450	2902	2284	54.41
gcc	1insn-recog.i	599	14782	113	57.50
gcc	1cccp.i	1344	21438	253	59.89
gcc	emit-rtl.i	173	18771	31	55.01
gcc	regclass.i	127	18660	23	56.70
gcc	explow.i	241	14969	45	56.80
swim	swim.in	273176	876	725	99.20
apsi		394597	1806	17112	64.63
fpppp	natoms.in	396365	980	5297	66.12
gccaverage		497	17724	93	57.18
intaverage		25183	5100	3345	60.01
fpaverage		354713	1221	7994	76.65
totalaverage		135026	3807	4799	65.56

Table 2. Misprediction Rates

Program	2-bit % misp	PAg(8) % misp	PAg(12) % misp	PAg(16) % misp	PAs(6,16) % misp	gshare(12) % misp
go	20.444	20.037	19.150	15.284	20.938	27.251
compress	11.815	10.955	10.550	10.154	10.601	10.006
li	10.738	5.301	3.880	2.560	5.207	4.896
ijpeg	9.051	6.239	5.905	5.571	6.294	5.855
perl	4.255	0.037	0.025	0.020	0.042	2.131
gcc	9.925	8.184	6.399	5.153	8.780	11.638
swim	0.266	0.197	0.197	0.197	0.197	0.213
fpppp	15.869	7.093	4.319	3.410	7.222	5.600
int average	11.04	8.46	7.65	6.46	8.64	10.30
fp average	8.06	3.64	2.26	1.80	3.71	2.91
total average	10.30	7.25	6.30	5.29	7.41	8.45

An examination of the distribution of mispredictions among branches shows that even fewer branches are responsible for the majority of mispredictions. For *perl*, less than 2% of all branch instructions, 55 branches, create over 90% of the total mispredictions. This again emphasizes the importance of identifying the branches that are generating the largest disturbances in the prediction accuracy. Thus, it is extremely valuable to the accuracy of a predictor if the branches which are referenced often or have high misprediction rates are discovered and targeted for special processing (e.g. dual path execution).

Increasing the depth of history allows more patterns to be differentiated; however, many of these new patterns may not be referenced often. The characteristics of branches examined previously can also be seen when looking at patterns. In *perl*, using the predictor PAg(12), 4 distinct patterns correspond to over 90% of the total references. It has been shown [7] that branches are highly biased toward a particular pattern or direction and since branches may map to more than one pattern throughout the execution of a program, targeting patterns can encompass more branch references and branch mispredictions.

3.5. The Relationship Between Accuracy and Coverage

Both accuracy and coverage (the percent of total references encompassed by the patterns) play a major role in deciding which patterns to include in the predicting set. Given that there is not an infinite set of resources to always allow fanouts without delay, it would be best to only include patterns in the predicting set that achieve an average accuracy above a certain threshold. However, the higher the accuracy threshold is set, the more patterns fall below that accuracy and compete for fanout resources.

If two branches compete for limited dual-path execution resources, then the first branch to be started will be allocated the resources regardless of the prediction accuracies of their respective patterns. Thus, a less accurate pattern may be forced to be predicted because the resources are already filled with a more accurately predicted pattern. This can lead to a reduction in overall accuracy when the predicting set does not contain enough patterns, implying that the coverage is too small. A compromise needs to be made between choosing patterns with high accuracies and those that create a large coverage. Including patterns with a high percentage of the total references would reduce how often fanouts would be attempted. Choosing a predicting set that targets either accuracy or coverage, while being optimal in the local point of view, may not improve overall performance.

Figure 2 illustrates the relationship between accuracy and coverage for each benchmark. As the number of patterns in the predicting set increases, the accuracy drops and the coverage increases. These curves were obtained by sorting the patterns according to accuracy and then the accuracy and coverage were determined as each pattern was added into the predicting set. These curves assume that those patterns that are not in the predicting set will have the resources to fanout. Since the coverage curve increases quickly, the predicting set can remain small, with the hope of not compromising the accuracy greatly. If the correct patterns could be identified, *go* could achieve an overall accuracy of over 95% with 60% coverage. This is a dramatic increase from its normal overall accuracy of just over 80% (translating to over a 75% drop in misprediction rate). Thus, the importance of choosing the optimal patterns can be seen.

3.6. Analysis of Patterns and the Predicting Set

There are a few different approaches that can be used to obtain the predicting set. Patterns can be analyzed in all the benchmarks and a set can be chosen that should perform well for all programs. Another approach would be to chose a separate set for each benchmark. While this method should attain higher accuracies for each program, a generic predicting set is more versatile. The following describes the techniques used to obtain one predicting set for all benchmarks.

When choosing the most beneficial patterns to include in the predicting set for both the accuracy and coverage concerns, it is helpful to look for trends among the patterns across the benchmarks. For each benchmark, two lists of patterns were crucial to determining the predicting set; patterns sorted in order of references and patterns sorted in order of accuracy. The list of the top referenced patterns revealed a noticeable trend. Table 3 lists the top 20 referenced patterns for each benchmark. In every benchmark the top two referenced patterns were those that contained either all ones or all zeros, meaning that the history is exclusively TAKEN or exclusively NOT-TAKEN. These patterns will be referred to as the **always-**

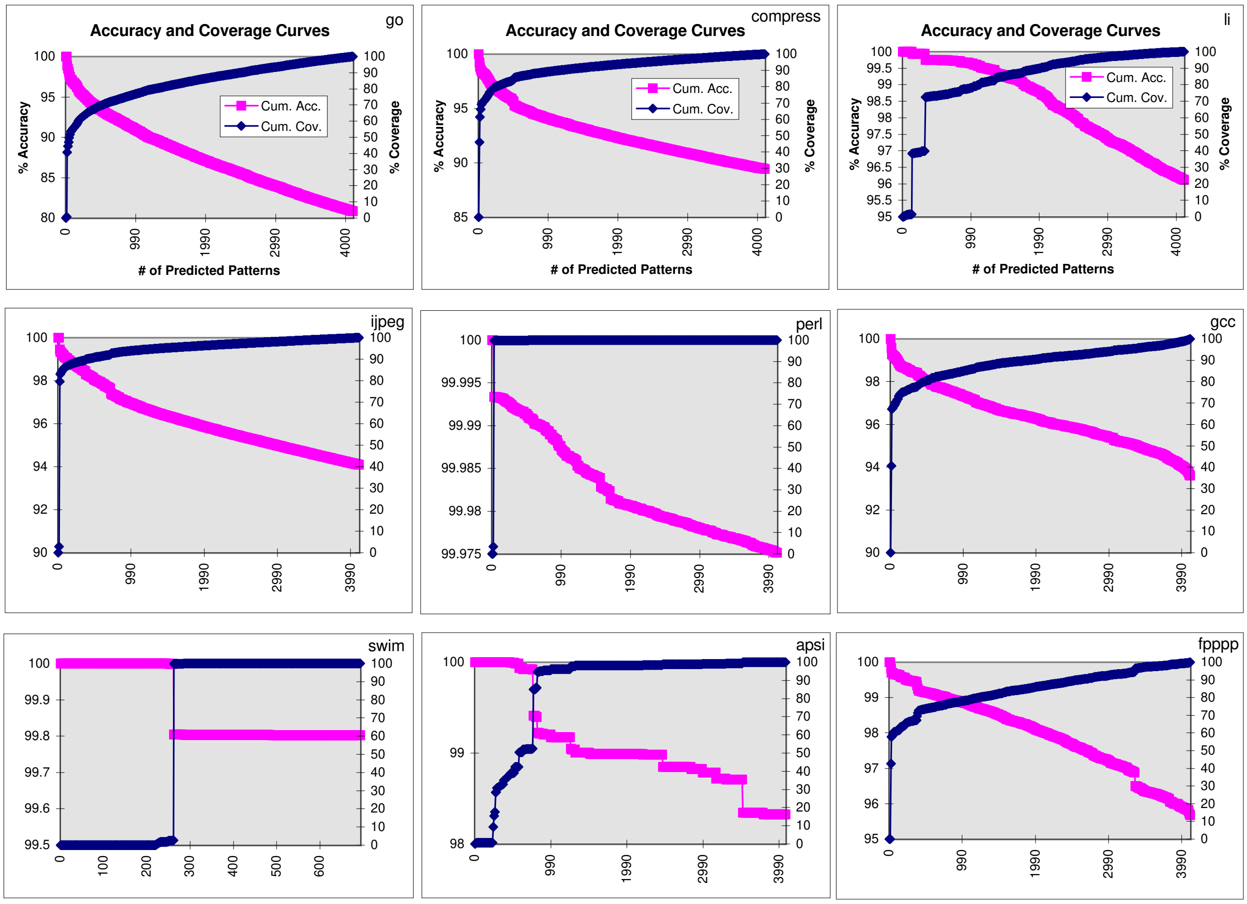

Figure 2: Relationship Between Coverage and Accuracy

taken pattern and the **always-not-taken** pattern, respectively.

This trend implies that these patterns are beneficial to the coverage component for choosing the predicting set. Fortunately, looking at the list of patterns that have the best accuracies, the always-taken and always-not-taken patterns are found in the top 1% of all patterns for most of the benchmarks and in the top 5% for all benchmarks. Thus, the always-taken and always-not-taken patterns are obvious candidates for the predicting set; they have good prediction accuracy as well as good coverage. When analyzing these two lists, the necessity of using both accuracy and coverage together is clear; many of the most accurate patterns obtain such a high accuracy due to very low reference counts.

Another interesting characteristic found in the sorted patterns is that patterns with only a single NOT-TAKEN (zero state) branch often have a high reference count and achieve fairly high accuracies. This set of patterns is referred to as the **almost-always-taken** patterns. This phenomena led to the classification of patterns into sets based on the number of TAKEN paths in the history. These sets were analyzed through graphs that showed how they compared in terms of references and accuracies.

Figure 3 shows these graphs for three benchmarks, *go*, *ijpeg*, and *fpppp*. Each point gives the accuracy or reference for the set of patterns with X number of TAKEN branches. For example, the pattern *0001000100000000* would be included in the set of patterns with 2 TAKEN branches. The overall accuracy is also shown on the graphs for comparison. The graphs shown use data obtained by the predictor PAg(16). These graphs reaffirm the wisdom of including the always-taken pattern and the always-not-taken pattern in the predicting set since these patterns consistently achieve higher accuracies than the overall accuracy. Both *go* and *ijpeg* show that the almost-always-taken patterns also give a higher accuracy than the overall accuracy. Thus the patterns with only one NOT-TAKEN path in their history are included in the predicting set.

Some of the benchmarks show that the almost-always-not-taken pattern set (those containing only one TAKEN branch in a stream of NOT-TAKEN branches) also achieves high accuracy. While this is not true for all the benchmarks, these patterns were chosen to be included in the predicting set due to the

Table 3. Patterns in Programs

go	ijpeg	perl	gcc	swim	apsi	fpppp
111111111111	111111111111	111111111111	111111111111	111111111111	111111111111	111111111111
000000000000	000000000000	000000000000	000000000000	000000000000	000000000000	000000000000
110111011101	101010101010	101101101101	010101010101	011111111111	101010101010	101110111011
101110111011	010101010101	110110110110	101010101010	111111111110	010101010101	110111011101
111011101110	111101111111	011011011011	011111111111	111111111101	010101010010	111011101110
011101110111	111110111111	101001100010	111111111110	101111111111	010010101010	011101110111
011111111111	111111011111	010100110001	111110111111	111111111011	111110111111	101101101101
111111111110	111111101111	001010011000	111111011111	110111111111	111111011111	110110110110
111111011111	011111111111	100110001010	111101111111	111011111111	101010010010	011011011011
111110111111	111111111110	000101001100	111111101111	111111110111	010101001001	010101010101
111111111101	011111110111	110001010011	111011111111	111111011111	010100100101	101010101010
101111111111	111011111110	100010100110	111111110111	111111101111	101001001010	110110111011
111111101111	101111111011	010011000101	101111111111	111101111111	100101010100	011011101110
111101111111	110111111101	011000101001	110111111111	111110111111	001010101001	110111011011
110111111111	101111111111	001100010100	111111111011	101010101010	101010100100	101101110111
111111111011	110111111111	101010101010	111111111101	010101010101	010010010101	011101110110
111111110111	111111111101	010101010101	000000000001	101110111011	100100101010	111011101101
111011111111	111011111111	111110111111	100000000000	011101110111	001001010101	111011011101
010101010101	111111110111	111111011111	000000100000	110111011101	111101111110	011101101110
101010101010	111111111011	111101111111	000001000000	111011101110	011111101111	101110110111

simplicity of the circuit that could identify these patterns. The references to these patterns will also help limit the number of fanouts attempted. A few benchmarks, such as *fpppp*, which do not have good accuracy for the almost-always-not-taken patterns, do not reference this pattern set often enough for it to have a negative effect on the accuracy of the new hybrid predictor.

Peaks can be seen in many of the programs for the pattern sets that were TAKEN during half the history and NOT-TAKEN for half. After further examination, it was found that the patterns that alternate between TAKEN and NOT-TAKEN paths were responsible for the majority of references in this pattern set. These patterns will be referred to as the alternating patterns. Since these patterns can be predicted well (The overall accuracy of the half TAKEN/half NOT-TAKEN is brought down by those patterns which are not alternating), they too are included in the predicting set. Thus, the predicting set used in the following sections of this paper will contain the always-taken pattern, the always-not-taken pattern, the almost-always-taken and almost-always-not-taken patterns, and the alternating patterns.

The accuracy and coverage that occur due to the chosen predicting set is shown in Table 4. The table shows the percentages for each parameter varying the history depth between 8, 12, 16 bits. The actual PAg(x) accuracy, without using any fanouts, is shown for comparison. As more history is maintained, the coverage decreases, attempting to fanout more often. The accuracy is seen to increase with the history. These graphs assume that there is always a DPE resource available when needed, making the only patterns to be predicted those in the predicting set. Given ample resources, it is clear that there would be a very beneficial increase in accuracy due to this new hybrid predictor.

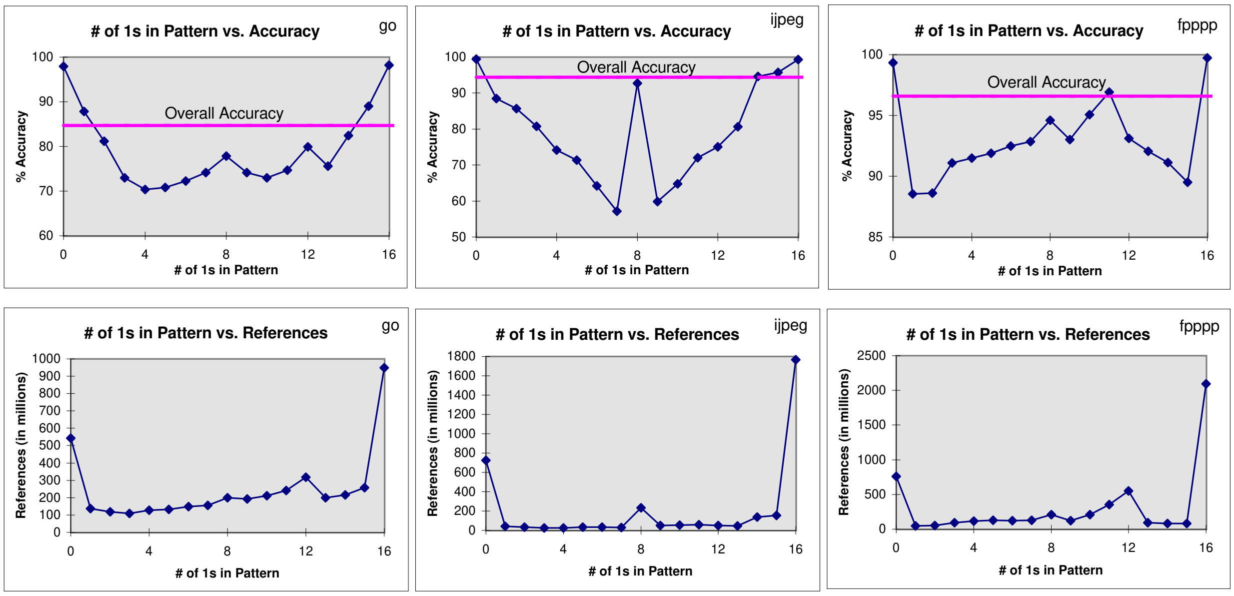

Figure 3: Pattern Sets Based on the Number of TAKEN Paths

Table 4. Accuracy and Coverage Assuming Unlimited DPE Resources

Program	DPE/PAg(8) % Cov	DPE/PAg(8) % Acc	PAg(8) % Acc	DPE/PAg(12) % Cov	DPE/PAg(12) % Acc	PAg(12) % Acc	DPE/PAg(16) % Cov	DPE/PAg(16) % Acc	PAg(16) % Acc
go	59.091	94.734	79.963	49.688	97.290	80.850	44.559	98.247	84.716
compress	78.330	96.252	89.045	71.011	97.620	89.450	65.781	98.602	89.846
li	75.185	98.690	94.699	71.716	99.449	96.120	70.535	99.623	97.440
ijpeg	88.277	99.099	93.761	84.394	98.804	94.095	81.563	99.200	94.429
perl	90.637	99.983	99.963	90.606	99.988	99.975	90.581	99.991	99.980
gcc	81.881	96.987	91.816	76.420	98.285	93.601	73.156	98.823	94.847
swim	99.947	99.803	99.803	99.938	99.803	99.803	99.929	99.803	99.803
fpppp	63.131	98.191	92.907	59.145	99.233	95.681	57.497	99.534	96.590

3.7. Fanout Constraints

The frequency of fanouts, or dual path execution, is dependent on the amount of resources available and the time it takes for a branch to be resolved. In order to accurately simulate the correct cycle count, one would need a full pipeline model and the results would be implementation dependent. We will later show the results of our approach on a full pipeline simulator, but for now, branch resolution will be based on instruction count; this enables a comparative analysis of alternate approaches to selecting the predicting set, while avoiding the complexities required for a complete cycle level simulation. After, this initial analysis, we will verify the results with full simulations of selected configurations.

When determining whether to allow a pattern that is not in the predicting set to fanout, the branch resolution time is compared to the number of branches that have occurred since the last fanout. If fanouts occurs too close together, then a prediction is forced. A resolution equal to **2** corresponds to the requirement that at least two branches must be predicted between fanouts. In this paper the effects of the branch resolution time is analyzed as it varies from one to six.

4. Evaluating Limited DPE Performance

As mentioned previously, the predicting set that was used in this work contained the always-taken, always-not-taken, almost-always-taken, almost-always-not-taken, and alternating patterns. This set will be referred to as the *fixed* predicting set since it does not change with the benchmark being used. The following sections give simulation results for the DPE/PAg predictor. The first section uses the fixed predicting set, while the second section uses an alternate method for partitioning patterns, between the DPE and predicting set, based on profiling information.

4.1. The Fixed Predicting Set

The misprediction rates for PAg and DPE/PAg are shown in Figure 4 (The results for *apsi* are not reported due to errors in the DPE runs.). The depth of history maintained is varied between 8, 12, and 16. The branch resolution is varied from 1 up to 6. These variations are referred to as H(R), where H is the number of history bits and R is the branch resolution. Thus, 12(5) would imply that the DPE/PAg predictor uses 12 bits to maintain the branch history and 5 branches are required to occur between fanouts.

This figure shows that ss the branch resolution decreases, the misprediction also decreases. A shorter branch resolution allows more fanouts to occur and thus more patterns with poor prediction accuracies do not have to predict. Looking at *li* with a history depth of 12 and a branch resolution of 1, there is a reduction of over 78% in the misprediction rate. Increasing the resolution to 6 while keeping the same number of history bits gives a reduction of the misprediction rate by 33%. Thus, even using a resolution of 6, approximately 30 instructions between fanouts, a significant increase in performance is seen.

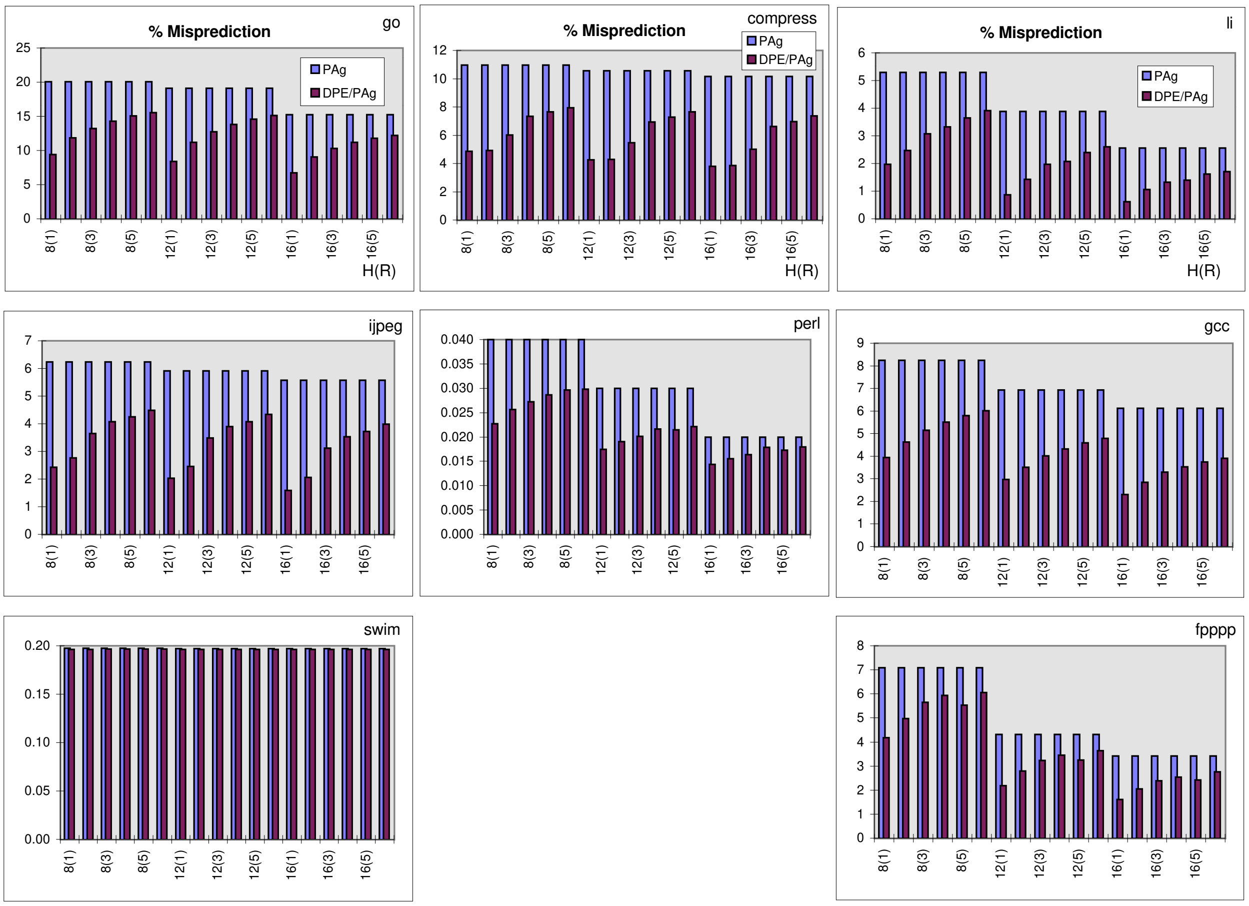

Figure 4: Misprediction Comparison of PAg and DPE/PAg

This allows the use of DPE even with minimal resources to be beneficial.

When analyzing the results of the hybrid DPE predictor, the makeup of the predicting set should not be ignored. The performance of the hybrid predictor is dependent on the accuracy and coverage of the predicting set. If the accuracy is kept too high, the coverage will be reduced and there will often be contention for the DPE resources between patterns; forcing a pattern with a low prediction to predict because a pattern with a higher accuracy has fanned out previously taking the resource. Thus the coverage of the predicting set is also very important.

Figure 5 shows the percent of patterns that were actually able to fanout. This percentage is in terms of patterns that were contending for the DPE resources, those not in the predicting set. Looking at a history depth of 12 and a branch resolution of 1, the fixed predicting set was able to perform between 60% and 94% of all fanouts, with an average of 77%. Increasing the branch resolution to 6, between 23% and 83% of all fanouts were given DPE resources, with an average of 41%. Thus, how much cost is to be devoted to dual path execution and the branch resolution have a large affect on the coverage of the predicting set.

Table 5 shows the number of fanouts executed and the percentage of total executions that avoided prediction for history depths of 8 and 12. An average of 16% of the total branches had patterns that were allowed to fanout when the history depth is 12 and the resolution is 1. Thus, the fixed predicting set does a good job of limiting the occurrence of dual path execution.

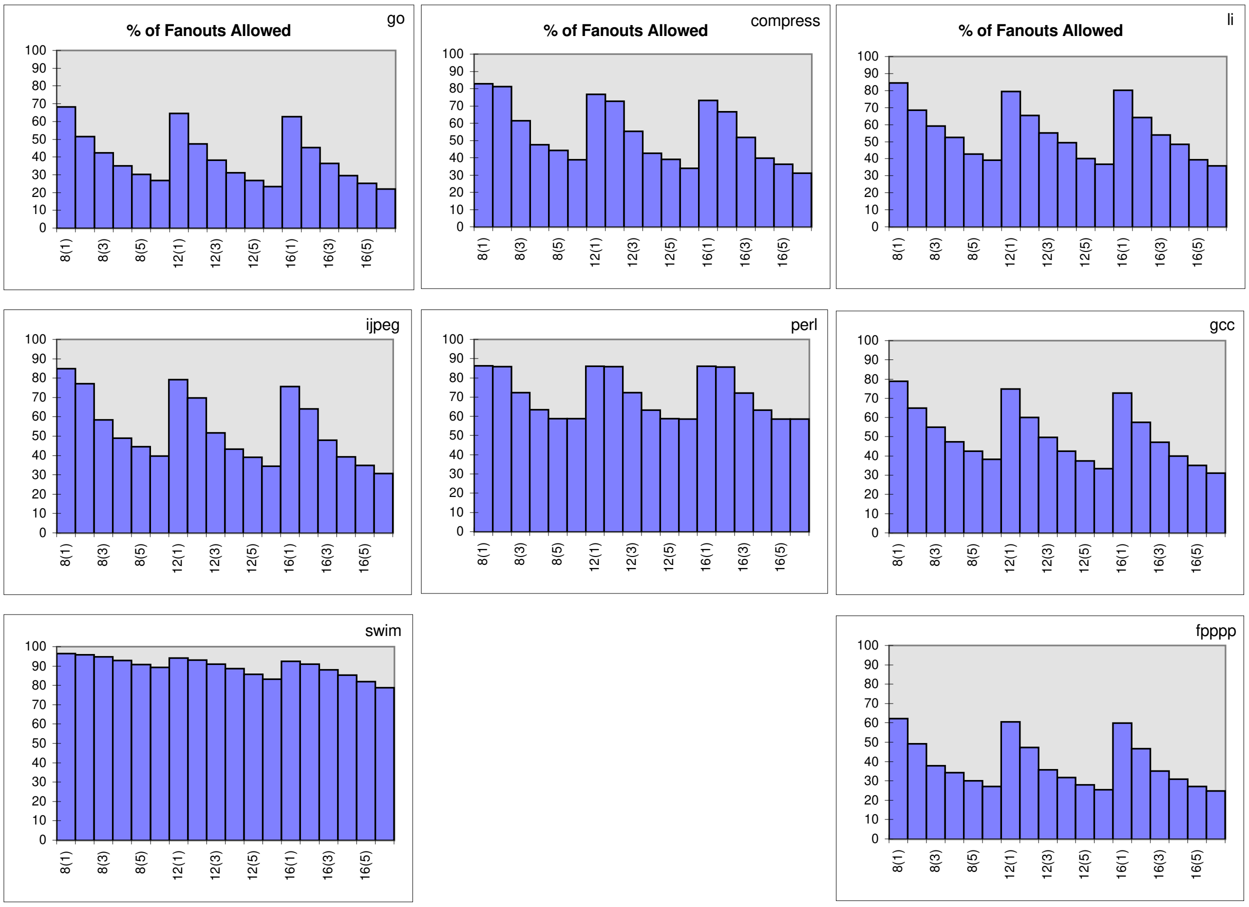

Figure 5: Analysis of Fanouts Allowed

4.2. Profile-based Predicting Sets

A second approach to creating the predicting set was used for comparison. A predicting set was created especially for each benchmark. Profiling was used in order to decide which patterns should be included in the predicting set. The patterns were sorted in order of decreasing accuracy and kept in a file along with their corresponding references. When the program was run, the desired coverage was specified for that run. At the start of a run, the profiled pattern file is read in. Starting from the most accurate pattern, patterns were included into the predicting set until the desired coverage was met. A bit corresponding to the pattern was set if the pattern was added to the predicting set. Then execution of the program continued, and when a branch occured, the bit associated with the branch pattern was checked. If the pattern was in the predicting set, then the pattern was predicted; otherwise dual path execution was attempted as before.

The advantage to using this scheme for creating the predicting set is that each benchmark only includes those patterns that it has previously found to achieve the highest accuracy. Patterns that may perform well in many other benchmarks, but not on this one, will not be included. Another advantage is that the coverage can be adjusted easily to restrict fanning out to only those patterns that really require it due to their poor prediction accuracy. The coverage from the fixed predicting set given before varies among benchmarks. A drawback to this scheme is that the program must be run prior to application execution in order to determine the predicting set.

Table 5. Number of Fanouts and the Percentage of the Total Executions

DPE/PAg(8)												
Program	Resolution = 1		Resolution = 2		Resolution = 3		Resolution = 4		Resolution = 5		Resolution = 6	
	# Fans	% Tot	# Fans	% Tot	# Fans	% Tot	# Fans	% Tot	# Fans	% Tot	# Fans	% Tot
go	1199	27.9	906	21.1	745	17.3	616	14.3	531	12.3	469	10.9
compress	2.30	17.9	2.26	17.6	1.71	13.3	1.32	10.3	1.24	9.6	1.08	8.4
li	2060	20.9	1671	17.0	1443	14.7	1281	13.0	1043	10.6	953	9.7
ijpeg	351	9.9	319	9.0	241	6.8	202	5.7	184	5.2	164	4.6
perl	184	8.1	184	8.0	155	6.8	135	5.9	126	5.5	126	5.5
gcc	16.2	14.2	13.3	11.7	11.2	9.9	9.7	8.6	8.7	7.7	7.9	7.0
swim	0.37	0.05	0.37	0.05	0.37	0.05	0.36	0.05	0.35	0.05	0.34	0.05
fpppp	1214	22.9	961	18.1	738	13.9	670	12.6	588	11.1	529	10.0
DPE/PAg(12)												
Program	Resolution = 1		Resolution = 2		Resolution = 3		Resolution = 4		Resolution = 5		Resolution = 6	
	# Fans	% Tot	# Fans	% Tot	# Fans	% Tot	# Fans	% Tot	# Fans	% Tot	# Fans	% Tot
go	1393	32.4	1021	23.8	826	19.2	674	15.7	576	13.4	506	11.8
compress	2.86	22.2	2.70	21.0	2.06	16.0	1.59	12.3	1.46	11.4	1.26	9.8
li	2212	22.5	1820	18.5	1532	15.6	1377	14.0	1114	11.3	1018	10.36
ijpeg	437	12.4	385	10.9	285	8.1	238	6.7	215	6.1	189	5.3
perl	184	8.1	183	8.0	154	6.7	135	5.9	126	5.5	126	5.5
gcc	20.0	17.6	16.1	14.2	13.3	11.7	11.3	9.9	10.0	8.8	8.9	7.8
swim	0.42	0.05	0.42	0.05	0.41	0.05	0.40	0.05	0.39	0.05	0.37	0.05
fpppp	1308	24.7	1023	19.3	774	14.6	687	12.9	602	11.4	549	10.4

Six different benchmarks were run with profile-based predicting sets. We found that when fixing the branch resolution, the misprediction rate decreases and then increases as the coverage is varied. If the coverage is too low, then patterns with low prediction accuracies may be forced to predict since there is a lot of contention for the DPE resources. If the coverage is too high, then the DPE resources are not being used enough and patterns with low accuracies are being included in the predicting set and thus are being forced to predict.

Each benchmark has its own ideal coverage for each length of branch resolution. For example, *ijpeg* performs best when the branch resolution is 2, when the coverage is 85, while *go* performs best with a coverage of 50. *gcc* performs best at a coverage of 70 when the branch resolution is 1, 2 and 3, yet benefits from a coverage of 85 when the resolution increases to 4. Thus, the architectural limits of DPE resources should be considered when choosing a coverage.

When comparing the misprediction rate for the ideal coverage and that of the coverage obtained by using the fixed predicting set, we found that the rates do not vary by much. Often the two coverages are even fairly close together. For a resolution of 2, *compress* has an ideal coverage of 70 and achieves a misprediction rate of 4.07. The fixed predicting set has a coverage of 71 and achieves a misprediction of 4.29. These correspond to decreases in misprediction rate of the DPE/PAg scheme of 78.7% and 77.6% respectively. The closeness in the coverages and the misprediction rates imply that the fixed predicting set does a good job of identifying the patterns that are good predictors and that those patterns really do not vary much with the benchmark. Thus, profiling may not be necessary.

4.3. Evaluating the Performance of LDPE on Cycle Time

Branch prediction rates give a good first approximation of the expected performance impact of the limited dual path execution scheme, but to fully evaluate the capabilities of this scheme it must be integrated into a processor pipeline. In order to evaluation the performance of the hybrid DPE mechanism on the overall execution time of an application, we built a cycle level simulator based on the SimpleScalar tool set. The tool set employs the SimpleScalar instruction set, which is a (virtual) MIPS-like architecture. All programs were compiled with GNU GCC (version 2.6.2), GNU GAS (version 2.5), and GNU GLD (version 2.5) with maximum optimization (-O3) and loop unrolling enabled (-funroll-loops). The Fortran codes were first converted to C using ATT F2C version 1994.09.27.

We simulated the execution of an agressive out-of-order pipeline very similar to that of the Pentium Pro (without the need to translate CISC instructions). The pipeline is capabile of issuing up to 4 instructions per cycle and perform a single branch prediction every cycle. Conditional branch instructions take a minimum of 12 cycles to complete (other branch latencies were evaluated, but space limitations require us to show one pipeline configuration).

Table 6 shows the number of cycles necessary to execute each of the listed benchmark programs. The second column shows the number of cycles necessary assuming perfect branch prediction, the third column the number of cycles required using the LDPE technique, the fourth column the number of cycles needed if using an unmodified 2-level branch predictor (PAg), the fifth column the percent slowdown over the minimum for the LDPE case, and the sixth the percent slowdown for the PAg case. This table shows that limited dual path execution is capable of greatly reducing the branch mispredict penalty, leading to a substantial improvement in IPC.

5. Conclusions

This work presents a hybrid branch predictor scheme that uses dual path execution and a predicting set based on patterns to effectively reduce the branch penalty in high performance processors. While dual path execution is in theory beneficial in eliminating branch penalty, it is generally unrealizable due to the excessive hardware cost required. Combining DPE with another current prediction scheme can limit the times that a fanout would occur reducing the cost.

When limiting DPE, it is beneficial to evaluate when the prediction is believed to be a sound one and to allow the DPE resources to be used when the prediction is likely to be wrong. In order to accomplish this, a predicting set is created to select when to predict and when to execute down both paths, creating a partial coverage predictor. This work targets patterns since branches are biased to a particular path or pattern. The accuracy and reference count of each pattern is used to assign a confidence to that pattern, which determines whether that pattern should be predicted or should attempt to avoid a prediction by executing down both paths.

Using the pattern confidences, a partial coverage predictor can be implemented. In the past, branches have either always been predicted or are never predicted. The idea of a partial coverage predictor takes a predictor that is known to typically predict well and limits the time it predicts to those high accuracy cases. The predicting sets described in this paper do a good job of allocating the DPE resources and allow the strengths of both the predictor and dual path execution to remain, while also reducing the

Table 6. Cycle Time Comparison of LDPE to Perfect Prediction and PAg

Program	Cycles Required Using Perfect Prediction	Cycles Required Using LDPE	Cycles Required Using PAg	% Increase in Execution Time (for LDPE)	% Increase in Execution Time (for PAg)
compress	1924374	1989416	2357989	3.38	22.53
gcc	256463944	303700801	415954749	18.42	62.19
go	526541349	609912176	884465133	15.83	67.98
li	498301747	651088394	884971198	30.66	77.60
perl	8932741	10598949	12931283	18.65	44.76

negative costs of DPE.

Comparing the DPE hybrid predictor model to the single scheme approach PAg, dramatic decreases in misprediction rate were seen. The PAg(16) gave an average misprediction of 5.29%. The DPE/PAg(16)/1 gives an average misprediction of 2.11% and the DPE/2-bit(16)/1 achieves a misprediction rate of 3.08%. These correspond to decreases of 60% and 36% respectively, which translates to an improvement of 10% to 20% in execution time. These results imply that dual path execution which often is thought to be a resource consuming method may be a worthy approach if restricted with a predicting set.

6. Future Work

The proposed hybrid branch predictor scheme can greatly help increase the performance of processors. While it is clear that this scheme provides an increase in prediction accuracy over current schemes, assumptions were made throughout the experiments that should be addressed in order to further validate the results of this predictor approach. Future work in this area will eliminate some of these assumptions.

- Other hybrid dual path execution variations should be studied. The current predictors used in this study were PAg and two-bit, however, other current predictors might prove useful if combined with DPE.
- The cost of implementing of dual path execution was not looked into for this study. The cost of varying the branch resolution time should also be studied. Knowing these costs would allow a better comparison between predictors to find the best predictor for a given cost.
- In this study, execution was limited to two instruction streams at a time. The effects of allowing more instruction streams to execute simultaneously would be an interesting analysis. This essentially allows for instructions that are an unresolved branch path to be executed, creating more instructions to fill the pipeline, however also creating the possibility for many more results to be found to be useless later.

7. Bibliography

[1] W.D. Conners, J. Florkowski and S.K. Patton, "The IBM 3033: An Inside Look," In Datamation, pages 198-218, May 1979.

[2] Augustus K. Uht and Vijay Sindagi, "Disjoint Eager Execution: An optimal form of speculative execution," in *Proceedings of the 28th Annual International Symposium on Microarchitecture (MICRO-28),* pages 313-325, Nov. 1995

[3] Gary Tyson, "The Effects of Predicated Execution on Branch Prediction", in *27th Annual International Symposium on Microarchitecture*, pages 196-205, Nov. 30-Dec. 2, San Jose, CA, 1994

[4] E. Rotenberg, S. Bennett, and J.E. Smith, "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching," in *29th Annual International Symposium on Microarchitecture,* Paris, France, Dec. 1996

[5] Scott McFarling, "Combining Branch Predictors", *Technical Report TN-36*, DEC-WRL, June 1993.

[6] Erik Jacobsen, Eric Rotenberg, and James E. Smith, "Assigning Confidence to Conditional Brnach Predictions", in *29th Annual International Symposium on Microarchitecture*, pages 142-152, December 1996.

[7] Kelsey Lick and Gary Tyson, "Hybrid Branch Prediction Using Limited Dual Path Execution", Department of Computer Science Technical Report #UCR-CS-96-7, University of California, Riverside, November 1996.