The Cascaded Predictor: Economical and Adaptive Branch Target Prediction

Karel Driesen and Urs Hölzle
Department of Computer Science
University of California
Santa Barbara, CA 93106
{karel,urs}@cs.ucsb.edu
http://www.cs.ucsb.edu/oocsb

Two-level predictors improve branch prediction accuracy by allowing predictor tables to hold multiple predictions per branch. Unfortunately, the accuracy of such predictors is impaired by two detrimental effects. Capacity misses increase since each branch may occupy many entries, depending on the number of different path histories leading up to the branch. The working set of a given program therefore increases with history length. Similarly, cold start misses increase with history length since the predictor must first store a prediction separately for each history pattern before it can predict branches with that history.

We describe a new hybrid predictor architecture, cascaded branch prediction, which can alleviate both of these effects while retaining the superior accuracy of two-level predictors. Cascaded predictors dynamically classify and predict easily predicted branches using an inexpensive predictor, preventing insertion of these branches into a more powerful second stage predictor. We show that for path-based indirect branch predictors, cascaded prediction obtains prediction rates equivalent to that of two-level predictors at approximately one fourth the cost. For example, a cascaded predictor with 64+1024 entries achieves the same prediction accuracy as a 4096-entry two-level predictor. Although we have evaluated cascaded prediction only on indirect branches, we believe that it could also improve conditional branch prediction and value prediction.

1. Introduction

Indirect branches, which transfer control to an address (recently) loaded into a register, are hard to predict accurately. Unlike conditional branches, they can have more than two targets, so that prediction requires a full 32-bit or 64-bit address rather than just a "taken" or "not taken" bit. Furthermore, their behavior is often directly determined by data loaded from memory, such as virtual function pointers in object-oriented programs written in languages such as C++ and Java.

Indirect branches occur frequently in some programs of widely used benchmark sets like the SPECint95 suite, although they remain less common than conditional branches. However, indirect branches are much more frequent in object-oriented languages. These languages promote a programming style in which late binding of subroutine invocations is the main instrument for clean, modular code design. Virtual function tables, the implementation of choice for most C++ and Java compilers, execute an indirect branch for every lately bound call. The C++ programs studied here execute an indirect branch as frequently as once every 50 instructions; other studies [CGZ94] have shown similar results. Java programs (where all non-static calls are virtual) are likely to use indirect calls even more frequently.

Even today, indirect branch misses can cause significant overheads. The overhead of virtual function calls in C++ programs on superscalar processors with a large BTB is as high as 29% [DH96]. Similarly, Chang, Hao, and Patt show that for the SPECint95 programs *perl* and *gcc* the indirect branch overhead is approximately 15% and 8% with a BTB [CHP97].

Two-level path-based prediction can reduce the indirect branch overhead considerably, compared to standard BTBs [DH98]. Unfortunately, the improved accuracy comes at the cost of much larger prediction tables. For example, to achieve a prediction accuracy of 90%, a path-based two-level predictor requires a table with 1024 entries.

We describe a new predictor architecture, cascaded branch prediction, which dynamically classifies branches into "easy" and "hard" branches and uses a simple BTB to handle the easy cases, preventing insertion of these branches into the more powerful second stage predictor. Cascaded predictors improve upon two-level and hybrid predictors by using a different update rule, significantly reducing the table size needed to achieve a given accuracy. For example, a cascaded predictor with a total of 288 prediction table entries achieves virtually the same prediction accuracy as the standard 1024-entry two-level predictor above.

Name	Description	Style	lines of code	# of indirect branches	instr. / indirect	cond. / indirect	virtual %	switch %	indirect %	1 target %	2 targets %	> 2 targets %	active branches	
													99 %	100 %
idl	IDL compiler[a]	OO	13,900	1,883,641	47	6	93.2	3.2	3.6	97.1	0.1	2.8	70	543
jhm	JHM[b] 6-12M	OO	15,000	6,000,000	47	5	93.6	1.2	5.2	58.7	1.4	39.9	34	155
self	Self-93 VM: 5-6M	OO	76,900	1,000,000	56	7	76.0	4.4	19.6	40.1	31.6	28.3	848	1855
xlisp	SPEC95	C	4,700	6,000,000	69	11	0.0	0.1	99.9	38.9	9.0	52.1	4	13
troff	GNU groff 1.09	OO	19,200	1,110,592	90	13	73.7	12.5	13.8	41.9	13.6	44.5	61	161
lcom	HDL[c] compiler	OO	14,100	1,737,751	97	10	63.2	36.8	0.0	33.5	54.0	12.5	87	328
AVG-100: instr/indirect < 100			23,967	2,955,331	68	9	66.6	9.7	23.7	51.7	18.3	30.0	184	509
perl	SPEC95	C	21,400	300,000	113	17	0.0	31.7	68.3	41.2	0.0	58.8	7	24
porky	scalar optimizer[d]	OO	22,900	5,392,890	138	19	70.6	23.8	5.6	15.6	8.1	76.3	89	285
ixx	IDL parser[e]	OO	11,600	212,035	139	18	46.5	52.2	1.3	37.1	6.4	56.5	91	203
edg	C++ front end	C	114,300	548,893	149	23	0.0	62.4	37.6	7.9	29.6	62.5	186	350
eqn	equation typesetter	OO	8,300	296,425	159	25	33.8	66.2	0.0	4.2	37.8	58.0	58	114
gcc	SPEC95	C	130,800	864,838	176	31	0.0	31.5	68.5	0.8	1.7	97.5	95	166
beta	BETA compiler	OO	72,500	1,005,995	188	23	0.0	2.3	97.7	18.7	28.1	53.2	135	376
AVG-200: 100 < instr/indirect < 200			54,543	1,231,582	152	22	21.6	38.6	39.9	17.9	16.0	66.1	94	217
AVG: instr/indirect < 200			40,431	2,027,158	113	16	42.4	25.3	32.4	33.5	17.0	49.5	136	352
AVG-OO: OO, instr/indirect < 200			28,267	2,071,037	107	14	61.2	22.5	16.3	38.5	20.1	41.3	164	447
AVG-C: C, instr/indirect < 200			67,800	1,928,433	127	21	0.0	31.4	68.6	22.2	10.1	67.7	73	138
m88ksim	SPEC95	C	12,200	300,000	1827	233	0.0	46.2	53.8	2.9	10.3	86.8	5	17
vortex	SPEC95	C	45,200	3,000,000	3480	525	0.0	30.7	69.3	23.1	16.9	60.0	10	37
ijpeg	SPEC95	C	16,800	32,975	5770	441	0.0	97.8	2.2	96.7	3.2	0.1	7	60
go	SPEC95	C	29,200	549,656	56355	7123	0.0	99.0	1.0	0.2	0.0	99.8	5	14
AVG-infreq: instr/indirect > 200			25,850	970,658	16858	2081	0.0	68.4	31.6	30.7	7.6	61.7	7	32

Table 1. Benchmarks and commonly shown averages (arithmetic means)

[a] SunSoft version 1.3
[b] Java High-level Class Modifier
[c] hardware description language compiler
[d] SUIF 1.0
[e] Fresco X11R6 library

2. Benchmarks

Our main benchmark suite consists of large object-oriented C++ applications ranging from 8,000 to over 75,000 non-blank lines of C++ code each (see Table 1), and *beta*, a compiler for the Beta programming language [MMN93], written in Beta. We also measured the SPECint95 benchmark suite with the exception of *compress* which executes only 590 branches during a complete run. Together, the benchmarks represent over 500,000 non-comment source lines.

All C and C++ programs except *self*[1] were compiled with GNU gcc 2.7.2 with options -O2 -multrasparc plus static linking (required by *shade*) and run under the *shade* instruction-level simulator [CK93] to obtain traces of all indirect branches. Procedure returns were excluded because they can be predicted accurately with a return address stack [KE91]. All programs were run to completion or until six million indirect branches were executed.[2] In *jhm* and *self* we excluded the

[1] *self* does not execute correctly when compiled with -O2 and was thus compiled with "-O" optimization. Also, *self* was not fully statically linked; our experiments exclude instructions executed in dynamically-linked libraries.

initialization phases by skipping the first 5 and 6 million indirect branches, respectively.

For each benchmark, Table 1 lists the number of indirect branches executed, the number of instructions executed per indirect branch, the number of conditional branches executed per indirect branch, and the source of the indirect branches (switch statements, virtual function calls, or indirect function calls). It also shows the percentage of indirect branch executions that correspond to the branch classes used in section 3, as well as the number of branch sites responsible for 99%, and 100% of the branch executions. For example, only 5 different branch sites are responsible for 99% of the dynamic indirect branches in *go*. The SPECint95 programs are dominated by very few indirect branches, with less than ten interesting branches for all programs except *gcc*. Four of the SPEC benchmarks execute more than 1,000 instructions per indirect branch. Since the impact of branch prediction will be very low for the latter four benchmarks, we exclude them when optimizing predictor performance. Most results below refer to the AVG misprediction rate, i.e., the average of all benchmarks with fewer than 200 instructions between indirect branches (see Table 1).

3. Background

Before discussing cascaded predictors, we briefly review path-based two-level indirect branch predictors [DH98] and categorize the different sources of misprediction for indirect branches.

3.1 Branch Target Buffers (BTB)

The simplest design for indirect branch target prediction is a Branch Target Buffer (see Figure 1). Its uses a selection of bits of the current branch address that serves as a key pattern into a table that stores the most recently observed target for the pattern. If the key pattern is longer than needed to index all entries of the table (as is likely with the 24-bit pattern shown above), the lower-order bits of the pattern are used to index into the table and the high-order bits are stored as tags in the table entry. Alternatively, a tagless table simply shortens the key pattern by discarding the high-order bits.

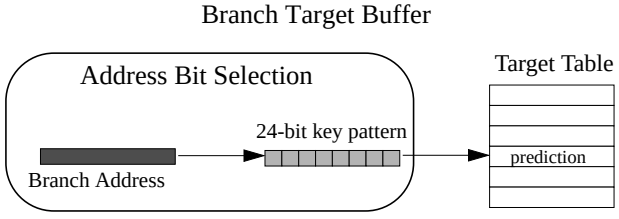

Figure 1. Branch Target Buffer. Address bits of a branch are used to access a Prediction Table that stores the most recently observed target addresses.

[2] We reduced the traces of three of the SPEC benchmarks in order to reduce simulation time. In all of these cases, the BTB misprediction rate differs by less than 1% (relative) between the full and truncated traces, and thus we believe that the results obtained with the truncated traces are accurate.

BTBs may mispredict for several reasons (Table 2). *Compulsory misses* occur the first time a branch is encountered. *Misprediction misses* occur when the branch is found in the predictor table but the predicted target is incorrect. For ideal BTBs, with full-precision addresses and unlimited, fully associative prediction tables, misprediction misses occur 25% of the time. *Capacity misses* occur when the prediction from the current branch was evicted from the table by a more recently executed branch. *Conflict misses* occur when the table is not fully associative and an entry is evicted by another branch with the same index (i.e., a conflict miss is like a capacity miss, but it is the size of the associativity set that is the limiting factor). In this study we use associativity sets of size four unless mentioned otherwise. For a table with 256 entries (64 associativity sets of size four), most BTB capacity and conflict misses disappear.

Miss cause	Summary	contribution to overall BTB misprediction rate
Compulsory	First branch execution (no target)	negligible
Misprediction	Target is wrong	25%
Capacity	Target got evicted	negligible for >256 entries
Conflict	Target got evicted from associativity set	negligible for associativity 4
Pattern Interference	Key pattern not precise enough	negligible for 24-bit pattern

Table 2. Indirect branch misprediction causes for a BTB

Finally, a *pattern interference miss* results from the projection of a full-precision address to the reduced size key bit pattern (some branches may project to the same key pattern). For our benchmark suite, using the 24 lowest order bits of the branch address suffices to eliminate all BTB pattern interference.

3.2 Two-level path based predictors

Unlike a BTB, a two-level predictor [YP91] can store more than one target for each branch. It does this by constructing a key pattern from the targets of recently executed branches leading up to the current branch. After combination with the branch address, this pattern serves as a key to lookup the most recent target in a prediction table. Since we are investigating indirect branch prediction, our predictors use bits from the target addresses of previous branches rather than the taken/not taken history bits commonly used in conditional branch prediction. We follow Nair's terminology [Nair95] and call this *path-based prediction*. The *path length* of a two-level predictor is the number of recent targets incorporated in the key pattern. Figure 2 shows a two-level predictor of path length three. A 24-bit global

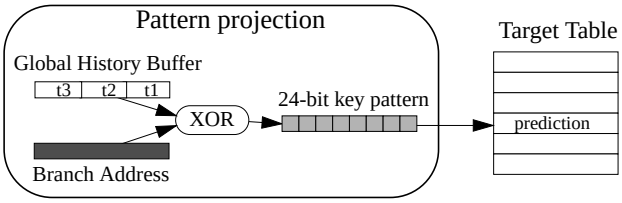

Figure 2. Two-level predictor of path length 3. The pattern projector constructs a key by concatenation of 8 bits of the most recently observed indirect branch targets, xor-ed with the current branch address. This key pattern is used to access the table, as in a BTB.

history buffer stores 8 bits each from the three most recent indirect branch targets (unless mentioned otherwise, all path-based predictors in this paper use a total path length of 24 bits). The lower-order branch address bits are xor-ed with this pattern, resulting in a 24-bit key pattern. Two-level predictors with longer path lengths use fewer bits of each target address, so that the length of the pattern remains constant. For example, a path length 6 predictor uses 4 bits from each target address. Note that a two-level predictor with path length 0 reverts to a BTB. The 24-bit key pattern is used, as in a BTB, to access the prediction table (History Table or Target Cache in [CHP97]) by splitting it into table index and tag as explained in the previous section.

A two-level predictor suffers from the same misprediction causes as a BTB, but to a different extent. Compulsory misses remain the same, since they only depend on the branch trace. Only few pattern interference misses occur due to reduced precision of target addresses, and conflict misses become negligible with 4-way associative tables (data not shown for space reasons).

Capacity misses are a different matter. The number of table entries stored per branch is proportional to the number of different paths leading up to the branch. Where a path length 0 predictor (BTB) stores one entry for every branch, a longer path length predictor stores an entry for every target combination that leads to the branch in a entire program run. This is exactly what makes two-level predictors work well: if the target pattern in the history buffer correlates with the current branch target, a two-level predictor captures this correlation by storing a different target address for each pattern. Unfortunately, longer path lengths can actually reduce prediction accuracy by inflating capacity misses; each branch needs a larger number of entries (one for each path), and the number of paths increases with path length.

3.3 Branch classes

To better understand the remaining miss causes (misprediction misses), we classified branches according to the number of different targets encountered in a program run, or branch *arity*. The *arity* of each branch was determined in a separate profiling run, and a subsequent simulation produced prediction data for each class of branches.

After some experimentation, we chose to form three classes: one target, two targets, and more than two targets. Branches with only one target (*monomorphic* branches) constitute 67% of all branches but are executed only 34% of the time. We divided *polymorphic branches* (branches with more than one target) into two classes. 18% of all branches jump to two targets, for 17% of all branch executions. Branches with three or more targets constitute 15% of all branches but are executed 49% of the time.

Figure 3 shows AVG misprediction rates per branch class and path length. (Recall that AVG includes all benchmarks except those executing indirect branches very infrequently, see Table 1). We first discuss monomorphic branches, which are perfectly predicted by a BTB. Longer path lengths increase the number of mispredictions since every different path leading to the branch causes an initial miss. These *cold start misses* are similar to compulsory misses, but occur once for each pattern, instead of once per branch. As the path length grows, so does the number of cold start misses incurred for monomorphic branches. Since monomorphic branches occupy multiple table entries, capacity misses increase as a secondary effect.

In contrast, branches with two targets benefit from history and have an optimal path length of two, causing few mispredictions. The bulk of mispredictions comes from branches with three or more targets, which are best predicted with path length three (for 1K-entry tables).

Miss cause	Summary	contribution to overall misprediction rate
Compulsory	First branch execution (no target)	negligible
Cold start	First time pattern is encountered (no target)	small, increase with path length
Misprediction	Target is wrong	5.8%, for optimal[a] path length 6
Capacity	Target got evicted	large, increase with path length
Conflict	Target got evicted from associativity set	negligible for associativity 4
Pattern Interference	Key pattern not precise enough	negligible for 24-bit pattern

Table 3. Indirect branch misprediction causes for 2-level predictor

[a] for a fully associative, unlimited prediction table (see [DH98])

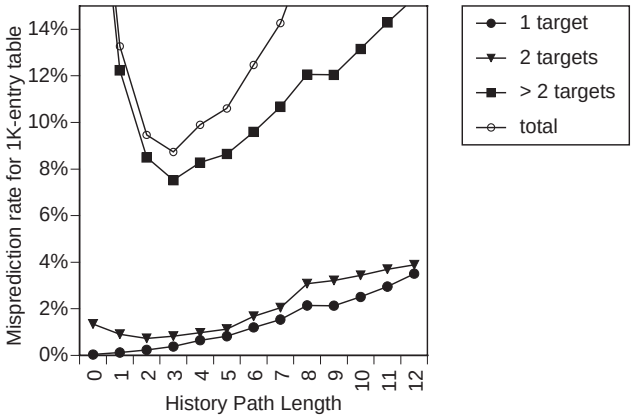

Figure 3. Misprediction rates for a 1K-entry 4-way associative prediction table

The differing behavior of these branch classes can be exploited to achieve better prediction with a classifying hybrid predictor which uses different component predictors for the different classes. In particular, if monomorphic branches were predicted with a BTB and never entered into a more expensive path-based predictor, the load on the second table could be significantly reduced (thus lowering its capacity misses as well). A simple experiment with a hypothetical classifying predictor confirms this intuition: for *jhm*, a 256-entry predictor predicting all branches experiences a table miss rate of 2.3%, but this miss rate falls to 0.9% if monomorphic branches are removed. The next section discusses a practical way to exploit this effect with dynamic target-based classification (i.e., requiring no separate profiling run, compiler changes, or changes to the instruction set architecture).

4. Cascaded prediction

A *cascaded predictor* is similar to a two-level predictor but uses a more sophisticated update rule. It classifies branches dynamically by observing their performance on a simple first-stage predictor. Only when this predictor fails to predict a branch correctly is a more powerful second-stage predictor permitted to store predictions for newly encountered history patterns of that branch. By preventing easily predicted branches to occupy prediction table space in the second-stage predictor, the first-stage functions as a filter. By filtering out easily predicted branches, the first-stage predictor prevents them from overloading the second-stage table, thereby increasing its effective capacity and overall prediction performance. As a result, the second-stage predictor's prediction table space is used to predict only those branches that actually need history-based prediction.

Figure 4 shows the prediction and update scheme for a cascaded predictor. If both predictors have a prediction, the second-stage predictor takes precedence. Therefore, the second-stage predictor's prediction table must have tagged entries, so that table misses can be detected.

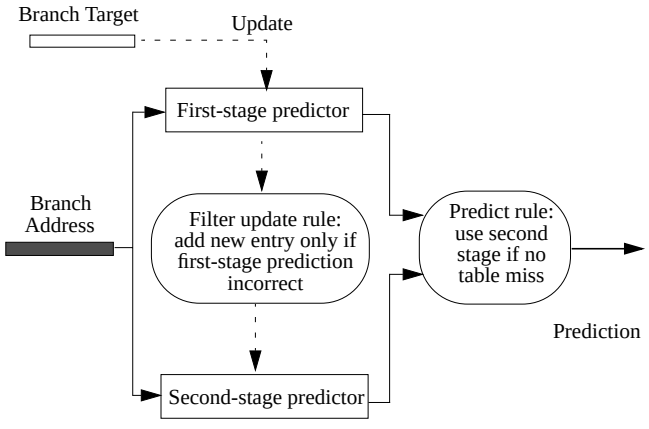

Figure 4. Cascaded predictor

The main difference to existing hybrid predictors lies in the handling of table updates. If the first-stage predictor predicted the branch correctly, we do not allow the second-stage predictor to create a *new* entry for the branch. However, if the entry is already present, the target is updated if necessary, allowing patterns already observed to require second-stage prediction to continue to be updated. In contrast, the first-stage predictor is always allowed to update its table.

We look specifically at the case where the first-stage predictor is a BTB, i.e., does not employ history. A BTB stores at most one entry per branch, which suffices for monomorphic branches. Non-monomorphic branches are better predicted by a longer path length predictor, as shown in section 3, and such branches will advance to the second stage of the predictor cascade.

We examined two variants of cascading predictors. Predictors with a *strict filter* only allow branches into the second-stage predictor if the first-stage predictor mispredicts (a target prediction is present in the table but it is wrong). In other words, branches only advance to the second stage if they are provably non-monomorphic. A strict filter thus prevents compulsory misses in the first stage from causing new entries in the second stage. In contrast, a *leaky filter* also allows new second-stage entries on first-stage table misses. Thus, the second-stage table may contain one entry for each monomorphic branch.

4.1 Strict filters

Successful prediction in the filter classifies a branch as monomorphic (if only temporarily). If a branch misses in the filter predictor, nothing conclusive is known: every branch incurs a compulsory table miss in the filter, even if it is purely monomorphic. To prevent compulsory misses of the filter to pass through to the second-stage predictor, strict filtering disallows new entry insertion in the second-stage predictor on a table miss (but it passes on *mispredicted* branches). To implement this strict filter design, the filter's table must be associative (i.e. it must check for pattern equality using tags).

Figure 5 shows the misprediction rates for three selected second-stage predictors (each with optimal path length for its size). We also show the misprediction rate of a cascaded predictor without filtering. Even without filtering, the first-stage predictor reduces overall misprediction rates compared to the stand-alone predictor (shown as filter size 0) by providing an educated guess in the case of a table miss in the second-stage predictor. In other words, a staged predictor consisting of a BTB and a path-based predictor reduces cold-start misses even without filtering.

Strict filters do not perform well for small filter table sizes. For 16 entries or less, overall misprediction rates are even higher than that of the stand-alone predictor. Essentially, a strict filter predictor recognizes branches as non-monomorphic only if the branch remains in the filter table long enough to incur a target change (and thus a misprediction). But with small filters, many branches are displaced from the filter before they mispredict, and thus they never enter the second

stage. When the filter becomes large enough to avoid most capacity misses, this effect disappears and filtering starts to pay off. At 256 entries, strict filtering performs as intended, i.e., it prevents monomorphic branches from overloading the second-stage predictor, resulting in lower misprediction rates than those of a non-filtering staged predictor.

4.2 Leaky filters

The sensitivity of strict filters to capacity misses is a serious flaw; the performance of a filtered predictor should remain at least as good as that of a stand-alone second-stage predictor. To prevent filter capacity miss problems, a leaky filter inserts an entry into both predictors upon a first-stage table miss. That is, only correctly predicted branches are stopped by the filter. Thus, every branch is introduced at least once into the second-stage predictor, but filtering still occurs for later executions of the same branch: as long as the branch remains in the filter table and doesn't mispredict, no further second-stage entries will be permitted. If the load on the second-stage predictor table is high, the compulsory entries for monomorphic branches will eventually be displaced by entries for non-monomorphic branches. Leaky filters are cheaper to implement than strict filters: since a misprediction and a table miss is treated the same way (new entry in second-stage table), the filtering predictor can use a tagless table. The second-stage predictor still needs tags in order to recognize a table miss.

Figure 6 shows the performance of leaky filters. Even for very small filters, the filtering effect is pronounced and improves misprediction rates compared to a non-filtering cascaded predictor. For example, a 32-entry BTB filter improves the misprediction rate of a 256-entry monopredictor from 11.7% to 10.7%. Filtering still helps even with very large (4K) predictors, reducing mispredictions by about 0.5%. For large filters of 256 entries or more, the leaky filter's misprediction rate is only slightly better than that of a strict filter (not shown in the figure).

Table 4 shows the best path length and misprediction rate for cascaded predictors with second stage two-level predictors. Even very small BTB filters increase the effective capacity of the second-stage table, allowing it to accommodate longer paths without incurring extensive capacity misses. For example, a 16-entry filter reduces the misprediction rate for all second-stage sizes to below that of a monopredictor with twice the number of entries and uses the path length of a monopredictor that is four times larger. A 64-entry filter lowers the misprediction rate below that of a monopredictor four times as large, and for all table sizes smaller than 1K entries, beyond that of a dual-path hybrid predictor of twice the size.

We also studied cascaded predictors that use dual-path hybrid predictors (see [DH98]) in the second stage, anticipating that filtering would again reduce second-stage misses and allow longer path lengths. For each filtered hybrid of table size T, we simulated the best path length couples of hybrid predictors with table sizes T, 2T and 4T

The resulting improvements were equally pronounced (Table 5). Again, filtering reduces the misprediction rate and increases the best path length choices for each table size. Whereas dual-path hybrid predictors need at least 1024 table entries to achieve misprediction rates below 9%, a filtered dual-path hybrid attains this threshold with only 544 entries (32-entry filter BTB plus 512-entry dual-path hybrid). Adding an 8-entry filter to a 1K-entry hybrid predictor lowers its misprediction rate from 9.0% to 8.2%. A 128-entry filter reduces this to 7.5%. Across all table sizes, cascaded prediction reduces table size by roughly a factor of two.

Figure 7 shows AVG misprediction rates for selected prediction schemes. For all schemes and table sizes, filtering reduces misprediction rates. For tables of 512 entries or less, the resulting misprediction rate of filtered monopredictors is equal to or lower than that of filtered dual-path hybrid predic-

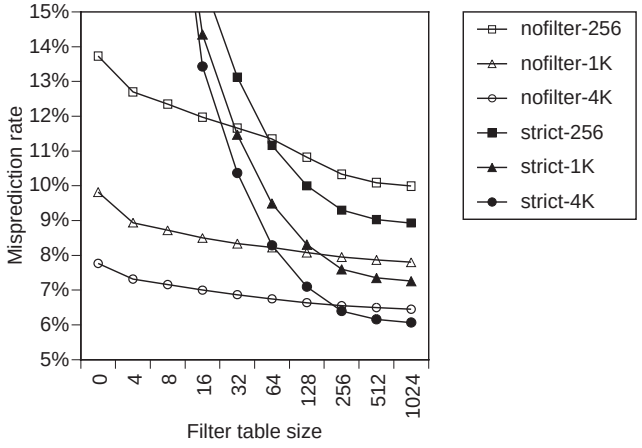

Figure 5. Misprediction rates, cascaded predictor with strict filter

Second-stage predictor table size is 256, 1K and 4K entries; both predictor tables are 4-way associative. Also shown is a cascaded predictor without filtering.

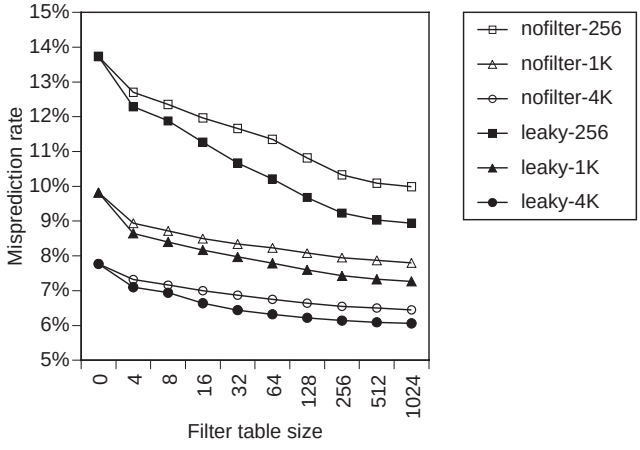

Figure 6. Misprediction rates, cascaded predictor with leaky filter

Second-stage predictor table size is 256, 1K and 4K entries; both predictor tables are 4-way associative. Also shown is a cascaded predictor without filtering.

Second stage table size	mono	Best P per filter table size								mono	Miss% per filter table size								hybrid
		4	8	16	32	64	128	256	512		4	8	16	32	64	128	256	512	
64	1	2	2	2	2	2	2	2	2	19.8	18.6	17.4	15.8	14.7	13.7	12.7	12.0	11.7	19.8
128	1	2	2	2	2	3	3	3	3	17.0	14.8	14.1	13.3	12.7	11.9	11.1	10.5	10.3	16.7
256	2	2	2	3	3	3	3	3	3	13.7	12.3	11.9	11.3	10.7	10.2	9.7	9.2	9.0	13.3
512	2	3	3	3	3	3	3	3	3	11.3	10.2	9.7	9.3	8.9	8.7	8.4	8.1	8.0	10.9
1024	3	3	3	3	4	4	4	5	5	9.8	8.6	8.4	8.2	8.0	7.8	7.6	7.4	7.3	9.0
2048	3	4	4	4	5	6	6	6	6	8.5	7.8	7.6	7.4	7.2	7.0	6.9	6.7	6.7	7.8
4096	3	4	4	6	6	6	6	6	6	7.8	7.1	6.9	6.6	6.4	6.3	6.2	6.1	6.1	6.7
8192	4	5	6	6	6	6	6	6	6	7.3	6.5	6.3	6.1	6.0	5.9	5.8	5.8	5.7	6.0
16384	5	6	6	6	6	6	8	8	8	6.8	6.2	6.0	5.8	5.7	5.7	5.6	5.6	5.5	5.5

Table 4. Path length and misprediction rate for second-stage monopredictors

For comparison, the table also shows the best monopredictor ("mono") and the best dual-path hybrid predictor ("hybrid") of equivalent size. The best hybrid predictors were chosen using data from [DH96].

Second level table size	hybrid	Best P per filter table size								hybrid	Misprediction rate (%) per filter table size							
		4	8	16	32	64	128	256	512		4	8	16	32	64	128	256	512
64	1.1	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	19.8	20.6	19.3	18.3	17.5	15.8	14.7	13.9	13.6
128	2.0	2.0	3.1	3.1	3.1	3.1	3.1	3.1	3.1	16.7	15.5	14.9	13.9	13.1	12.4	11.4	10.7	10.4
256	2.0	3.1	3.1	3.1	3.1	3.1	3.1	3.1	3.1	13.3	12.3	11.8	11.3	10.9	10.5	9.9	9.3	9.1
512	3.1	3.1	5.1	5.1	5.1	5.1	5.1	5.1	5.1	10.9	10.0	9.6	9.3	9.0	8.8	8.5	8.1	8.0
1024	3.1	5.1	5.1	6.2	6.2	6.2	6.2	6.2	6.2	9.0	8.2	8.0	7.7	7.5	7.3	7.1	6.9	6.8
2048	5.1	6.2	6.2	6.2	6.2	6.2	6.2	6.2	6.2	7.8	7.1	6.9	6.7	6.5	6.5	6.4	6.3	6.2
4096	6.2	6.2	7.2	7.2	7.2	7.2	7.2	7.2	7.2	6.7	6.3	6.1	6.0	5.9	5.9	5.8	5.8	5.7
8192	6.2	8.2	8.2	8.2	8.2	8.2	8.2	8.2	8.2	6.0	5.6	5.5	5.5	5.4	5.4	5.4	5.3	5.3
16384	7.2	8.2	8.2	8.2	8.2	8.2	8.2	8.2	8.2	5.5	5.3	5.2	5.2	5.2	5.1	5.1	5.1	5.1

Table 5. Path length and misprediction rate for second-stage dual-path hybrid predictors

For comparison, the table also shows the best the best dual-path hybrid predictor ("hybrid") of equivalent size. The best hybrid predictors were chosen using data from [DH96].

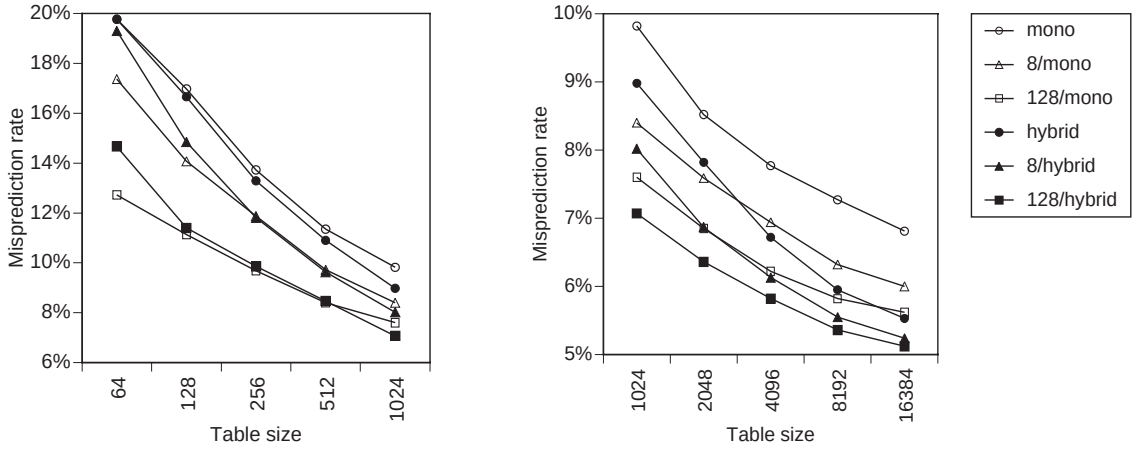

Figure 7. AVG Misprediction rates for mono and hybrid predictors without filters and with 8 and 128-entry filters. We use different scales for small and large table sizes to increase visibility.

tors. This is due to a filter's cold-start miss reduction (it usually has a zero path length prediction if the second stage has not yet encountered the longer path). A dual-path hybrid predictor does not get extra benefit from this effect at small table sizes, since its shorter component's path length is already close or equal to zero. Most of the misprediction rate reduction of small hybrid predictors is therefore due solely to the filter's capacity miss reduction. Two-level predictors, with path lengths of two or three, benefit both from capacity miss reduction and cold-start miss reduction. The resulting misprediction rates end up being fairly similar for two-level and hybrid predictors. However, for second-stage tables of 1K entries and larger, the filtered hybrid predictor's misprediction rate is substantially lower than that of a filtered monopredictor at all filter sizes. Since the short component path length is two in most of these cases, a filter's cold-start miss reduction also benefits a dual-path length hybrid predictor. At the high end of the table size range (+8K), conflict and capacity misses become less frequent, and the benefit of filtering starts to diminish.

Figure 8 shows misprediction rates for the *self*, *edg* and *gcc* benchmarks. The former two programs have the largest number of active branches in the benchmark suite. The reduction in misprediction rate is higher than the reductions on AVG of Figure 7, which shows that the benefit of filtering is especially pronounced for large programs.

Although we did not measure the actual performance impact of improved prediction in this study, previous studies indicate that it can be significant. For example, Chang et al. [CHP97] measure an execution overhead of 15% and 8% for *perl* and *gcc* with a BTB, respectively. A 32+256 cascaded predictor reduces the misprediction ratio from 5.6% to 0.1% for *perl* (7.5% to 2.7% for *gcc*), suggesting considerable savings in execution time.

5. Related work

Branch classification was first proposed for conditional branches in [CHP94]. Conditional branches are divided in six classes, corresponding to the frequency with which a branch is taken in a profiling run, with boundary values 5%, 10%, 50%, 90%, 95% and 100%. The authors present a GAs predictor with multiple branch history length and shared prediction table. A dynamically classifying predictor uses a fully associative branch address cache (BAC), consisting of 2-bit saturating counters that indicate the component predictors which best predicts a given branch. Since we use a simple predictor both to predict indirect branches and to classify them as "hard-to-predict", a BAC is unnecessary in a dynamically classifying cascaded predictor. By combining profile-guided classification for mostly monomorphic branches with dynamic classification for mixed-direction branches (between 10% and 90% taken), prediction accuracy of 96.4% is achieved in [CHP94] for conditional branches of the SPECint92 benchmark suite.

Chang et al. [CEP96] study a conditional branch predictor which uses a BTB to filter out easily predicted branches. Their predictor inhibits the history table update if the BTB's confidence counter is at maximum (e.g., "strongly taken"). In comparison, the update rule of cascaded predictors is more flexible and potentially more accurate since it allows a branch to be predicted by more than one component predictor. For example, consider a branch that is always taken except for one (long) path history for which it is always not taken. A two-stage cascaded predictor can perfectly predict this branch using one entry in each table, whereas the predictor proposed by Chang et al would incur misses.

Indirect branch prediction has been studied by Lee and Smith [LS84] (several forms of BTBs), Jacobson et al. [J+96] (path-based history schemes), Emer and Gloy [EG97] (single-level indirect branch predictors), Chang et al. [CHP97] (two-level indirect branch prediction), and Driesen and Hölzle [DH96] (two-level and hybrid indirect branch prediction). In [CHP97] a limited range of two-level predictors for indirect branches is explored and the resulting speedups of selected SPECint95 programs are measured by simulation for a superscalar processor. The misprediction rate of a BTB is reduced by half to 30.9% for *gcc* with a Pattern History Tagless Target Cache with configuration gshare(9), resulting in 14% speedup. In [DH96], a comparable non-hybrid predictor (p=3, tagless 512-entry) reaches a misprediction ratio of 31.5% for *gcc*. The cascaded predictor in presented in this study, with 4-entry filter, obtains 23.7% misprediction rate with a 512-entry, four-way associative dual-path hybrid predictor as second stage.

Hybrid prediction for conditional branches was first proposed in [McFar93]. Recent results can be found in [CHP95] and [ECP96]. Chen et al. [CCM96] propose Partial Prefix Matching prediction for conditional branch prediction and show that a PPM predictor performs better than a two-level predictor for a similar hardware budget. Since a PPM predictor chooses the prediction of the longest pattern for which a prediction is available (choosing progressively shorter path lengths until a prediction is found), a cascaded predictor's prediction rule mimics this behavior.

6. Conclusions and future work

We have described a new predictor architecture, the *cascaded branch predictor*, which dynamically classifies easily predicted branches using a simple first-stage predictor. By preventing correctly predicted branches from entering a more expensive second-stage predictor, the latter's prediction table is more effectively used. Not only does a cascaded predictor reduce the capacity misses of the second-stage predictor, it also reduces cold-start misses for monomorphic branches (branches with a single target). We tested two different update rules to implement this filtering effect: *strict filtering*, which prevents compulsory misses from entering the second stage, and *leaky filtering*, which passes on first-stage table misses. We found that the performance of strict

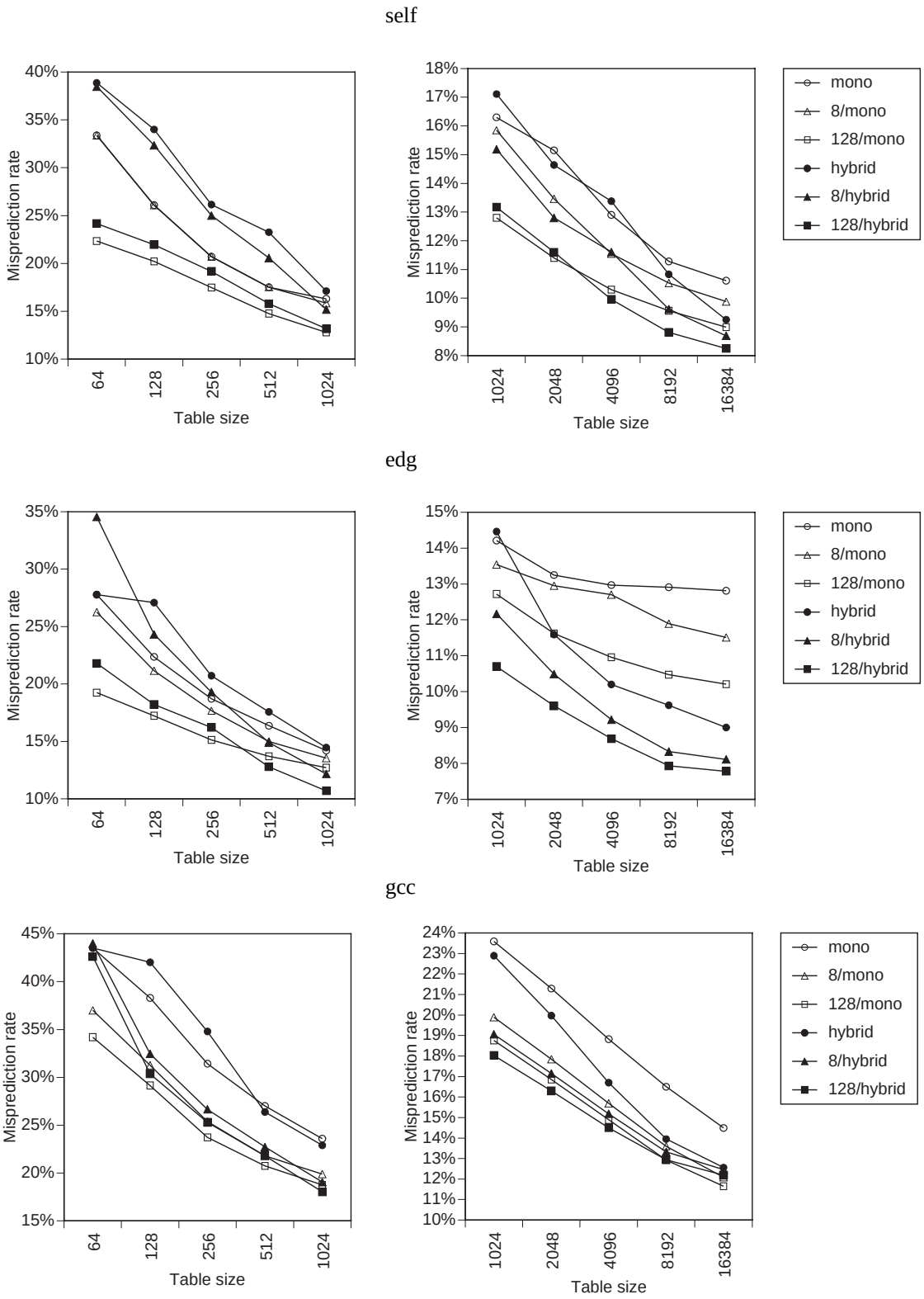

Figure 8. Misprediction rates for mono and hybrid predictors without filters and with 8 and 128-entry filters, for the *self*, *edg* and *gcc* benchmarks. Best mono predictor path lengths are determined separately for each benchmark; hybrid predictor path lengths are picked from the AVG path length choices.

filtering deteriorates badly for small tables, and that leaky filtering performs better at all table sizes, in spite of the compulsory misses it allows.

Cascaded prediction with leaky filtering achieves prediction rates equivalent to that of the previously best known predictors for roughly half their table size. For example, prefixing a 1K-entry predictor with a tiny fully-associative 4-element BTB filter reduces the overall misprediction rate from 9.8% to 8.7% for our benchmark suite which contains over 500,000 non-comment source lines. A 64-entry, 4-way associative filter reduces this further to 7.8%, the same performance as a stand-alone predictor with 4096 prediction table entries. In other words, adding the 64-entry filter allows us to reduce the size of the path-based predictor by a factor of four. Similarly, combining a 64-entry filter with a 1K-entry dual-path hybrid predictor reduces the misprediction rate from 9.0% to 7.3%, lower than the misprediction rate of a 2K-entry dual-path hybrid predictor and more than three times better than a BTB of equivalent size. Even a relatively small predictor with 32+256 entries achieves a respectable prediction accuracy of 89.3%. To our knowledge, these cascaded predictors improve upon any indirect branch predictor reported to date.

Cascaded prediction works so well because the first-stage predictor reduces the load on the second-stage predictor. Generally speaking, longer branch histories require larger tables. The filtering effect of cascaded predictors removes virtually all entries for branches that do not actually require a long path length, and thus the overall table size can be reduced without compromising prediction accuracy. Cascaded prediction also generalizes naturally to predictors with more than two components.

In principle, the filtering effect should occur for any application of path- or history-based predictors where the dynamic frequency of easily predicted cases is high. In particular, it appears plausible that conditional branch prediction or load value prediction should behave in qualitatively the same way. Therefore, we believe that cascaded predictors might also perform well in those areas. Of course, only empirical work can confirm this hypothesis, and thus we are planning to explore these questions in future work.

Acknowledgments. This work was supported in part by National Science Foundation CAREER grant CCR-9624458, an IBM Faculty Development Award, Sun Microsystems, and the State of California MICRO program. We would like to thank Anurag Acharya and Raimondas Lencevicius for their comments on earlier versions of this paper.

7. References

[CHP94] Po-Yung Chang, Eric Hao, Yale N. Patt. Branch Classification: A New Mechanism for Improving Branch Predictor Performance. *MICRO '27 Proceedings*, November 1994.

[CHP95] Po-Yung Chang, Eric Hao, Yale N. Patt. Alternative Implementations of Hybrid Branch Predictors. *MICRO '28 Proceedings*, November 1995.

[CEP96] Po-Yung Chang, Marius Evers, and Yale Patt. Improving Branch Prediction Accuracy by Reducing Pattern History Table Interference. *PACT '96 Proceedings*, October 1996.

[CHP97] Po-Yung Chang, Eric Hao, Yale N. Patt. Target Prediction for Indirect Jumps. *ISCA'97 Proceedings*, July 1997

[CCM96] I-Cheng K.Chen, John T.Coffey, Trevor N. Mudge. Analysis of Branch Prediction via Data Compression. *ASPLOS'96 Proceedings*.

[CGZ94] Brad Calder, Dirk Grunwald, and Benjamin Zorn. *Quantifying Behavioral Differences Between C and C++ Programs*. Journal of Programming Languages 2(4):313-351, December 1994.

[CK93] Robert F. Cmelik and David Keppel. *Shade: A Fast Instruction-Set Simulator for Execution Profiling*. Sun Microsystems Laboratories, Technical Report SMLI TR-93-12, 1993.

[DH96] Karel Driesen and Urs Hölzle. The Direct Cost of Virtual Function Calls in C++. In *OOPSLA '96 Conference proceedings*, October 1996.

[DH98] Karel Driesen and Urs Hölzle. Accurate Indirect Branch Prediction. *ISCA '98 Conference Proceedings*, July 1998.

[EG97] Joel Emer and Nikolas Gloy. A Language for Describing Predictors and its Application to Automatic Synthesis. *ISCA'97 Proceedings*, July 1997.

[ECP96] Marius Evers, Po-Yung Chang, Yale N. Patt. Using Hybrid Branch Predictors to Improve Branch Prediction Accuracy in the Presence of Context Switches. *Proceedings of ISCA'96*.

[HP95] Hennessy and Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1995.

[J+96] Quinn Jacobson, Steve Bennet, Nikhil Sharma, and James E. Smith. Control Flow Speculation in Multiscalar Processors. *HPCA-3 proceedings*, February 1996.

[KE91] David Kaeli and P. G. Emma. Branch History Table Prediction of Moving Target Branches due to Subroutine Returns. *ISCA '91 Proceed*ings, May 1991.

[LS84] J. Lee and A. Smith. Branch Prediction Strategies and Branch Target Buffer Design. *IEEE Computer 17(1)*, January 1984.

[LS95] James Larus and Eric Schnarr. EEL: Machine-Independent Executable Editing. In *PLDI '95 Conference Proceedings*, June 1995.

[MMN93] Ole Lehrmann Madsen, Birger Moller-Pedersen, Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley 1993.

[McFar93] S. McFarling. Combining Branch Predictors. *WRL Technical Note TN-36*, Digital Equipment Corporation, June 1993.

[Nair95] Ravi Nair. Dynamic Path-Based Branch Correlation. *Proceedings of MICRO-28*, 1995.

[USS97] Augustus K. Uht, Vijay Sindagi, Sajee Somanathan. Branch Effect Reduction Techniques. *IEEE Computer*, May 1997.

[YP91] Tse-Yu Yeh and Yale N. Patt. Two-level Adaptive Branch Prediction. *MICRO 24 Proceedings*, November 1991.

[YP93] Tse-Yu Yeh and Yale N. Patt. A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History. *Proceedings of ISCA'93.*