Improving Indirect Branch Prediction With Source- and Arity-based Classification and Cascaded Prediction

Karel Driesen and Urs Hölzle
Department of Computer Science
University of California
Santa Barbara, CA 93106

Technical Report TRCS98-07
March 15, 1998

Abstract. Indirect branch prediction is likely to become more critical to program performance because indirect branches occur more frequently in object-oriented programs. We study indirect branch behavior in both procedural and object-oriented programs in order to build more accurate predictor architectures. First we use a statically classifying hybrid predictor with a shared history table and separate history buffers tuned for different branch classes. Opcode-based classification, which classifies branches according to their source code origin, leads to only minor improvements in prediction performance. Arity-based classification, which classifies branches according to the number of different targets, obtains better performance, competitive with recently proposed dual-path length hybrid predictors. Finally, we present cascaded branch predictors, which dynamically classifies easily predicted branches using an inexpensive predictor. By preventing insertion of these branches into a more powerful second stage predictor, cascaded prediction obtains prediction rates equivalent to that of the previously best known indirect branch predictors, but at roughly half the cost. Specifically, a cascaded predictor with a total of 288 history table entries achieves 89.3% prediction accuracy.

1. Introduction

Programs written in an object-oriented programming style execute indirect branches more frequently than programs written in a procedural language (see Table 1). The overhead of mispredicted indirect branches increases with longer pipelines and higher instruction issue [HP95]. Together, these trends make accurate indirect branch prediction increasingly critical to performance [CHP97].

In this study, we use a suite of large object-oriented and procedural programs to:

- determine whether the two programming styles show substantial difference in indirect branch behavior; and to
- determine whether indirect branch predictor can exploit this behavior to improve prediction performance.

First, we study whether knowing a branch's source-level origin (indirect call, virtual call, or switch statement) could improve branch prediction using branch classification [CHP94]. Although some behavioral aspects of these three branch classes are quite different, it appears to be hard to exploit in a predictor. We evaluate the prediction performance of a range of two-level predictors [CHP97], [DH98] on each class. A statically classifying hybrid predictor reduces mispredictions from 9.8% to 9.5% for a predictor using a 1K-entry, 4-way associative, shared history table.

A second experiment classified branches based on the number of different targets encountered in a program run (the *arity* of a branch). This classification reduces the misprediction rate of the 1K-entry predictor to 8.9%, slightly better than the 9.0% of a dual-path length hybrid predictor.

Although these experiments indicate that it is hard to exploit the behavioral differences of indirect branches with static classification, a dynamically classifying predictor can significantly improve performance. The third section of this study describes and evaluates *cascaded branch prediction*. A cascaded predictor classifies branches into two classes: those that are perfectly predicted (if only temporarily) by a simple predictor, and those that are not. When the first-stage predictor mispredicts a branch, the branch is entered into a more powerful but also more costly second-stage path-based predictor. By filtering out easily predicted branches, the first-stage prevents them from overloading the second-stage predictor, thereby increasing its effective capacity and overall prediction performance.

Given the right update strategy, even very small filters improve the performance of the second-stage predictor. Using only a fraction of the resource budget of a 1K-entry second-stage predictor, a fully-associative BTB filter with 4 entries reduces the overall misprediction rate from 9.8% to 8.7%. A 64-entry, 4-way associative filter reduces this further to 7.8%, the same performance as a stand-alone predictor with four times as many history table entries, or a dual-path hybrid predictor with twice the number of entries. 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. To our knowledge, the prediction accuracy of this cascaded predictor is better than any other indirect branch predictor reported to date.

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 (options -O2 -multrasparc plus static linking) 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 initialization phases by skipping the first 5 and 6 million indirect branches, respectively.

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.

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.

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 percentage of indirect branch executions that correspond to the branch classes used in section 3 and section 4, 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 (by minimizing the AVG misprediction rate).

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

3. Opcode-based classification

In a first experiment, we classified branches into three classes: switch, virtual call, and indirect call. The GNU compiler translates *switch statements* with more than seven cases into an indirect jump; no other C/C++ constructs use an indirect jump, and thus this class is easy to recognize. To recognize *virtual function calls*, we used EEL [LS95] to look for the characteristic five-instruction sequence generated by gcc and marked the corresponding call instruction accordingly. All other indirect calls were classified as indirect calls (i.e., calls through function pointers).[1]

			OO			C		
Class	Source structure	Detection	dyn%	stat%	#targets	dyn%	stat%	#targets
switch	switch statements, >7 cases	by opcode (JMP in SPARC)	22.5	2.6	12.7	31.4	35.6	4.9
virtual	virtual function calls in C++	with EEL (5-instr. sequence)	61.2	69.4	2.1	N/A	N/A	N/A
indirect	all other indirect branches	by opcode (JMP and link)	16.3	28.0	2.2	68.6	64.4	5.1

Table 2. Indirect branch classes with dynamic and static frequency, number of targets for OO and C programs

For each branch class, Table 2 shows the average dynamic frequency, static frequency, and average number of targets. Not surprisingly, virtual function calls dominate in OO programs (61% of all branches) whereas indirect calls dominate in C programs (69%). The branch classes differ substantially; in particular, switch branches have a much higher number of targets than the other branches. We hope to exploit this difference by fitting a monopredictor to each class and combining these in a hybrid predictor.

Before discussing classifying predictors, we briefly discuss the basic monopredictor that serves as the basic building block for more advanced predictors. (For a detailed study of this predictor design we refer to [DH98].) Figure 1 shows a two-level predictor with path length 8. A 24-bit global

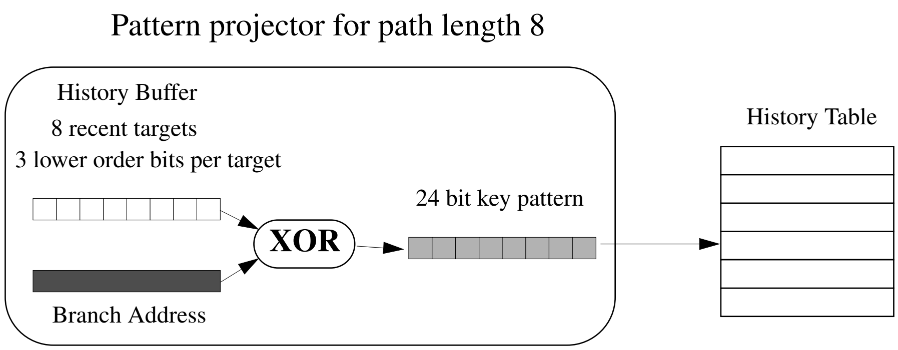

Figure 1. Two-level indirect branch predictor. For each path length between 0 and 12, the pattern projector constructs a key of 24 bits, which is then used to access a History Table that stores target addresses.

history buffer stores three bits of the most recent eight targets. The branch address is xor-ed with this pattern, giving a 24-bit key pattern. This pattern is used to access a history table storing the most recently observed target address of each pattern. Longer path lengths use fewer bits of each

[1] Obviously, the distinction between virtual and indirect in hardware would require the introduction of a new opcode or special register usage conventions.

target address, so that the length of the pattern remains constant[1]. Some bits of the key pattern are used to index into the history table, whereas the rest form the tag stored in each table entry (except for tagless tables). Unless mentioned otherwise, we use 4-way associative history tables, and table entries are updated only after two consecutive mispredictions. We will refer to this setup as a *mono-predictor* of path length p.

A classifying hybrid predictor uses one component predictor for each branch class. The predictors may use different branch history organizations. While every component predictor's history pattern is updated by all branches, only one is selected for prediction based on the branch class. The classifying predictor can use a shared history table for all component predictors (as shown in Figure 2) or three separate tables.

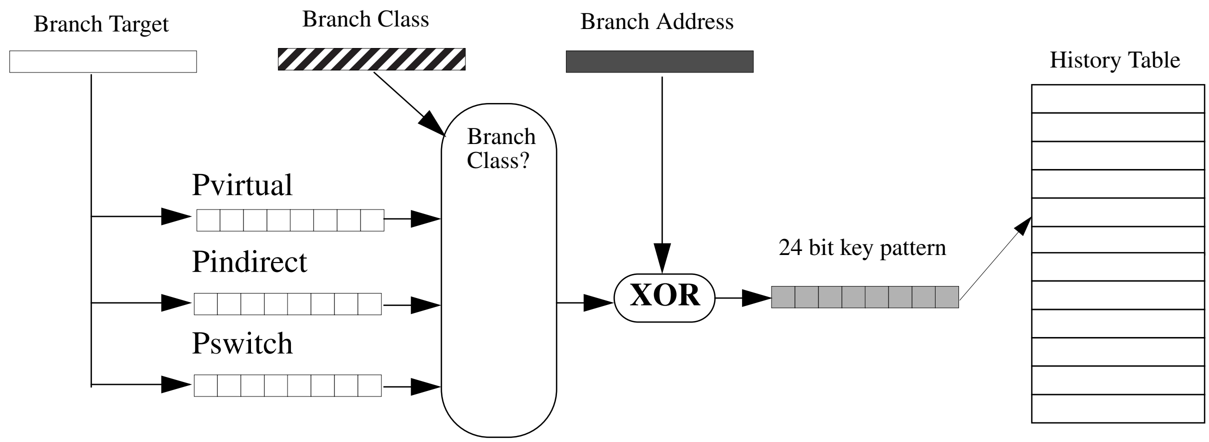

Figure 2. Classifying hybrid predictor with shared history table

We first determined the path length that results in the lowest misprediction rate for each branch class. The best path length varies according to table size and associativity, since longer paths

[1] For a 24-bit key, the loss of prediction performance due to reduced precision is negligible [DH98].

require larger tables to be effective [DH98]. Figure 3 shows AVG misprediction rates per branch class and path length for a 1K-entry, 4-way associative table.

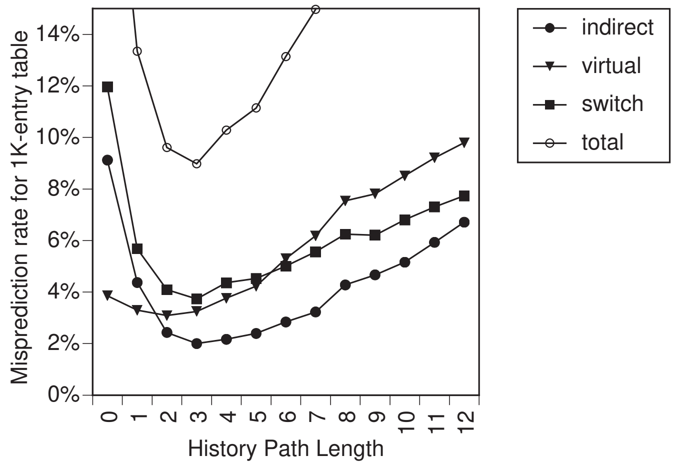

Figure 3. Misprediction rates per branch class for a 1K-entry 4-way associative history table, for path lengths 0 (BTB) to 12.

Switch branches behave similarly to *indirect* branches even though they have more targets; both classes reach minimal prediction rates at path length three. *Virtual* branches differ from the other classes in that their ideal path length is slightly shorter, i.e. they correlate with more recently executed branches.

After finding the best path length for each branch class separately, we calculated the misprediction rate of a classifying predictor using separate tables for each class. For a given total table size T, one of the component predictors uses a table of T/2 entries, the other two components use T/4 entries. From the three possible configurations we selected the one with the lowest miss rate. We then used the best path lengths for separate component predictors of size T, 2T and 4T, to simulated a shared table classifying predictor of size T.

Table 3 and Figure 4 show the resulting predictor performances in comparison with a non-classifying monopredictor of equal size. Path lengths for the indirect and switch branch class are nearly identical over all table sizes. Virtual branches are better predicted by shorter path lengths.

Clearly, classifying predictors with separate tables don't perform well, having uniformly higher misprediction rates than a monopredictors. These predictors do not utilize their table space efficiently since the relative frequency of the different branch classes varies widely across benchmarks. For instance, C programs do not have virtual branches, so at least a quarter of the total table space is unused. A shared-table classifying predictor performs better, obtaining a uniformly lower misprediction rate than a monopredictor of equivalent size. This result demonstrates that per-class path length determination does pay off, although the gain is quite small so that the higher hardware cost of opcode-based prediction does not seem justified.

4. Arity-based classification

In a second experiment, we classified branches according to a dynamic measure, the number of different targets encountered in a program run, or branch *arity*. The *arity* of a branch can be deter-

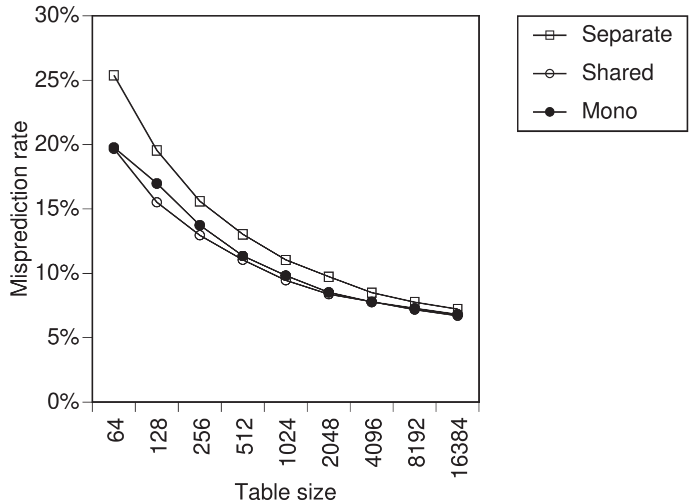

Figure 4. Misprediction rates for opcode-based classifying hybrid predictors

Table size	Separate Hybrid			Shared Hybrid		Mono	
	Class with T/2	P I.V.S	mis s%	P I.V.S	mis s%	P	mis s%
64	Indirect	2.0.2	25.4	2.0.2	19.7	1	19.8
128	Indirect	2.0.2	19.5	2.0.2	15.5	1	17.0
256	Switch	2.0.2	15.6	2.0.2	13.0	2	13.7
512	Indirect	2.0.2	13.0	3.1.2	11.0	2	11.3
1024	Indirect	3.1.2	11.0	3.2.3	9.5	3	9.8
2048	Switch	3.1.3	9.7	3.2.3	8.4	3	8.5
4096	Switch	3.2.3	8.5	4.2.3	7.8	3	7.8
8192	Indirect	4.2.3	7.8	4.3.4	7.2	4	7.3
16384	Switch	4.3.4	7.2	5.3.5	6.7	5	6.8

Table 3. Misprediction rates for separate and shared opcode-based classifying predictors.
Also shown are the branch class with the largest table (for classifying predictors with separate tables), and the best path length combinations (indirect/virtual/switch) for opcode-based classifying hybrid predictors.

mined in a profiling run, or estimated by source code analysis. Annotation of this arity in the branch opcode would allow a classifying hybrid predictor to function in the same manner as the opcode-based classifying predictor shown earlier.

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. 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 5 shows AVG misprediction rates per branch class and path length. Monomorphic branches are perfectly predicted by a monopredictor without history, i.e., a BTB. Longer path lengths increase the number of mispredictions since every different path leading to the branch causes an extra cold-start miss. Also, each monomorphic branch may occupy multiple table entries, thus increasing capacity misses. Branches with two targets have an optimal path length of two and cause few mispredictions. The bulk of mispredictions comes from branches with three or more targets, with optimal path length three.

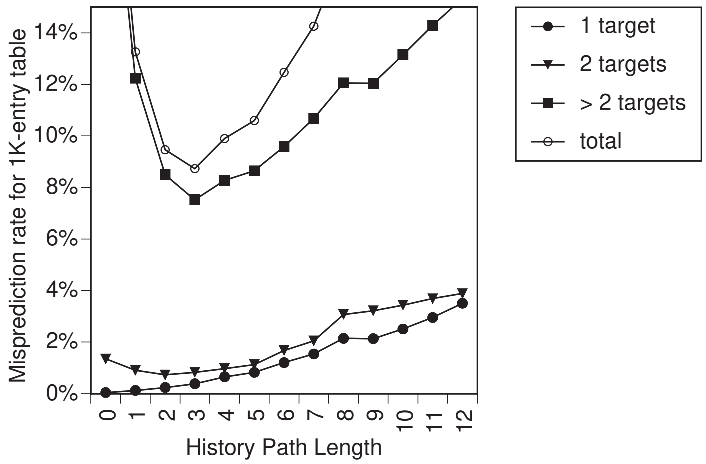

Figure 5. Misprediction rates for a 1K-entry 4-way associative history table

Table 4 shows the best arity-based predictor per table size for both separate and shared table predictors. The best path length for monomorphic branches is always zero, but path lengths for the other branches increase with table size. In a component predictor with separate tables, the largest table always goes to polymorphic branches, and the best path length for these branches is nearly always the same as the best path length for a monopredictor.

Table size	Separate Hybrid			Shared Hybrid		Mono	
	Class T/2	P 1.2.>2	miss %	P 1.2.>2	miss %	P	miss %
64	> 2 targets	0.0.1	23.3	0.0.2	18.8	1	19.8
128	> 2 targets	0.0.2	17.3	0.0.2	14.7	1	17.0
256	> 2 targets	0.0.2	13.5	0.0.2	12.1	2	13.7
512	> 2 targets	0.1.2	11.3	0.1.2	10.5	2	11.3
1024	> 2 targets	0.2.3	9.7	0.2.3	8.9	3	9.8
2048	> 2 targets	0.2.3	8.4	0.2.3	8.0	3	8.5
4096	> 2 targets	0.2.3	7.8	0.2.4	7.4	3	7.8
8192	> 2 targets	0.2.4	7.3	0.2.4	6.9	4	7.3
16384	> 2 targets	0.3.5	6.9	0.3.5	6.5	5	6.8

Table 4. Mispredictions rates for separate and shared arity-based classifying predictors.
Also shown are the branch class with the largest table (for classifying predictors with separate tables), and the best path length combinations (indirect/virtual/switch) for opcode-based classifying hybrid predictors.

Figure 6 illustrates that the misprediction rate of a separate component hybrid is nearly the same as that of a monopredictor. These predictors benefit from shorter path lengths which generate fewer different patterns per branch, so that fewer table entries are used for branches of arity 1 and 2. On the other hand, polymorphic branches are restricted to use half of the total table size. Both effects appear to nearly cancel each other out, as shown in Figure 6. With a shared table, the latter restriction is removed, and the misprediction rate is always lower than that of a monopredictor.

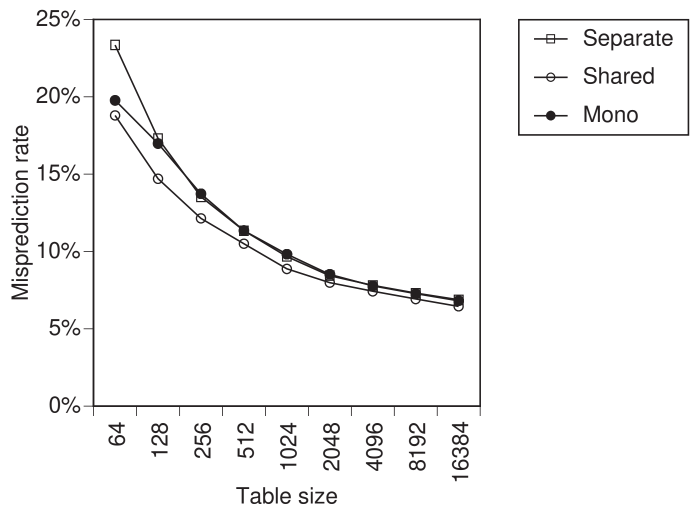

Figure 6. Misprediction rates for arity-based classifying hybrid predictors

Although an arity-based classifying hybrid predictor performs significantly better than a monopredictor, especially for smaller tables, it has significant practical disadvantages:

- Branch instructions need to be annotated with an arity counter, requiring an instruction set extension.
- Arity must be determined by profiling, which may not remain accurate over different program runs, or program analysis, which may not be precise enough to lower the overall misprediction rate.

For these reasons, arity-based classification may not be practical. However, its performance shows that classification on the number of targets is more promising than opcode-based classification. In particular, arity-based classification helps reduce the load on a path-based predictor by removing monomorphic branches. This effect significantly reduces capacity misses: for example, for *jhm* the 256-entry predictor of a separate-table classifying predictor of size 512 experiences a table miss rate of only 0.9% versus 2.3% for a 256-entry monopredictor. To exploit this effect, we will use dynamic target-based classification in the next section, which is more adaptive and does not require changes to the instruction set architecture.

5. Cascaded prediction

A *cascaded predictor* 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 history table space in the second-stage predictor, the first-stage predictor functions as a filter. By filtering out easily predicted branches, the first-stage 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 history table space is used to predict only those branches that actually need history-based prediction.

Figure 7 shows the prediction and update scheme for a cascaded predictor. If both predictors have

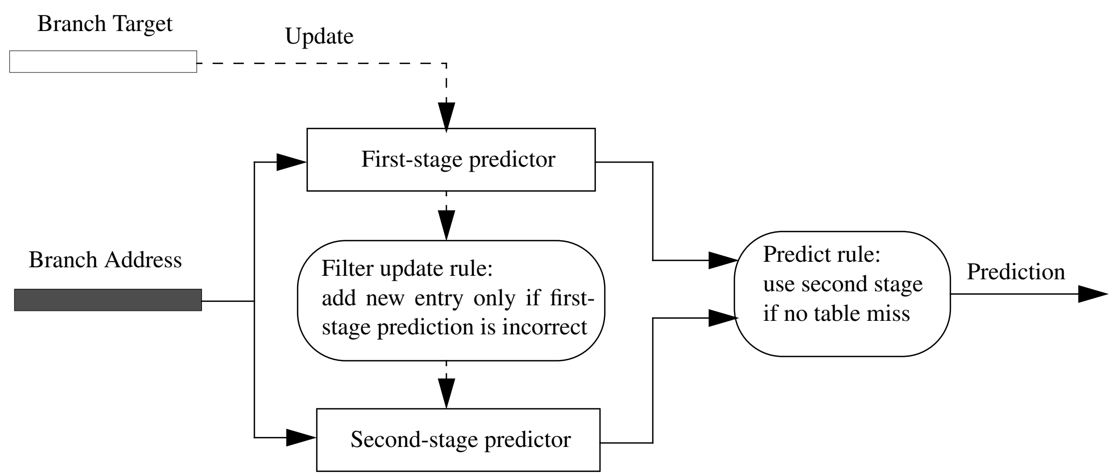

Figure 7. Cascaded predictor scheme

a prediction, the second-stage predictor takes precedence. Therefore, the second-stage predictor's history table must have tagged entries, so that table misses can be detected.

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 4, 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 (but not if it misses). In other words, branches only advance to the second stage if they are provably non-monomorphic. In contrast, a *leaky filter* also allows new second-stage entries on first-stage misses. Thus, the second-stage table may contain entries for monomorphic branches, but only if these weren't found in the BTB.

5.1 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 cold-start 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 miss (but not on a mispredict). To implement this strict filter design, the filter's table must be tagged.

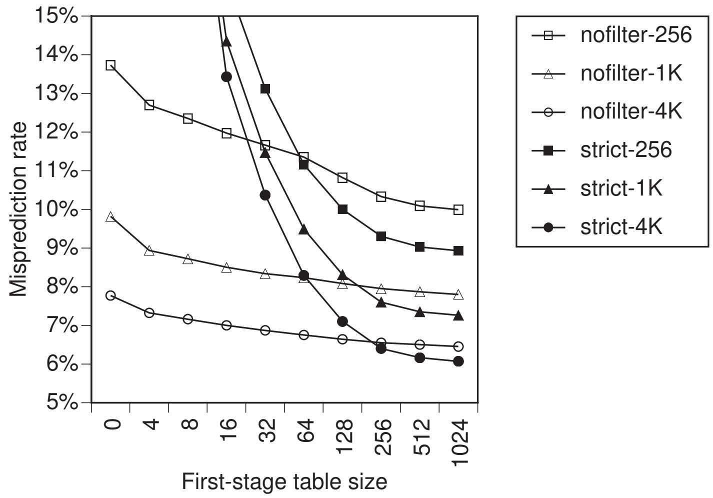

Figure 8. Misprediction rates for a cascaded predictor with strict filter

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 8 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 hybrid 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 polymorphic 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 polymorphic 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 cascaded predictor.

5.2 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 cold-start entries for monomorphic branches will eventually be displaced by entries for polymorphic 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.

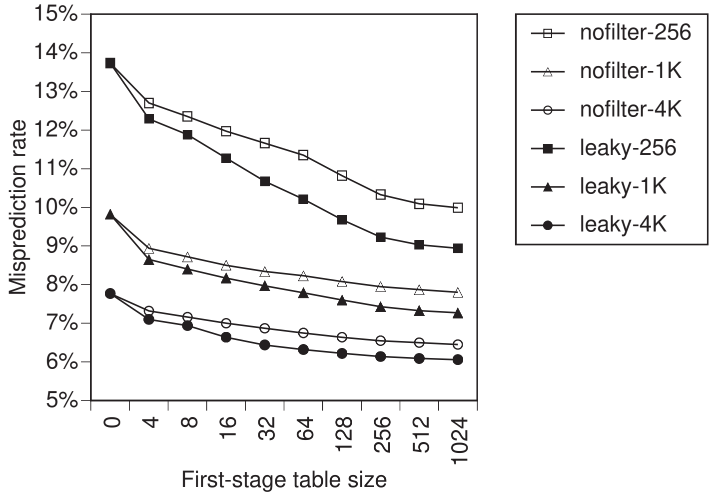

Figure 9. Misprediction rates for a cascaded predictor with leaky filter
Second-stage predictor table size is 256, 1K and 4K entries; both predictor tables are 4-way associative. Also shown is a cascaded predictor without filtering.

Figure 9 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 5 shows the best path length and misprediction rate for cascaded predictors with second stage monopredictors. 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 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 non-cascaded dual-path hybrid predictors with table sizes T, 2T and 4T from [DH98].

The resulting improvements were equally pronounced (Table 6). Again, filtering reduces the misprediction rate and increases the best path length choices for each table size. Whereas non-cascaded 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 mispre-

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 5. 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 [DH98].

Second level table size	hybrid	Best P per filter table size								hybrid	Miss% 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 6. 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 [DH98].

diction 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.

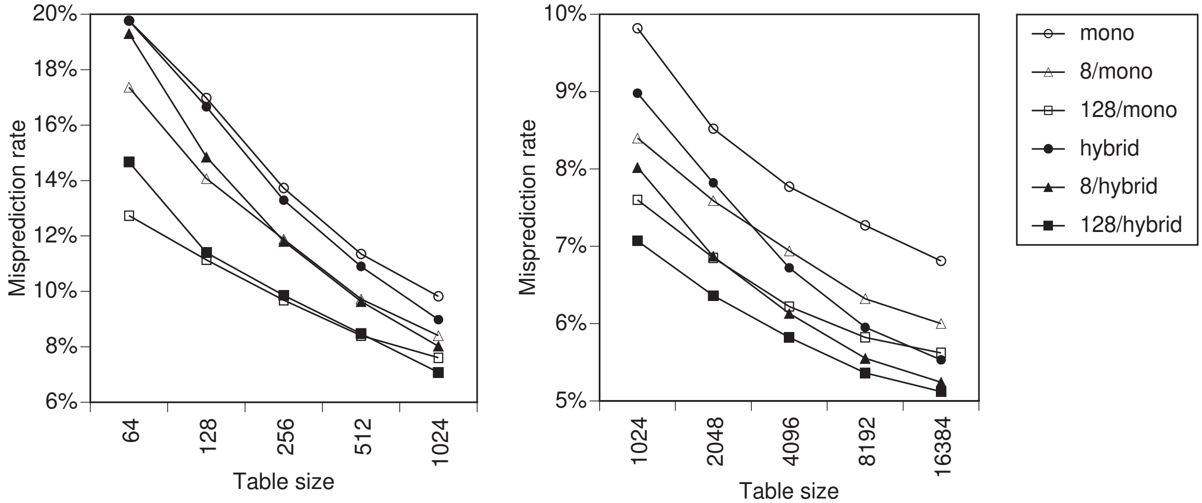

Figure 10. AVG Misprediction rates for mono and hybrid predictors without filters and with 8 and 128-entry filters. We use different scales for small and large table sizes to increase visibility.

Figure 10 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 or lower than that of filtered dual-path hybrid predictors. 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. Mono 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 mono 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 11 shows misprediction rates for the s*elf*, *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 10, which shows that the benefit of filtering is especially pronounced for large programs.

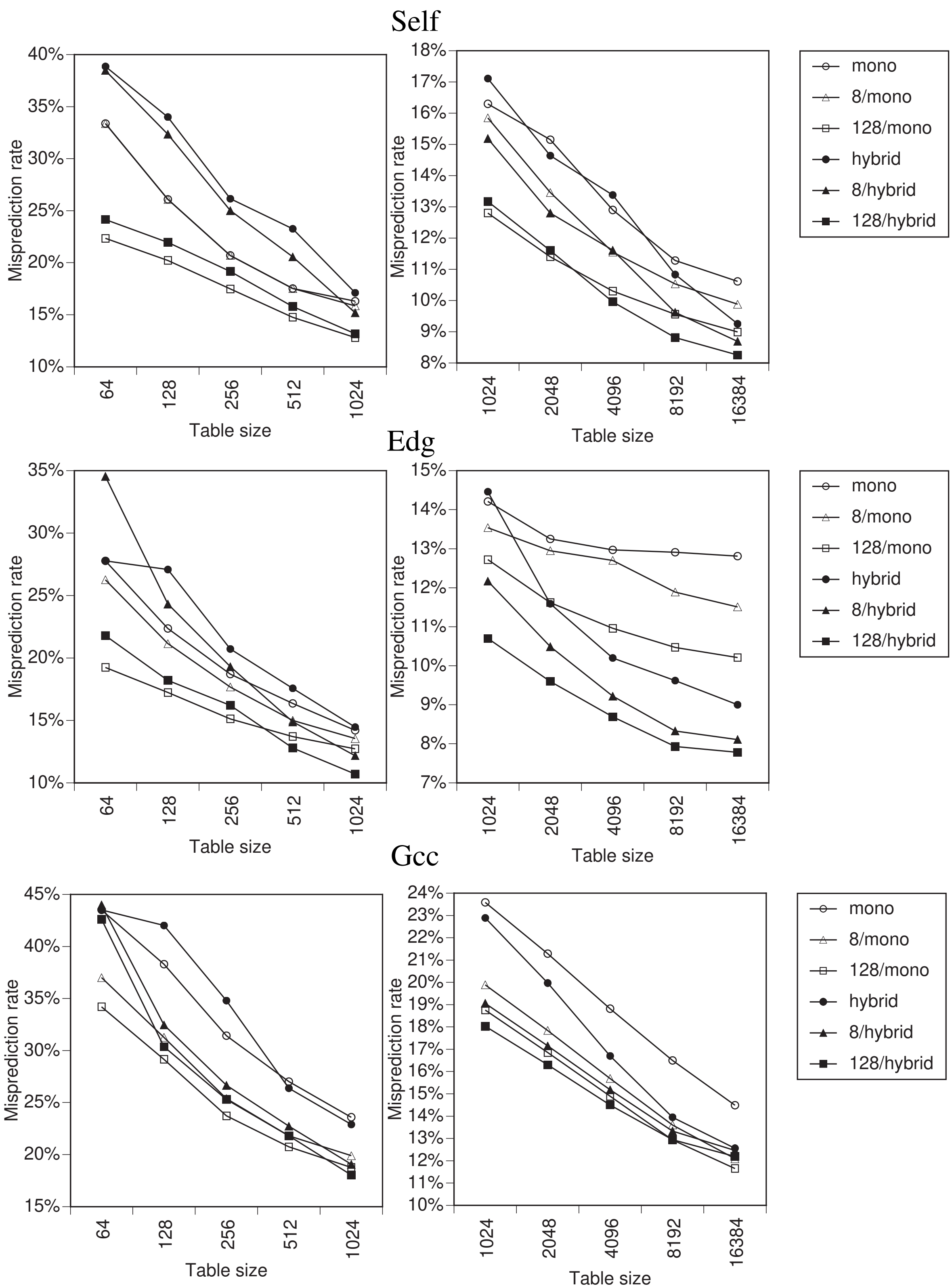

Figure 11. Misprediction rates for mono and hybrid predictors without filters and with 8 and 128-entry filters, for the self, edg and gcc benchmarks. Best mono predictor path lengths are determined seperately for each bencmark. Hybrid predictor path lengths are picked from the AVG path length choices.

6. 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 history table, which is similar to the classifying shared-table hybrid predictor used in this study. 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 benchmarks suite.

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 [DH98] (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 [DH98], 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.

7. Conclusions

Source-based, static branch classification does not appear to significantly improve the prediction of indirect branches in object-oriented or procedural programs. *Opcode-based classification* tunes a global history buffer’s path length separately to branch classes based on the source code origin of a branch (switch statements, virtual function calls, and other). *Arity-based classification* classifies branches according to the number of different targets they branch to. Both schemes led to relatively minor improvements in prediction accuracy, indicating that easily obtainable source-level information is not sufficiently indicative of indirect branch behavior. On the other hand, the two studies did indicate that some branches are much easier to predict than others, so that classification per se might be very successful if it could recognize these differences reliably enough.

A new predictor architecture, the *cascaded branch predictor*, 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 history table is more effectively used. 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 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. Using only a fraction of the resource budget of a 1K-entry second-stage mono predictor, a fully-associative BTB filter with 4 entries reduces the overall misprediction rate from 9.8% to 8.7%. A 64-entry, 4-way associative filter reduces this further to 7.8%, the same performance as a stand-alone predictor with four times as many history table entries. 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. To our knowledge, this improves 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. In principle, this effect should occur for any application of path-based predictors where the dynamic frequency of easily predicted cases is high. In particular, conditional branch prediction or load value prediction could 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.

8. 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.

[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*.

[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. Also published as Technical Report CSE-TR 93-06-06, University of Washington, 1993.

[DH98] Karel Driesen and Urs Hölzle. Accurate Indirect Branch Prediction. To appear in the *ISCA '98 Conference Proceedings*, July 1998 (techreport available at http://www.cs.ucsb.edu/oocsb/papers/TRCS97-19.html)

[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*, pp. 291-300, La Jolla, CA, June 1995. Published as *SIGPLAN Notices 30(6)*, 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

[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,* 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.*