Limits of Indirect Branch Prediction

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

Technical Report TRCS97-10
June 25, 1997

Abstract. Indirect branch prediction is likely to become more important in the future because indirect branches tend to be more frequent in object-oriented programs. With indirect branch prediction misprediction rates of around 25% on current processors, such branches can incur a significant fraction of branch misses even though indirect branches are less frequent than the more predictable conditional branches. We investigate the predictability of indirect branches to determine whether the inferior accuracy of the current indirect branch prediction mechanism (branch target buffers) results from an intrinsic unpredictability of indirect branches or is caused by suboptimal branch prediction hardware. Using programs from the SPECint95 suite as well as a suite of C++ applications, we show that prediction accuracy can exceed 95% on average for these benchmarks, assuming an unlimited hardware budget. This result suggests that better indirect branch prediction hardware can significantly outperform current branch target buffers.

1. Introduction

Indirect branches, which transfer control to an address (recently) stored in 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 the heap, such as virtual function pointers in object-oriented programs written in languages such as C++ and Java. This behavior suggests that such indirect branches may represent a blind spot for branch prediction strategies such as two-level branch prediction, which are driven by dynamic control flow information.

Indirect branches are quite common in widely used benchmark sets like the SPECint95 suite, although they occur less frequently than conditional branches. Indirect branches are much more frequent in object-oriented languages. These languages promote a more polymorphic 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 polymorphic 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.

Current processors improve the performance of indirect branches with a branch target buffer (BTB) that caches the most recent target address for a particular indirect branch and uses this address to predict the outcome of the next execution of this branch. While reasonably useful, BTBs typically have much lower prediction rates than the best predictors for conditional branches. For example, an ideal (infinite) BTB achieves an average prediction hit ratio of only 64% on the SPECint95 benchmarks, whereas the average conditional branch prediction hit ratio can exceed 95% depending on the prediction scheme used.

The goal of this study is to determine whether the inferior accuracy of current indirect branch prediction results from an intrinsic unpredictability of indirect branches or is caused by suboptimal branch prediction mechanisms. In other words, do BTBs already capture most of the predictability of indirect branches, or is it worth exploring new indirect branch prediction hardware schemes?

In order to answer this question, we test a variety of indirect branch prediction schemes inspired by successful conditional branch prediction methods. Our aim is to explore the limits of indirect branch prediction, and thus we assume unlimited hardware resources in this first stage of research, so that our results will not be obscured by implementation artifacts such as conflict misses caused by direct-mapped tables. We test a range of costly, and likely impractical schemes, and combine them to get a best-case prediction rate. The results show that indirect branch prediction can be vastly improved over the current hardware-implemented schemes, approaching a prediction rate of 94% for two-level predictors and over 97% for hybrid predictors.

1.1 Sources of branch prediction misses

Our methodology is geared towards estimating the intrinsic predictability of indirect branches for a given benchmark. Intuitively, a predictor fails to predict a given branch correctly for one of two reasons: either the branch has never jumped to the given target before (a compulsory miss), or the predictor does not capture the behavior of the program accurately. While the former case is unavoidable unless a default static prediction can be used, a number of factors influence the latter:

- The program may have intrinsically unpredictable behavior, so that the only adequate predictor is a simulation of the program itself (a variant of the Halting Problem). One of the goals of this study was to determine the extent of this intrinsic unpredictability.
- The predictor may be blind to those parts of the machine state that influences the branch's behavior. For example, an object-oriented program invoking a method on each element of a polymorphic collection of objects can result in unpredictable target sequences because the types of the objects in the collection (which determine the targets of the call) are invisible to a control-flow driven branch predictor.

  This problem is mitigated by the fact that regularities in the unseen data stream will be expressed into regularities of the branch target sequences. (If this weren't the case, then the information would be useless for branch prediction in the first place.) Therefore, a predictor may capture this regularity through other information such as the history of previous targets (path-based prediction [Nair95]).
- Finally, the predictor may have access to all the relevant machine state influencing the branch, but may still be unable to capture its regularities adequately. For example, a pattern-based two-level branch predictor may have history tables that are too small.

This last case is hardest to analyze because many factors can contribute to the inability to capture regularities of the information stream. Capacity and conflict misses arise from implementation limitations such as relatively small prediction tables or direct-mapped tables. Similarly, path-based prediction may suffer from path interference or insufficient path lengths. Since we aim to reach the intrinsic misprediction rate, i.e., the unpredictability inherent in the program itself, we avoid these sources of inaccuracy by simulating arbitrarily large, fully associative prediction tables and full instruction addresses as table indices or path history elements. In cases where the ideal organization is not obvious (e.g., path lengths) we simulate a range of parameter values that is large enough to ensure that prediction accuracy is not constrained.

2. The benchmarks

We used three benchmark suites to evaluate indirect branch prediction schemes. The C++ benchmark suite consists of C++ applications that range from 8,000 to 50,000 lines of C++ code each (see Table 1).[1] We also measured the programs of the SPECint95 benchmark suite (see Table 2) with the exception of *compress* which executes only 590 branches during a complete run [CGZ94].

All programs except *self*[2] 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 except procedure returns (which were excluded because they can be predicted easily with a return address stack [KE91]). All programs were run to completion or until six million indirect branches were executed. 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.

Name	Description	lines of code	# of indir. branches	% virtual	active branch sites			
					90%	95%	99%	100%
eqn	typesetting program for equations	8,300	296,425	34%	17	23	58	114
idl	SunSoft's IDL compiler (version 1.3)	13,900	1,883,641	93%	6	15	70	543
ixx	IDL parser, part of the Fresco X11R6 library	11,600	212,035	47%	31	46	91	203
lcom	compiler for hardware description language	14,100	1,737,751	63%	8	17	87	328
porky	SUIF 1.0 scalar optimizer	22,900	5,392,890	71%	35	51	89	285
troff	GNU groff version 1.09	19,200	1,110,592	74%	19	32	61	161
self	Self-93 system	50,000[a]	6,000,000	80%	239	381	756	2658

Table 1. C++ Benchmarks

[a] estimated (the Self-93 system is 120,000 lines but not all of them are exercised during the benchmark run)

For each benchmark, the tables list the number of indirect branches executed and the percentage of these branches that corresponds to virtual function calls. For example, only 34% of the indirect branches in *eqn* are due to virtual function calls; the rest represent indirect calls through function pointers, indirect branches of `switch` statements, etc. In addition, the tables list the number of

1 We are currently expanding our benchmark suite to include additional applications; the final paper will likely include at least two more applications consisting of over 100,000 lines each.

2 *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.

Name	# of indir. branches	% virtual	active branch sites			
			90%	95%	99%	100%
gcc	864,838	0%	38	56	95	166
go	549,656	0%	2	2	5	14
ijpeg	32,975	0%	3	5	7	60
m88ksim	300,000	0%	3	4	5	17
perl	300,000	0%	6	6	7	27
vortex	3,000,000	0%	6	6	9	37
xlisp	6,000,000	0%	3	3	4	13

Table 2. SPECint95 benchmarks

indirect branch sites responsible for 90%, 95%, 99%, and 100% of the indirect branches. For example, only 2 different branch sites execute 95% of the dynamic indirect branches in *go*.

99% of the indirect branches in the C++ and SPEC programs are executed from less than 100 indirect branch sites, except for *self* which contains a much larger number of active indirect branches (756). Several factors may contribute to this difference, including *self*'s larger size its programming style which is more object-oriented than other C++ programs, many of which make only sparing use of the object-oriented features of C++.

The SPECint95 programs are even more dominated by very few indirect branches, with less than ten interesting branches for all programs except *gcc*. Because there are so few distinct indirect branches in these programs, they are much more sensitive to variations in indirect branch prediction schemes since a change in the prediction accuracy of a single indirect branch may significantly affect the overall prediction rate. We have included the SPECint95 programs mostly for comparison purposes since we do not believe that they are very meaningful when evaluating indirect branch predictors. In effect, the SPEC benchmarks are microbenchmarks as far as indirect branch prediction is concerned. In our evaluation of indirect branch prediction schemes we will therefore focus on the behavior of the C++ programs.

3. Prediction schemes

3.1 Branch Target Buffer

Current processors use a branch target buffer (BTB) to predict indirect branches, and thus we use BTB as the baseline against which other predictors are compared. When an indirect branch is fetched, the predictor uses the branch address as a key into a table (the BTB) which stores the last target address of the branch. (Alternatively, the buffer can store one or more instructions from the predicted target address to allow branch folding).

Since we are interested in the limits of predictability, we simulate BTBs with unlimited table size and perfect conflict resolution (full associativity). However, BTBs with realistic organizations can come quite close to this ideal BTB: a recent study [DH96] showed that there is no improvement in prediction rates beyond a BTB size of 256 entries for a suite of similar C++ benchmarks. This result is not surprising given the relatively small number of relevant indirect branches in these programs.

We consider two variants: "BTB" is a standard BTB which updates its target address after each branch execution. "BTB-2bc" is a BTB with two-bit counters which updates its target only after two consecutive mispredictions. In conditional branch predictors, the latter strategy is implemented with a two-bit saturating counter (2bc), hence the name. For an indirect branch, one bit suffices to indicate whether the entry had a miss the last time it was consulted. We chose to retain the name "two-bit counter" because its functionality is analogous to two-bit counters for conditional branch prediction.

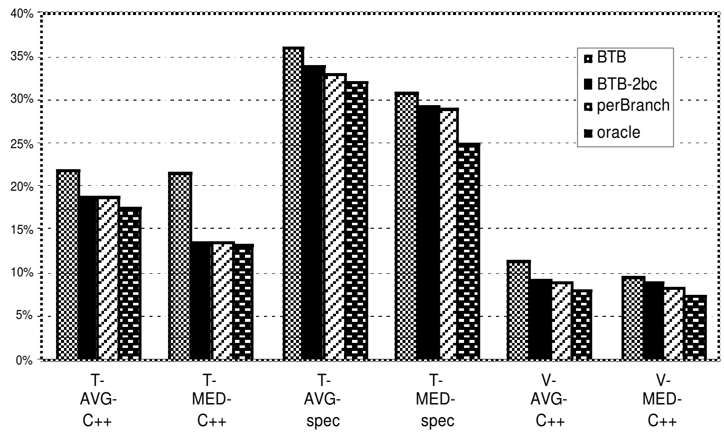


Figure 1. BTB misprediction ratios

T-AVG and T-MED refer to the average or median of the original C++ or SPEC benchmarks. V-AVG/MED refer to the indirect branches within the C++ benchmarks that correspond to virtual function calls.

Figure 1 shows the misprediction ratios for BTB and BTB-2bc. The latter performs better in virtually all cases, most likely because it works better for polymorphic branches that occasionally switch their target but are dominated by one most frequent target, a situation that is quite frequent in object-oriented programs [AH96], [D+96]. But even with two-bit counters, BTB accuracy is quite poor, ranging from median misprediction ratios of 13% in C++ programs to 28% for SPECint95. (We show both medians and means because the latter are sometimes skewed by outliers.) BTBs appear to predict virtual function calls better than other indirect branches in C++ programs, as shown by the rightmost two groups in the graph.

Although two-bit counters improve prediction accuracy overall, they do not improve accuracy for every indirect branch, as is shown by the *perBranch* and *oracle* data. The *perBranch* predictor statically selects either BTB or BTB-2bc for every indirect branch [LS84][CHP94], and *oracle* dynamically selects between BTB and BTB-2bc assuming a perfect hybrid predictor. That is, for every indirect branch execution *oracle* selects the correct predictor (if there is one). For the SPEC benchmarks, *perBranch* and *oracle* show improvements over both BTB and BTB-2bc, indicating that neither BTB variant is strictly superior to the other.

3.2 Two-level prediction for indirect branch paths

With BTBs, the table storing the predicted target is accessed using the current branch address as the index. In two-level branch predictors, the index is a history pattern based on previously executed branches [YP91], [YP93]. The goal of two-level branch prediction is to map branch execution patterns to branch targets, allowing the prediction to use past behavior for better prediction. Most of the variations in two-level predictors come from different answers to two basic questions, which we treat in the following two sections (see Figure 2).

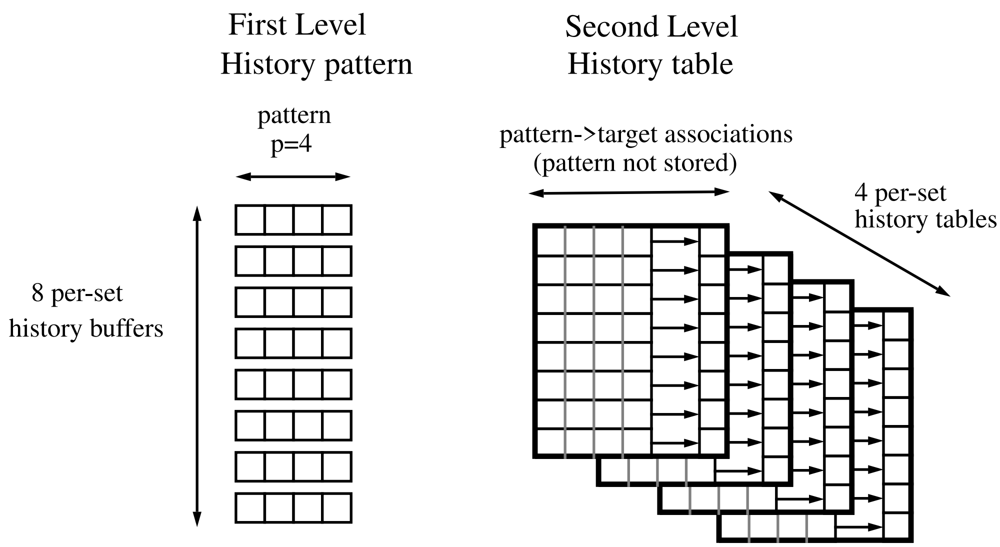


Figure 2. Two-level branch predictor

3.2.1 First level: what comprises the history pattern?

For conditional branches, a branch history of length *p* consists of the taken/not-taken bits of the *p* most recently executed branches of a particular category [YP93]. Categories can be defined in several ways, leading to different history patterns. One one end of the spectrum of choices, a *global history* (correlation branch prediction) uses a single history, and all branches are predicted using the outcome pattern of the *p* most recently executed branches. In contrast, with a *per-address history* each branch keeps its own history of its *p* previous invocations, so that branches do not influence each other's prediction. Finally, *per-set history* prediction forms a compromise by using a separate history for each set of branches, where a set may be determined by the branch opcode, a compiler-assigned branch class, or a particular address range. Thus, with per-set histories a branch is influenced only by the branches in the same set.

In contrast to conditional branches, indirect branches are unconditional on most architectures, i.e., they are always taken. Thus, keeping a history of taken/not-taken bits would be ineffective. Instead, the history must consist of previous target addresses (or bits thereof). Such a path-based history could also be used to predict conditional branches, but since taken/not-taken bits summarize the target addresses of a conditional branch more succinctly, conditional branch predictors usually do not employ target address histories (but see [Nair95]).

In this study, we assume that indirect branch histories consist of full 32-bit target addresses of past branches. We investigate global histories, per-branch histories, and per-set histories where the set is determined by address bits of the branch. Bits are numbered from lowest to highest significance, so bit 0 is the LSB. All branches with the same values in bits *s*..31 fall into a set, i.e., a set contains all branches in a memory region of size 2^s bytes. With this parametrization, a global history corresponds to *s*=31 and per-branch histories correspond to *s*=0[1].

[1] On most architectures, s=1 and s=2 also imply per-branch histories since branch instructions are four bytes long.

3.2.2 Second level: how is the history pattern mapped to a target?

Given a history pattern, a two-level predictor uses it as an index into a history table that stores predicted targets. Again, branches may or may not share the same table, allowing for a global history table where the branch address is not taken into account, or per-set or per-branch history tables. As before we use the branch address to determine set membership, mapping all indirect branches with identical address bits h..31 into the same set.

3.2.3 Path length

The path length p of a two-level predictor determines the number of branch targets in the history pattern. Normally, p is limited by the maximum size of the history table(s) since the table size is 2^p. We operate with unlimited tables in this study and thus are free to explore a wide range of path lengths. In theory, longer paths are better since a predictor cannot capture regularities in branch behavior with a period longer than p. Shorter paths, however, have the advantage that they adapt more quickly to new phases in the branch behavior. A long path captures more regularities, but the "warm-up"-time for long patterns can prevent the predictor from taking advantage of this knowledge before the program behavior changes again. We studied path lengths up to 18bits in order to investigate both trends and see where they combine for the best prediction rate.

parameter	meaning	range	discussed in ...
s	history sharing	0..31	section 4.1
h	history table sharing	0..31	section 4.2
p	history (path) length	0..18	section 4.3

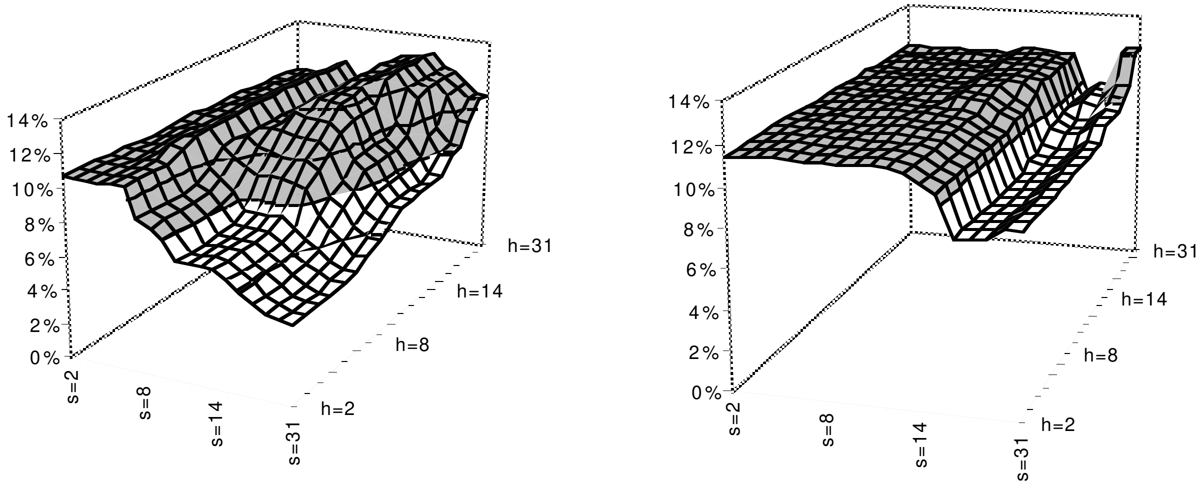
Table 3. Summary of two-level indirect branch prediction parameters

In summary, we simulate the full range of two-level indirect branch prediction parameters (see Table 3). In addition, since we observed that adding two-bit counters reduces the overall misprediction rate of every tested indirect branch predictor (with very few exceptions), all predictors include two-bit counters. We will discuss the effect of discarding the counters in section 5.3.

4. Results

To find the best indirect branch predictors, we first ran an exhaustive simulation using all combinations of s, h, and p for the C++ suite (excluding *self*), resulting in a total of 4,800[1] simulations which took several weeks on multiple UltraSPARC workstations. These runs gave us an overview of the interesting regions in the parameter space of two-level predictors. We established the limits beyond which the misprediction rate stops improving and obtained some preliminary values for path length, history pattern sharing, and history table sharing (p=8, s=31, h=0). The misprediction rate varies smoothly in the three dimensions of the parameter space in the vicinity of the global minimum (see Figure 3), allowing us to minimize one parameter at a time for the full benchmark suite while still finding the global minimum. We examined all C++ benchmarks (except *self*), and although there are local minima in the miss rate surface, the global minimum is surrounded by a

[1] We omitted parameter values that are provably identical to others. In particular, the values 0 and 1 as well as 22..30 for s and h are redundant since no benchmark executable except *self* exceeded 2^{21} bytes in size, and the difference between s=22 and s=31 was very small for *self*.



porky (p=8) troff (p=8)

Figure 3. Smoothness of parameter space

relatively smooth area in all directions. Figure 3 shows the miss rate surfaces of two representative samples (*porky*'s virtual branches and *troff*'s indirect branches).

4.1 First level: History Sharing

We first determined the impact of path history sharing. Figure 4 shows the average misprediction rates for different values of *s*, the parameter that determines the size of the partial address set an indirect branch belongs to. (The appendix contains data for each individual benchmark.) The path length *p* remained at 8, and each branch had its own history table (*h*=2).

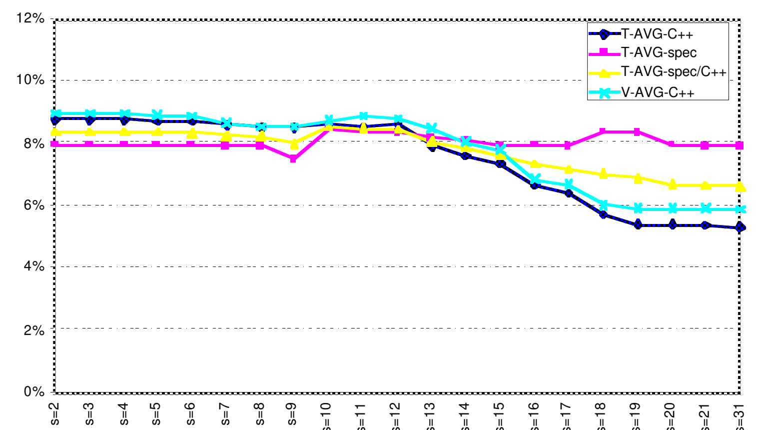


Figure 4. Influence of history sharing

two-level predictor for pure indirect traces, path length p=8, per-address history table (h=0), two-bit counters

In general, a global history outperforms local histories. The average misprediction rate for all benchmarks declines from 8.4% (per-address paths) down to 6.6% (global path). The C++ programs benefit most from sharing paths, with misprediction rates falling from 8.8% to 5.4%. The SPEC benchmarks show a curious dip for s=9 (i.e., if branches within 512-byte code regions share paths). On closer observation, the dip is due to *xlisp* where only three indirect branches are responsible for 95% of the dynamic indirect branch executions. For *xlisp*, moving from s=8 to s=9 reduces mispredictions by a factor of three. Similarly, at s=10 *go*'s misprediction ratio jumps from 26% to

33% (*go* is dominated by two indirect branches). Similarly, the bump at s=18 and s=19 results from a transition in the misprediction rate of two benchmarks dominated by 4 and 6 indirect branches. This result illustrates the difficulty of spotting trends based on averages of "microbenchmarks" (for the purpose of indirect branch prediction). SPEC's misprediction rate with a global history is still nearly minimal, with 7.9% vs. 7.5% in the dip.

We conclude that a global history outperforms per-set histories based on address ranges for indirect branch prediction. This result indicates a substantial correlation between different branches (i.e., inter-branch correlation) in our benchmark suite, a correlation not limited by code distance. If all that mattered to an indirect branch was its own behavior, shared paths would give higher misprediction rates than per-address paths, since a branch's own history would be crowded out by the irrelevant histories of other indirect branches.

Incidentally, a shared path is the fastest to implement in silicon, since the first level of the predictor does not require a table lookup.

4.2 Second level: History Table Sharing

A predictor accesses its prediction table using a combination of the past history and the branch address as an index. In this section we explore the importance of including the branch address in the index by varying the number of bits of the branch address to include. In the experiment below, we include the bits [*h*..31] of the branch address and measure all meaningful values of *h*; recall that *h*=2 implies per-branch history tables and *h*=31 implies a single shared history table. Figure 5 shows the average misprediction rates for different values of *h*; the path history length *p* remains at 8, and one path is shared globally between indirect branches (*s*=31, as found in the previous section).

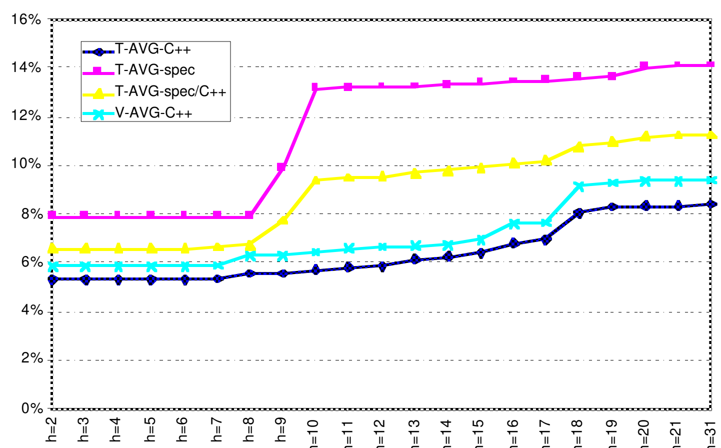


Figure 5. Influence of history table sharing

two-level predictor for pure indirect traces, path length p=8, global shared path (s=31), two-bit counters

The average misprediction rate for all benchmarks increases from 6.6% for per-address history tables to 11.3% for a globally shared history table. Looking at the individual benchmark suites, the misprediction rate of the C++ programs increases from 5.4% to 8.5%, and that of the SPEC benchmarks rises from 7.9% to 14.2%. The SPEC average shows a very sharp increase at h=9 and h=10, resulting from jumps in the misprediction rates of *go* and *xlisp* (see the appendix for details). As discussed before, these programs are dominated by less than a handful of indirect branches.

These results show that the address of the branch matters, at least for a branch predictor which sees a pure indirect branch trace. If the history table is indexed by just the global path history (ignoring the branch address), as with a global table, indirect branches that reside on different control flow paths after an indirect branch will share and disrupt each others target predictions. In conditional branch prediction, where the history pattern and the prediction consist of taken/not taken bits, this effect can sometimes give a lower misprediction rate (positive interference). With indirect branches, positive interference is very unlikely, and often impossible since two indirect branches or calls may not even share any common targets. With per-address history tables, each branch on a different control flow path after an indirect branch builds its own associations between the global history patterns that reach it and its effective targets, thus improving prediction accuracy.

Now that we have chosen values for path sharing (global) and history table sharing (per-branch), we will vary the path length in the next section.

4.3 Path length

Figure 6 shows the impact of the history path length on the misprediction rate for all path lengths from 0 to 18. For per-address history tables, a path length of 0 reduces the two-level predictor to a BTB predictor, since every indirect branch has exactly one target stored in its single-entry table.

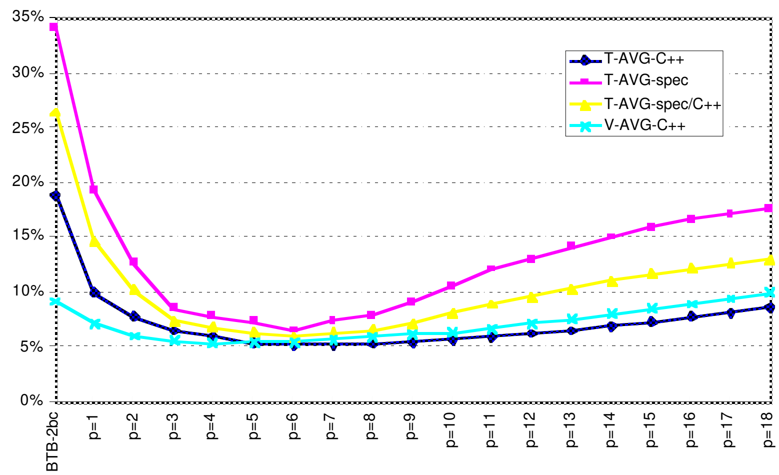


Figure 6. Mispredicition rates as a function of path length

two-level predictor for pure indirect traces, path length p=8, global shared path (s=31), per-address history tables (h=2), two-bit counters

The average misprediction rate drops dramatically from 26.5% for p=0 (BTB) to 7.5% for p=3 and then gently slopes down to a minimum of 5.9% at path length 6. Then the misprediction rate starts to rise again and keeps on rising for larger path lengths up to the limit of our testing range at p=18. All benchmark suites follow this pattern, although the SPECint95 benchmarks show uniformly higher misprediction rates than the C++ programs.

This result indicates that most regularities in the indirect branch traces have a relatively short period. In other words, a predictable indirect branch execution is usually correlated with the execution of a branch less than three to six steps before it. Increasing the path length captures some longer term correlations, but at path length six cold-start misses begin to negate the advantage of a longer history. At this point, adding an extra branch target to the path may still allow longer-term correlations to be exploited, but on the other hand it will take the branch predictor one step longer to learn a full new pattern association for every branch that changes its behavior due to a phase transition in the program.

Apparently, branches representing virtual function calls have shorter average correlation lengths since their misprediction rate reaches a minimum of 5.4% at path length 4. Thus, if we used a predictor with p=4 for virtual branches, and p=6 for other indirect branches, the total misprediction rate would be lower since the combination could avoid the compromise between the two differently behaved branch classes. We test this hypothesis in the next section.

5. Variations

In this section we explore further variations, including hybrid predictors that combine multiple path lengths, predictors that include conditional branches in the history path, and predictors without two-bit counters.

5.1 Multiple Path Length Hybrid Prediction

Chen et al. [CCM96] use a branch predictor with all path lengths from 1 to p to estimate the limit of conditional branch prediction. With Prediction by Partial Matching (PPM), a conditional branch resolution pattern with length p is first looked up in full. If it is not in the history table, the pattern is successively shortened is shortened by dropping the oldest branch and looked up again until a match is found (with default values for path length 0). This scheme avoids cold start misses at the cost of using twice the number of table entries of a standard two-level predictor with path length p.

We can mimic this technique by running several path length predictors in parallel and choosing for each branch execution the longest predictor with a match. But instead of limiting ourselves to a particular predictor selector algorithm (sequential search on history pattern inclusion), we go further and stipulate an oracle that chooses the predictor that correctly predicts the branch target. The prediction rate of this *oracle* predictor is an upper bound on the performance of any hybrid predictor that combines predictors with multiple path lengths. To inject a little bit more realism, we also stipulate a *perBranch* predictor which statically (at compile time) chooses one predictor path length for each branch [LS84], [CHP94].

We simulated all possible hybrid predictors consisting of combinations of predictors with path length p=2, 4, 6, and 8. We also added a BTB (p=0) as a component to the largest hybrid predictor. Figure 7 shows the misprediction rate of the best hybrid predictors with two, three and four components. The lower *perBranch* misprediction rate shows that some branch sites have intrinsically shorter correlation path lengths, while the *oracle* misprediction rate shows that even more branch sites benefit from shorter path lengths only during certain program phases. Most of the improvement for *perBranch* predictors is reached with a p=2,8 two-component predictor (5.2% vs. 5.9% for p=6). Adding p=0,4,6 does not decrease the miss rate much further (4.9%). *oracle* performs much better; the best meta-predictor (*oracle* with p=0,2,4,6,8) reaches an astounding 2.3%. Clearly, phase transitions for individual branch sites are an important remaining source of misprediction, but dynamic meta-prediction can remove much of the associated warm-up cost.

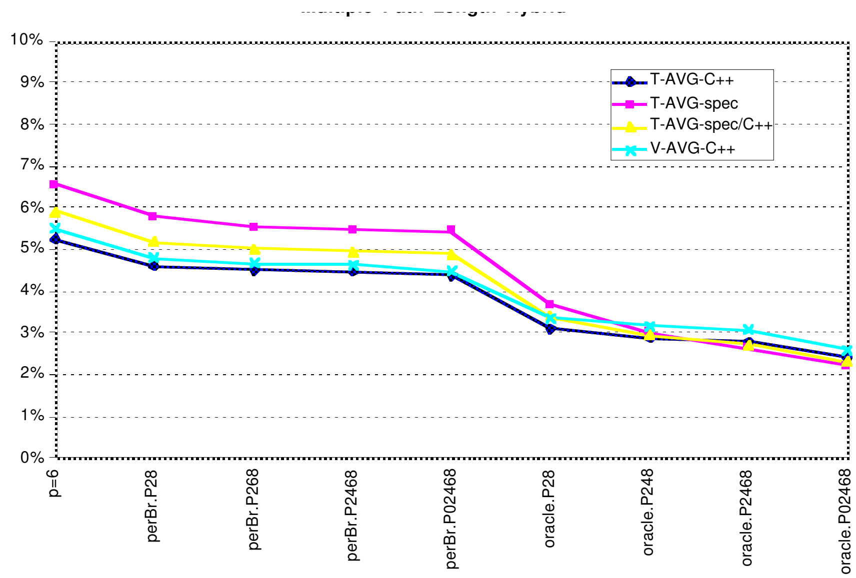


Figure 7. Misprediction rate limits for multiple path length predictors

two-level predictor for pure indirect traces, path length p=8, global shared path (s=31), per-address history tables (h=2), two-bit counters

5.2 Mixed conditional/indirect traces

Together, conditional and indirect branches determine the execution path leading to an indirect branch. So far, we have used only histories of indirect branches; a history containing both conditional and indirect branches may be even better correlated with the next indirect branch target. In this section we experiment with adding conditional branches to the trace.

Since the full traces would have exceeded our disk capacity, we used traces that included the 32 preceding conditional branches (encoded as target addresses) for every indirect branch invocation. For those benchmarks where the full trace was available we verified that the results were not substantially different from the results we present here.

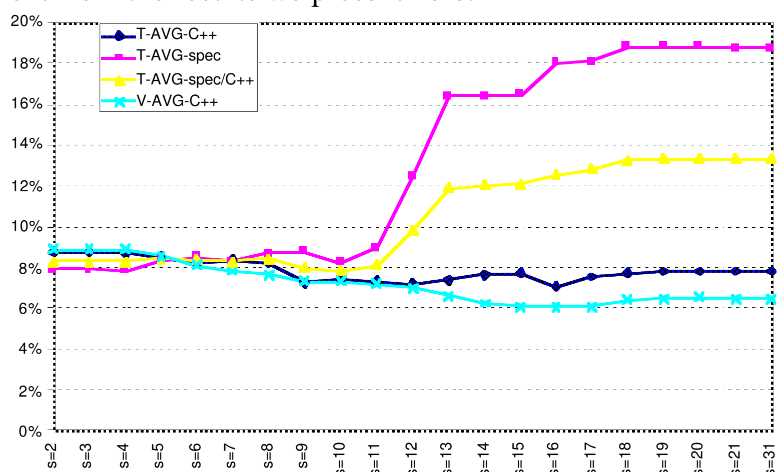


Figure 8. Influence of history sharing for conditional/indirect path history

two-level predictor for pure indirect traces, path length p=8, global shared path (s=31), per-address history tables (h=2), two-bit counters

Figure 8 shows the average misprediction rate for different values of *s*. The path length *p* remained at 8, and each branch has its own history table (*h*=2). For per-address history patterns (*s*=2), the misprediction rates are identical to that of the predictor running on a pure indirect trace: the conditional branches are filtered out of the pattern because their address differs from the indirect branch. As *h* increases, more neighboring conditional and indirect branches contribute to a branch's history pattern, the misprediction rate of the SPEC benchmarks increases even though the rate of the pure indirect trace decreases over this parameter range (see Figure 4). In contrast, the misprediction rate of the C++ suite decreases to a minimum of 7.1% at s=16. This rate is still higher than that of an identical predictor running on a pure indirect trace (6.7%). The average of SPEC and C++ reaches minimum at *s*=10 with a misprediction rate of 7.1%, which is lower than the 8.5% of an identical predictor for pure indirect traces, but higher than the 6.6% of a globally shared history.

We believe the inclusion of conditional branch targets in the history pattern does not pay off because they crowd out indirect branch targets which correlate well with each other. The SPEC suite suffers more from conditional branch inclusion because the density of indirect branches is lower than in the C++ suite, so that even moderate amounts of sharing (*s*=14) allow conditional branches to push recently executed indirect branches out of the history pattern.

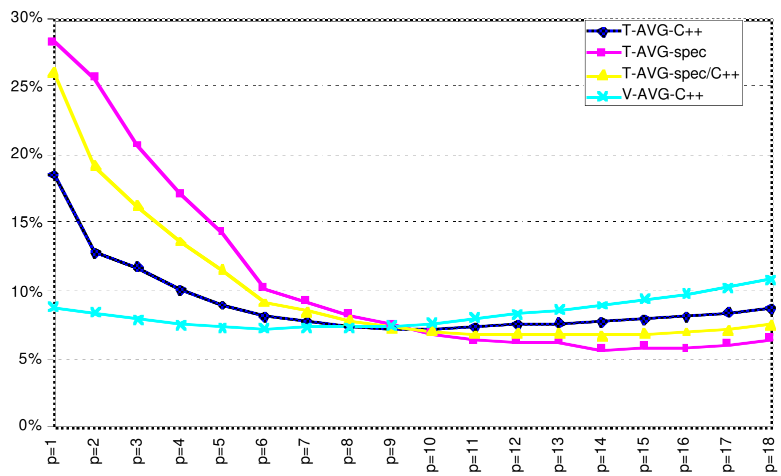


Figure 9. Misprediction rates as a function of path length for conditional/indirect history

two-level predictor for pure indirect traces, path length p=8, partially shared path (s=10), per-address history tables (h=2), two-bit counters

Figure 9 shows misprediction rates as a function of path length for 32-prefix conditional/indirect mixed traces[1]. Comparing the graph to Figure 6, the minimum misprediction rate attained (6.8%) is higher than that for pure indirect traces (5.9%), and this minimum needs a longer path length (p=14 instead of p=6).

We conclude that while conditional branch information may be useful for indirect branch prediction, it appears to be far less relevant than indirect branch information. It may still be possible to use conditional branch information in a less disruptive way, e.g., in a multiple-stream hybrid predictor. We plan to investigate this question in the future.

[1] We do not show the graph for history table sharing since it leads to the same conclusion as for pure indirect traces: per-address history tables perform best.

5.3 Dropping the two-bit counter rule

All predictors presented so far use the two-bit counter rule, i.e., update their predictions only after two consecutive misses. We also measured all configurations without two-bit counters. The resulting graphs look almost identical, except that all the misprediction averages increase by between 0.5% and 1% across the board. Two-bit counters filter out one-time misses that break the regularity seen so far; it appears that it is always beneficial to filter out such events. In other words, the patterns currently stored should only updated by new patterns, and a single event is not a good enough indication of the start of a new pattern (others have found similar effects [SLM95], [J+96]). It may be interesting to look at alternative rules such as predictors that only change an association after three consecutive misses, or only if the target of the second miss was identical to the first one.

6. Related work

Lee and Smith [LS84] describe several forms of BTBs, including a *perBranch* variant. Jacobson et al. [J+96] study efficient ways to implement path-based history schemes and observe that BTB hit rates increase substantially when using a global path history. Their Correlated Task Target Buffer (CTTB) reached misprediction rates of 18% and 15% for *gcc* and *xlisp* with path length 7; our study found misprediction rates of 12% and 1.5% for p=7. The different results can be explained by several factors: different benchmark version (SPEC92 vs. SPEC95), inputs, and radically different architectures (e.g., the multiscalar processor's history information will likely omit some branches in the immediate past). Finally, Jacobson et al. include conditional branches in the path histories, which is probably responsible for the difference in *xlisp* (in our study, *xlisp*'s misprediction rate rises to 12.7% when including conditional branches).

Chang et al. [CHP97] propose an indirect branch prediction scheme realizable in hardware and simulate the resulting speedups of selected SPECint95 programs for a superscalar processor. The study explores only a narrow range of prediction schemes but still achieved significant improvements, reducing the misprediction rate of a BTB-2bc by half for *gcc* and *perl* to 30.9% and 30.4% with a Pattern History Tagless Target Cache with configuration gshare(9). This predictor XORs a global 9-bit history of taken-non taken bits from conditional branches with the branch address, and uses the result as a key into a globally shared, direct-mapped 512-entry history table. In comparison, the best non-hybrid predictor in our study reached a misprediction ratio of 11.6% for *gcc* and 0.5% for *perl*, indicating that substantial improvements may be possible. The technique we tested that resembles most the predictor proposed by Chang et al. (in spirit, not in implementation cost) is a two-level predictor with a global path of length p=9 and per-address history tables which obtained a misprediction rate of 20.9% running on a mixed conditional/indirect trace. This is better than the hardware-constrained predictor, as expected, since our predictor ignores pattern interference and table conflict or capacity misses. These comparisons should be regarded with caution, however, since the two experiments differed in architectures (HPS vs. SPARC), compilers, and benchmark inputs.[1] Furthermore, as observed before, we believe that the SPECint95 benchmarks are ill suited for indirect branch studies since their behavior is dominated by very few branches.

Emer and Gloy [EG97] describe several single-level indirect branch predictors based on combinations of the values of PC, SP, register number, and target address, and evaluate their performance on

[1] We were unable to obtain the benchmark inputs used by Chang et al.

a subset of the SPECint95 programs. For these programs, the best predictor shown achieved a misprediction ratio of 30%, although the authors allude to a better predictor that achieves 15%. The latter predictor comes surprisingly close to the misprediction rate of a two-level predictor with p=6 (10.5%), but again we wish to caution that comparisons based on the SPEC programs may not be meaningful.

Calder and Grunwald proposed the two-bit counter update rule for BTB target addresses [CG94] and showed that it improved the prediction rate of a suite of C++ programs. Chen et al. [CCM96] propose PPM prediction for conditional branch prediction and show that a PPM predictor performs better than a two-level predictor for a similar hardware budget. Chen et al. also establish a correlation between the compression ratio of branch traces and branch miss prediction rate, and suggest that compression ratios are good indicators of the predictability of a trace. The *gzip* compression rates of our pure indirect traces for all benchmarks had a correlation of 0.9 with the indirect branch prediction rates of a two-level predictor with path length p=8.; the correlation of a BTB with either of these was 0.4.

Nair [Nair95] introduced path-based branch correlation for conditional branches and showed that a path-based predictor with two-bit partial addresses attained prediction rates similar to a pattern-based predictor with taken/not taken bits (for similar hardware budgets).

Many predictors in this study were inspired by conditional branch predictors. We refer to [USS97] for a recent general overview, to [YP93] for the classification of two-level predictors used in this paper, and [ECP96] for recent hybrid prediction results.

7. Conclusions and future work

We have explored the limits of predictability of indirect branches. The results show that there is a large potential for improvement in indirect branch prediction. The best non-hybrid predictor, two-level prediction with a global shared path of length 6 and per-address history tables, attains a misprediction rate of 5.9% on average for a suite of programs consisting of large C++ applications and SPECint95, a four-fold improvement over an ideal BTB with two-bit counter update rule which reaches 26%.

The substantial differences in behavior between C++ programs and SPECint95, as well as the extremely small number of relevant branches in SPECint95 (except for *gcc*) suggest that the SPEC programs should not be used as the primary benchmark suite for evaluating indirect branch prediction mechanisms.

We also show that hybrid predictors allow further improvements. A combination of two-level predictors with different path lengths appears promising; this form of hybrid prediction avoids the “cold start” overhead of long paths while allowing long-range correlations to be exploited. In our test case (2,4,6 and 8 path length components), “perBranch” static meta-prediction already reduces the misprediction rate from 5.9% to 4.9%. The best meta-predictor measured, an oracle that chooses the correct component predictor (if any) reaches an astounding 2.3% misprediction rate.

We have also explored some areas of the design space of predictors that were sub-optimal. In particular, we found that:

- including conditional branch targets in the history pattern lowers prediction performance by pushing the more relevant indirect branch information out of the history buffer; and
- updating targets on every miss lowers the prediction rate in virtually every case, compared to two-bit counters.

7.1 Future work

We plan to further explore the indirect branch prediction scheme design space in three directions:

- Realistic predictors may be derived by constraining the most promising prediction schemes while preserving as much of their predictive power as possible.
- Better combinations of existing and new component predictors for hybrid predictors (including exploring ways of utilizing conditional branch information) may further reduce mispredictions.
- A predictor could predict not only the target of a branch but also the address of the next indirect branch to be executed. This disambiguates branches that lie on different conditional branch control flow paths but share the same indirect branch path, and allows a predictor to run, in principle, arbitrarily far ahead of execution.

Acknowledgments. The authors would like to thank Mike Smith and Raimondas Lencevicius for their comments on an early version of this paper. This work was supported in part by National Science Foundation CAREER grant CCR-9624458, an IBM Faculty Development Award, and by Sun Microsystems.

8. References

[AH96] Gerald Aigner and Urs Hölzle. Eliminating Virtual Function Calls in C++ Programs. *ECOOP '96 Proceedings*, Springer Verlag, July 1996.

[CGZ94] Brad Calder, Dirk Grunwald, and Benjamin Zorn. *Quantifying Behavioral Differences Between C and C++ Programs*. Technical Report CU-CS-698-94, University of Colorado, Boulder, January 1994.

[CG94] Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in C++ programs. In *21st Symposium on Principles of Programming Languages*, pages 397-408, 1994.

[CHP94] Po-Yung Chang, Eric Hao, Yale N. Patt. Branch classification: A new mechanism for improving branch predictor performance. *MICRO '27 Proceedings*, November 1994.

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

[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.

[D+96] Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov, and Craig Chambers. Vortex: An Optimizing Compiler for Object-Oriented Languages. *Proceedings of OOPSLA '96*, San Jose, CA, October, 1996.

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

[EG97] Joel Emer and Nikolas Gloy. A language for describing predictors and its application to automatic synthesis. To appear in the *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*.

[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.

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

[SLM95] Stuart Sechrest, Chich-Chieh Lee, and Trevor Mudge. The role of adaptivity in two-level adaptive branch prediction. *Proceedings of MICRO-29*, November 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,* 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.*

Appendix A Detailed Data

Figure 10 shows the misprediction rates for the individual C++ and SPECint95 benchmarks.

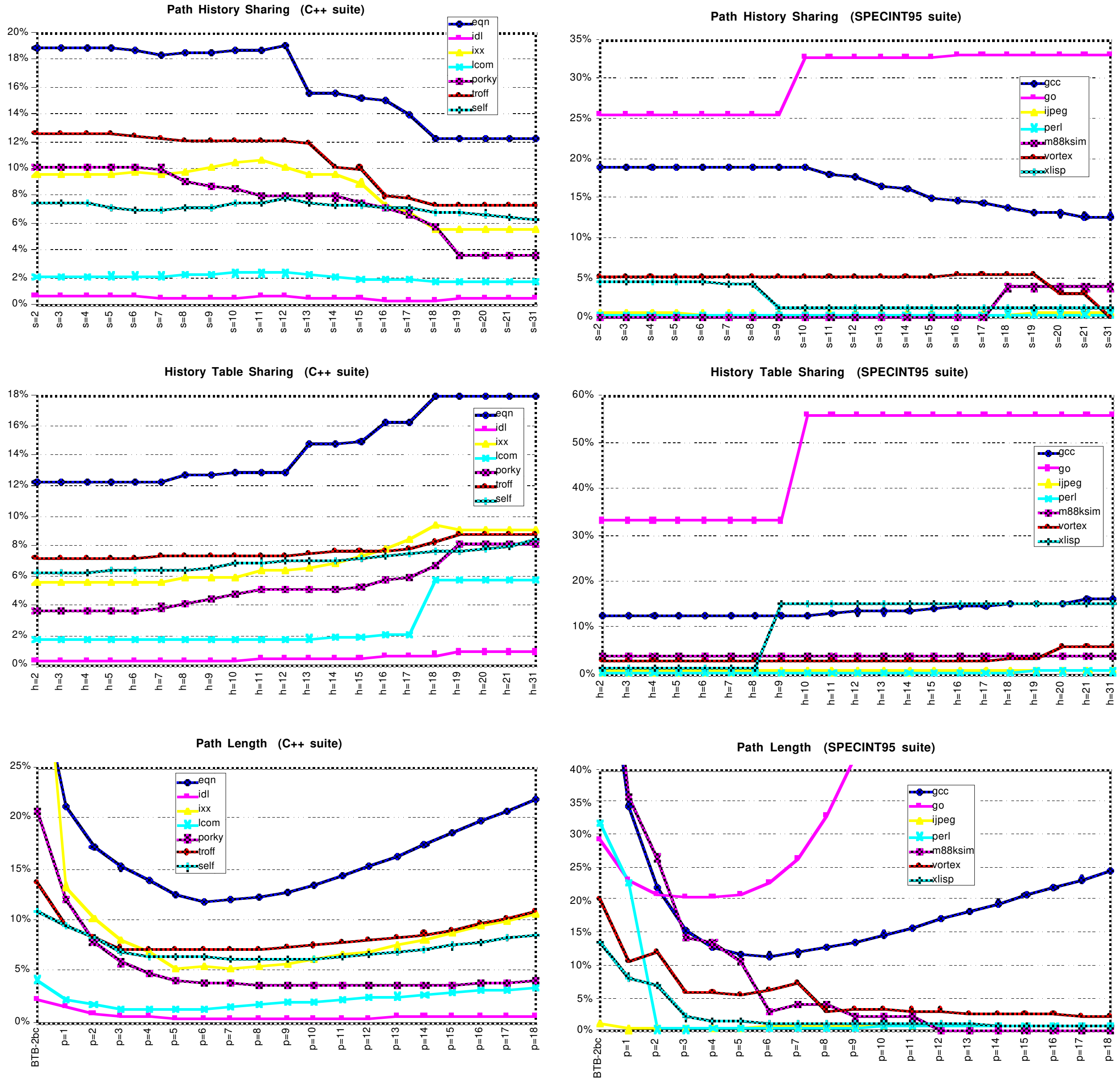


Figure 10. Individual benchmark results

p=8,h=2,s=	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	31
eqn	19.0	19.0	19.0	19.0	18.8	18.4	18.6	18.6	18.7	18.8	19.1	15.7	15.6	15.2	15.1	14.1	12.3	12.3	12.3	12.3	12.3
idl	0.6	0.6	0.6	0.6	0.6	0.5	0.6	0.6	0.6	0.6	0.6	0.5	0.5	0.5	0.4	0.4	0.4	0.5	0.5	0.5	0.5
ixx	9.6	9.6	9.6	9.5	9.7	9.7	9.8	10.1	10.5	10.6	10.2	9.7	9.5	9.0	7.3	6.9	5.6	5.6	5.6	5.6	5.6
lcom	2.1	2.1	2.1	2.1	2.1	2.2	2.2	2.3	2.4	2.4	2.3	2.2	2.1	2.0	1.8	1.8	1.8	1.8	1.8	1.8	1.8
porky	10.1	10.1	10.1	10.1	10.1	10.1	9.1	8.8	8.5	8.0	8.0	8.0	8.0	7.5	7.1	6.7	5.7	3.7	3.7	3.7	3.7
troff	12.7	12.7	12.7	12.7	12.5	12.3	12.1	12.1	12.1	12.1	12.0	11.9	10.1	10.1	8.0	7.8	7.3	7.3	7.3	7.3	7.3
self	7.4	7.4	7.4	7.2	7.0	7.1	7.2	7.2	7.5	7.5	7.9	7.5	7.4	7.3	7.2	7.2	6.8	6.8	6.7	6.6	6.3
T-AVG-C++	8.8	8.8	8.8	8.8	8.7	8.6	8.5	8.5	8.6	8.6	8.6	7.9	7.6	7.4	6.7	6.4	5.7	5.4	5.4	5.4	5.3
T-MED-C++	9.6	9.6	9.6	9.5	9.7	9.7	9.1	8.8	8.5	8.0	8.0	8.0	8.0	7.5	7.2	6.9	5.7	5.6	5.6	5.6	5.6
gcc	18.9	18.9	18.9	18.9	18.9	18.9	18.9	18.9	18.9	18.1	17.7	16.8	16.4	15.2	14.8	14.6	13.9	13.3	13.2	12.8	12.8
go	25.7	25.7	25.7	25.7	25.7	25.7	25.7	25.7	32.7	32.7	32.7	32.7	32.7	32.7	33.1	33.1	33.1	33.1	33.1	33.1	33.1
ijpeg	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.5	0.5	0.5	0.6	0.6	0.6	0.5	0.5	0.6	0.6	0.7	0.7	0.7	0.7
perl	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.5	0.6	0.6	0.6	0.6
m88ksim	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.2	0.2	0.2	3.9	3.9	3.9	3.9	3.9
vortex	5.3	5.3	5.3	5.3	5.3	5.3	5.3	5.3	5.3	5.3	5.3	5.3	5.2	5.2	5.4	5.4	5.3	5.5	3.2	3.2	3.2
xlisp	4.6	4.6	4.6	4.6	4.6	4.4	4.4	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.3	1.3	1.3	1.3	1.3	1.3	1.3
T-AVG-spec	8.0	8.0	8.0	8.0	8.0	7.9	7.9	7.5	8.5	8.4	8.3	8.2	8.1	7.9	8.0	7.9	8.4	8.3	8.0	7.9	7.9
T-MED-spec	4.6	4.6	4.6	4.6	4.6	4.4	4.4	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.3	1.3	3.9	3.9	3.2	3.2	3.2
T-AVG-TOT	8.4	8.4	8.4	8.4	8.3	8.3	8.2	8.0	8.5	8.5	8.5	8.1	7.9	7.6	7.3	7.2	7.0	6.9	6.7	6.7	6.6
T-MED-TOT	6.4	6.4	6.4	6.2	6.2	6.2	6.2	6.2	6.4	6.4	6.6	6.4	6.3	6.2	6.3	6.1	5.5	4.7	3.8	3.8	3.8

Table 4. Path history sharing misprediction rate in %

p=8,s=31,h=	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	31
eqn	12.3	12.3	12.3	12.3	12.3	12.3	12.8	12.8	12.9	12.9	12.9	14.8	14.8	14.9	16.3	16.3	18.0	18.0	18.0	18.0	18.0
idl	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.6	0.6	0.6	0.7	0.7	0.8	0.9	0.9	0.9	0.9
ixx	5.6	5.6	5.6	5.6	5.6	5.6	6.0	6.0	6.0	6.3	6.4	6.5	6.8	7.4	7.9	8.5	9.5	9.1	9.1	9.1	9.1
lcom	1.8	1.8	1.8	1.8	1.8	1.8	1.8	1.8	1.8	1.8	1.8	1.9	1.9	2.0	2.1	2.1	5.8	5.9	5.9	5.9	5.9
porky	3.7	3.7	3.7	3.7	3.8	4.0	4.3	4.5	4.9	5.2	5.2	5.2	5.2	5.3	5.8	5.9	6.8	8.2	8.2	8.2	8.2
troff	7.3	7.3	7.3	7.3	7.3	7.3	7.3	7.3	7.4	7.4	7.4	7.5	7.6	7.7	7.8	7.9	8.2	8.8	8.8	8.8	8.8
self	6.3	6.3	6.3	6.4	6.4	6.4	6.5	6.6	6.9	7.0	7.0	7.1	7.1	7.2	7.3	7.5	7.8	7.8	7.9	8.1	8.4
T-AVG-C++	5.4	5.4	5.4	5.4	5.4	5.4	5.6	5.6	5.8	5.9	5.9	6.2	6.3	6.5	6.8	7.0	8.1	8.4	8.4	8.4	8.5
T-MED-C++	5.6	5.6	5.6	5.6	5.6	5.6	6.0	6.0	6.0	6.3	6.4	6.5	6.8	7.2	7.3	7.5	7.8	8.2	8.2	8.2	8.4
gcc	12.8	12.8	12.8	12.8	12.8	12.8	12.8	12.8	12.8	13.2	13.5	13.6	13.9	14.3	14.6	14.9	15.2	15.4	15.4	16.1	16.1
go	33.1	33.1	33.1	33.1	33.1	33.1	33.1	33.1	55.7	55.7	55.7	55.7	55.7	55.7	55.7	55.7	55.7	55.7	55.7	55.7	55.7
ijpeg	0.7	0.7	0.7	0.7	0.7	0.7	0.7	0.7	0.7	0.8	0.8	0.8	0.8	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
perl	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.7	0.7	0.7	0.7
m88ksim	3.9	3.9	3.9	3.9	3.9	3.9	3.9	3.9	3.9	3.9	3.9	3.9	3.9	3.9	3.9	3.9	3.9	3.9	3.9	3.9	3.9
vortex	3.2	3.2	3.2	3.2	3.2	3.2	3.2	3.2	3.2	3.2	3.2	3.2	3.2	3.2	3.2	3.2	3.5	3.6	6.4	6.4	6.4
xlisp	1.3	1.3	1.3	1.3	1.3	1.6	1.6	15.5	15.5	15.5	15.5	15.5	15.5	15.5	15.5	15.5	15.5	15.5	15.5	15.5	15.5
T-AVG-spec	7.9	7.9	7.9	7.9	7.9	8.0	8.0	10.0	13.2	13.3	13.3	13.3	13.4	13.4	13.5	13.5	13.6	13.7	14.1	14.2	14.2
T-MED-spec	3.2	3.2	3.2	3.2	3.2	3.2	3.2	3.9	3.9	3.9	3.9	3.9	3.9	3.9	3.9	3.9	3.9	3.9	6.4	6.4	6.4
T-AVG-TOT	6.6	6.6	6.6	6.6	6.6	6.7	6.8	7.8	9.5	9.6	9.6	9.8	9.8	9.9	10.2	10.3	10.9	11.0	11.2	11.3	11.3
T-MED-TOT	3.8	3.8	3.8	3.8	3.9	4.0	4.1	5.2	5.4	5.8	5.8	5.9	6.0	6.3	6.6	6.7	7.3	8.0	8.0	8.1	8.3

Table 5. History table sharing misprediction rate in %

s=31,h=2,p=	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
eqn	34.8	21.3	17.2	15.3	13.9	12.6	11.9	12.1	12.3	12.7	13.5	14.4	15.4	16.3	17.4	18.6	19.8	20.9	22.1
idl	2.4	1.5	1.0	0.6	0.6	0.4	0.4	0.5	0.5	0.5	0.5	0.5	0.5	0.6	0.6	0.6	0.7	0.7	0.7
ixx	45.7	13.4	10.3	8.2	6.9	5.5	5.5	5.4	5.6	5.8	6.2	6.7	7.0	7.6	8.2	8.8	9.4	10.1	10.8
lcom	4.3	2.3	1.7	1.3	1.3	1.3	1.4	1.6	1.8	1.9	2.1	2.3	2.5	2.6	2.8	3.0	3.1	3.3	3.5
porky	20.8	12.2	7.8	5.9	4.8	4.2	4.0	3.9	3.7	3.7	3.6	3.6	3.6	3.7	3.7	3.8	3.9	4.0	4.2
troff	13.7	9.5	8.3	7.3	7.2	7.1	7.2	7.2	7.3	7.4	7.6	7.8	8.0	8.3	8.7	9.2	9.7	10.3	10.9
self	10.9	9.6	8.5	7.0	6.6	6.6	6.4	6.3	6.3	6.2	6.3	6.5	6.7	7.0	7.2	7.6	7.9	8.3	8.7
T-AVG-C++	18.9	10.0	7.8	6.5	5.9	5.4	5.3	5.3	5.4	5.5	5.7	6.0	6.3	6.6	7.0	7.4	7.8	8.2	8.7
T-MED-C++	13.7	9.6	8.3	7.0	6.6	5.5	5.5	5.4	5.6	5.8	6.2	6.5	6.7	7.0	7.2	7.6	7.9	8.3	8.7
gcc	65.7	34.6	22.1	15.6	13.0	11.8	11.6	12.0	12.8	13.7	14.8	16.0	17.2	18.4	19.6	20.8	22.1	23.3	24.6
go	29.3	23.1	20.9	20.4	20.4	21.0	22.8	26.5	33.1	42.1	51.9	60.9	68.7	75.4	81.1	85.7	89.2	91.7	93.3
ijpeg	1.3	0.3	0.4	0.5	0.5	0.6	0.6	0.7	0.7	0.7	0.8	0.8	0.8	0.8	0.8	0.9	0.9	0.9	0.9
perl	31.8	22.7	0.3	0.3	0.4	0.4	0.5	0.5	0.6	0.6	0.6	0.7	0.7	0.7	0.7	0.7	0.8	0.8	0.8
m88ksim	76.4	36.0	26.6	14.4	13.4	10.6	3.1	3.9	3.9	2.1	2.1	2.1	0.2	0.2	0.2	0.2	0.2	0.2	0.2
vortex	20.2	10.6	12.1	6.1	5.8	5.7	6.3	7.4	3.2	3.5	3.4	3.1	3.1	2.7	2.5	2.4	2.5	2.4	2.4
xlisp	13.5	8.3	7.0	2.1	1.5	1.5	1.4	1.4	1.3	1.2	1.2	1.1	1.1	1.0	0.9	0.8	0.8	0.8	0.8
T-AVG-spec	34.0	19.4	12.8	8.5	7.9	7.4	6.6	7.5	7.9	9.1	10.7	12.1	13.1	14.2	15.1	15.9	16.6	17.2	17.6
T-MED-spec	29.3	22.7	12.1	6.1	5.8	5.7	3.1	3.9	3.2	2.1	2.1	2.1	1.1	1.0	0.9	0.9	0.9	0.9	0.9
T-AVG-TOT	26.5	14.7	10.3	7.5	6.9	6.4	5.9	6.4	6.6	7.3	8.2	9.0	9.7	10.4	11.0	11.6	12.2	12.7	13.1
T-MED-TOT	20.5	11.4	8.4	6.5	6.2	5.6	4.8	4.6	3.8	3.6	3.5	3.4	3.4	3.2	3.3	3.4	3.5	3.7	3.8

Table 6. Path length misprediction rate in %