Target Encoding for Efficient Indirect Jump Prediction*

Juan Carlos Moure[1], Domingo Benitez[2], Dolores Isabel Rexachs[1], and Emilio Luque[1]

[1] Computer Architecture and Operating Systems Department,
Universidad Autónoma de Barcelona, 08193 Barcelona, Spain
{JuanCarlos.Moure,Dolores.Rexachs,Emilio.Luque}@uab.es
[2] University of Las Palmas G.C., 35017 Las Palmas, Spain
dbenitez@dis.ulpgc.es

Abstract. Accurate indirect jump prediction is critical for some applications. Proposed methods are not efficient in terms of chip area. Our proposal evaluates a mechanism called target encoding that provides a better ratio between prediction accuracy and the amount of bits devoted to the predictor. The idea is to encode full target addresses into shorter target identifiers, so that more entries can be stored with a fixed memory budget, and longer branch histories can be used to increase prediction accuracy. With a fixed area budget, the increase in accuracy for the proposed scheme ranges from 10% to up to 90%. On the other hand, the new scheme provides the same accuracy while reducing predictor size by between 35% and 70%.

1 Introduction

Dynamic control-flow prediction is a key task on current processors. This work proposes an efficient mechanism for predicting indirect jumps. Although they are less frequent than conditional branches, for some applications the lack of a specialized indirect jump predictor may degrade performance significantly [7], [8].

Common sources of indirect jumps are case statements and virtual function calls used in object-oriented languages. While some indirect branches jump to a unique target address during the program's execution (*monomorphic* jumps), and are easy to predict, many of them (called *polymorphic*) jump to several target addresses, depending on input data, and their prediction is complex. Accurate prediction for those jumps requires a multiple-choice predictor, rather than a mere binary (taken/not-taken) predictor, and storing several target addresses per jump.

Indirect jump predictors proposed in the literature match the scheme depicted in Fig. 1. One or more tables are indexed using the jump's address and *branch history*, which codifies the outcomes of recently executed branches (indirect or not) leading up to the jump. Tables contain full target addresses and additional data that is used to select the final predicted address. Tables are trained using the outcome of indirect jumps once they are retired from the processor pipeline.

* This work was supported by the MCyT-Spain under contract TIN 2004-03388, the Generalitat de Catalunya - Grup Recerca Consolidat 2001 SGR-00218, and the HiPEAC European Network of Excellence

J.C. Cunha and P.D. Medeiros (Eds.): Euro-Par 2005, LNCS 3648, pp. 497–507, 2005.

The predictor's accuracy is mostly limited by its memory size and by the length and quality of the branch history. A separate entry is allocated into the predictor's tables for each combination of jump address and branch history. As the history gets larger, the probability of containing a previous branch that correlates with the predicted branch also gets larger, increasing prediction accuracy [5]. However, more entries are required in the predictor's tables or otherwise many prediction cases will map into the same entry and create *aliasing*. The indexing and selection methods try to reduce the effect of aliasing, making efficient use of the available predictor's memory.

Although larger tables provide higher accuracy, they do not handle information efficiently, since the same long target addresses are replicated several times. We present and evaluate a method to encode full target addresses into shorter target identifiers. The proposed two-stage mechanism consists of (1) predicting a short target identifier using the scheme shown in Fig. 1, and then (2) translating it into a full target address. Since encoded targets are shorter, more entries can be stored with a fixed memory budget, and then longer histories can be used to increase prediction accuracy. The table used in the second stage holds full target addresses and still requires large entries, but since each address is stored only once, it requires considerably fewer entries.

Results obtained in simulation indicate that the design achieves a better ratio between prediction accuracy and predictor size. This increase of storage efficiency may be used to increase performance or to lower area requirements and the predictor's power consumption. The proposed two-stage scheme increases the indirect predictor's average response latency, but this increase is shown to have very little effect on performance. On a 4-way superscalar processor with a realistic memory hierarchy, with a penalty of 2 cycles for using the two-level jump predictor, the performance improvement due to target encoding ranges from 0.1% to 2.5%, depending on the benchmark.

Section 2 reviews some related work. Sections 3 and 4 describe the baseline and the proposed indirect jump predictors. Section 5 presents the experimental methodology and some preliminary results. Full results are presented and discussed in section 6. The final section outlines the conclusions and introduces future lines of research.

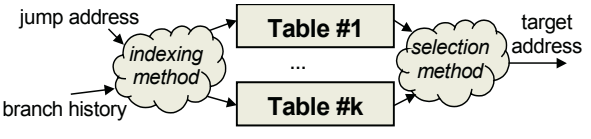

Fig. 1. General scheme of an indirect jump predictor

2 Related Work

A Branch Target Buffer (BTB) [13] provides a simple method for accurately predicting monomorphic indirect jumps or jumps whose target changes infrequently, but provides weak results for polymorphic jumps. Adding a hysteresis bit to limit the update of the target address only after two consecutive mispredictions [3] provides

small gains [5, 6]. A better method for dealing with polymorphic jumps is the Target Cache (TC) [5]. It adapts the two-level prediction methods previously proposed for conditional branches [14], to indirect jumps. [5] analyzes several methods to track branch history, several indexing methods, and the use of tags on the TC.

A variation on the TC is the Cascaded Indirect Jump Predictor [6], which significantly reduces the table size needed to achieve a given accuracy. It dynamically identifies easily predicted jumps and devotes them a simple and low cost predictor, preventing insertion of these jumps into a more powerful second stage predictor. The result is that easily predicted jumps avoid most cold start misses and do not waste entries in the second stage predictor, which is better exploited by the remaining indirect jumps. Using a BTB as the first-stage filter, as in the Intel Pentium M and Pentium 4 processors [7, 8], provides good results with a simple and efficient design. We evaluate our proposal using this scheme as the baseline design (described in the next section).

Prediction by Partial Matching (PPM) [12] exploits variable-length path correlation to improve prediction accuracy. Several tables are accessed in parallel, each one addressed by an index containing a branch history of a different length. The table using the longest history and having a valid prediction provides the final target. The potential of varying the history length for each specific branch is not analyzed in this paper.

Control-flow prediction performance is improved by either increasing accuracy, width (instructions retired per prediction), or rate (predictions per cycle). We have previously proposed a two-level hierarchy [14], but aimed to increase prediction rate and not to improve the ratio between prediction accuracy and predictor size.

A method to increase prediction width is path-based next trace prediction [9]. It uses a cascaded, two-level scheme to predict instruction traces, rather than jumps or branches. Our baseline design uses the exclusive-or-fold method proposed there.

3 Baseline: A BTB-Based Cascaded Predictor

The baseline design used to evaluate our proposal is a cascaded indirect jump predictor [6, 7] using two tables (see Fig. 2). The first table is a Branch Target Buffer (BTB) [13], which identifies branches in the instruction fetch stream and provides a target address for each branch. Since the BTB always correctly predicts monomorphic jumps, a specialized Indirect Jump Predictor (IJPred) exploiting branch correlation is only required for polymorphic indirect jumps. We assume that an extra 32-entry return address stack (RAS) [11] (not shown) is used to predict indirect return jumps.

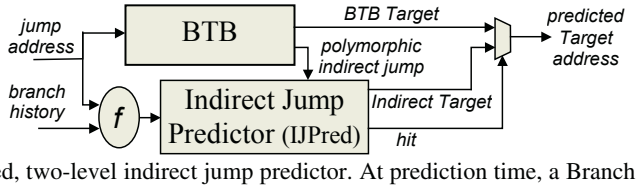

Fig. 2. Cascaded, two-level indirect jump predictor. At prediction time, a Branch Target Buffer (BTB) identifies branches and filters the use of a specialized Indirect Jump Predictor (IJPred)

Extensive simulation has been performed to obtain a realistic, highly tuned baseline design. We have simulated IJPred sizes from 256 to 16K entries, IJPred tag sizes from 0 to 16 bits, history lengths from 1 to 61, hysteresis counters from 0 to 3 bits, different ways of building and codifying branch history, and several indexing and selection algorithms. The most important results are explained in the following description.

Updating the BTB and IJPred at Retire Time (Not Speculatively)
We assumed a BTB with 4K entries and 16-bit tags, which suffers a low miss rate. At retire time, a BTB entry is allocated for each branch that misses in the BTB, and initialized with the branch type and target address. If the branch is an indirect jump (not a return), the type field is set as monomorphic. If the same indirect jump is later retired and its target address does not match the address stored in the BTB, then the type field is set as polymorphic but the target address prediction is **not** modified.

The IJPred is direct-mapped and 4-bit tagged. It is updated at retire time only for indirect jumps identified as polymorphic by the BTB, and only when their target address differs from the BTB prediction. This filtering scheme prevents prediction cases that are well-handled by the BTB from wasting IJPred memory space.

A hysteresis bit is used to avoid replacing IJPred entries that frequently provide correct predictions. The bit is cleared when the entry is first allocated, and is set on correct target predictions, and cleared on wrong predictions. On an IJPred miss, the previous entry contents are replaced by the new ones only when the hysteresis bit is found to be cleared. If the bit is found set, then it is cleared.

Using Two Types of Global Path History
Two separate 61-bit history registers are used: *cghr* is updated for each conditional branch taken, and *ighr* is updated for each indirect jump (including returns). As was noted in [12], maintaining two history registers of different types and dynamically choosing one of these for each static jump provides improved accuracy compared to using a unique history register. Our approach merges the contents of *cghr* and *ighr* using an exclusive-or operation before using history to generate the IJPred index (Fig. 4 on next page). It provides a similar improvement in accuracy (from 2% to 30%, depending on the benchmark) with a simpler implementation.

Both history registers, *cghr* and *ighr*, are speculatively updated using the outcome of the current prediction, and are corrected on branch mispredictions using a very small amount of recovery data. The update consists of shifting the contents *s* bits to the left and adding the exclusive-or of the *s* lower bits of the branch address and the branch target address, as shown in Fig. 3. The *history length*, *l*, is the number of branches whose histories are held in the history register, ($l = 61 / s$).

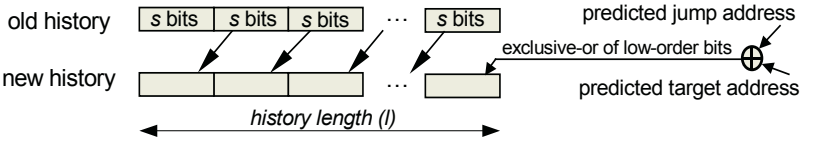

Fig. 3. Speculative update of branch history registers (corrected on branch mispredictions)

As other authors [10],[12], we have found that *l* highly influences accuracy. Also, for each IJPred table size, using the optimal *l* for each single benchmark (*BestHist*) instead of using the optimal *l* for all the benchmarks (*BestHistALL*) increases accuracy between 15% and 45%. In our experiments, we have used *BestHistALL* most of the time but has also validated that results do not significantly vary if using *BestHist*.

Indexing and Selection Algorithms at Prediction Time

The goal of an indexing scheme is to map the whole input data into *n* output bits so that the resulting index is evenly distributed, and aliasing is reduced. Fig. 4.a shows an scheme to fold a value, *v*, into an *n*-bit value using an exclusive-or-fold method [9].

The index for the BTB (Fig. 4.b) is the result of *xor-folding* the address of the branch to be predicted into chunks of different size (sizes are prime numbers) and the combination of these chunks into a large value that is again *xor-folded* into a final n-bit index. The best results are obtained with a skewed-associative scheme [1], which generates a different BTB set index for each possible BTB way.

The index for the IJPred uses the address of the jump to be predicted and the history registers (Fig. 4.c). These complex indexing methods increase accuracy slightly with respect to simpler ones but, most importantly, provide highly homogeneous results for all the configurations evaluated. The scheme is not intended to be an implementation proposal, but a reference for exploring cheaper and faster methods that merge fewer bits in this latency-critical step.

On a BTB miss, the fall-through address is predicted as the jump's target. On a hit, the BTB provides the target prediction, unless the indirect jump is identified as polymorphic or as a return. For polymorphic jumps, the IJPred is accessed and provides the prediction only if the IJPred access hits. Return jumps are handled by the RAS.

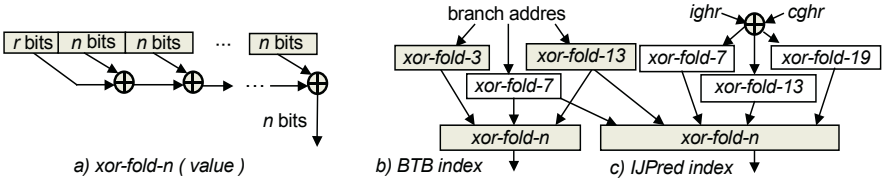

Fig. 4. Index generation scheme for BTB and IJPred using an xor-fold scheme

4 Indirect Target Encoding

The IJPred contains two types of data: (1) which of the possible paths a jump will take, and (2) at which address this path begins. Separating these two types of data on two different tables provides more efficient memory usage. If a jump can take *k* different paths, then $\log_2 k$ bits are required to codify a path identifier (*pathID*). Then, if the IJPred holds *pathID*'s instead of full addresses, it may contain more entries, use longer histories, and then increase prediction accuracy.

A second table, the Indirect Target Table (ITT), is required to provide the full target address (see Fig. 5). Using both the *pathID* and the jump's address to index the

ITT provides the best results. The ITT should ideally be able to hold all the target addresses taken by all jumps, but in practice only useful targets need to be stored, since storing those targets that rarely involve a correct prediction only marginally improves performance.

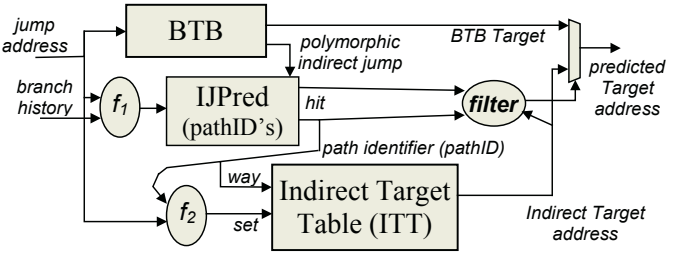

Fig. 5. Indirect Target Encoding. A short path identifier (*pathID*) replaces target addresses in the IJPred. A second-level Indirect Target Table (ITT), indexed by the jump address and the *pathID*, provides the full indirect target address

ITT Access at Prediction Time

Exploiting freedom in the target-encoding algorithm allows implementing a *k*-way set-associative ITT with the small access delay and power consumption of a direct-mapped tag-less table. The scheme is similar to the next cache line and set predictor [4]. The low order bits of the *pathID* codify the ITT way in which the target address is located. The target address is hashed (like in Fig. 4) to provide the *pathID*'s high-order bits. With this mechanism, the complexity of the associative indexing scheme is avoided at prediction time (where may affect performance), and is suffered at retire time, but only on IJPred mispredictions.

The ITT way is obtained from the *pathID* read from the IJPred. The ITT set is obtained by hashing the *pathID*'s high-order bits and the jump's address (like in Fig. 4). A filter tests the IJPred prediction validity by comparing the *pathID* read from the IJPred with the *pathID* computed from the target address read from the ITT.

Target Search and ITT Update at Retire Time

When a mispredicted indirect jump is retired, its final target address is searched for in the ITT. Since these cases are unfrequent and we will show that the ITT update latency does not affect performance, the search operation on ITT ways may be done serially to reduce H/W complexity. The *pathID*'s high-order bits are generated from the correct target address and combined with the jump's address to index each one of the ITT ways. As for the BTB, a skewed-associative scheme provided the lowest miss rate [1].

Each ITT entry contains a 4-bit saturating counter that is increased with each correct target use. When a new target address must be allocated, the counter of all the entries that are a potential placement are decremented, and only a zero counter enables the replacement. This policy prioritizes useful target addresses and reduces the number of ITT replacements, which also reduces the occurrence of ITT misses.

5 Experimental Methodology, Results, and Discussion

We use a trace-based simulation to measure prediction accuracy and tune the main design parameters. Accurate cycle-by-cycle simulation is used to measure the effect on prediction accuracy of the delayed update of prediction tables (BTB, IJPred, ITT), and the effect of prediction latency and accuracy on the processor's performance.

First, we analyze the design space of the Indirect Target Table (ITT) and select an optimal configuration. Then, we explore the best size for the IJPred tags and *pathID* field. We compare the baseline design and the proposed target encoding design with respect to prediction accuracy and predictor size. Then we verify that the effect on prediction accuracy of the delayed update of prediction tables is insignificant, and that increasing the indirect predictor's latency reduces performance slightly.

Metrics, Simulator, and Benchmarks

Prediction accuracy is measured as the average number of instructions between jump mispredictions (Kilo-instructions per misprediction). Predictor size is measured in KBytes (KB). Processor performance is measured in instructions per cycle (IPC).

We have used the Simplescalar-Alpha tool set [2] to generate the dynamic instruction trace of the first 20 billion instructions for some programs of the SPEC benchmark suites. Table 1 shows the selected benchmarks and their inputs (Alpha ISA, cc DEC 5.9, –O4). They have been selected because they have the lowest accuracy when a simple BTB is used for indirect jump prediction (6^{th} column of Table 1), and then may benefit more from using a specialized predictor (col. 7-8 for an IJPred of 512 and 4K entries). Table 1, columns 4-5, shows the number of static polymorphic indirect jumps and targets.

Table 1. Simulation data and simulation results for selected SPEC benchmarks

Benchmark	input	SPEC	polymorphic		instructions / misprediction		
			jumps	targets	BTB	IJPred 512	IJPred 4K
crafty	reference	*int00*	15	89	877	1457	1908
eon	cook	*int00*	10	20	681	45123	$> 10^5$
gap	reference	*int00*	49	146	1609	46724	$> 10^5$
gcc	expr.i	*int00*	183	975	439	1765	3137
perl	difference	*int00*	65	659	118	763	2190
vpr	route	*int00*	4	19	4951	21291	$> 10^5$
m88ksim	reference	*int95*	7	26	1407	18355	56805
li	reference	*int95*	7	71	989	5381	14799
facerec	reference	*fp00*	9	47	2766	$> 10^7$	$> 10^7$
fma3d	reference	*fp00*	12	23	2977	$> 10^7$	$> 10^7$
sixtrack	reference	*fp00*	8	49	601	4248	9795

ITT Configuration

Results have shown that only a small subset of all the target addresses of polymorphic jumps needs to be held in the ITT for near-optimal performance. Although some benchmarks have more than 500 targets (see Table 1), a 64-entry ITT is enough to achieve a miss rate lower than 0.1%, except for benchmark *perl*, which requires 128 entries. For such sizes, an 8-way set-associative ITT provides the best performance.

A 4-bit *pathID* is enough to maintain the ITT miss rate below 0.1% for all benchmarks except for *gcc* and *perl*, which require a 5-bit and a 6-bit *pathID*, respectively.

IJPred Configuration

The baseline IJPred configuration, with entries containing full addresses, may afford a large tag and a large hysteresis counter to try to reduce IJPred misses. Results have shown that a tag larger than 4 bits, or a hysteresis counter larger than 1 bit improves performance very slightly on a direct-mapped IJPred organization.

When target encoding is used to improve chip area utilization, it is more effective to reduce the tag size to 3 bits and devote 5 bits to codify the *pathID*. The filter method described in section 4 detects a significant part (20-70%) of the misses not detected by the shortened tag. It also also detects (but does not correct) cases where ITT replacements have made the *pathID* in the IJPred indicate a wrong ITT way.

Accuracy Versus Storage Size

Figure 6 displays accuracy versus storage size for some representative benchmarks and for different predictor designs with an IJPred of 512-4K entries. The storage size of the baseline design accounts for the target address' size (32 bits), the tag's size (3 bits), and the hysteresis counter's size (1 bit). Target encoding replaces the target address by the *pathID* (5 bits instead of 32 bits) and must account for the ITT size (64 entries, each containing a full target address and a 4-bit replacement counter).

Results in Fig. 6 show that the ratio of accuracy versus predictor size is always better for the encoded design. Benchmarks *li* and *sixtrack* are depicted together because they have very similar results. For these benchmarks, correlation is highly effective in increasing accuracy. Given a fixed predictor size, the encoded scheme exploits correlation better than the baseline (the line depicting accuracy versus size separates for larger predictors). For example, with a 5-KB predictor, accuracy improves by 90%.

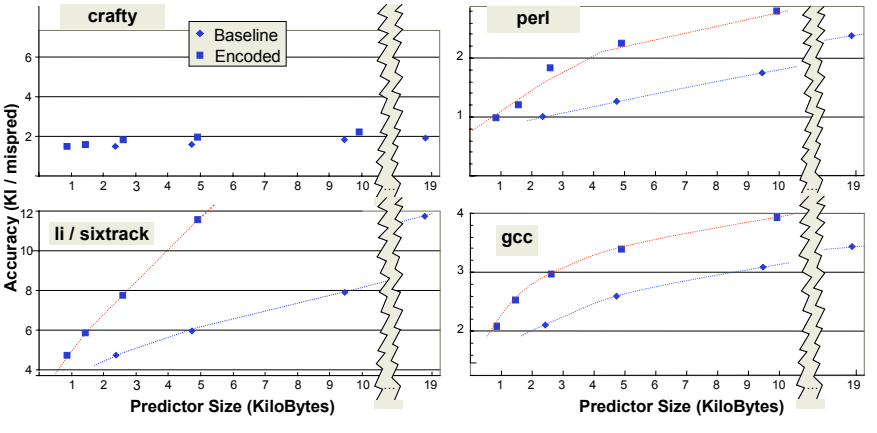

Fig. 6. Accuracy versus storage size on selected benchmarks for the baseline and encoded predictors, with IJPred sizes from 512 to 4K. 64 ITT entries, *pathID* length is 5 bits, IJPred tag length is 3 bits, history length (l) is *BestHistALL*, which depends on IJPred size: 512 (l=4), 1K (l=8), 2K (l=14), 4K (l=20))

The encoded scheme achieves accuracy improvements of around 50% for a fixed predictor size for benchmarks *gcc* and *perl*. With a large working set of indirect target addresses, the small ITT and short *pathID* provokes a moderate amount of ITT misses (<2%) that reduces the potential accuracy improvement by only around 10%.

For a relatively large area budget for the indirect predictor, two kind of benchmarks cannot benefit from the encoded scheme to improve processor performance. The first example is *crafty*, which benefits little from history correlation, and increases accuracy very slowly with higher storage. The other example are the benchmarks not shown in Fig. 4, which provide near-perfect prediction with a relatively small IJPred of around 1K entries. The encoded scheme, however, is still very useful in reducing storage (and power) requirements. For example, averaging for all benchmarks, a 3-KB encoded predictor provides the same accuracy as a 10-KB baseline predictor.

Table 2. Microarchitecture parameters for the cycle-accurate simulations

Front-End	Back-End (Execution Core)	Memory System
Decoupled: Fetch Target Queue (FTQ) holds up to 24 fetch blocks	Up to 4 instructions **renamed** and **dispatched** per cycle	Perfect Memory Disambiguation, Store to Load forwarding of any size
I-Cache Prefetches: two I-cache checks per cycle using FTQ contents. If not found, prefetch from L2 cache.	Up to 6 instructions **issued** and **retired** per cycle	**I-Cache**: 256 sets x 2 ways x 32B blocks **D-Cache**: 256 sets x 4 ways x 16B blocks **L2-Cache**: 1K sets x 8 ways x 64B blocks **L3-Cache**: 8K sets x 8ways x 128B blocks
Fetch Predictor: predicts one conditional branch per cycle, one return every two cycles, one indirect jump every *k* cycles.	**Fetch Queue**: 12 instructions **Issue Queue**: 24 instructions **Reorder Buffer**: 124 instructions **Load/Store Queue**: 48/24 instr.	**I-Cache** / **D-Cache**: 2 ports, 2-cycle latency **L2-Cache**: 7-cycle load-use latency **L3-Cache**: 40-cycle load-use latency **Memory**: 200-cycle load-use latency
Gshare: 64K entries, 2-bit counters	Operation **latencies** like Pentium IV	**Bandwidth** (L2/L3/Mem): 25,6 / 12,8 / 6,4 GB/s

Performance Measures

Cycle-level simulations have been performed by modeling a 4-way processor back-end and a realistic memory system (see Table 2). The simulated front-end is decoupled and predicts one full basic-block per cycle. Our first result was that prediction accuracy is not degraded by the delayed update of prediction tables. Accuracy varies very slightly when increasing the pipeline depth (and then the predictor update delay) from 12 to 30 cycles. As argued in other papers, a higher update delay increases the predictor's hysteresis, which does not necessarily causes a worse behavior.

Figure 7 shows the effect on performance of varying IJPred size and varying the indirect predictor's latency. On the one hand, doubling IJPred size, and therefore increasing indirect jump prediction accuracy, provides an average IPC increase of around 0.5% for the benchmarks considered in this paper (Fig. 7.a). Benchmark *perl* (Fig. 7.b) is the one that benefits most from a larger IJPred (average IPC increase of 1.2% when doubling IJPred size). The average penalty of indirect jumps has been experimentally found to be around 21 cycles, which explains why avoiding mispredictions results in a significant gain in performance.

On the other hand, a two-cycle delay penalty for using the IJPred table reduces IPC by less than 0.05% of the average. There are two main reasons for this result. First, jump mispredictions are not too frequent (less than 1 every 100 instructions), since many of the indirect jumps (30-70%) are predicted by the BTB. Second, a sub-

stantial part (more than 95%) of the delay due to using the IJPred and ITT tables instead of the BTB, is overlapped by other stalls occurring later in the pipeline. More than 60% of the overlap is due to the decoupled front-end scheme, which compensates *IJPred-use* stall cycles with cycles where a branch prediction provides several instructions (full basic blocks) to the front-end. However, as predicted by Amdhal's law, the indirect prediction latency becomes more critical for values larger than 3 cycles or if the execution width of the processor is scaled to 8 instructions per cycle.

Given that prediction latency is not critical, power consumption is afforded by delaying the IJPred access until the BTB access has been completed and a polymorphic jump has been identified. Similarly, power is saved by delaying the ITT access until a valid *pathID* from the IJPred table is read.

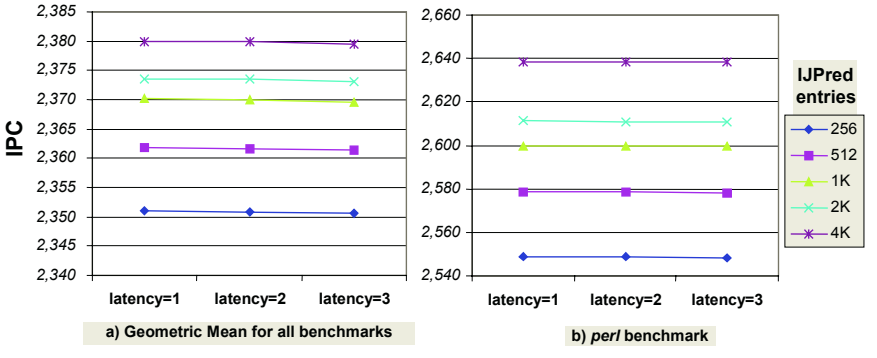

Fig. 7. IPC for varying IJPred sizes and IJPred latencies

6 Conclusions and Future Lines

We have presented and evaluated target encoding as a method for improving the indirect jump prediction accuracy to cost ratio. This improvement can be used to increase processor performance for benchmarks that execute a moderate amount of polymorphic indirect jumps. On a realistic 4-way superscalar processor with a realistic memory hierarchy, the additional latency of the proposed two-level predictor has very slight effects on performance. Assuming a two-cycle increase in latency, the performance increase ranges from 0.1% to 2.5%. For benchmarks that benefit little from larger IJPred tables, the scheme may be used to reduce chip area and power consumption. For example, a 3-KB encoded predictor (with direct-mapped access) provides the same accuracy as a 10-KB baseline predictor (with set-associative access).

The target-encoding scheme works well because indirect jumps have a small working set of target addresses, which can be effectively cached in a table with 64 entries. The careful design of the table achieves several conflictive issues: high logical associativity to reduce conflict misses, and a small latency and low power consumption due to its direct mapped access. Basically, the freedom of the target-encoding algorithm allows for the implementation of a way prediction mechanism for free. Also, the replacement policy is designed to prioritize useful targets instead of frequent targets.

The relatively small effect on performance of the enhanced indirect predictor is very related to the low frequency of indirect branches in the SPECint2000 workload. A future extension to this work is analyzing more object-oriented workloads such as SPECjvm98.

Static and profile analysis may improve indirect jump prediction in several ways. First, if the most frequent target for each indirect jump is identified, it may be used to initialize the BTB and then reduce the storage requirements of the IJPred and its usage rate. This option requires an ISA extension to allow access to the BTB. Second, embedded systems tuned at design time can use the static analysis to select the best configuration for the indirect predictor (IJPred and ITT size, *pathID*/tag length, history length, …). In particular, we have found that tuning history length for an specific benchmark may yield an accuracy improvement between 15% and 45%. Adapting history length dynamically, either for a full application, like in [10], or for each specific branch, like in [12], is another future line for improving accuracy.

References

1. Bodin, F., Seznec, A.: Skewed associativity improves program performance and enhances predictability. IEEE Trans. on Computers, vol. 46(5) (1997) 530–544
2. Burger, D., Austin, T.M.: The SimpleScalar tool set. Univ. Wisconsin-Madison Computer Science Department, Tech. Report TR-1342 (1997)
3. Calder, B., Grunwald, D.: Reducing Indirect Function Call Overhead in C+ Programs. Proc. 21th Int. Symp. on Principles of Programming Languages (1994) 397–408
4. Calder, B., Grunwald, D.: Next Cache Line and Set Prediction. Proc. 22nd Int. Symp. on Computer Architecture (1995) 287–296
5. Chang, P.-Y., Hao E., Patt, Y. N.: Target Prediction for Indirect Jumps. Proc. 24th Int. Symp. on Computer Architecture (1997) 274–283
6. Driesen, K., Hölzle, U.: The cascaded predictor: economical and adaptive branch target prediction. Proc. 31st Intl. Symp. on Microarchitecture (1998) 249–258
7. Gochman, S., et. al.: The Intel Pentium M processor: Microarchitecture and Performance. Intel Technology Journal, vol. 7(2), (2003) 21–36
8. Hinton, G., et. al.: The microarchitecture of the Pentium 4 processor. Intel Technology Journal, Q1 (2001)
9. Jacobson, Q., Rotenberg, E., Smith, J. E.: Path-based next trace prediction. Proc. 30th Int. Symp. on Microarchitecture (1997) 14–23
10. Juan, T., Sanjeevan, S., Navarro, J.J.: A third level of adaptivity for branch prediction. Proc. 25th Int. Symp. on Computer Architecture (1998) 155–166
11. Kaeli, D.R., Emma, P.G.,: Branch History Table Prediction of Moving Target Branches due Subroutine Returns. Proc. 18th Int. Symp. on Computer Architecture (1991) 34–41
12. Kalamatianos, J., Kaeli, D.R.: Predicting indirect branches via data compression. Proc. 31st Int. Symp. on Microarchitecture (1998) 272–281
13. Lee, J. K. F., Smith, A. J.: Branch Prediction Strategies and Branch Target Buffer Design. IEEE Computer Vol. 17(2) (1984) 6–22
14. Moure, J. C., Rexachs, D. I., Luque, E.: Optimizing a decoupled front-end architecture: the Indexed Fetch Target Buffer (iFTB). Lecture Notes in Computer Science, Vol. 2790. Euro-Par'03. Springer-Verlag, (2003) 566–575
15. Yeh, T.-Y., Patt, Y.: Two-Level Adaptive Branch Prediction. Proc. 24th Int. Symp. on Microarchitecture (1991) 51–61