Value Based BTB Indexing for Indirect Jump Prediction

Muhammad Umar Farooq, Lei Chen, and Lizy Kurian John

Department of Electrical and Computer Engineering, The University of Texas at Austin
ufarooq@mail.utexas.edu, lei777@gmail.com, ljohn@ece.utexas.edu

Abstract

History-based branch direction predictors for conditional branches are shown to be highly accurate. Indirect branches however, are hard to predict as they may have multiple targets corresponding to a single indirect branch instruction.

We propose the Value Based BTB Indexing (VBBI), a correlation-based target address prediction scheme for indirect jump instructions. For each static hard-to-predict indirect jump instruction, the compiler identifies a 'hint_instruction', whose output value strongly correlates with the target address of the indirect jump instruction. At run time, multiple target addresses of the indirect jump instruction are stored and subsequently accessed from the BTB at different indices computed using the jump instruction PC and the hint_instruction output values. In case the hint_instruction has not finished its execution when the jump instruction is fetched, a second and more accurate target address prediction is made when the hint_instruction output is available, thus reducing the jump misprediction penalty.

We compare our design to the regular BTB design and the best previously proposed indirect jump predictor, the tagged target cache (TTC). Our evaluation shows that the VBBI scheme improves the indirect jump target prediction accuracy by 48% and 18%, compared with the baseline BTB and TTC designs, respectively. This results in average performance improvement of 16.4% over the baseline BTB scheme, and 13% improvement over the TTC predictor. Out of this performance improvement 2% is contributed by target prediction overriding which is accurate 96% of the time.

Keywords: indirect branches, correlation-based branch prediction, compiler guided branch prediction, branch target prediction.

1 Introduction

Branches can be classified into conditional or unconditional and direct or indirect. Branch prediction research has shown that for a conditional branch instruction, its direction can be predicted with high accuracy [12] [19] [26] [27]. Furthermore, a direct branch has a fixed target encoded in the instruction, thus making conditional or unconditional direct branches easier to predict. Unlike direct branches, indirect branch instructions are hard to predict as they may have multiple targets corresponding to a single indirect branch instruction. Function call returns are a special type of indirect branch predicted accurately using a return address stack [14]. Although not as frequent as conditional direct branches, indirect jumps are becoming more common with the increase in programs written in object oriented programming languages such as Java, C++ and C#. Some common programming constructs including virtual function calls, switch-case statements and function pointers are implemented using indirect jump instructions. Predicated execution [1] [17] further increases the significance of indirect jump prediction since it eliminates hard to predict direct conditional branches. With the increase of indirect branches and their high misprediction rate, indirect branch misprediction penalty is becoming a sizable fraction of overall branch misprediction penalty. Figure 1 shows the number and percentage of indirect branch mispredictions per 1K instructions (MPKI) for the simulated benchmarks with a 4-way, 4K-entry BTB. Realizing the performance impact due to indirect branch misprediction, recent processors from Intel, including Intel Pentium M, have an indirect branch predictor [10].

Indirect branch target prediction research has focused mainly on two approaches – history-based [3] [6] [7] [8] [15] [16] and precomputation-based [23]. In a history-based target prediction scheme, specialized jump predictors use target history of indirect branches to predict the next target address. Target precomputation, on the other hand, dy-

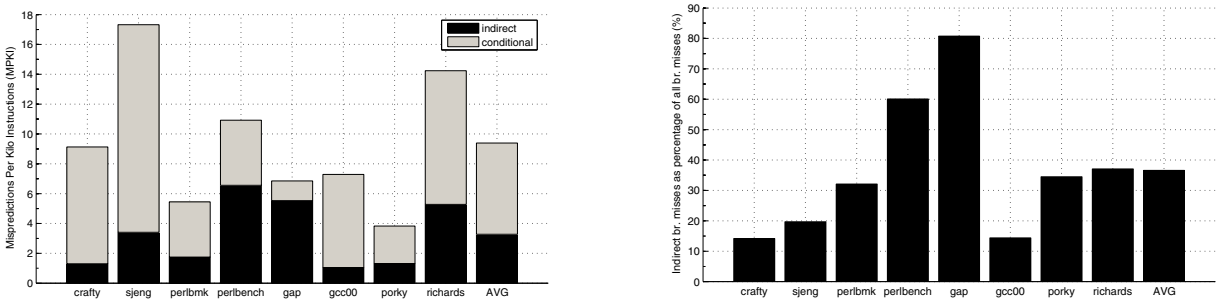

Figure 1. MPKI for conditional and indirect branches (left), indirect branch mispredictions as percentage of all branch mispredictions (right)

namically captures the target generation process and uses a small execution engine to calculate the target before the corresponding jump instruction is fetched. Despite using specialized hardware, prediction accuracy of existing indirect jump prediction techniques remains low.

This paper proposes *Value Based BTB Indexing (VBBI)*, a target address prediction mechanism for indirect jump instructions. For every hard-to-predict static indirect jump instruction, the compiler identifies an instruction whose output value strongly correlates with the target address taken by the jump instruction. When this correlated instruction (which we refer to as *'hint_instruction'*) produces a value (which we refer to as *'hint_value'*), we store the calculated target address of the indirect jump instruction in the BTB at an index computed by hashing the PC of the jump instruction with the hint_value.[1] Different hint_values correspond to different target addresses of a static jump instruction, and these targets are stored in the BTB at different indices computed by hashing the PC of the jump instruction with the corresponding hint_values. Next time when the jump instruction is fetched, we index the BTB using its PC and the new hint_value to get the predicted target address. In order to maintain strong correlation between the target and the hint_value, the current hint_value should be used. In cases where the latest hint_value is not available when the jump instruction is fetched, target prediction is made using *old* hint_value. However, in these cases a *second* and *more accurate* target prediction is made using the new hint_value when it becomes available. A more accurate second prediction effectively reduces the impact of jump mispredictions on performance by decreasing the jump misprediction penalty.

Our evaluation shows that VBBI prediction scheme improves the indirect jump prediction accuracy over the commonly used BTB based prediction by 60% for a less aggressive 2-wide 8-stage pipeline processor, and by 44% for a more aggressive 4-wide 24-stage pipeline processor. The average IPC is improved by 15.47% for the more aggressive processor, and by 9.53% for the less aggressive processor. We also show that target prediction overriding achieves average prediction accuracy of above 96%, resulting in an average saving of 35 cycles for each successfully overridden misprediction. We extensively compared VBBI prediction scheme with a previously proposed tagged target cache (TTC) predictor [3], which a recent study [13] showed to be the best indirect jump predictor for a similar set of benchmarks. We found that VBBI prediction scheme improves the indirect jump prediction accuracy by 18% on average over TTC, resulting in 13% performance (IPC) improvement.

This paper makes the following contributions:

1. We propose Value Based BTB Indexing (VBBI): a low cost, compiler guided, correlation based indirect jump target address prediction mechanism with high target prediction accuracy. Multiple targets of a jump instruction are stored in the BTB at indices computed by hashing the PC of the jump instruction with the output value of the correlated instruction.

2. We propose target address prediction overriding using a *second* and *more accurate* target address prediction. To our knowledge, target address prediction overriding has not yet been proposed. Prior studies have proposed branch direction outcome overriding for conditional branches [4] [9].

3. We showed that the VBBI can be implemented in an aggressive out-of-order processor using existing BTB with little extra hardware support.

[1] *hint_instruction* refers to any normal instruction in the program whose output value has strong correlation with the target address of an indirect jump instruction, and *hint_value* refers to the output value of the hint_instruction.

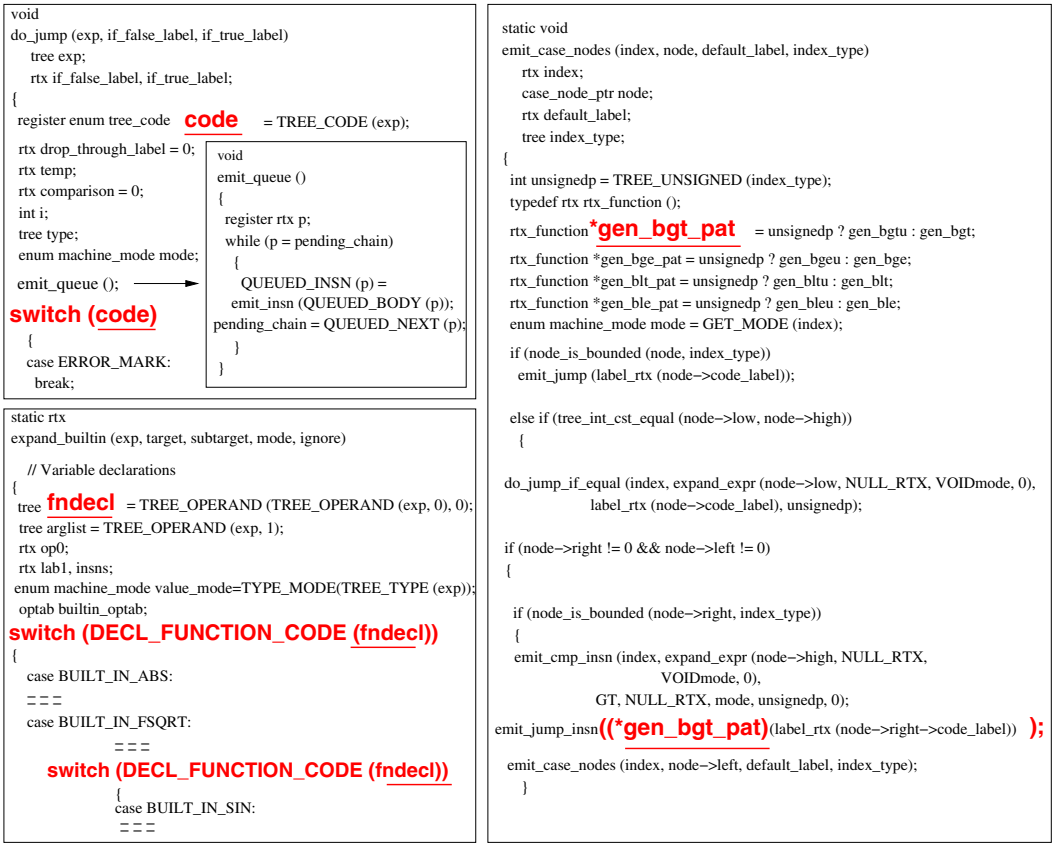

Figure 2. Code examples showing the distance between an indirect jump instruction and its associated 'hint instruction'. Code snippets are taken from 176.gcc benchmark in SPEC CPU2000 suite.

2 Why does VBBI work?

Figure 2 shows code examples applicable to the VBBI prediction scheme. These examples are taken from 176.gcc benchmark in SPEC CPU2000 suite. Switch-case statements in the two examples on the left will be compiled into indirect jumps. The 'definition' and 'use' of the case variables which these indirect jumps are dependent on are underlined. In the example on the top left, a function is invoked between the 'def' and 'use' of the case variable. This provides enough dynamic instructions for the case variable assignment operation to be completed before the subsequent indirect jump instruction is fetched. Similarly, in the example on the bottom left, the same case variable is used by two nested switch-case statements, each of which at increasing distance from the case variable assignment operation. Finally, the example on the right produces an indirect jump instruction due to a function call made using function pointer. The function pointer assignment and the pointer function call are separated by several calls to other functions. We denote the definition of the highlighted variable as the hint_instruction. For the simulated benchmarks, Table 1 shows the average number of dynamic instructions between a hint_instruction and its corresponding indirect jump instruction. Higher the number of dynamic instructions between the hint-jump instruction pair, higher the likelihood that the current hint_value will be available before the jump instruction is fetched.

3 Related Work

Previous research on indirect branch prediction has focused on two approaches: history-based [3] [6] [7] [8] [15] [16] [20], and precomputation-based [23]. Lee and Smith [20] used branch target buffer (BTB) to predict indirect branches. A BTB predicts the last taken target of the branch as the current target. This scheme works well for direct conditional or unconditional branches since they have only one taken target. Indirect branches often have multiple targets for the same static instruction. Therefore, a BTB-based predictor, though simple in design, is inaccurate for predicting indirect branches with multiple targets. Chang et al. [3] proposed a branch history based two-level predictor known as the "target cache" for predicting the target address of in-

Table 1. Average number of dynamic instructions between a hint_instruction and its corresponding indirect jump instruction

	crafty	sjeng	perlbmk	perlbench	gap	gcc00	porky	richards	AVG
Avg. dynamic instr. count between hint and jump instr.	78	39	7	18	24	43	21	38	33

direct jumps. It uses the same concept as a two-level branch direction predictor [26]. Target addresses from recently executed indirect jump instructions are recorded in a target history register. When an indirect jump is fetched, the fetch address and the target history register is used to index in the target cache to predict the next target address. Upon retiring the indirect jump, the target cache entry and the target history register is updated with the actual target address. Several configuration parameters such as size of target history register, number of target address bits shifted in the target history register, bit position from where these bits are taken etc. have direct impact on the performance of target cache predictor. Section 6.3 provides experimental evaluation of the target cache design with various parameter settings.

Li et al. [21] proposed rehashable BTB (R-BTB) scheme that dynamically identifies hard-to-predict indirect branches and stores their targets in the traditional BTB using a different indexing function. This new index is computed by XOR-ing the target history register with the jump PC. Other branches including the easy-to-predict indirect branches continue to index the BTB using their PC value.

Driesen et al. [7] [8] combined multiple target predictors using a cascaded predictor. Cascaded predictor is a hybrid predictor consisting of a simple predictor for easy-to-predict indirect branches, and a more complex predictor for hard-to-predict indirect branches.

Kalamatianos et al. [15] proposed predicting indirect branches via data compression. Their predictor uses the prediction by partial matching (PPM) algorithm of order three, which is a set of four Markov predictors of decreasing size, indexed by an indexing function formed by a decreasing number of bits from previous targets in the target history register.

Recently, Kim et al. [16] proposed VPC prediction which uses the existing conditional branch predictor for predicting the indirect branches. Conceptually, VPC treats an indirect branch instruction with *t* number of targets as *t* direct branches, each with its own unique target address. When an indirect jump instruction is fetched, the VPC prediction algorithm accesses the conditional branch predictor for *MAX_ITER* times, each time as a different *'virtual direct branch'* of the same indirect branch. This iterative process stops either when a *'virtual direct branch'* is predicted to be taken, or *MAX_ITER* number is reached, in which case the processor is stalled until the indirect branch is resolved. Here, *MAX_ITER* determines the number of attempts made to predict an indirect branch. Each attempt takes one cycle during which no new instruction is fetched. Results from a recent study (Figures 13 and 14 in [13]) show that performance of VPC prediction degrades significantly for workloads with higher number of dynamic targets.

Precomputation-based target prediction tries to calculate the target address in anticipation of having to make a prediction. Roth et al. [23] proposed a precomputation-based prediction scheme specifically for virtual function calls. It dynamically captures the sequence of instructions involved in the target generation process. Whenever the first instruction in the sequence completes, it quickly executes the rest of the instruction sequence using a separate execution engine, and computes the target before the actual call instruction is encountered. Although this technique avoids using specialized jump predictor, it requires significant hardware for capturing the target generation instructions along with a fast execution engine to pre-compute the target. Furthermore, this technique is very specific to v_call's target prediction, as their target generation process consists of a fixed pattern of three dependent loads followed by an indirect call.

Kaeli et al. [14] proposed the case block table (CBT), a hardware table that stores case addresses of a switch-case statement. When a switch-case statement is encountered, instead of executing a series of conditional branches it uses the value of the case block variable to fetch the case address from the CBT. By jumping directly to the appropriate case block instead of executing a series of conditional branches, the dynamic instruction count is reduced by 9%. However, modern compilers already generate the jump table corresponding to the switch-case statement if the number of cases exceeds certain threshold [3]. If the value of the case variable is not known, CBT redirection is delayed until it is available, resulting in performance degradation for deeply pipelined superscalar processors.

Finally, Joao et al. [13] proposed dynamic predication of hard-to-predict indirect jump instructions. When a hard-to-predict indirect jump instruction is fetched, the processor starts fetching from N different targets of the jump instruction, thereby increasing the probability of fetching from the correct target path at the expense of executing more instructions. They showed that N=2 is a good trade-off between performance and complexity.

Our approach is different from the above mentioned history-based or precomputation-based schemes in that we use correlation-based target address prediction for indirect

branches. The compiler identifies an instruction whose output strongly correlates with the target taken by the indirect jump instruction. At run time, multiple targets of the jump instruction are stored in the BTB at an index computed by hashing the output of the correlated instruction with the jump PC. We compared our results with the best performing history-based indirect branch predictor.

4 Value Based BTB Indexing (VBBI) Prediction

4.1 Overview

VBBI prediction scheme relies on the compiler to identify hint-jump instruction pair for hard-to-predict indirect jump instructions in the program. Multiple jump instructions can be bound to the same hint instruction. Different targets of an indirect jump instruction are stored in the BTB at different indices computed by hashing the hint_value with the PC of the corresponding jump instruction. When a hint instruction is executed, its output value is stored in a buffer. Subsequently, when the corresponding jump instruction is fetched, it reads the hint_value and uses it to compute the BTB index *'i'* for predicting the target address. When the jump instruction commits, the BTB is updated with the correct target (if different from the predicted target) using the same index *'i'*.

4.2 Implementation Details

4.2.1 Compiler Support

In our proposed VBBI prediction scheme, the compiler analyzes the source code to identify the 'last definition' of the variable on which an indirect jump instruction is dependent. During code generation, *offset* of the jump instruction from the instruction holding the 'last definition' (i.e. the hint_instruction) is encoded within the indirect jump instruction. Since indirect jump instructions specifies their target using an architectural register instead of an absolute address, some of the bits in the instruction encoding are unused and are available for providing hints [5]. Figure 3 shows the Alpha ISA jump instruction format augmented with the VBBI related hint information. If there are more than one path defining the variable, profiling can be used to select the most frequently executed path. For all the benchmarks we have simulated, we have not found any case where hint variable is reaching the indirect jump instruction through multiple paths.

4.2.2 Hardware Support

Figure 4 shows the overall operation of the VBBI prediction scheme. Central to the operation is a hardware structure called the *Hint Instruction Buffer (HIB)*. Each entry in the HIB structure has 3 fields, jump instruction PC (jmp_pc), corresponding hint_instruction PC (hint_pc), and

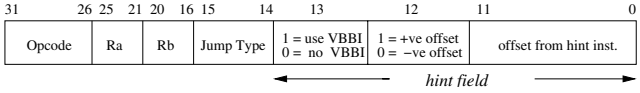

Figure 3. Alpha ISA jump instruction format augmented with the VBBI related hint information.

the hint_value. HIB is accessed in two different pipeline stages: fetch and write-back. In the fetch stage, indirect jump instructions read the hint_value from the HIB to form the BTB index, while other instructions access HIB to see if they are the hint_instruction for an indirect jump instruction. In the write-back stage, instructions (only those that are marked as hint_instruction) write their output value into the HIB.

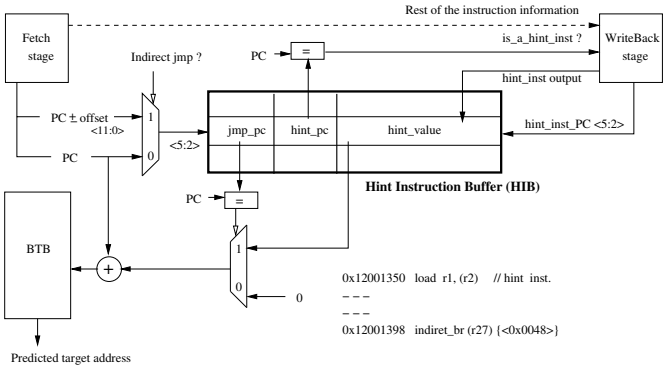

Figure 4. VBBI Prediction Hardware

When an indirect jump instruction with the 'use VBBI' bit (bit 13 in the fetched branch instruction) ON is fetched with no corresponding entry already in the HIB, an entry is created in the HIB at an index computed using the hint_instruction's PC. The hint_instruction PC can be calculated as jump PC $\pm$ offset value, with offset value provided in the lower 12 bits of the instruction word[2]. The jump instruction places part of its PC as tag in the jmp_pc field, and (jump PC $\pm$ offset value) as another tag in the hint_pc field. When a non-jump instruction is fetched, it will use its PC to index into the HIB. If the hint_pc field matches the instruction's PC, it is marked as a hint_instruction. When a hint_instruction finishes execution, its output is written in the hint_value field of the HIB entry. When the indirect jump instruction is fetched again, it reads the hint_value field and forms the BTB index by hashing the hint_value with its PC. When the jump instruction commits, the BTB is updated with the actual target (if different from the predicted target) using the same BTB index computed in the fetch stage.[3]

[2]Reason for using hint_instruction PC instead of jump instruction PC for computing the HIB index is that the same HIB entry will be accessed by hint_instruction as well for updating the hint_value.

[3]Update of the hint_value is done at the write-back stage, when the

Table 2. Characteristics of evaluated benchmarks

	crafty	sjeng	perlbmk	perlbench	gap	gcc00	porky	richards	AVG
Static Indir. br.	11	8	32	79	35	79	257	18	65
Dynamic Indir. br. (K)	222	551	779	956	1237	195	585	977	687
Indir. br. selected for VBBI	6	5	1	5	4	8	2	1	4
% Indir. misses covered by selected VBBI branches (%)	98	96	85	95	99	80	81	99	92
Baseline IPC	0.92	0.63	0.62	0.64	0.72	0.88	0.79	0.62	0.73

4.3 Target Prediction Overriding

For the VBBI target address prediction to be more accurate, the hint_instruction should have finished its execution before the subsequent indirect jump instruction is fetched. In cases where the jump instruction is fetched before the latest hint_value is written to the HIB entry, the jump instruction will use *old* hint_value to compute the BTB index. When the latest hint_value becomes available, another prediction is made using the updated value, if it is different from the old hint_value. This prediction will override the initial prediction if they are different. A successful target address prediction overriding will reduce the jump misprediction penalty by providing a more accurate prediction, cycles before the resolution.

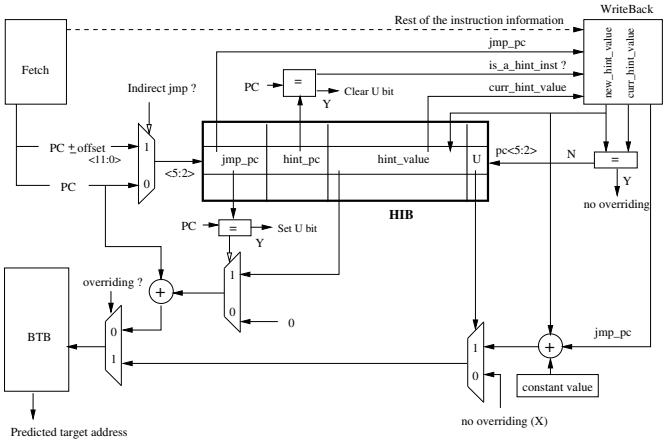

Figure 5. VBBI Prediction Hardware with Target Prediction Overriding Support

Figure 5 shows the augmented VBBI prediction hardware with support for target prediction overriding. Each HIB entry includes a 1-bit *Used (U)* flag. It indicates whether or not the jump instruction has *used* the old hint_value in making the target address prediction. When a hint_instruction is fetched, the associated U-bit is cleared. It is set when a subsequent associated jump instruction is fetched. When the hint_instruction finishes its execution, and the new hint_value is different from the old hint_value, the U-bit is checked to see whether or not the jump instruction has already been fetched and has used the old hint_value. If this is the case, a *second* target address prediction is made using the BTB index formed by hashing the new hint_value with the jmp_pc and a constant value. The reason for hashing with a constant value is explained below.

Update of BTB: In a VBBI scheme with target prediction overriding, the BTB is logically partitioned into two halves. The upper half stores the actual target at an index computed from the *old* hint_value which greatly helps the initial prediction when the jump is fetched but the *new* hint_value is not yet available. The lower half stores the actual target at a different index computed using the *new* hint_value which is highly correlated with the actual target. It is utilized when the hint_value is available for prediction overriding. Update and lookup in the lower half of the BTB is done by hashing the computed index with an additional constant value, which we set to be equal to half the number of sets in the BTB.

4.4 Implementation Cost

Not every static indirect jump instruction in the program needs to be predicted using the VBBI prediction scheme. Through profiling, compiler identifies hard-to-predict indirect jump instructions. This information is passed to the hardware using jump instruction encoding (see bit 13 in Figure 3). At run time, only those jump instructions that are marked by the compiler will need to occupy entries in HIB. Based on our observation, a small set of the static indirect jump instructions account for the majority mispredictions. For the selected benchmarks shown in Table 2, on average there are 65 static indirect jumps for each application. Selecting 4 out of the 65 indirect branches covers 92% of the BTB indirect jump mispredictions. We implemented a 16-entry HIB. Each HIB entry consists of two 26-bit tags (jmp_pc and hint_pc), a 12-bit hint_value, and a 1-bit U-bit flag. The total size of the HIB is 1040 bits. As shown in Figure 5, we added a control logic consisting of a few muxes and comparators to access the BTB for making predictions with or without target prediction overriding.

5 Experimental Methodology

5.1 Simulation Methodology

We extended SimpleScalar [2] to evaluate the VBBI prediction scheme. Table 3 shows the baseline parameters for

instruction type is known. Only those instructions producing results need to access the HIB.

Table 4. MPKI: baseline vs VBBI

	crafty	sjeng	perlbmk	perlbench	gap	gcc00	porky	richards	AVG
Cond. br. MPKI (baseline)	7.8	13.9	3.7	4.3	1.3	6.2	2.5	8.9	6.1
Indir. br. MPKI (baseline)	1.3	3.4	1.7	6.6	5.5	1.1	1.3	5.3	3.3
Cond. br. MPKI (VBBI)	8.0	14.6	3.7	4.9	1.0	6.3	2.6	8.4	6.2
Indir. br. MPKI (VBBI)	0.5	1.7	1.2	4.5	0.0	0.5	0.7	3.0	1.5

Table 3. Baseline processor parameters

Pipeline depth	16 stages;
Instr. Fetch	4 instructions per cycle; fetch ends at first pred. taken br;
Execution Engine	4-wide decode/issue/execute/commit; 512-entry RUU; 128-entry LSQ;
Branch Predictor	12KB hybrid pred. (8K-entry bimodal and selector, 32K-entry gshare); 4K-entry, 4-way BTB with LRU repl.; 32-entry return addr. stack; 15 cycle min. br mispred. penalty;
Caches	16KB, 4-way, 1-cycle L1 D-cache; 16KB, 2-way, 1-cycle L1 I-cache; 1MB, 8-way, 10-cycle unified L2 cache; All caches have 64B block size with LRU replacement policy;
Memory	150-cycle memory latency (first chunk), 15-cycle (rest);

our processor. Branch mispredictions are resolved in the write-back stage. Our workload includes 4 SPEC CPU2000 INT benchmarks [24], 2 SPEC CPU2006 INT benchmarks. We use 2 other C++ benchmarks, *richards* [25] and *porky* [18]. *Richards* simulates the task dispatcher in the kernel of an operating system, and *porky* is a middleware that makes various transformations in the SUIF compiler. We use those benchmarks in SPEC INT 2000 and 2006 suites that gives at least 5% performance improvement with a perfect indirect branch predictor.

We use the SimPoint approach [22] to find a representative program execution slice for each benchmark using the reference input data set. All binaries are compiled using gcc-4.1.1 with -O2 optimization running on Compaq Tru64 UNIX V5.1B. Each benchmark is run for 100M Alpha instructions. Table 2 shows the characteristics of simulated SimPoint for each benchmark.

5.2 Identifying Hint Instructions

For the experiments in this paper, we profiled the application to identify indirect branches that are hard-to-predict using traditional BTB based prediction. For each static hard-to-predict indirect branch instruction in assembly language, we identify its corresponding location at the source code level. We then use an algorithm similar to the one described in [11] to identify the 'last assignment' to the variable on which the indirect branch instruction is dependent. Finally, we trace the 'last assignment' to the variable back to the assembly language instruction, which we call the 'hint instruction'. Note that, hard-to-predict indirect branches can be dynamically identified with minor extensions to the existing BTB, thus avoiding the dependency on profiling.[4]

[4]In this case, the compiler will identify the hint for all indirect jumps, not just hard-to-predict ones. At run time, indirect jumps giving mispredictions will be predicted using VBBI instead of traditional BTB prediction.

6 Results

6.1 Performance of VBBI

Figure 6 (top) shows the indirect branch prediction accuracy for the baseline BTB and the VBBI prediction. On average, VBBI improves the prediction accuracy by 47.7% over the baseline BTB, from 51.7% (baseline BTB) to 76.4% (VBBI). Higher the number of instructions between hint and jump instruction, higher the improvement in prediction accuracy from the baseline (see Table 1 for average dynamic instruction count between a hint instruction and its corresponding indirect jump instruction). The *GAP* benchmark is an exception which we will explain later in the section. Table 4 shows the effect of VBBI on conditional and indirect MPKI. On average, VBBI reduces the indirect branch MPKI by 55% compared to baseline, with negligible impact on conditional branch MPKI.

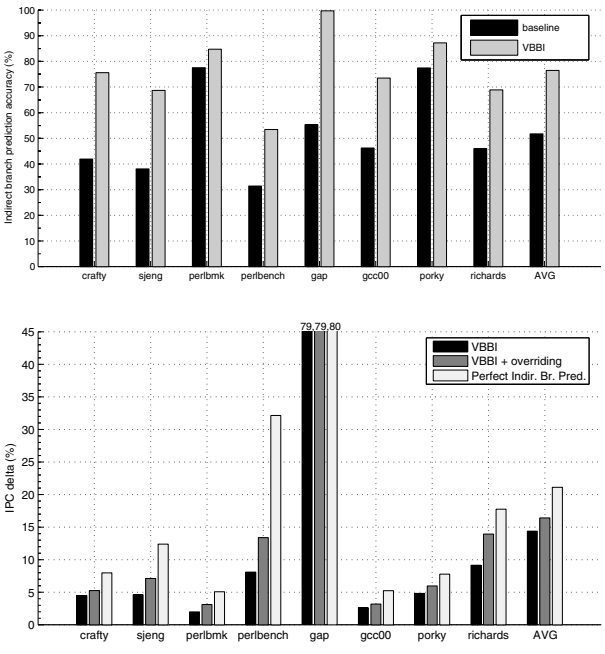

Figure 6. Performance of VBBI prediction: Indirect branch prediction accuracy (top), IPC improvement over baseline (bottom)

Figure 6 (bottom) shows the performance improvement of VBBI prediction over the baseline. VBBI predic-

tion achieves average performance improvement of 14.4%. When target prediction overriding is enabled, our scheme achieved an overall performance improvement of 16.42%. We also experimented with a perfect indirect branch predictor, which predicts all the indirect branch targets with a 100% accuracy. It shows an upper bound of performance improvement of 21.12% over baseline.

Analysis of the GAP benchmark: With VBBI prediction the *gap* benchmark achieves an almost perfect prediction accuracy of 99.6%. The benchmark analysis reveals that it has 4 dominant indirect jump instructions which produce nearly all of the indirect misses. Each indirect jump has only two targets taken alternatively, making it unpredictable by a traditional BTB. Although dynamic instruction count between a hint instruction and its corresponding indirect jump instruction is not very high, VBBI exploits the fact that the targets of the indirect jump alternate. Since there is no third target, each target is always preceded by the same previous target. As the hint_values are strongly correlated with the targets taken by a jump instruction, each target will have the same *old* hint_value in the HIB, if its current hint_value is unavailable when making the prediction. Thus the jump always uses the same *old* hint_value to lookup and update the BTB. In general, if a target T_a of an indirect jump is always preceded by another target T_b of the same jump, then T_a can be accurately predicted using the *old* hint_value i.e. the hint_value correlated to T_b. The Four indirect jumps have no obvious execution pattern, making it hard for a global history based indirect jump predictor to predict the targets accurately (see section 6.3 for results of TTC predictor). VBBI treats each jump separately (using its PC), therefore targets of one jump instruction do not affect the targets of the other jump instructions.

6.2 Performance of Target Prediction Overriding

It can be seen from Figure 6 (bottom) that target address prediction overriding helps close 30% performance gap between VBBI prediction and perfect indirect jump prediction. Figure 7 (bottom) shows that on average, target prediction overriding saves 35 cycles for each successfully overridden misprediction. This saving comes from the more accurate second target address prediction using the updated hint_value, cycles before the jump resolution. When making a second prediction, there are five possibilities, compared with the initial prediction.

1. **Already Correct:** The newly predicted target address is the same as the one predicted before when the jump was fetched. On average, this happens 65.56% of the overriding attempts.
2. **Useful:** Overriding target prediction is successful when the second predicted target address is correct and the jump was originally mispredicted. This will re-direct the fetch to the correct address, cycles before the jump resolution. On average, this happens 30.49% of the overriding attempts. Combined with *Already Correct* case, overriding target address prediction is correct 96% of the time. This shows the strong correlation between the output of the hint_instruction and the target address taken by the corresponding jump instruction.
3. **Neutral:** This happens when the jump was mispredicted but the overriding prediction does not predict the correct target address either. This happens 1.42% of the overriding attempts.
4. **Harmful:** This occurs when the overriding target address is different from the originally predicted target address, and the jump was initially correctly predicted. This will re-direct the fetch from the correct path to the wrong path until the jump is finally resolved. However, this harmful case happens only 1.07% of the overriding attempts.
5. **BTB Miss:** The second predicted target address is not present in the BTB. It happens 1.45% of the overriding attempts.

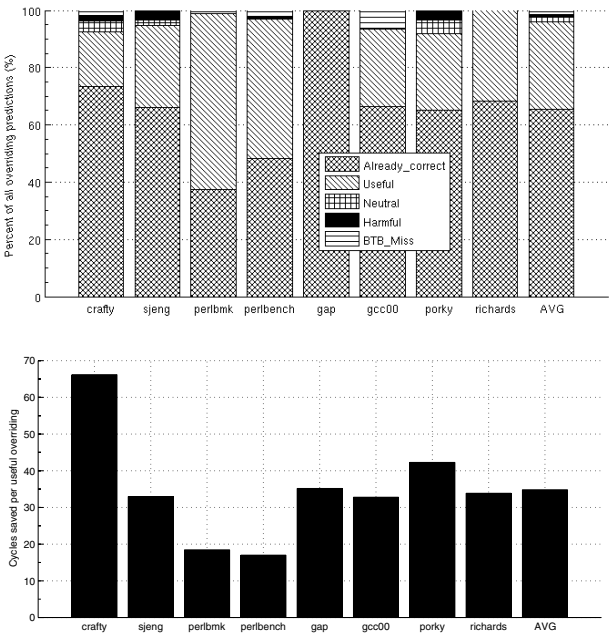

Figure 7. Performance of target prediction overriding: breakdown of all indirect target prediction overriding attempts (top), cycles saved per useful overriding (bottom)

6.3 Comparison with Tagged Target Cache

We compared VBBI prediction with a previously proposed tagged target cache (TTC) predictor [3], which a recent study [13] showed to be the best indirect jump predictor for a similar set of benchmarks (in Figure 17). Tagged target cache works as follows: When an indirect jump is fetched, the jump address and the target history register are used to form an index into the target cache. The target cache is accessed and the resident address is predicted as the target address. When the indirect jump retires, the computed target address for the jump is written into the target cache using the same index. When updating the history information, several bits from the target address are shifted into the target history register.

As shown in [3] [6], target cache performance depends on the size *s* of the global target history register, number of target address bits *t* shifted into the target history register, and the bit position *p* from where *t* bits are taken from the target address. We simulated various configurations for the tagged target cache design by varying values of *s, t* and *p*. Our configurations include two 4-way set-associative TTC designs with 512 sets (s=9) and 16k sets (s=14). Each TTC entry holds a 4-byte target address and a 2-byte tag, giving a total size of 12KB and 384KB for the two TTC designs respectively. Additionally, we explored different values of *t*={2,3,5,7,9} and *p*={2,3} for each TTC design.

Figure 8 shows the indirect branch prediction rate for the two TTC designs with different values of *t* and *p*. Figure 8 shows that for different benchmarks, TTC gives the best prediction accuracy for different values of *t* and *p*. For example, in Figure 8 (top), *t*=2 and *p*=2 gives best prediction accuracy for *perlbmk*, but worst prediction accuracy for *gcc00*. We selected the configuration that gives *the best average* prediction accuracy (shown by 'x' in Figure 8). On average, VBBI prediction improves indirect branch prediction accuracy by 25% over a 12KB TTC, and by 18% over a 384KB TTC. Figure 8 (bottom) compares the performance of VBBI predictor with TTC predictor. On average, VBBI improves the performance by 14.2% for a smaller 12KB TTC and by 13% for a larger 384KB TTC. VBBI outperforms TTC in all but one benchmark, *richards*, for which a 384KB TTC performs better than VBBI with a 1.01KB HIB. We also include a hypothetical case where each benchmark is simulated with the best value of *t* and *p* for that benchmark. On average, VBBI improves the IPC by 10.67% compared to this hypothetical case.

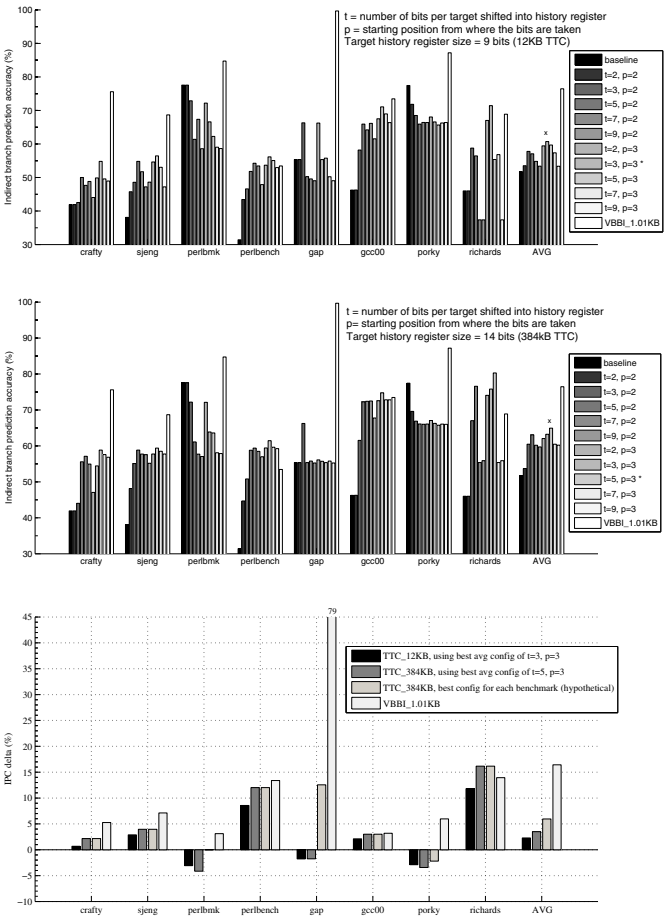

Figure 8. VBBI vs. tagged target cache (TTC): indirect branch prediction accuracy with 12KB TTC (top), with 384KB TTC (middle), performance improvement over baseline (bottom)

6.4 Sensitivity of VBBI to Microarchitectural Parameters

6.4.1 Different Issue Width and Pipeline Depth

Let $C_{h\_fetch \rightarrow j\_fetch}$ represents the number of cycles between the fetch of hint_instruction and the subsequent fetch of corresponding jump instruction, and $C_{h\_fetch \rightarrow h\_execute}$ represents the number of cycles between the fetch of hint_instruction and its execution. Processor issue width affects $C_{h\_fetch \rightarrow j\_fetch}$, and processor pipeline depth affects $C_{h\_fetch \rightarrow h\_execute}$. If $C_{h\_fetch \rightarrow j\_fetch}$ is larger than $C_{h\_fetch \rightarrow h\_execute}$, VBBI will make better prediction with updated values. Otherwise, VBBI will use *old* hint_value to compute BTB index which *may* result in a correct prediction. We evaluated VBBI prediction with different issue widths and pipeline depths. Figure 9 (top) shows that indirect jump prediction accuracy is higher for narrow issue width and short pipeline depth, and gradually decreases as the issue width and/or pipeline depth increases. On average, prediction accuracy ranges from 82.7% (for 2 issue, 8 stage pipeline) to 74.4% (for 4 issue, 24 stage pipeline). Figure 9 (bottom) shows the impact of different issue width and pipeline depth on IPC

improvement.

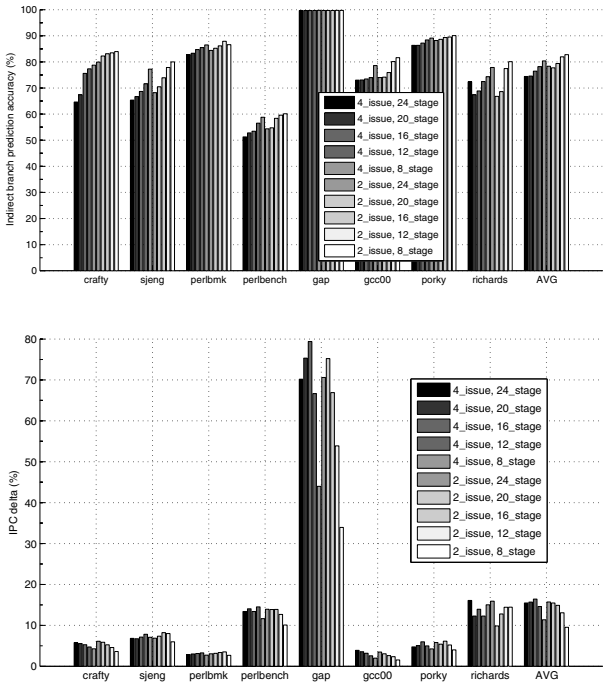

Figure 9. Performance of VBBI prediction for different issue width and pipeline depth: Indirect branch prediction accuracy (top), IPC improvement over baseline (bottom)

6.4.2 Different BTB Sizes

We evaluated VBBI prediction with BTB sizes of 1024, 2048 and 4096 entries. As can be seen from Table 5, VBBI prediction scheme achieves significant performance improvement even with smaller BTB sizes.

Table 5. Effect of different BTB sizes on VBBI performance

BTB entries	Baseline		VBBI Prediction	
	indirect MPKI	IPC	indirect MPKI	IPC Δ
1024	3.28	0.72	**1.66**	**15.86%**
2048	3.27	0.73	**1.60**	**16.22%**
4096	3.27	0.73	**1.53**	**16.42%**

7 Conclusion

Commonly-used BTB-based branch prediction scheme stores a single target per static branch instruction. It is not effective in predicting the targets of indirect jumps, which often have multiple targets per static indirect jump. This paper proposed and evaluated the *Value Based BTB Indexing (VBBI)* for predicting target addresses of indirect jumps. The key idea of the VBBI scheme is to store multiple targets of an indirect jump in the BTB at different indices computed by hashing the jump PC with the output value of a hint_instruction that strongly correlates with the target address taken by the indirect jump instruction. As such, VBBI enables the use of existing BTB structure to predict the targets of an indirect jump without requiring an extra structure specialized for storing multiple indirect jump targets. If the output value of the hint_instruction is not available when the corresponding jump instruction is fetched, a second target address prediction is made once the value becomes available. This overriding target address prediction reduces the penalty of indirect jump misprediction by providing a more accurate prediction, cycles before the jump resolution.

We evaluated VBBI on a set of 8 applications with significant indirect jumps. Our results show that VBBI improves indirect jump prediction accuracy by 48% over a commonly-used BTB-based indirect jump predictor, resulting in average performace improvement of 16.4%. We also compared VBBI against one of the best previously proposed indirect jump predictor, the tagged target cache (TTC). Our results show that compared to TTC, VBBI improves the prediction accuracy by 18%, resulting in 13% average performance improvement over TTC.

References

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of Control Dependence to Data Dependence. In *POPL-10*, pages 177–189, 1983.

[2] D. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997.

[3] P. Chang, E. Hao, and Y. N. Patt. Target Prediction for Indirect Jumps. In *ISCA-24*, pages 274–283, 1997.

[4] L. Chen, S. Dropsho, and D. H. Albonesi. Dynamic Data Dependence Tracking and its Application to Branch Prediction. In *HPCA-9*, pages 65–76, 2003.

[5] COMPAQ. Alpha Architecture Handbook, V4, Oct. 1998.

[6] K. Driesen and U. Hölzle. Accurate Indirect Branch Prediction. In *ISCA-25*, pages 167–178, 1998.

[7] K. Driesen and U. Hölzle. The Cascaded Predictor: Economical and Adaptive Branch Target Prediction. In *MICRO-31*, pages 249–258, 1998.

[8] K. Driesen and U. Hölzle. Multi-stage Cascaded Prediction. In *Euro-Par*, 1999.

[9] A. Farcy, O. Temam, R. Espasa, and T. Juan. Dataflow Analysis of Branch Mispredictions and its Application to Early Resolution of Branch Outcomes. In *MICRO-31*, pages 59–68, 1998.

[10] S. Gochman, R. Ronen, I. Anati, A. Berkovits, T. Kurts, A. Naveh, A. Saeed, Z. Sperber, and R. C. Valentine. The Intel Pentium M Processor: Micorarchitecture and Performance. In *Intel Technology Journal, 7(2)*, May 2003.

[11] M. Harrold and M. Soffa. Computation of Interprocedural Definition and Use Dependencies. In *International Conference on Computer Languages*, pages 297–306, Mar 1990.

[12] D. A. Jiménez and C. Lin. Dynamic Branch Prediction with Perceptrons. In *HPCA-7*, pages 197–206, 2001.

[13] J. A. Joao, O. Mutlu, H. Kim, R. Agarwal, and Y. N. Patt. Improving the Performance of Object-Oriented Languages with Dynamic Predication of Indirect Jumps. In *ASPLOS-13*, pages 80–90, 2008.

[14] D. R. Kaeli and P. G. Emma. Improving the Accuracy of History-Based Branch Prediction. *IEEE Transactions on Computers*, 46:469–472, 1997.

[15] J. Kalamatianos and D. R. Kaeli. Predicting Indirect Branches via Data Compression. In *MICRO-31*, pages 272–281, 1998.

[16] H. Kim, J. A. Joao, O. Mutlu, C. J. Lee, Y. N. Patt, and R. Cohn. VPC Prediction: Reducing the Cost of Indirect Branches via Hardware-Based Dynamic Devirtualization. In *ISCA-34*, pages 424–435, 2007.

[17] H. Kim, O. Mutlu, J. Stark, and Y. Patt. Wish Branches: Combining Conditional Branching and Predication for Adaptive Predicated Execution. In *MICRO-38*, pages 54–65, Nov. 2005.

[18] M. S. Lam. The SUIF Group. `http://suif.stanford.edu/`.

[19] C.-C. Lee, I.-C. K. Chen, and T. N. Mudge. The Bi-Mode Branch Predictor. In *MICRO-30*, pages 4–13, 1997.

[20] J. Lee and A. Smith. Branch Prediction Strategies and Branch Target Buffer Design. *Computer*, 17(1):6–22, Jan. 1984.

[21] T. Li, R. Bhargava, and L. K. John. Rehashable BTB: An Adaptive Branch Target Buffer to Improve the Target Predictability of Java Code. In *HiPC-02*, 2002.

[22] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder. Using SimPoint for Accurate and Efficient Simulation. In *SIGMETRICS '03*, pages 318–319, 2003.

[23] A. Roth, A. Moshovos, and G. S. Sohi. Improving Virtual Function Call Target Prediction via Dependence-Based Pre-Computation. In *ICS-13*, pages 356–364, 1999.

[24] SPEC. Standard Performance Evaluation Corporation. `http://www.spec.org`.

[25] M. Wolczko. Benchmarking Java with the Richards benchmark. `http://research.sun.com/people/mario/java_benchmarking/richards/richards.html`.

[26] T.-Y. Yeh and Y. N. Patt. Two-Level Adaptive Training Branch Prediction. In *MICRO-24*, pages 51–61, 1991.

[27] T.-Y. Yeh and Y. N. Patt. A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History. In *ISCA-20*, pages 257–266, 1993.