Compiler Support for Value-Based Indirect Branch Prediction*

Muhammad Umar Farooq[1], Lei Chen[2], and Lizy Kurian John[1]

[1] Department of ECE, The University of Texas at Austin
ufarooq@utexas.edu,ljohn@ece.utexas.edu
[2] Intel Architecture Group, Intel Corporation
lei777@gmail.com

Abstract. Indirect branch targets are hard to predict as there may be multiple targets corresponding to a single indirect branch instruction. *Value Based BTB Indexing (VBBI)*, a recently proposed indirect branch prediction technique, utilizes the compiler to identify a 'hint instruction', whose output value strongly correlates with the target address of an indirect branch. At run time, multiple targets are stored at different branch target buffer (BTB) locations indexed using the branch PC and the hint instruction output value.

In this paper, we present compiler support for the VBBI prediction scheme. We also propose compiler and run time optimizations to increase the dynamic instruction count between the indirect branch and its corresponding hint instruction. The more the dynamic instructions between the hint-jump instruction pair, the more likely that the hint value will be available when making the prediction.

Our evaluation shows that the proposed compiler and run time optimizations improve the VBBI prediction accuracy from 66% to 80%. This translates into performance improvement from 17.2% (baseline VBBI) to 24.8% (optimized VBBI) over the traditional BTB design and from 11% (baseline VBBI) to 17.3% (optimized VBBI) over the best previously proposed indirect branch prediction scheme.

Keywords: branch prediction, indirect branches, compiler guided branch prediction, compiler optimizations, compiler-microarchitecture interaction.

1 Introduction

Several high level programming language constructs such as virtual function calls, switch-case statements, function pointers are implemented using indirect branches. With object oriented programming languages gaining more popularity in various computing arenas, indirect branches will become more prevalent in future applications. As a result, whether or not the indirect branches can be

* This research was partially supported by NSF grant 1117895. The opinions and views expressed in this paper are those of the authors and not those of NSF.

M. O'Boyle (Ed.): CC 2012, LNCS 7210, pp. 185–199, 2012.
© Springer-Verlag Berlin Heidelberg 2012

accurately predicted will be a limiting factor of the overall system performance. This trend is recognized by commercial microprocessor manufacturers including Intel, whose recent processor includes a dedicated indirect branch predictor [8]. Figure 1 shows the mispredictions per 1K instructions (MPKI) for different applications using different indirect branch prediction schemes. On average, indirect branch mispredictions account for 38%, 31% and 22% of the overall mispredictions, using the branch target buffer (BTB) [13], the tagged target cache (TTC) [2] and the value-based BTB indexing (VBBI) [7] designs respectively.

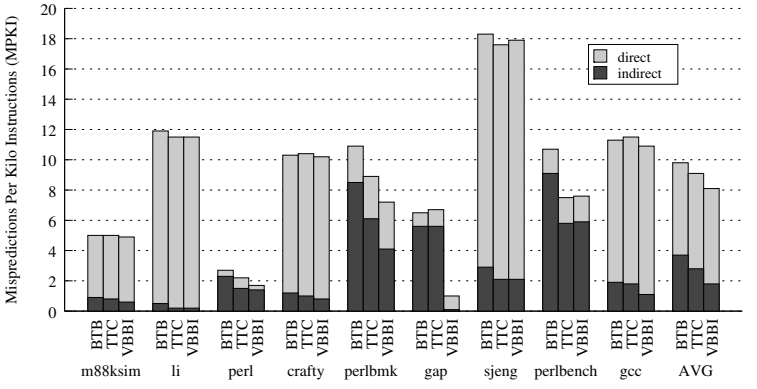

Fig. 1. MPKI for BTB, TTC and baseline VBBI prediction schemes

Prior research on indirect branch prediction has mainly focused on history-based target prediction schemes [2,4,5,6,11,12]. In these schemes, branch history information is used to distinguish different dynamic instances of the same indirect branch. These purely dynamic schemes have the advantage of not requiring compiler support and invisible to the software. However, hardware has limited view of program execution and may not be able to capture certain program behavior with reasonable cost.

The recently proposed VBBI scheme [7] shows that by tracing back the indirect branch data dependence chain, an instruction can be found whose output is directly related to the target taken by the indirect branch. This correlated instruction is referred to as the *hint instruction*, and its output as *hint value*. The key idea of VBBI is to store multiple targets of an indirect branch at different BTB indices computed by hashing the branch PC with the hint value.

Previous work on VBBI presented performance improvements without details on compiler implementation [7]. In this paper, we propose compiler support for the VBBI prediction scheme. For every static indirect branch instruction, the compiler analyzes the source code to find the 'most recent definition' of the variable on which the indirect branch is dependent. During code generation this information is encoded in the indirect branch to be used at run time. In order to maintain strong correlation between the target and the hint value, the current hint value should be used for making the prediction, i.e. the hint instruction should have finished its execution before the indirect branch is fetched. To this

end, we propose the compiler and run time optimizations for improving the VBBI prediction accuracy by increasing the dynamic instruction count between the hint instruction and the corresponding indirect jump instruction.

We show the performance improvement from these optimizations and compare with the traditional BTB design and the tagged target cache (TTC) design [2]. Our evaluation shows that the proposed compiler and run time optimizations improve the VBBI prediction accuracy from 66% (baseline VBBI) to 80% (optimized VBBI). In terms of performance, the optimized VBBI improves the performance from 17.2% (baseline VBBI) to 24.8% (optimized VBBI) over the traditional BTB design and from 11% (baseline VBBI) to 17.3% (optimized VBBI) over the TTC design.

This paper makes the following contributions:

1. We added compiler support for a recently proposed indirect branch prediction technique, the VBBI scheme. The implementation is based on GCC v4.2.1.
2. We propose compiler and run time optimizations to improve the VBBI prediction accuracy. These optimizations are applicable to similar schemes, as well as other design ideas exploiting data dependences and improving memory operations.

Rest of the paper is organized as follows. Section 2 gives the VBBI background. Compiler analysis for the VBBI scheme is introduced in section 3. Section 4 presents compiler and run time optimizations. Our simulation methodology is outlined in section 5. We discuss our results in section 6. Section 7 presents the related work and we conclude the paper in section 8.

2 Value Based BTB Indexing (VBBI) Background

The VBBI prediction scheme relies on the compiler to identify a *'hint instruction'* whose output value strongly correlates with the target taken by the indirect jump instruction. Dynamically, multiple targets are stored at different BTB locations indexed using the jump PC and the hint instruction output value. 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. When the branch commits, the BTB is updated with the correct target (if different from the predicted target) using the same index.

Figure 2 shows the overall operation of the VBBI prediction scheme. Each entry in the *Hint Instruction Buffer (HIB)* has 3 fields, branch PC (jmp_pc), corresponding hint instruction PC (hint_pc), and the hint value. HIB is accessed in fetch and write-back (WB) stages. In the fetch stage, indirect jump instructions read the hint value from the HIB to compute 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, hint instructions write their output value into the HIB.

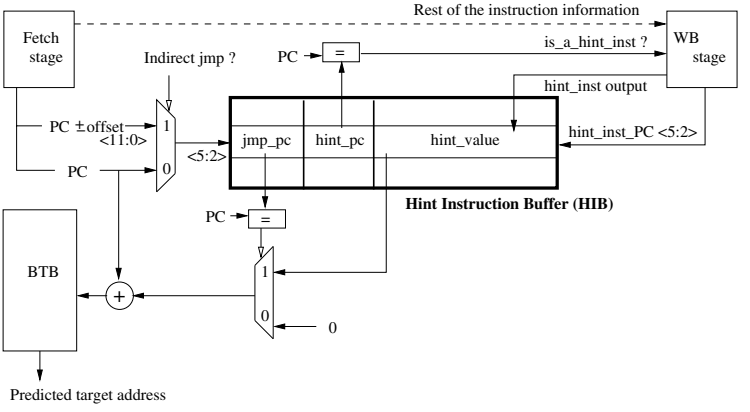

Fig. 2. VBBI Hardware Design (from [7])

Target Prediction Overriding. VBBI scheme will be more accurate, if the prediction is made using the *current* output of the hint instruction. In cases where the jump instruction is fetched before the hint instruction has produced its output, the jump instruction will use *stale* hint value for making the prediction. When the latest hint value becomes available, another prediction is made using the updated hint value, if it is different from the old value. This prediction will override the initial prediction and redirect the fetch to the correct path, cycles before the jump resolution.

3 Compiler Analysis for VBBI

VBBI prediction scheme relies on compiler to identify the 'most recent definition' of the variable on which an indirect jump instruction depends on. This variable can be a switch-case control variable, a pointer to a function or an object, etc. During code generation, *offset* of the jump instruction from the instruction holding the 'most recent definition' (i.e. the hint instruction) is encoded in the indirect jump instruction. Since an indirect jump instruction specifies its 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 [3].

Implementation Details. We modified GCC v4.2.1 to support hint instruction identification for the VBBI scheme. Figure 3 explains the modified passes using a hand written example code. In Figure 3(a), the switch-case statement will be compiled into an indirect jump instruction. The target taken by the jump depends on the definition of the underlined variable **p**.

1. **pass_uncprop:** This pass is executed before coming out of the SSA form (in pass_del_ssa). While the code is still in the SSA form, we modified this pass to identify the 'last definition' of the variable (i.e., the hint instruction), on which the indirect jump is dependent on. Figure 3(b) shows that in the SSA

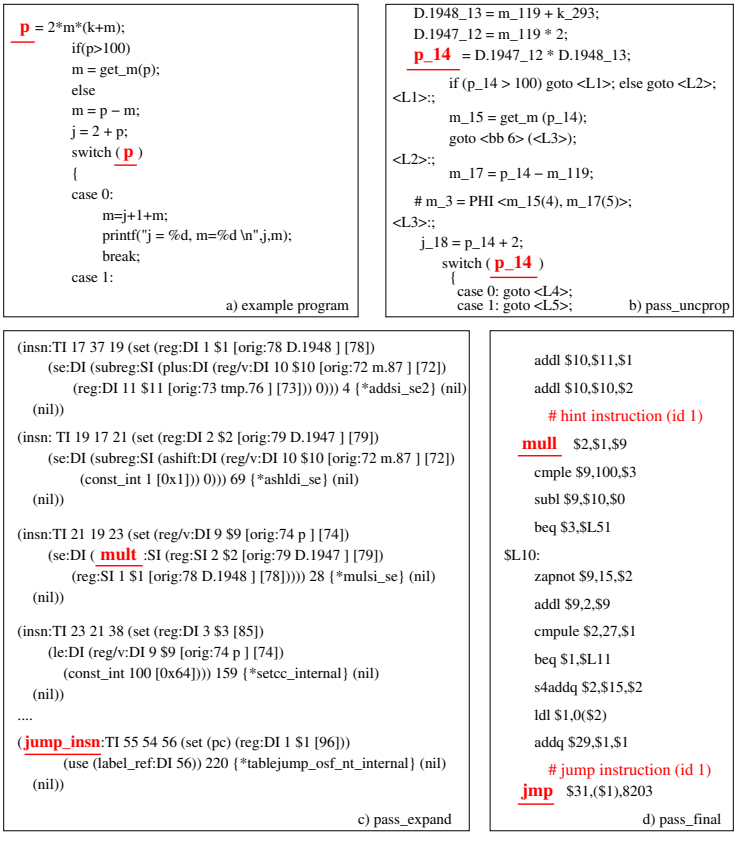

Fig. 3. Modified compiler passes for VBBI hint instruction analysis

form, the program variable **p** is renamed to **p_14**. All the uses of **p** including the switch-case statement reached by **p's** assignment are also renamed to **p_14**. Since in the SSA form, each USE has a unique DEF, we identify the hint instruction as the one containing the DEF of the SSA variable **p_14**.

2. **pass_expand:** This pass converts program statements from TREE format into the RTL code. For every hint instruction identified in the earlier pass, its corresponding RTL code is also marked as hint instruction. Figure 3(c) shows RTL code for some of the statements in the example program. Note the underlined **mult** code generated for statement 'p_14 = D.1947_12 * D.1948_13'. Similarly, the underlined **jump inst** code is generated corresponding to the 'switch (p_14)' statement.
3. **pass_final:** This pass looks at the list of instructions in the RTL format and outputs their corresponding assembly code. While going through the list of instructions, a counter keeps track of the number (and size) of instructions between the hint instruction and its corresponding jump instruction. This gives the PC offset of the hint instruction from the jump instruction, which

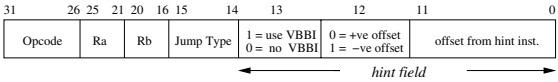

Fig. 4. Alpha indirect branch instruction augmented with VBBI hint bits

is then encoded in the jump instruction assembly code. Figure 3(d) shows the assembly code generated for the example program. The underlined **mull** instruction is the identified hint instruction for the underlined **jmp** instruction. The third argument of the **jmp** instruction is the VBBI related hint information encoded in the jump instruction format shown in Figure 4, i.e. bits 0 through 13 in the instruction. In this example, the hint information shows that the offset between the hint-jump instruction pair is 11 instructions (8203 = 2ˆ13 + 11).

4 Optimizations for Improving VBBI Prediction Accuracy

As indicated by the almost perfect prediction accuracy with VBBI target prediction overriding [7], the VBBI scheme is highly accurate when the current hint value is used, i.e. the hint instruction output is available when making the prediction. We apply compiler and run time optimizations to increase the dynamic instructions between the hint-jump instruction pair. This results in higher probability of the hint value being available when making the prediction. The techniques presented in this section are applicable to similar prediction schemes, as well as other design ideas exploiting data dependences to improve instruction level parallelism (ILP) and memory operation performance.

4.1 Compiler Optimizations

Instruction Hoisting. In this optimization, the hint instruction is moved away from its dependent jump instruction, thus creating more dynamic instructions between the hint-jump pair. Figure 5(a) shows a code example from SPEC95 099.go benchmark that is suitable for such an optimization. In this example, the switch-case statement will be compiled into an indirect jump instruction. The target taken by the jump instruction depends on the underlined computation of **shapes[sh].where** which can be hoisted up to the beginning of the **for** loop, thus increasing the dynamic distance between the hint instruction and the jump instruction.

Function Inlining. If the control variable of an indirect jump instruction is passed as a function argument, that function can be inlined to increase the dynamic instruction count between the hint and the jump instruction. Figure 5(b) shows a code example from SPEC2000 186.crafty benchmark that can benefit

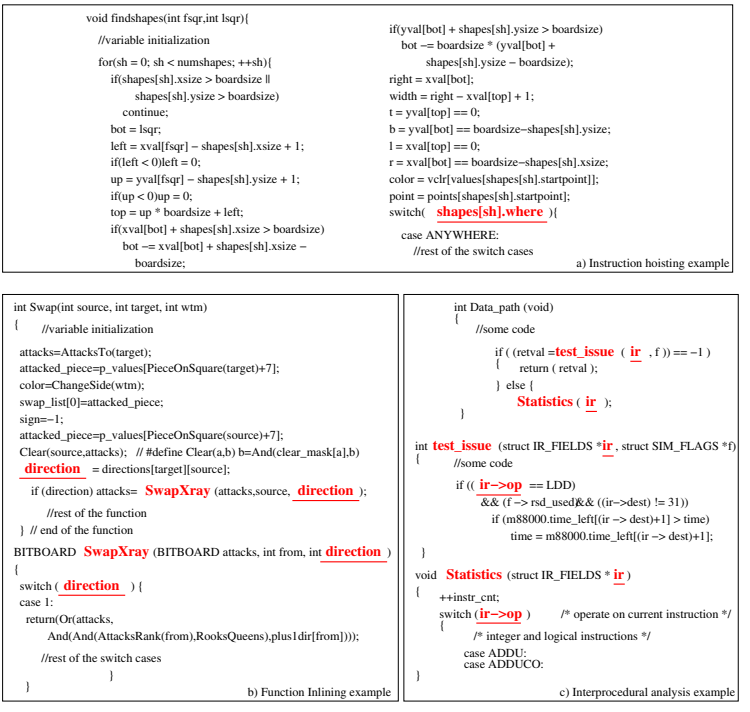

Fig. 5. Code examples showing compiler optimizations for VBBI

from function inlining optimization. In this example, the indirect jump instruction in function **SwapXray** depends on the underlined variable **direction** which is passed as an argument to the function. Without inlining, the instruction that POPS the variable **direction** from the stack will be marked by the compiler as the hint instruction. However, after inlining, this indirect jump instruction is now dependent on the *def* of the variable **direction** in function **Swap**. Note that this *def* of variable **direction** can be further moved to the beginning of the function using the previous instruction hoisting optimization.

Inter-procedural Dataflow Analysis. This optimization aims at identifying the hint instruction in a function other than the one containing the indirect jump instruction. Figure 5(c) shows a code example suitable for such an optimization. In this example, taken from SPEC95 124.m88ksim benchmark, the function **Data_path** calls two other functions, **test_issue** and **Statistics**, passing them the same pointer **ir**. The indirect jump instruction in function **Statistics** depends on the underlined computation **ir->op**. The same computation is also performed in an earlier function **test_issue**. Marking the hint instruction in function **test_issue** instead of **Statistics** greatly increases the possibility that the latest hint instruction outcome is available for making the prediction.

```
int Search(int alpha, int beta, int wtm, int depth, int ply, int do_null)
{
// some code
      while ((current_phase[ply]=(in_check[ply]) ? NextEvasion(ply,wtm) :
                                    NextMove (ply,wtm))) {
         extended_reason[ply]&=check_extension;
            // some other code
        MakeMove (ply, current_move[ply] ,wtm);
           //rest of the code
       } //end of while
 } //end of Search
```

```
void MakeMove (int ply,  int move , int wtm)
{
  register int piece, from, to, captured, promote;
    // some code
 piece = Piece( move ); // define Piece(a) (((a)>>12)&7)
    // some code
  switch ( piece ) {
      case pawn:
```

```
int NextMove (int ply, int wtm)
{
 register int *bestp, *movep, *sortv, temp;
 register int history_value, bestval, done, index;

 switch (next_status[ply].phase) {

 case HASH_MOVE:
  next_status[ply].phase=GENERATE_CAPTURE_MOVES;
  if (hash_move[ply]) {
   current_move[ply] = hash_move[ply];
   if (ValidMove(ply,wtm,current_move[ply])) return(HASH_MOVE);
   else printf("bad move from hash table, ply=%d\n",ply);
  }
 case CAPTURE_MOVES:
  if (next_status[ply].remaining) {
   current_move[ply] = *(next_status[ply].last);
   *next_status[ply].last++=0;
   next_status[ply].remaining--;
   if (!next_status[ply].remaining) next_status[ply].phase=KILLER_MOVE_
   return(CAPTURE_MOVES);
  }
  next_status[ply].phase=KILLER_MOVE_1;
```

Prediction Accuracy (%)			
Jump PC	**BTB only**	**VBBI baseline**	**VBBI with load–store address matching**
0x120022930	55%	61%	88%

Fig. 6. Load to store address matching example (taken from SPEC2000.186.crafty)

4.2 Hardware Optimization

Load to Store Address Matching. Dynamic tracking of load and store dependences has been used to support data speculation in code scheduling [9] [14]. We apply this technique to improve target prediction for indirect branches. When the hint instruction for an indirect jump is a *load* instruction, the address of the load instruction can be recorded in a *Hint Store Buffer (HSB)*. Subsequently, each store address is compared against load addresses in the HSB. If a match is found, the store value will be placed as the hint value in the corresponding HIB entry. Figure 6 shows an example taken from SPEC2000 186.crafty benchmark. In this example, the *while* loop in function *Search()* computes the next move by calling function *NextMove()*, which stores the computed move in **current_move[ply]**. Subsequently, **current_move[ply]** is passed as an argument to the function MakeMove() where **current_move[ply]** is used as the switch-case control variable. Dynamically tracking the load-store dependence and using the *store* value as opposed to the *load* value as the hint value increases the possibility of making the prediction using the latest hint value. As shown in Figure 6, the load-to-store address matching optimization improved the prediction accuracy of this jump to 88% compared to 61% when the hint instruction originally identified by the compiler was used for making predictions (VBBI baseline).

5 Simulation Methodology

We extended SimpleScalar [1] to simulate a 4-issue, 24-stage pipeline for evaluating VBBI prediction scheme. Table 2 shows the baseline parameters for our

Table 1. Characteristics of evaluated benchmarks

	m88ksim	li	perl	crafty	perlbmk	gap	sjeng	perlbench	gcc
Static Indir. br.	59	39	4	8	59	35	8	62	543
Dynamic Indir. br. (K)	161	62	289	215	1252	1238	532	1170	474
Baseline IPC	0.98	0.69	0.78	0.88	0.74	0.73	0.62	0.62	0.59

processor. Our workload includes nine benchmarks, three each from SPEC95, SPEC2000 and SPEC2006 suites [17]. Currently our compiler work for identifying hint instruction only support benchmarks written in C language. We plan to extend the support for C++ and Java benchmarks.

We use SimPoint [15] to find a representative program execution slice for each benchmark using the reference input data set. All binaries are compiled using modified GCC v4.2.1 with -O3 optimization running on Compaq Tru64 UNIX V5.1B. Each benchmark is run for 100M Alpha instructions. Table 1 shows the characteristics of simulated SimPoint for each benchmark.

Table 2. Processor parameters

Pipeline depth	Evaluated multiple configurations ranging from 8 to 24 stages;
Instr. Fetch	4 instructions/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; 16-entry HIB; 32-entry HSB;
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	300-cycle memory latency (first chunk), 15-cycle (rest);

6 Results

In this section we compare the performance of the baseline VBBI [7] with the optimized VBBI. Sections 6.1 and 6.2 use the traditional BTB and the TTC designs respectively as the reference point for comparison.

6.1 VBBI versus Traditional BTB

We compare the optimized VBBI prediction accuracy with the baseline VBBI and with the traditional BTB scheme in Figure 7. On average, the VBBI prediction accuracy is improved from 66% (baseline VBBI) to 80% (optimized VBBI).

Table 3. Average dynamic instruction count between hint-jump instruction pair

	m88ksim	li	perl	crafty	perlbmk	gap	sjeng	perlbench	gcc	AVG
VBBI baseline	45	58	10	63	12	31	47	15	51	37
VBBI optimized	168	110	16	158	43	39	174	24	64	88

Table 4. MPKI with traditional BTB, VBBI baseline and optimized VBBI

	m88ksim	li	perl	crafty	perlbmk	gap	sjeng	perlbench	gcc	AVG
Indir. br. MPKI (BTB only)	0.9	0.5	2.3	1.2	8.5	5.6	2.9	9.1	1.93	3.7
Indir. br. MPKI (VBBI baseline)	0.6	0.2	1.4	0.8	4.1	0.1	2.1	5.9	1.1	1.8
Indir. br. MPKI (VBBI optimized)	0.1	0.1	0.5	0.5	3.2	0.1	0.8	4.9	1.1	1.3

The BTB mispredictions are reduced by 2.1x using the optimized VBBI scheme. The increase in prediction accuracy is due to the fact that more predictions are made using the current and highly correlated hint value. Table 3 shows that the proposed optimizations increase the average dynamic distance between hint-jump instruction pair from 37 instructions to 88 instructions. Table 4 shows the number of mispredictions for indirect branches per 1K instructions (MPKI) for different prediction techniques. The optimized VBBI slashes the indirect branch MPKI by a third compared with traditional BTB design, and by a half compared with the baseline VBBI design.

Performance comparison of the optimized and the baseline VBBI scheme over the traditional BTB design is shown in Figure 8. For a 4-issue, 24-stage pipeline, the proposed VBBI optimizations enhance the baseline VBBI performance by 5.5%, achieving a 20.7% performance improvement over the traditional BTB design. When target prediction overriding is also enabled, the VBBI scheme achieves an overall performance improvement of 24.8% over the traditional BTB technique.

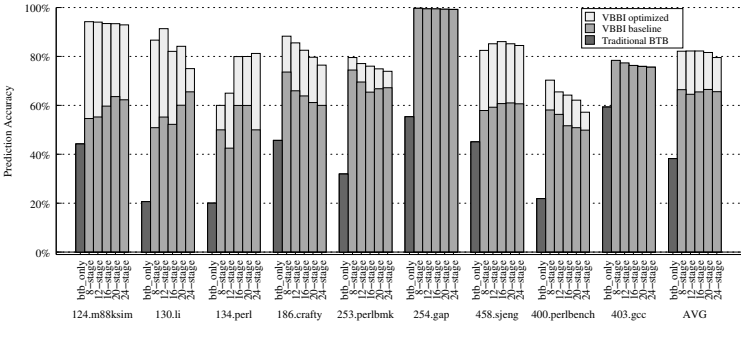

Fig. 7. VBBI Indirect branch prediction accuracy

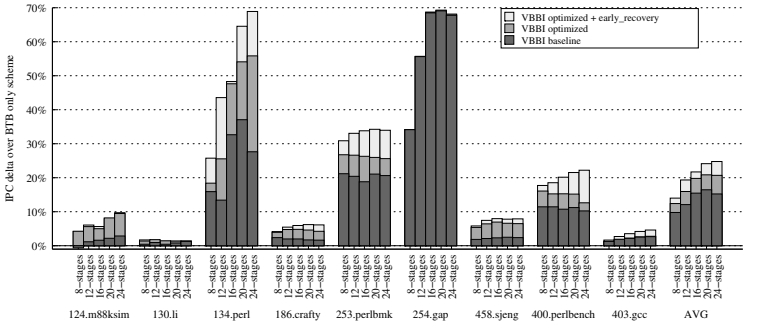

Fig. 8. IPC improvement of VBBI over traditional BTB scheme

6.2 VBBI versus Tagged Target Cache

We also compare the VBBI design with the TTC predictor [2], which is shown to be the best previously proposed jump predictor in a recent study by [10] (in Figure 17). In the TTC scheme, target addresses from recently executed indirect jump instructions are recorded in a target history register. When an indirect jump is fetched, the target cache is indexed using the XOR of the indirect branch PC and target history register, and the address stored at that index is predicted as the next target address. When the indirect jump retires, the computed target address is written into the target cache using the same index. When updating the history information, few bits from the target address are shifted into the global target history register. Farooq et al. [7] show that the TTC gives the best prediction accuracy with 14-bit global target history register. Upon update, 5 bits of target address (starting from the 3rd bit) are shifted into the target history register.

Figure 9 compares the VBBI prediction accuracy with the TTC design. On average, the optimized VBBI achieves a prediction accuracy of 80%, compared to 56% achieved by the best performing TTC configuration. Figure 10 compares the performance of the VBBI predictor with the TTC predictor. On average, the baseline VBBI outperforms the TTC design by 9.1% with just 130B of additional storage [7] compared to 384KB storage of the TTC design. The optimized VBBI further enhance the baseline VBBI performance by 4.7%, achieving 13.8% improvement over the TTC design. With target prediction overriding, VBBI achieves an overall performance improvement of 17.3% over the TTC predictor.

7 Previous Work

Lee and Smith [13] proposed branch target buffer (BTB) to predict indirect branches. This technique predicts the same target for the current execution of the branch that was taken in the last execution of that branch. Though simple in design, this scheme does not work well for indirect branches that may switch between multiple targets at run time.

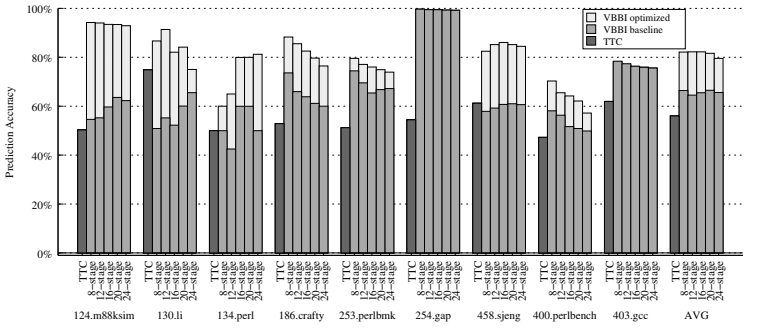

Fig. 9. Indirect branch prediction accuracy: VBBI prediction vs. Tagged Target Cache (TTC)

History based two-level indirect branch predictor was first proposed by Chang et al. [2]. This mechanism, known as 'target cache', uses the branch history information to distinguish different dynamic instances of the same indirect branch, a concept similar to 2-level conditional branch predictor [18]. When an indirect jump is fetched, the jump address and the global 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. Upon retiring the indirect jump, the target cache entry and the target history register is updated with the actual target address.

Driesen et al. [5] [6] focused on improving the indirect branch prediction accuracy by combining multiple 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. [11] proposed predicting indirect branches via data compression. Their predictor uses 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.

Kim et al. [12] utilized the existing conditional branch predictor for predicting indirect branches as well. The mechanism, known as the 'VPC prediction', treats an indirect branch instruction with t targets as t direct branches, each with its own unique target address. On fetching an indirect jump, the VPC prediction algorithm makes *MAX_ITER* attempts for predicting an indirect branch target, 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. *MAX_ITER* determines the maximum number of attempts made to predict an indirect branch. Each attempt takes one cycle during which no new instruction is fetched. A more recent study ([10] in Figure 13 and 14) shows that performance of VPC prediction degrades significantly for workloads with higher number of dynamic targets.

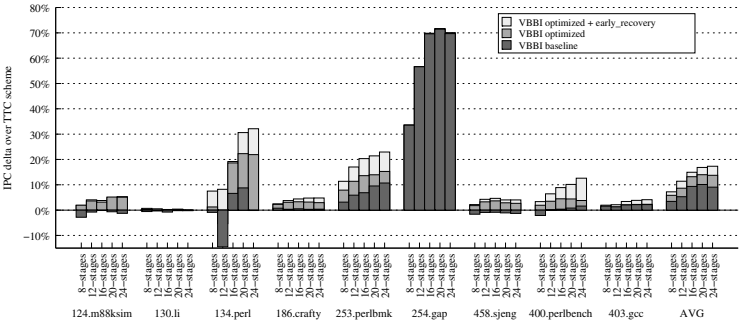

Fig. 10. Performance improvement: VBBI prediction vs. Tagged Target Cache (TTC)

Roth et al. [16] took a different approach for predicting indirect branch targets, precomputating them in anticipation of having to make a prediction. Proposed specifically for virtual function calls, the scheme dynamically captures the sequence of instructions involved in the target generation process. Whenever the first instruction in the sequence completes, it uses a separate, fast 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 target prediction of virtual function calls , as their target generation process consists of a fixed pattern of three dependent loads followed by an indirect call.

Joao et al. [10] proposed a new way of handling indirect jumps, dynamically predicating them. Instead of fetching from a single control path, when a hard-to-predict indirect jump instruction is fetched, the processor starts fetching from N different targets of the jump instruction. By fetching from more than one target, the processor increases 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.

Recently, Farooq et. al [7] proposed a compiler-guided, correlation-based target address prediction scheme that combines data dependences with indirect branch target prediction. The proposed technique, known as Value based BTB indexing (VBBI), relies on the compiler's ability to statically capture data dependences, and uses the hardware to exploit the correlation between the data and branch target at run time. The key idea of VBBI is to identify a 'hint instruction' whose output is highly correlated with the target taken by the jump. At run time multiple targets of an indirect branch are stored at different BTB indices computed by hashing the branch PC with the output of the hint instruction. They show that by off-loading dependence analysis to the compiler, the hardware predictor size can be kept much smaller.

8 Conclusion

The recently proposed VBBI prediction scheme uses a novel BTB indexing technique that allows multiple targets of an indirect branch to be stored at different BTB indices. This technique relies on the compiler to identify a *'hint instruction'* whose output strongly correlates with the target taken by the indirect branch. At run time multiple targets are stored at different BTB indices computed by hashing the branch PC and the hint instruction output value.

In this paper we propose the compiler support for identifying the hint instruction for the VBBI prediction scheme. We also propose the compiler and run time optimizations that improve the VBBI prediction accuracy by increasing the dynamic instructions between the hint-jump instruction pair. The more the dynamic instructions between this instruction pair, the more likely that the hint instruction outcome will be available when making the prediction.

Our evaluation shows that the proposed optimizations improve the VBBI prediction accuracy from 66% to 80%. Compared to traditional BTB design, this translates into average performance improvement from 17.2% (baseline VBBI) to 24.8% (optimized VBBI). We also compare the VBBI with the best previously proposed indirect jump predictor, the tagged target cache (TTC). Compared to the TTC design, the proposed optimizations improve the performance by 6.3%, from 11% (baseline VBBI) to 17.3% (optimized VBBI).

References

1. Burger, D., Austin, T.M.: The SimpleScalar Tool Set, Version 2.0. SIGARCH Comput. Archit. News 25(3), 13–25 (1997)
2. Chang, P., Hao, E., Patt, Y.N.: Target Prediction for Indirect Jumps. In: ISCA-24, pp. 274–283 (1997)
3. COMPAQ. Alpha Architecture Handbook, V4 (October 1998)
4. Driesen, K., Hölzle, U.: Accurate Indirect Branch Prediction. In: ISCA-25, pp. 167–178 (1998)
5. Driesen, K., Hölzle, U.: The Cascaded Predictor: Economical and Adaptive Branch Target Prediction. In: MICRO-31, pp. 249–258 (1998)
6. Driesen, K., Hölzle, U.: Multi-stage Cascaded Prediction. In: Amestoy, P.R., Berger, P., Daydé, M., Duff, I.S., Frayssé, V., Giraud, L., Ruiz, D. (eds.) Euro-Par 1999. LNCS, vol. 1685, pp. 1312–1321. Springer, Heidelberg (1999)
7. Farooq, M.U., Chen, L., John, L.K.: Value Based BTB Indexing for Indirect Jump Prediction. In: HPCA-16, pp. 1–11 (January 2010)
8. Gochman, S., Ronen, R., Anati, I., Berkovits, A., Kurts, T., Naveh, A., Saeed, A., Sperber, Z., Valentine, R.C.: The Intel Pentium M Processor: Microarchitecture and Performance. Intel Technology Journal 7(2) (May 2003)
9. Intel. Intel software college: `http://developer.intel.com/software/products/college/itanium/`
10. Joao, J.A., Mutlu, O., Kim, H., Agarwal, R., Patt, Y.N.: Improving the Performance of Object-Oriented Languages with Dynamic Predication of Indirect Jumps. In: ASPLOS-13, pp. 80–90 (2008)
11. Kalamatianos, J., Kaeli, D.R.: Predicting Indirect Branches via Data Compression. In: MICRO-31, pp. 272–281 (1998)

12. Kim, H., Joao, J.A., Mutlu, O., Lee, C.J., Patt, Y.N., Cohn, R.: VPC Prediction: Reducing the Cost of Indirect Branches via Hardware-Based Dynamic Devirtualization. In: ISCA-34, pp. 424–435 (2007)
13. Lee, J.K.F., Smith, A.J.: Branch Prediction Strategies and Branch Target Buffer Design. Computer 17(1), 6–22 (1984)
14. Lin, J., Chen, T., Hsu, W.C., Yew, P.C.: Speculative Register Promotion Using Advanced Load Address Table (ALAT). In: Proceedings of First Annual IEEE/ACM International Symposium on Code Generation and Optimization, pp. 125–134. IEEE Computer Society (2003)
15. Perelman, E., Hamerly, G., Biesbrouck, M.V., Sherwood, T., Calder, B.: Using SimPoint for Accurate and Efficient Simulation. In: SIGMETRICS 2003, pp. 318–319 (2003)
16. Roth, A., Moshovos, A., Sohi, G.S.: Improving Virtual Function Call Target Prediction via Dependence-Based Pre-Computation. In: ICS-13, pp. 356–364 (1999)
17. SPEC. Standard Performance Evaluation Corporation, `http://www.spec.org`.
18. Yeh, T.-Y., Patt, Y.N.: Two-Level Adaptive Training Branch Prediction. In: MICRO-24, pp. 51–61 (1991)