Energy-Efficient Branch Prediction with Compiler-Guided History Stack

Mingxing Tan, Xianhua Liu, Zichao Xie, Dong Tong, Xu Cheng
Microprocessor Research and Development Center, Peking University, Beijing, China
{tanmingxing, liuxianhua, xiezichao, tongdong, chengxu}@mprc.pku.edu.cn

Abstract—**Branch prediction is critical in exploring instruction level parallelism for modern processors. Previous aggressive branch predictors generally require significant amount of hardware storage and complexity to pursue high prediction accuracy. This paper proposes the *Compiler-guided History Stack* (CHS), an energy-efficient compiler-microarchitecture cooperative technique for branch prediction. The key idea is to track very-long-distance branch correlation using a low-cost compiler-guided history stack. It relies on the compiler to identify branch correlation based on two program substructures: loop and procedure, and feed the information to the predictor by inserting guiding instructions. At runtime, the processor dynamically saves and restores the global history using a low-cost history stack structure according to the compiler-guided information. The modification on the global history enables the predictor to track very-long-distance branch correlation and thus improves the prediction accuracy. We show that CHS can be combined with most of existing branch predictors and it is especially effective with small and simple predictors. Our evaluations show that the CHS technique can reduce the average branch mispredictions by 28.7% over gshare predictor, resulting in average performance improvement of 10.4%. Furthermore, it can also improve those aggressive perceptron, OGEHL and TAGE predictors.**

I. Introduction

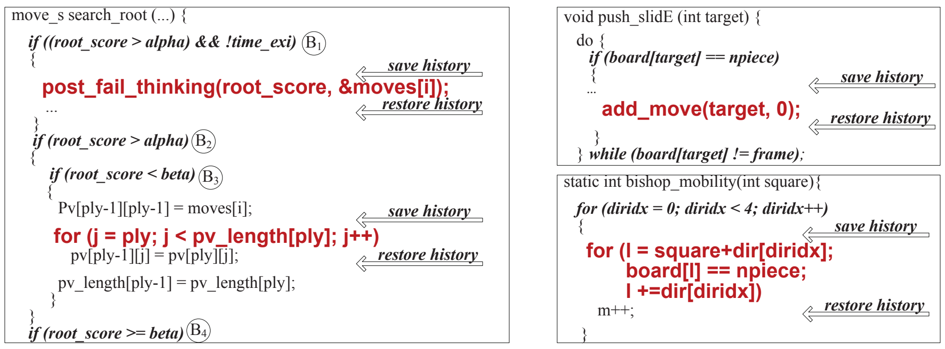
Energy efficiency is the new fundamental limiter of modern processor performance [3], so modern chip multiprocessor architecture tends to use several less aggressive cores to achieve high throughput and energy efficiency. Recent research [11] shows that the energy consumed in the instruction fetch path is becoming a large overhead. Accurate branch prediction can improve both the performance and energy efficiency by reducing pipeline flushes and invalid speculative operations in the instruction fetch path. To purse accurate branch prediction, a number of aggressive branch predictors [13], [18], [19] are proposed. However, the aggressive branch predictor itself may consume a large number of energy because of its large storage and complex control logics. For example, recently proposed perceptron predictors [13], [12] and PPM-based predictors [5], [19], [20] generally require 64K∼256Kbits storage and complex training algorithm. These aggressive predictors can achieve relatively high prediction accuracy, but they significantly increase the hardware complexity [16] and the energy consumption [2]. As a result, these aggressive predictors can be hardly adopted in modern energy-efficient processors.

This work was supported in part by the National High Technology Research and Development Program of China under Grant No. 2009ZX01029-001-002.

Recently, some researchers propose to improve branch prediction accuracy based on simple and small predictors by optimizing the global branch history. Branch history is successfully used in branch prediction because branches often correlate with previously executed branches [7], [16]. Generally, longer branch history enables the predictor to view a larger window of previously executed branches and learn based on the correlation with those branches. However, previous techniques such as skewed history [15] and artificial global history [6], [16] generally cannot track very-long-distance branch correlation. This is because tracking very-long-distance branch correlation requires the high-level program substructure information, which can be hardly captured at runtime.

In this paper, we propose an energy-efficient compiler-microarchitecture cooperative technique, called Compiler-guided History Stack (CHS), which improves branch prediction accuracy by enabling the predictor to tracking very-long-distance branch correlation using a low-cost history stack. It relies on the compiler to identify branch correlation based on two high-level program substructures: loop and procedure. In these program substructures, two correlated branches can be very-long-distance because of the large number of unrelated branches in the unrelated code regions, so the predictor can hardly track such correlation at runtime. However, we show that it is straightforward for the compiler to identify such very-long-distance branch correlation based on high-level program substructure information. With the help of the compiler-guided information, the processor dynamically saves and restores the global branch history using a low-cost history stack structure, and then uses the updated branch history to track very-long-distance branch correlation. We show that the CHS technique is low-cost, requiring only 512-bit storage and a simple control logic, so it can be easily implemented in modern processors.

Our CHS technique can be combined with most of existing branch predictors including gshare predictor [14], perceptron predictor [13], OGEHL predictor [18] and TAGE predictor [19], [20]. However, it is especially effective with smaller, simpler predictors, allowing those predictors to be competitive with more expensive and complex variants. When working with the simple and highly effective gshare predictor, our technique significantly reduces the branch Misprediction Per Kilo Instructions (MPKI) by 28.7% and improves the average performance by 10.4%. Even when working with the state-of-the-art aggressive TAGE predictor, our technique can still further reduce MPKI by 8.4%.



```
move_s search_root (...) {
  if ((root_score > alpha) && !time_exi) B1
  {
    post_fail_thinking(root_score, &moves[i]);
    ...
  }
  if (root_score > alpha) B2
  {
    if (root_score < beta) B3
    {
      Pv[ply-1][ply-1] = moves[i];
     for (j = ply; j < pv_length[ply]; j++)
        pv[ply-1][j] = pv[ply][j];

      pv_length[ply-1] = pv_length[ply];
    }
  }
  if (root_score >= beta) B4
```

```
void push_slidE (int target) {
  do {
    if (board[target] == npiece)
    {
    ...
      add_move(target, 0);
    }
  } while (board[target] != frame);
```

```
static int bishop_mobility(int square){
  for (diridx = 0; diridx < 4; diridx++)
  {
    for (l = square+dir[diridx];
          board[l] == npiece;
          l +=dir[diridx])
      m++;
  }
```

Fig. 1. Code examples showing very-long-distance branch correlation due to loops and procedure calls. Examples are taken from 458.sjeng benchmark.

II. Motivation

Longer branch history can improve the prediction accuracy by tracking more distant branch correlation, but it usually requires more hardware cost. Furthermore, the large number of unrelated branches in loops and procedures can make the branch correlation be *very-long-distance*. Figure 1 shows some code examples taken from 458.sjeng benchmark in SPEC CPU 2006 suite [22]. The two branches B_1 and B_2 in the left example have strong correlation with each other, i.e., if B_1 is resolved as 'taken' then B_2 must be 'taken' too. Unfortunately, the procedure '*post_fail_thinking*' contains a lot of unrelated branches, resulting in that branch predictors can hardly track the very-long-distance correlation between B_1 and B_2. Similarly, branches B_3 and B_4 also have strong correlation with each other, but the large number of unrelated branches in the *for-loop* statement causes a long distance between the two correlated branches. The two examples on the right also show such very-long-distance branch correlation due to the large number of unrelated branches in loops or procedures.

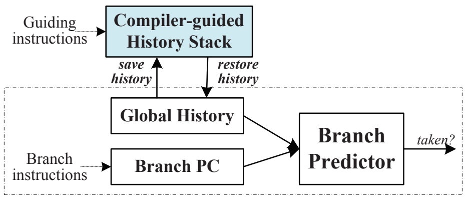


Fig. 2. Overview structure of our CHS-based predictor.

The very-long-distance branch correlation can be hardly tracked by traditional branch predictors even using large storage. However, it is straightforward for the compiler to identify such branch correlation. Based on this observation, we propose the CHS technique as shown in Figure 2. It uses a low-cost compiler-guided history stack to track very-long-distance branch correlation by dynamically saving and restoring the global history. The history stack is operated using guiding instructions inserted by the compiler. For example, to track the correlation between B_1 and B_2 in Figure 1, two guiding instructions are inserted around the procedure call of '*post_fail_thinking*'. When B_2 is fetched, the history of B_1 has been restored into the global history, so the predictor can use the updated global history to aid branch prediction.

III. Related work

Branch prediction is well studied in a great number of works [24], [14], [13], [18], [19]. Yeh and Patt [24] firstly propose the Global History Register (GHR) to track global branch correlation in their two-level predictor. The benefits of the GHR are further demonstrated by gselect/gshare predictors [14]. Traditional counter-based predictors usually track short-distance correlation using 10$\sim$15 bits GHR. Generally, longer branch history enables branch correlation with more distant branches, so recent aggressive predictors tend to adopt more complicated hardware to track longer history. Michaud et al. [15] propose the gskew predictor, which uses more than 20 bits of branch history to index multiple prediction tables. Jimenez et al. [13] propose the perceptron predictor, which uses 34 bits of global history to train a simple neural network. Perceptron predictor can be extended to utilize more than 50 bits of global history with larger storage [12]. Seznec [18] proposes the OGEHL predictor to make use of geometric history lengths ranging from 100 to 200. Seznec [19], [20] further proposes the TAGE predictor, which can use more than 600 bits of branch history. Although these aggressive predictors can achieve high prediction accuracy, they also significantly increase hardware storage and complexity, thus deterring their adoption in modern processors [16].

Besides more complicated predictors, some researchers try to improve the branch correlation by applying some optimizations to the global history. Yeh and Patt [24] point out that branches usually have strong correlation with the global branch history. Evers et al. [7] further point out that even though many branches are highly correlated with a small number of prior branches, the predictability may hardly be captured. Gao and Sair [8] reuse existing return address stack to exploit intra-procedure branch correlation. Porter and Tullsen [16] observe that not all regions of control flow are correlated with recently executed branches. Based on this observation, they propose various artificial modifications on the global history for single-thread [16] and multithread execution [6]. Recently, Sazeides et al. [17] propose the affectors and affectees, which rely on the compiler to eliminate unrelated branch history. Our technique differs from these works in that it mainly enables the predictor to track very-long-distance branch correlation rather than simply eliminates unrelated branch history.

IV. COMPILER-GUIDED HISTORY STACK (CHS)

A. *Overview of CHS Technique*

CHS relies on the cooperation between the compiler and the microarchitecture. First of all, the compiler identifies the very-long-distance branch correlation based high-level program substructures and explicitly inserts guiding instructions to indicate such correlation. At runtime, when a guiding instruction is executed, the processor uses a low-cost history stack to dynamically save and restore the global history according to the compiler-guided information. Subsequently when a branch instruction is fetched, the predictor uses the modified global branch history to aid branch prediction.

B. *Compiler Support*

The compiler identifies very-long-distance correlation based on two program substructures: loops and procedure calls.

1) Loops

Loops are detected using the traditional Dominator-Join graph based algorithm [23]. Figure 3 (a) and (b) show an example for a loop control flow graph (CFG). Given such a loop, the CHS compiler collects all preceding (Pred-) and succeeding (Succ-) branches of the loop. For each succeeding branch, the compiler finds its affectors and affectees using the algorithm proposed in [17]. If most of the affectors and affectees are preceding branches, meaning that succeeding branches have strong correlation with preceding branches, then the compiler inserts guiding instructions around the loop.

As shown in Figure 3(b), a loop may contain multiple entering edges and some of edges come from blocks inner the loop. In such a case, a preheader block is created immediately before the header block and all entering edges coming from outer loop are redirected to the preheader block. Similarly, a tail block is created after the loop exit point. Both preheader and tail block will be executed exactly once no matter how many times the loop iterates. At last, the compiler inserts a *guide_save_history* instruction into the preheader block and a *guide_restore_history* instruction into the tail block.

2) Procedure Calls

Procedure calls can be categorized as direct calls or indirect calls, which can be easily identified by the compiler. Figure 3 (c) and (d) show an example of procedure call. The compiler performs data dependence analysis based on affectors and affectees [17]. If succeeding branches have strong correlation with preceding branches, then the compiler inserts two guiding instructions: a *guide_save_history* before the procedure call and a *guide_restore_history* after the procedure call.

3) Guiding Instructions

The compiler hints the branch predictor using two guiding instructions: *guide_save_history* and *guide_restore_history*. These guiding instructions can be implemented by reusing reserved bits in ISA instructions. In our simulation, guiding instructions are implemented by operating a reserved system register. Assuming sr is a reserved system register, setting sr to 0 represents executing the *guide_save_history*, while setting sr to 1 represents the *guide_restore_history*.

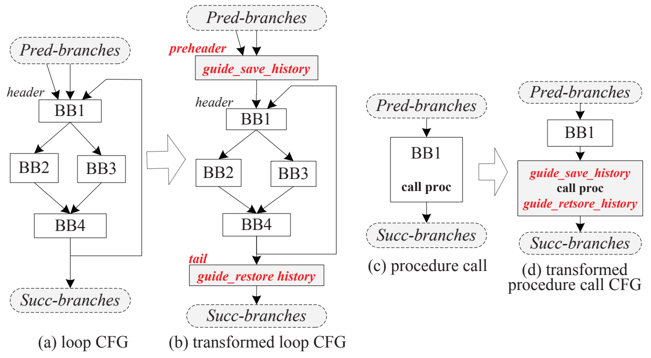


Fig. 3. Transformation of loops and procedure calls.

Ideally, the *guide_restore_history* should be executed before the corresponding branch instructions are fetched, but it may be violated because of the pipeline latency. To execute the *guide_restore_history* earlier, we adopt two optimizations. First, guiding instructions are scheduled to increase the dynamic distance to their corresponding branch instructions. Second, since guiding instructions are independent with other instructions and their executions have no effect on program correctness, the processor can actually execute their saving and restoring operations at the decode stage.

C. *CHS-Based Branch Predictor*

Generally, the CHS technique can be combined with most of existing branch predictors. In this section, we introduce the gshare-CHS predictor, which combines CHS with the highly effective gshare predictor [14], [1], [10].

1) Branch Predictor Structure

Figure 4 shows the hardware structure of gshare-CHS predictor. The predictor mainly consists of four components:

- BTB (Branch Target Buffer) contains all target addresses of conditional and indirect branches.
- PHT (Pattern History Table) contains a serial of saturated 2-bit counters, which is used to predict branch direction.
- GHR (Global History Register) contains the global branch history, which is used to generate PHT index.
- CHS (Compiler-guided History Stack) contains the compiler-identified very-long-distance branch correlation.

Compared with the original gshare predictor, the gshare-CHS predictor adds an extra CHS component. The CHS component is a low-cost circular stack buffer, which is operated using guiding instructions. If the guiding instruction indicates to save branch history (*guide_save_history*), then the GHR is pushed into CHS; otherwise it indicates to restore branch history (*guide_restore_history*) and the top entry of CHS is popped out for GHR modification. Since the CHS is implemented as a circular buffer, a simple TOS (Top of Stack) pointer is enough to maintain the push and pop semantics.

Besides the CHS structure, it also requires a simple control logic consisting of an adder gate to operate the CHS structure and a shifter gate to update the GHR. Note that our technique does not affect the prediction and update flow, so it does not add extra latency to the instruction fetch path.

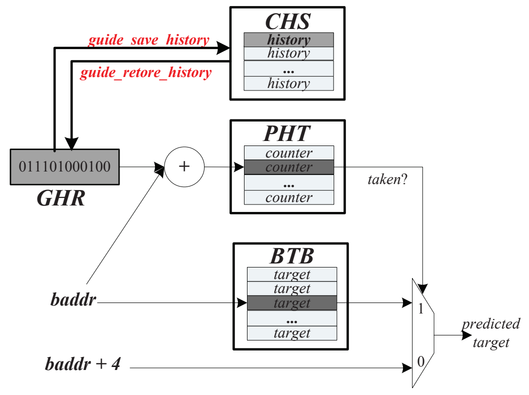


Fig. 4. Hardware structure of gshare-CHS predictor.

2) Operation on CHS Structure

The CHS structure is operated using guiding instructions, as shown in Figure 5. The code example contains three branches: B_1, B_2 and B_3. Although B_3 has strong correlation with B_1 and B_2, the correlation is disrupted by the *for-loop* statement. Since the *for-loop* iterates multiple times, the dynamic distance between B_2 and B_3 can be very long, so traditional predictors can hardly track such correlation. Figure 5(b) shows how our technique tracks such very-long-distance branch correlation:

(1) Before entering the *for-loop*, a *guide_save_history* instruction is executed, which pushes GHR into CHS. Since the current GHR contains the resolved outcome of B_1 and B_2, the branch history bits of B_1 and B_2 are saved into CHS.

(2) On the loop exit point, the loop has iterated many times. Since GHR is filled by those unrelated loop history, the branch history bits of B_1 and B_2 are discarded from GHR.

(3) When the *guide_restore_history* instruction is executed, the processor pops out the CHS top entry and updates GHR based on it. In other words, the discarded branch history bits of B_1 and B_2 are restored into GHR. Note that setting the whole GHR to CHS top entry may hurt prediction accuracy if subsequent branches depend on some of the branches in the *for-loop* statement, so the processor actually updates the GHR by concatenating the original GHR with the CHS top entry.

(4) When B_3 is fetched, the processor looks up PHT at the index computed by hashing GHR and branch address. Since branch history of B_1 and B_2 has been restored into GHR, the predictor can accurately predict the outcome of B_3 based on the very-long-distance branch correlation of B_1 and B_2.

The GHR may be not updated by guiding instructions at the time of branch prediction because of the pipeline latency. In such cases, our technique performs the same as the original predictor. Note that guiding instructions are actually executed at the decode stage, so such cases seldom happen. However, the wrong-path guiding instructions can push invalid history into CHS at the decode stage. To achieve better accuracy, we can repair CHS by referring to the RAS repair mechanism [21], but it will complicate the hardware. Since such cases are rare ($< 2\%$), we don't repair CHS in our simulation.

3) Hardware Cost and Complexity

The hardware cost of our technique mainly comes from the CHS structure and the control logic. In our simulation,

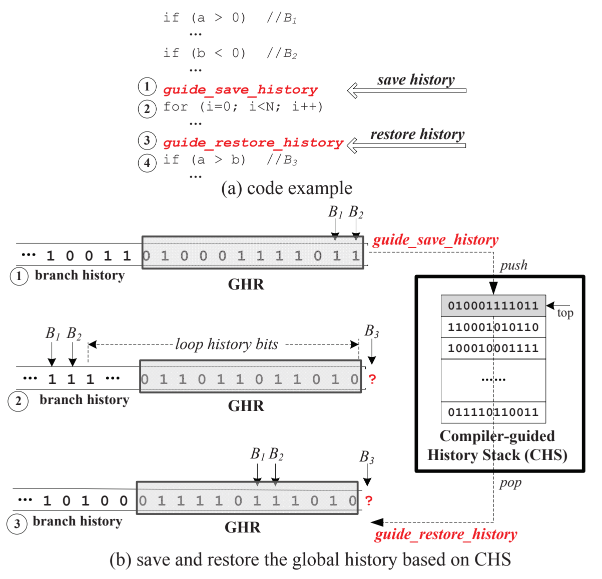


Fig. 5. An example to illustrate the operation on CHS.

the CHS is configured as a 512-bit stack buffer, featuring 32 entries and each entry contains 16 bits. The control logic mainly consists of an adder and a shifter. In conclusion, we show that the CHS technique requires only small storage and a simple control logic, so it is very applicable to modern energy-efficient processors.

V. EXPERIMENTAL METHODOLOGY

We extend the SimpleScalar/Alpha 3.0 tool set [4] to evaluate the CHS technique. TABLE I shows the parameters of the baseline processor. The baseline processor uses a 32K-entry gshare predictor and 4K-entry BTB for branch prediction. Perceptron [13], OGEHL [18] and TAGE [19], [20] predictors are also implemented in the baseline processor. All branch mispredictions are resolved in the commit stage.

The CHS technique is evaluated using 4 SPEC CPU2000 INT benchmarks and 5 SPEC CPU2006 INT benchmarks [22]. We choose those benchmarks in SPEC INT 2000 and 2006 suites that gain at least 8% performance improvement with a perfect branch predictor. We use the SimPoint to find a representative simulation region for each benchmark using the reference input data set. Each benchmark is run for 100M instructions. TABLE II shows the characteristics of simulated SimPoint for each benchmark.

We extend GCC-4.2 [9] to provide the compiler support. All benchmarks are compiled with the -O2 optimization level.

TABLE I: BASELINE PROCESSOR CONFIGURATION

Front End	4 instruction per cycle; fetch ends at the first predicted taken branch or indirect jump instruction
Execution Core	4-wide decode/issue/execute/commit; 512-entry ROB; 128-entry LSQ
Branch Predictor	32K-entry gshare predictor; 4K-entry, 4-way BTB with LRU replacement; 32-entry return address stack; 15 cycle minimal branch misprediction penalty
On-chip Caches	16KB, 4-way, 1-cycle L1 D-cache and L1 I-cache; 1MB, 8-way, 10-cycle unified L2 cache; All caches have 64B block size with LRU replacement
Memory	150-cycle latency (first chunk), 15-cycle (rest)

TABLE II: CHARACTERISTICS OF EVALUATED BENCHMARKS

	gzip	mcf	parser	twolf	gcc06	gobmk	hmmer	sjeng	astar	AVG
Static number of branches	190	95	632	208	7391	3976	179	1188	178	1560
Dynamic number of branches(K)	9553	23000	14262	9157	16539	12857	12483	15180	11276	13812
Branch MPKI	10.0	13.9	6.1	8.1	8.1	17.6	11.6	15.1	21.6	12.4
Branch prediction accuracy (%)	88.9	93.4	95.2	90.5	94.5	86.1	90.6	89.5	72.0	89.0
base IPC	1.41	1.15	1.53	1.33	1.29	0.97	1.34	1.03	0.69	1.169

TABLE III: STATISTICS OF EXECUTED GUIDING INSTRUCTIONS (G_INSTRUCTION)

	gzip	mcf	parser	twolf	gcc06	gobmk	hmmer	sjeng	astar	AVG
Number of static g_instruction	72	26	166	35	210	87	19	190	25	92
Number of dynamic g_instruction	850K	1512K	2874K	1003K	2493K	764K	84K	1384K	140K	1233K

VI. RESULTS

A. Performance of CHS-based Branch Prediction

In general, our CHS technique can benefit most when working with small and simple branch predictors. In this section, we firstly evaluate the performance and characteristics of the CHS technique by combing it with gshare predictor. We choose gshare predictor as the baseline predictor because it is a simple, small and highly effective predictor, and it has been widely used in commercial processors [1], [10]. Further evaluations on aggressive predictors will be discussed later.

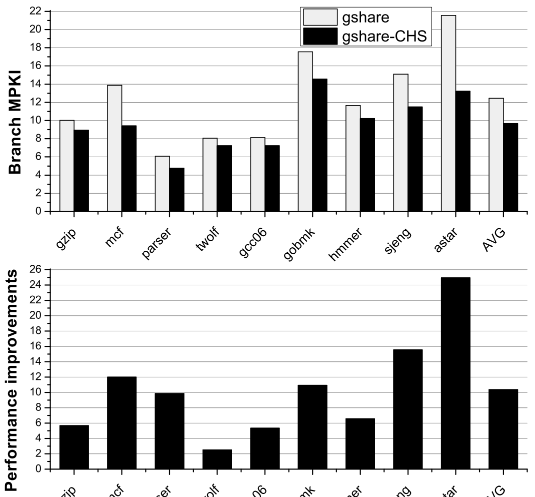


Fig. 6. Performance of gshare-CHS branch prediction: Branch MPKI (top) and Performance improvement (bottom).

Figure 6 shows the performance of gshare-CHS predictor. On average, the CHS technique significantly reduces branch MPKI by 28.7% over the baseline gshare predictor, resulting in average performance improvement of 10.4%.

The CHS technique improves the gshare predictor for two reasons. First, it enables the predictor to track very-long-distance branch correlation. The original gshare predictor uses 15-bit GHR as input, so branch correlation whose distance is longer than 15 will be discarded. On the contrary, the CHS technique tracks those very-long-distance branch correlation by dynamically saving and restoring branch history using the compiler-guided history stack. Second, the technique also reduces aliasing by eliminating history noise caused by those unrelated branches. Since aliasing is a significant source of mispredictions in gshare predictor [15], it also improves branch prediction by reducing aliasing.

B. Impact of Guiding Instructions

TABLE III shows the statistics of executed guiding instructions. On average, each program runs 92 static guiding instructions and 1233K dynamic guiding instructions in the simulated region. Since we run 100M instructions for each program, the percentage of dynamic guiding instructions is about 1.2%. Note that results in Figure 6 have counted the extra overhead for executed guiding instructions. The outstanding performance has shown guiding instructions are very effective.

C. CHS on Different Predictor Sizes

Figure 7 shows the performance of the CHS technique with different predictor sizes. Results show that the benefits of the CHS technique are most substantial when working with small predictors. This is because small predictors can only track short history, leaving a large number of distant branch correlation to be tracked by the CHS technique. However, even when working with 128K-entry PHT structure, it still reduces MPKI by 24% and improves performance by 9.4%.

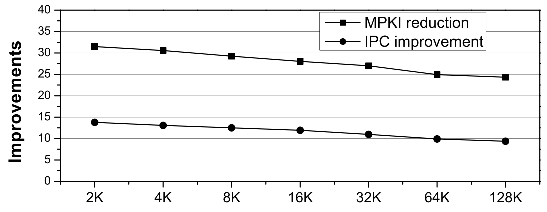


Fig. 7. CHS performance on different predictor sizes.

D. CHS on Different CHS Sizes

TABLE IV shows the performance of the CHS technique on different CHS sizes. Generally, it performs better with large CHS structure because large CHS structure can buffer more very-long-distance branch correlation and have less aliasing. As the number of CHS entries increases from 16 to 128, the average MPKI reduction increases from 26% to 32%, resulting in performance improvement from 9.6% and 11.4%.

TABLE IV: CHS PERFORMANCE ON DIFFERENT CHS SIZES

CHS entries	16	32	64	128
MPKI reduction	25.9%	28.7%	30.3%	31.8%
Performance improvement	9.6%	10.4%	11.0%	11.4%

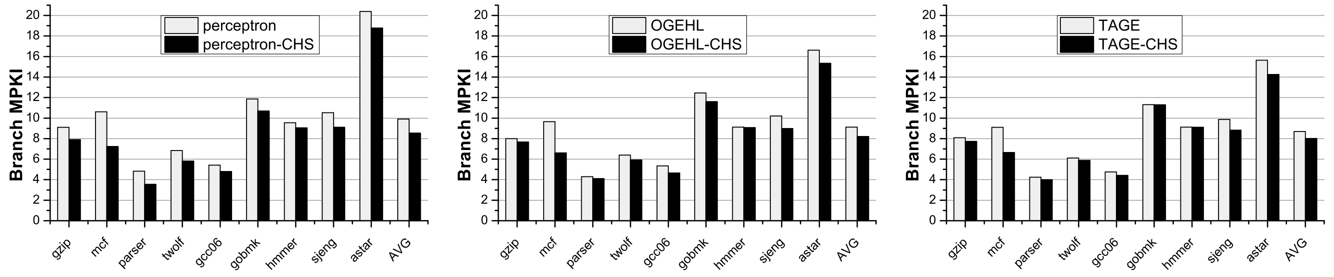


Fig. 8. CHS performance on perceptron(left), OGEHL(middle) and TAGE (right) predictors.

TABLE V: CONFIG OF PERCEPTRON,OGEAL AND TAGE PREDICTOR.

perceptron	64KB, featuring a 1024-entry perceptrons table. Each entry contains 64 weights w_j, using 64-bit GHR as input.
OGEHL	64Kbits, including 8 predictor tables. Tables are indexed with hash functions of 200-bit GHR, path history and PC.
TAGE	64Kbits, including 7 tagged tables and 1 bimodal table. Tables are indexed with hashing of 200-bit GHR, path history and PC.

E. CHS on Aggressive Branch Predictors

We also evaluate the CHS technique on three aggressive branch predictors: perceptron predictor [13], OGEHL predictor [18] and TAGE predictor [19], [20]. TABLE V shows the configuration for these branch predictors. Note that the CHS technique only modifies the global branch history in these predictors, leaving path history and branch PC unaffected.

Figure 8 shows the branch MPKI reduction of the CHS technique on the three aggressive predictors. On average, the CHS technique can further reduce the average branch MPKI by 18% over perceptron predictor, 11% over OGEHL predictor, and 8.4% over TAGE predictor. Results show that the CHS technique can also improve these aggressive branch predictors by tracking very-long-distance branch correlation.

TABLE VI shows the comparison of CHS on gshare, perceptron, OGEHL, and TAGE predictors. The CHS technique can improve all of these predictors, but it is more effective when working with the simple gshare predictor. Especially, our low-cost CHS technique can significantly reduce the gshare MPKI from 12.4 to 9.67, which makes it very competitive with those aggressive perceptron, OGEHL and TAGE predictors.

VII. CONCLUSION

In this paper, we proposed and evaluated the CHS technique, an energy-efficient compiler-microarchitecture cooperative technique for branch prediction. It improves branch prediction accuracy by tracking very-long-distance branch correlation using a low-cost compiler-guided history stack. Unlike previous techniques, the CHS technique requires only small hardware storage and a simple control logic, so it is very applicable for modern energy-efficient processors. We show that the CHS technique can be combined with most of existing branch predictors and it is especially effective with smaller, simpler predictors, allowing those predictors to be competitive with more expensive and complex variants. Evaluations show that it can reduce the average MPKI by 28.7% and improve performance by 10.4% over the baseline gshare predictor. Furthermore, it can also improve those aggressive perceptron, OGEHL and TAGE predictors.

TABLE VI: COMPARISON OF CHS ON DIFFERENT BRANCH PREDICTORS.

	MPKI			Performance		
	Original	CHS	Reduction	Original	CHS	Improvement
gshare	12.4	9.67	29%	1.169	1.292	10.4%
perceptron	10.1	8.54	18%	1.286	1.371	6.7%
OGEHL	9.12	8.21	11%	1.340	1.388	3.5%
TAGE	8.69	8.01	8.4%	1.355	1.398	3.2%

REFERENCES

[1] ARM. Cortex-a9 technical reference manual, version r3p0, 2008.

[2] A. Baniasadi and A. Moshovos. Sepas: A highly accurate energy-efficient branch predictor. In *ISLPED'04*, pages 38–43, 2004.

[3] S. Borkar and A. A. Chien. The future of microprocessors. *Communications of the ACM*, vol.54(5):67–77, 2011.

[4] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, vol.25(3):13–25, 1997.

[5] I.-C. K. Chen, J. T. Coffey, and T. N. Mudge. Analysis of branch prediction via data compression. In *ASPLOS-VII*, pages 272–281, 1996.

[6] B. Choi, L. Porter, and D. M. Tullsen. Accurate branch prediction for short threads. In *ASPLOS-XIII*, pages 125–134, 2008.

[7] M. Evers, S. J. Patel, R. S. Chappell, and Y. N. Patt. An analysis of correlation and predictability: what makes two-level branch predictors work. In *ISCA-25*, pages 52–61, 1998.

[8] F. Gao and S. Sair. Exploiting intra-function correlation with the global history stack. In *SAMOS-5*, pages 213–223, 2005.

[9] GCC. The gnu compiler collection. http://gcc.gnu.org/.

[10] S. Gochman et al. The intel pentium m processor: Microarchitecture and performance. *Intel Technology Journal*, vol.7(2), 2003.

[11] R. Hameed et al. Understanding sources of inefficiency in general-purpose chips. In *ISCA-37*, pages 37–47, 2010.

[12] D. A. Jiménez. Piecewise linear branch prediction. In *ISCA-32*, pages 382–393, 2005.

[13] D. A. Jiménez and C. Lin. Dynamic branch prediction with perceptrons. In *HPCA-7*, pages 197–201, 2001.

[14] S. McFarling. Combining branch predictors. *Technical Note TN-36, Digital. Equipment Corporation Western Research Laboratories*, 1993.

[15] P. Michaud, A. Seznec, and R. Uhlig. Trading conflict and capacity aliasing in conditional branch predictors. In *ISCA-24*, 1997.

[16] L. Porter and D. M. Tullsen. Creating artificial global history to improve branch prediction accuracy. In *ICS-23*, pages 266–275, 2009.

[17] Y. Sazeide, A. Moustakas, K. Constantinides, and M. Kleanthous. The significance of affectors and affectees correlations for branch prediction. In *HiPEAC-3*, pages 243–257, 2008.

[18] A. Seznec. Analysis of the o-geometric history length branch predictor. In *ISCA-32*, pages 394–405, 2005.

[19] A. Seznec. The l-tage branch predictor. *Journal of Instruction Level Parallelism*, vol.9, 2007.

[20] A. Seznec. A new case for the tage branch predictor. In *MICRO-44*, 2011.

[21] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Improving prediction for procedure returns with return-address-stack repair mechanisms. In *MICRO-31*, pages 259–271, 1998.

[22] SPEC. Standard performance evaluation corporation. http://www.spec.org.

[23] V. C. Sreedhar, G. R. Gao, and Y.-F. Lee. Identifying loops using dj graphs. *ACM Transactions on Programming Languages and Systems*, vol.18(6):649–658, 1996.

[24] T.-Y. Yeh and Y. N. Patt. Two-level adaptive training branch prediction. In *MICRO-24*, pages 51–61, 1991.