Store-Load-Branch (SLB) Predictor: A Compiler Assisted Branch Prediction for Data Dependent Branches

M. Umar Farooq Khubaib Lizy K. John

Department of Electrical and Computer Engineering
The University of Texas at Austin
ufarooq@utexas.edu, khubaib@ece.utexas.edu, ljohn@ece.utexas.edu

Abstract

Data-dependent branches constitute single biggest source of remaining branch mispredictions. Typically, data-dependent branches are associated with program data structures, and follow store-load-branch execution sequence. A set of memory locations is written at an earlier point in a program. Later, these locations are read, and used for evaluating branch condition. Branch outcome depends on data values stored in data structure, which, typically do not have repeatable pattern. Therefore, in addition to history-based dynamic predictor, we need a different kind of predictor for handling such branches.

This paper presents Store-Load-Branch (SLB) predictor; a compiler-assisted dynamic branch prediction scheme for data-dependent direct and indirect branches. For every data-dependent branch, compiler identifies store instructions that modify the data structure associated with the branch. Marked store instructions are dynamically tracked, and stored values are used for computing branch flags ahead of time. Branch flags are buffered, and later used for making predictions. On average, compared to standalone TAGE predictor, combined TAGE+SLB predictor reduces branch MPKI by 21% and 51% for SPECINT and EEMBC benchmark suites respectively.

1 Introduction

Most branch prediction techniques rely on branch history information for predicting future branches. Some use short history [11] [21] [36], while others use longer history [16] [25] [30] [32]. History-based dynamic branch prediction schemes have shown to reach high prediction accuracy for all except few hard-to-predict branches. Figures 1 and 2 show branch mispredictions per 1K instructions for EEMBC and SPECint benchmark suites using several history-based branch predictors. As can be seen from the figures, several benchmarks have higher branch mispredictions even when using long history branch predictor.

This work is based on the following observation: Hard-to-predict data-dependent branches are commonly associated with program data structures such as arrays, linked lists, trees etc., and follow store-load-branch execution sequence similar to one shown in listing 1. A set of memory locations is written while building and updating the data structure (line 2, listing 1). During data structure traversal, these locations are read, and used for evaluating branch condition (line 7, listing 1).

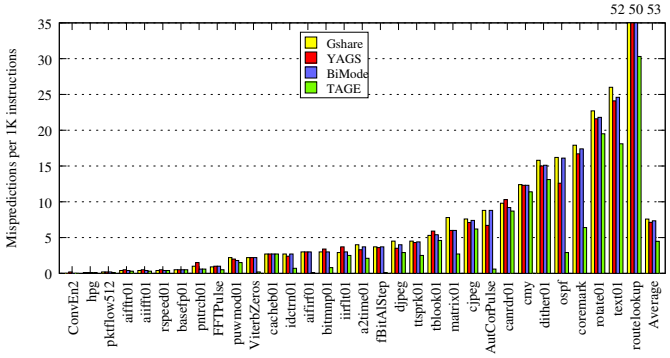

Figure 1. MPKI for EEMBC benchmark suite

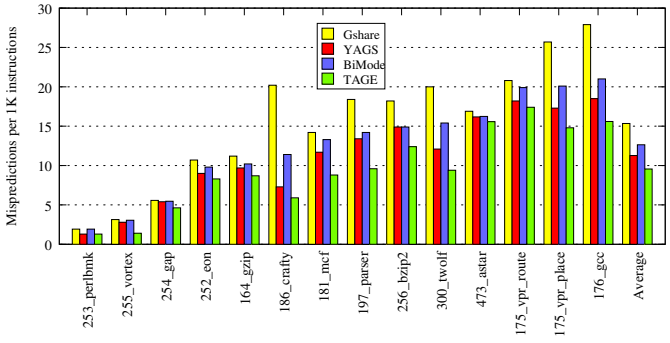

Figure 2. MPKI for SPECint benchmark suite

This paper proposes Store-Load-Branch (SLB) predictor, a compiler-assisted dynamic branch prediction scheme

Listing 1. Store-Load-Branch execution sequence

```
1  for(node=head; node!=NULL; node=node→next) {
2      node→key = ... ;
3  }
4  ..
5  node = head;
6  while(node!=NULL) {
7      if(node→key <condition>) {
8          ...
9      }
10     node=node→next;
11 }
```

for data-dependent branches using data value correlation. Compiler identifies all program points where data structure associated with a hard-to-predict data-dependent branch is referenced and modified. At run-time, hardware tracks marked store instructions that modify the data structure, computes branch condition flags ahead of time using store data values, and buffers them in a structure at store addresses. Later, during data structure traversal, pre-computed flags are read using *predicted* load address, and used for predicting branch outcome.

Typically, instruction that loads data structure values is quickly followed by branch instruction for evaluating branch condition using loaded values. Therefore, actual load address is usually not available before branch instruction gets fetched. Hence, we use predicted load address for reading pre-computed branch flags. Addresses for simple data structures such as arrays are easier to predict using stride-based address predictor. Bekerman et al. proposed load address predictor for irregular data structures such as linked list and tree [4], which we adapted according to our requirement.

We compared our design with the state-of-the-art TAGE branch predictor [32]. Results show that, for several benchmarks, top mispredicting branches in the TAGE predictor are accurately predicted using SLB predictor. On average, compared to standalone TAGE predictor, combined TAGE+SLB predictor reduces branch mispredictions per 1K instructions (MPKI) by 21% for SPECint [34] benchmark suite. Similarly, for EEMBC [12] benchmark suite, MPKI is reduced by 51%.

This paper makes following contributions:

1. We investigate program patterns that manifest hard-to-predict, data-dependent branches.
2. We propose SLB prediction scheme for data-dependent branches, which predicts branch outcome using pre-computed branch flags.
3. Our implementation of SLB requires adding couple of hint instructions to ISA, and light-weight hardware structures.

Rest of the paper is organized as follows: Section 2 presents motivating examples from real benchmarks. Section 3 describes SLB prediction scheme. Simulation methodology and results are presented in Sections 4 and 5 respectively. Section 6 discusses related work. Finally, Section 7 concludes the paper.

Listing 2. LD-BR execution sequence (cjpeg)

```
#define DCTSIZE2        64                                          1
#define DIVIDE_BY(a,b)  a /= b                                      2
                                                                    3
void forward_DCT ( ... )                                            4
{                                                                   5
    /* work area for FDCT subroutine */                             6
    DCTELEM workspace[DCTSIZE2];                                    7
                                                                    8
    /* Perform the DCT */                                           9
    (*do_dct) (workspace);                                          10
                                                                    11
    register DCTELEM temp, qval;                                    12
    register int i;                                                 13
    register JCOEFPTR output_ptr = coef_blocks[bi];                 14
                                                                    15
    for (i = 0; i < DCTSIZE2; i++) {                                16
        qval = divisors[i];                                         17
        temp = workspace[i];                                        18
                                                                    19
        if (temp < 0) {                                             20
            temp = -temp;                                           21
            temp += qval>>1;                                        22
            DIVIDE_BY(temp, qval);                                  23
            temp = -temp;                                           24
        } else {                                                    25
            temp += qval>>1;                                        26
            DIVIDE_BY(temp, qval);                                  27
        }                                                           28
        output_ptr[i] = (JCOEF) temp;                               29
    }                                                               30
}                                                                   31
```

2 Motivating Examples

2.1 Example 1: Data-Dependent Direct Branches

We will use *cjpeg* benchmark from eembc-consumer suite as a motivating example. The benchmark performs standard JPEG compression on a given image. Input image is broken into block of 8x8 pixels, and each block goes through discrete cosine transform (DCT), quantization and entropy coding steps. As shown in *listing* 2, *do_dct* function (line 10) populates an array, *workspace*, with DCT coefficients whose values range between -1024 to 1023. During quantization, each DCT coefficient is read (line 18), compared (line 20) to see if it is positive or negative, and quantized accordingly. The branch **'if (temp $<$ 0)'** (line 20) is

Listing 3. ST execution sequence (cjpeg)

```
1   GLOBAL(void)
2   jpeg_fdct_islow (DCTELEM * data)
3   {
4       DCTELEM *dataptr;
5       dataptr = data;
6
7       for (ctr = DCTSIZE-1; ctr >= 0; ctr--) {
8           dataptr[DCTSIZE*0] = ... ;
9           dataptr[DCTSIZE*4] = ... ;
10          ...
11          dataptr++;
12      }
13  }
```

an example of a hard-to-predict data-dependent branch. Table 1 (row 3) shows branch characteristics and prediction accuracy of this branch using short and long history predictors. It shows that using longer history TAGE predictor does not improve prediction accuracy for this branch. Instead, if branch condition flags are computed and buffered while populating array *workspace* in *do_dct* function[1] (see *listing* 3, lines 8-9), a simple buffer lookup can yield perfect branch prediction.

2.2 Example 2: Data-Dependent Indirect Branches

Indirect branches are generally harder to predict than direct branches as they may have multiple targets corresponding to a single static indirect branch.

Listing 4 shows a ray tracer program, *Eon*, taken from SPECint2000 suite. Each ray must be tested for intersection with all the objects in the scene. A scene consists of several different types of objects with a common base class *mrSurface*, as shown in the *listing* 4 (lines 1-8). While reading the scene, these objects are stored into a 3D data-structure called *grid* using its *mrGrid::insert* method (line 10). The *grid* is later traversed in the *mrGrid::viewingHit* method (line 21), to see if the incoming ray hits any object in the grid. During each iteration of the *while* loop, the next object's pointer (*oPtr*) is read from the grid (line 36), and depending on the type of the object, corresponding *viewingHit()* method is invoked (line 41). Since the sequence of objects stored in the grid does not have a repeatable pattern, predicting target address for virtual function call (line 41) using history-based indirect branch predictor results in lower prediction accuracy (see table 1, row 4). Instead, if *viewingHit()* function addresses corresponding to different types of objects in the grid are buffered while inserting objects in the grid (line 15 and 18), a buffer lookup during grid traversal yields correct function address corresponding to the read object. Note that, in addition to accurately predicting target address for '**oPtr→viewingHit()**' function call (line 41), another hard-to-predict branch at line 37, '**if (oPtr)**', can also be accurately predicted since it also depends on the same object read from the grid.

[1] do_dct is a pointer function, and is assigned as: fdct→do_dct = jpeg_fdct_islow;

Listing 4. ST-LD-BR execution sequence (eon)

```
class mrBox : public mrSurface {                                    1
}                                                                    2
                                                                     3
class mrSphere : public mrSurface {                                  4
}                                                                    5
                                                                     6
class mrSolidTexture : public mrSurface {                            7
}                                                                    8
                                                                     9
void mrGrid::insert(double time1, double time2,                      10
                                    mrSurface *obj_ptr ) {           11
    // other piece of code                                           12
 if(grid(i, j, k) == 0) {                                            13
     grid(i, j, k) = new mrSurfaceList();                            14
     ((mrSurfaceList*)grid(i, j, k))→Add(obj_ptr);                   15
 }                                                                   16
 else                                                                17
     ((mrSurfaceList*)grid(i, j, k))→Add(obj_ptr);                   18
}                                                                    19
                                                                     20
ggBoolean mrGrid::viewingHit(                                        21
     const ggRay3& r,                                                22
     double time,                                                    23
     double tmin,                                                    24
     double tmax,                                                    25
     mrViewingHitRecord& VHR,                                        26
     ggMaterialRecord& MR                                            27
     ) const                                                         28
{                                                                    29
 double tCellMin = 0;                                                30
 double tCellMax = 0;                                                31
 mrSurface *oPtr = 0;                                                32
                                                                     33
 ggGridIterator<mrSurface*> iterator(r,grid,tmin);                   34
                                                                     35
 while(iterator.Next(oPtr,tCellMin, tCellMax)) {                     36
     if(oPtr) {                                                      37
         tCellMax = ggMin(tmax, tCellMax);                           38
         tCellMin = ggMax(tCellMin, tmin);                           39
                                                                     40
         if(oPtr→viewingHit(r, time, tCellMin,                       41
                                tCellMax, VHR, MR))                  42
             return ggTrue;                                          43
     }                                                               44
 }                                                                   45
 return ggFalse;                                                     46
}                                                                    47
```

Benchmark	Br. location	Dynamic count	Taken count	Mispredict count		Miss rate	
				Bi-Mode	TAGE	Bi-Mode	TAGE
cjpeg	line 20	576000	297216 (51.6%)	282816	274889	49.1%	47.7%
252.eon	line 41	573306	573306 (100%)	251636	247841	43.8%	43.2%
252.eon	line 37	884626	311320 (35.2%)	258697	120297	29.2%	13.5%

Table 1. Characteristics and prediction accuracy of hard-to-predict branches in motivating examples

3 Store-Load-Branch (SLB) Predictor

3.1 Overview

SLB predictor is a compiler-assisted dynamic branch prediction technique specifically targeted at improving prediction accuracy of data-dependent branches. Most data-dependent branches are associated with program data structures such as array, linked list, tree etc. During traversal, these branches operate on elements of data structure. Branch outcome depends on data values stored in the structure, which, typically do not have repeatable patterns. Therefore, instead of relying on branch history information, we compute branch flags and use them for predicting branch direction. Due to deep processor pipeline, data values (and resulting branch flags) are often not available before branch instruction gets fetch. Therefore, instead of using load values, we compute branch flags ahead of time using store values while updating the data structure.

3.2 Implementation Details

3.2.1 Compiler and Architecture Support

For a data-dependent branch, SLB scheme relies on compiler to identify its store-load-branch (ST-LD-BR) sequence. Starting with branch instruction, compiler identifies load instruction(s) on which branch is dependent. It then identifies store instruction(s) feeding load instruction(s). ST-LD-BR sequence for a branch is encoded and passed down to hardware using special *load_hint* (HLD) and *store_hint* (HST) instructions shown in figure 3.

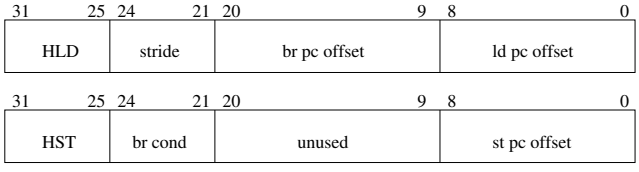

Figure 3. *load_hint* **and** *store_hint* **instruction format**

Compiler inserts an HLD/HST instruction for every static load/store associated with the branch. Lower 9 bits encode load/store pc offset from HLD/HST instruction. HLD bits 20:9 specify branch pc offset from HLD instruction. On seeing an HLD/HST instruction, hardware uses pc offset values for computing absolute address for load or store instruction associated with the branch instruction. HLD bits 24:21 specify load stride value. HST bits 24:21 specify condition code for evaluating branch outcome at store time.

3.2.2 Hardware Support

For identifying ST-LD-BR sequence at run time, hardware provides three main tables, the *store table*, the *load table*, and the *branch table*, as shown in figure 4.

Populating Load Table: On executing an HLD instruction, an entry is created in load table if it does not already exist. 'Tag' field in load table is populated with lower 12 bits of load instruction address which is computed using 'ld pc offset' in HLD instruction. 'Br pc' field is populated with 12 bits of branch instruction address which is computed using 'br pc offset' in HLD instruction. Rest of the fields in load table are associated with load address predictor, and are explained later.

Populating Store Table: Similar to load table, on executing an HST instruction, an entry is created in store table. 'Tag' field is populated with lower 12 bits of store instruction address which is computed using 'st pc offset' in HST instruction. 'Br cond' field is populated with 4-bit branch condition code specified in HST instruction.

Populating Branch Table: 'Tag' field in branch table is populated with lower 12 bits of branch instruction address which is computed using 'br pc offset' in HLD instruction.

Once ST-LD-BR sequence is populated in the tables, computing and consuming branch flags can begin.

Computing Branch Flags at Store Time: During code generation, compiler inserts *compare* instruction prior to store instruction for comparing store value with branch condition. This compare instruction sets the flag register. When store instruction executes and matches store table tag, branch flag is computed using flag register value and corresponding branch condition code in store table. Computed branch flag is stored in *T/NT prediction table* at store address. For more information on condition codes, flag register and compare/branch instruction, see ARMv7-A architectural manual [2].

Consuming Branch Flags at Fetch Time: During data structure traversal, when a data-dependent branch matches

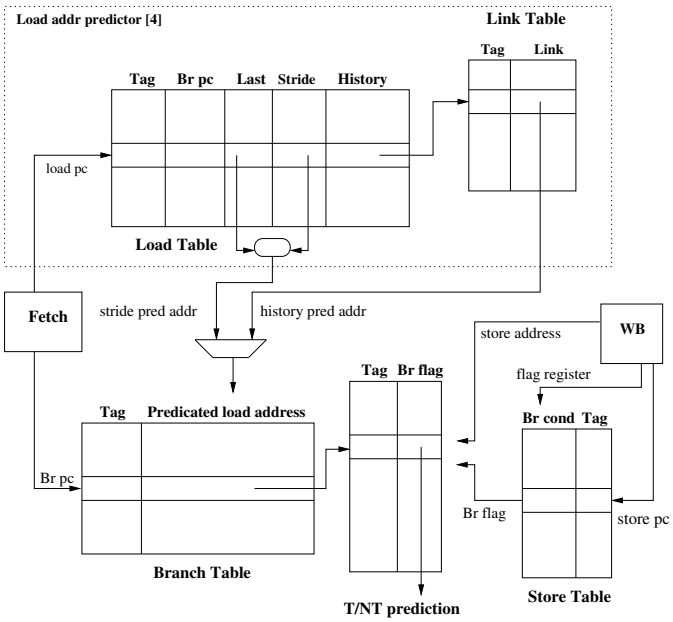

Figure 4. SLB predictor block diagram

branch table tag, prediction table is looked up using *predicted load address* for making taken/not-taken prediction. *Predicted load address* is generated using load instruction which appears earlier in program order than branch instruction, therefore, load address prediction is not on critical path of making branch prediction.

Generating Predicted Load Address: Since branch prediction is made early in the pipeline, and data structure traversal often occurs in tight loop, *'predicted load address'* is used for accessing T/NT prediction table. Figure 4 (dotted box) shows load address predictor, adapted from Bekerman's load address predictor [4]. Address prediction was originally proposed for reducing load instruction latency. In this paper, we propose an alternate use of load address predictors: *using predicted load addresses for predicting data-dependent branches*. For regular data structures (e.g. arrays) that are traversed linearly, stride-based load address predictor is sufficient [3][8]. Bekerman et al. proposed advance load address predictor for recursive data structures (e.g. linked lists and trees) [4]. It uses a two-level scheme for predicting the next load address. First level is a per-static-load table, the *load table*, where each entry records history of recent addresses seen by the associated load. The history is then used to index a second level table, the *link table*, which provides the predicted address. Typically, recursive data structure addresses can be accurately predicated by keeping last two addresses in the history. See [4] for more details on load address predictor.

3.2.3 Support for Indirect Branches

As oppose to a direct branch, an indirect branch can have multiple targets. Therefore, predicting an indirect branch requires predicting branch *target address* as oppose to branch *direction*. Most indirect branch prediction schemes maintain a history of recent targets taken by an indirect branch, and uses history to index into a 'target cache' for predicting the next target address [6] [9] [10] [17]. Similar to direct branches, data-dependent indirect branches typically do not follow history.

SLB indirect branch prediction scheme uses traditional BTB to store multiple targets of an indirect branch at different BTB indices. Figure 5 shows the block diagram for SLB indirect branch target address prediction (only necessary changes from figure 4 are shown).

Populating Store Data Array: When a store instruction executes and matches store table tag, lower 12 bits of store data value is buffered in *store data array* at store address.

Predicting Branch Target Address: When an indirect branch is fetch, it reads stored data value from *store data array* using 'predicted load address', hash it with branch pc, and index into branch target buffer (BTB) for predicting branch target address. Different store data values correspond to different targets of an indirect branch, which are stored and subsequently accessed at different BTB indices.

Updating the BTB: If an indirect branch mispredicts, either, because target address is seen for the first time, or it is replaced by another branch target, BTB is updated. Same index computed at BTB lookup time, is used for updating the BTB.

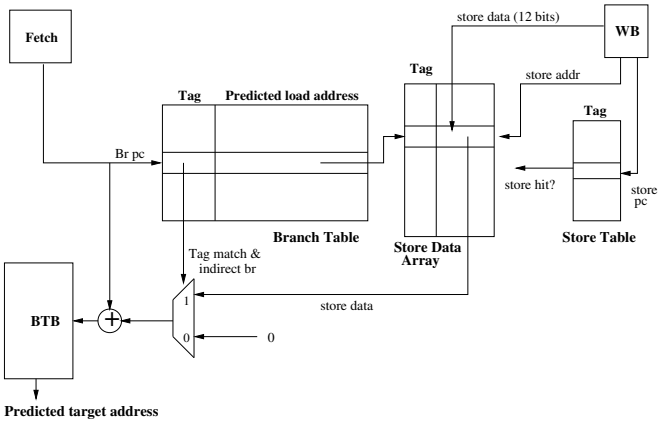

Figure 5. Support for indirect branches

3.2.4 Choosing Between TAGE and SLB Predictor

In a combined TAGE+SLB predictor configuration, not every branch is predicted using SLB predictor. If a branch address matches branch table tag in figure 4, prediction is

Load Address Predictor		
Load Table	32 entries x (12+32+32+10+64)	4800 bits
Link Table	1k entries x (12+32)	45056 bits
SLB Conditional Branch Predictor		
Branch Table	32 entries x (12+32)	1408 bits
Store Table	48 entries x (12+4)	768 bits
T/NT Prediction	1k entries x (12+1)	13312 bits
SLB Indirect Branch Predictor		
St. Data Array	1k entries x (12+12)	24576 bits

Table 2. Implementation cost of SLB predictor

taken from SLB predictor, else prediction is taken from default TAGE predictor.

3.3 Implementation Cost

Load, store, and branch tables shown in figure 4 are all per-static-instruction tables, while link table and T/NT prediction table holds dynamic values. We have sized various structures in SLB predictor based on number of SLB branches observed in benchmarks. Table 4 shows number of SLB branches identified in each benchmark along with their associated load and store instructions. Table 2 shows storage cost for implementing SLB predictor[2]. For reducing implementation cost, link table, T/NT table, and store data array in figures 4 and 5 can be made tagless. Total storage required for implementing SLB conditional branch predictor is 63.8 Kbits (39.8 Kbits, if tagless). An additional 24 Kbits (12 Kbits, if tagless) storage is needed for implementing SLB indirect branch predictor.

4 Experimental Framework

4.1 Simulation Methodology

Results presented in this paper are collected from an ARM performance simulator running benchmarks from EEMBC, SPECint2000 suites [12][34]. In addition, top mispredicting benchmark from SPECint2006 suite, 473.astar, is also included in the experiment. Benchmarks are compiled with ARM RealView compilation tool (RVCT 4.1) [28] with -O3 optimization flag. Our detailed cycle accurate simulator models a superscalar out-of-order processor core with 4-wide, 15 stages integer pipeline, 32KB L1 I and D cache, and 1MB L2 cache. Table 3 shows simulation parameters for the front-end pipeline. We used Bi-Mode branch predictor [21] as our baseline. We then compared the baseline with 1) standalone TAGE predictor [32], and 2) combined TAGE+SLB predictor. We obtain TAGE predictor code from [31] and integrated with our timing simulator.

[2]We assume 32 bits for data and address, and 12 bits for tag.

Table 3. Front-end pipeline parameters

Instruction Fetch	4-wide, 5-deep;
BTB	2048-entry;
Return Address Pred	16-entry;
Conditional Branch Predictors	
Bi-Mode (baseline)	48 Kbits, taken/not-taken/sel each 8k-entry; 13-bit global history register;
TAGE	64 Kbits, 14 components; 130 bit max hist;
SLB	68 Kbits, 32-entry load table; 32-entry branch table; 48-entry store table; 1K-entry T/NT table; 1K-entry link table;
Indirect Branch Predictors	
Target Cache (baseline)	512-entry;
SLB	1k-entry store data array;

Listing 5. Identifying load/store instructions

```
foreach dynamic instruction i, {                                    1
    if(i is store)                                                  2
      store_array.push(store_pc, store_address);                    3
    endif                                                           4
    if(i is a hard-to-predict branch)                               5
      mark load instruction(s) feeding into                         6
      the branch (through register ID matching)                     7
    endif                                                           8
    if(i is marked_load_instruction)                                9
      foreach(store_array.read_store_address()                      10
                                  == load_address) {                11
          mark the store instruction                                12
       }                                                            13
    endif                                                           14
}                                                                   15
```

4.2 Identifying Relevant Load/Store Instructions

SLB predictor relies on compiler to identify program points where data structure associated with a data dependent branch is referenced and modified. For experiments in this paper, we have written a binary profiler for identifying load/store instructions associated with a data-dependent branch. Listing 5 shows pseudo code for our binary profiler.

5 Results and Analysis

5.1 Store to Branch Delay

SLB predictor uses store data for computing branch flags, therefore, store data should be available before branch instruction gets fetched. Table 4 shows cycle count between store data becoming available and branch flags computed using store data getting consumed. Following two program characteristics explain high cycle count between store-branch instruction pair. Firstly, data structure update and traversing typically happen in different program phases. Secondly, even when update and traversal are adjacent to each other, dynamic instructions created by update loop

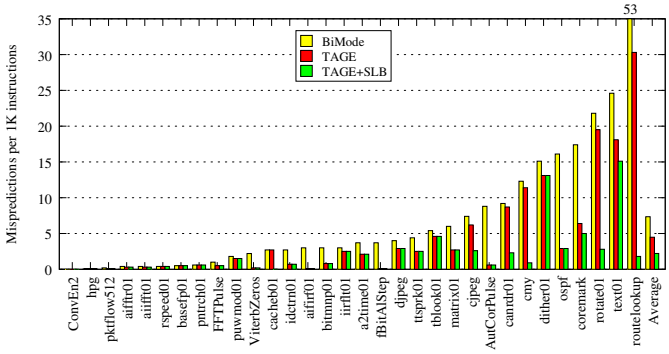

Figure 6. MPKI (EEMBC)

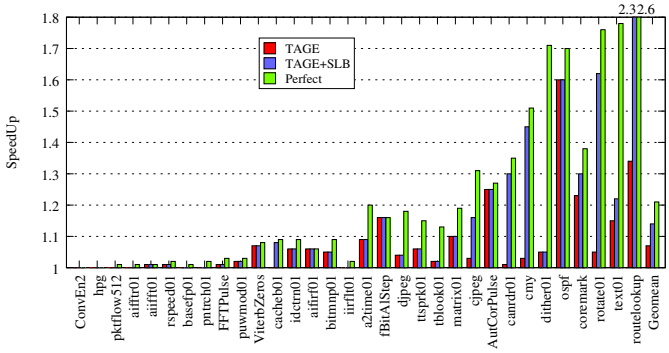

Figure 7. Speedup over baseline predictor. (EEMBC)

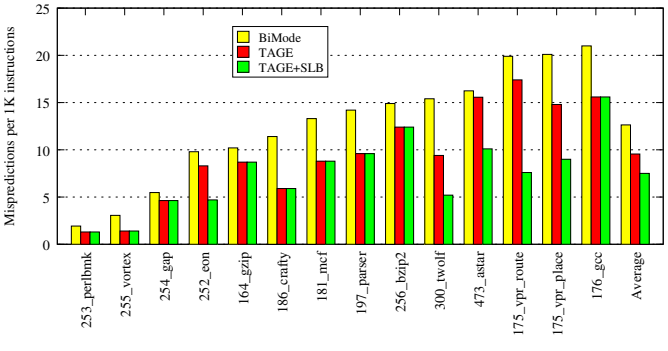

Figure 8. MPKI (SPECint)

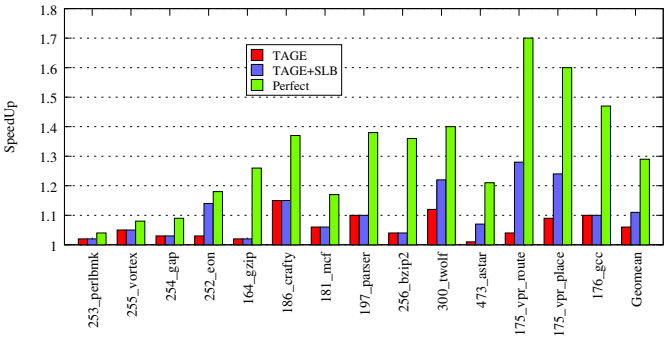

Figure 9. Speedup over baseline predictor. (SPECint)

generates necessary cycles between the first store operation of update loop and the first load-branch operation of traversal loop.

5.2 Branch MPKI

Table 4 shows that only a handful of static branch instructions are predicted using SLB predictor. Yet, these hard-to-predict, data-dependent branches are top contributors to overall misprediction (see table 7 and 8 for top mispredicting branches in each benchmark). Figure 6 and 8 compares MPKI of baseline Bi-Mode predictor with 1) standalone TAGE predictor, and 2) combined TAGE+SLB predictor. As shown in these figures, MPKI remains high for several benchmarks even when using state-of-the-art TAGE predictor. On average, combined TAGE+SLB predictor reduces MPKI for EEMBC suite from 4.48 to 2.21, a reduction of 51%. Similarly, for SPECint suite, MPKI is reduced from 9.56 to 7.50, a reduction of 21%.

Table 7 and 8 show up to 10 top mispredicting branches in benchmarks we have studied. In most cases, except coremark and text01, SLB predictor is able to accurately predict top mispredicting branches of Bi-Mode and TAGE predictor. Branches not handled by SLB predictor are marked as 'x'. In coremark, top mispredicting branch is an indirect branch resulting from a *switch-case* statement inside a *for* loop. In some *case* statements, control variable on which the indirect branch is dependent is also updated. This update is not available before next iteration of the *for* loop. As a result, SLB predictor reads stale value from *store data array*, therefore, mispredicts branch target address.

There are several benchmarks where combined TAGE+SLB predictor has same MPKI as that of the standalone TAGE predictor. This is because we have excluded these benchmarks from SLB analysis due to one of the following three reasons: 1) Benchmark MPKI is already low (e.g. EEMBC benchmarks on the left), 2) Mispredicting branches do not follow ST-LD-BR execution sequence (e.g. EEMBC/dither01, SPECint/164.gzip, SPECint/256.bzip2), and 3) Identifying ST-LD-BR sequence without compiler analysis is hard (e.g. SPECint/176.gcc, SPECint/197.parser).

5.3 SLB Performance Impact

Figure 7 and 9 compares performance improvement when using standalone TAGE predictor, combined TAGE+SLB predictor, and perfect predictor. For EEMBC suite, combined TAGE+SLB predictor doubled the speedup of standalone TAGE predictor over the baseline Bi-Mode predictor. A perfect predictor shows a potential of another

	cacheb 01	cjpeg	canrdr 01	cmy	core--mark	rotate 01	text 01	route lookup	252. eon	300. twolf	473. astar	175.vpr route	175.vpr place
Num of SLB BR	1	1	3	2	11	2	4	1	8	9	17	3	7
Num of SLB LD	1	1	3	1	10	2	5	1	4	9	9	2	4
Num of SLB ST	4	2	21	3	9	2	11	3	24	14	12	9	36
ST-BR Delay (cycles)	1241	774	71	3626	16K	785	1004	53	799	556	256	534	1445K

Table 4. Number of SLB-enabled branches, and associated load/store instructions

7% performance improvement over the baseline. Similarly, for SPECint suite, combined TAGE+SLB predictor shows performance improvement of 11% over baseline Bi-Mode predictor. This is almost double the speedup of standalone TAGE predictor over the same baseline. Perfect predictor shows a potential for 29% performance improvement.

5.4 SLB Area and Power Analysis

Area and power overhead of SLB scheme was estimated using McPAT [22], a framework for modeling processor area and power. Parameters of a generic McPAT RISC core (Alpha21364) were adjusted to match simulated ARM core. TAGE and SLB predictor components were added for estimating their area and power overhead. Table 5 shows area and power overhead of SLB predictor for a 32 nm process technology node.

Table 5. SLB area and power overhead (at 32 nm)

Metric	Core	Fetch Unit	TAGE	SLB	SLB overhead (%)
Area (mm2)	13.99	1.64	0.10	0.07	0.5
Dy. Power (W)	8.61	1.13	0.013	0.015	0.17
St. Power (W)	6.66	0.74	0.023	0.028	0.42

5.5 SLB Timing Analysis

Figure 10 shows critical timing paths in fetch unit. First two paths are through TAGE and SLB branch direction predictor, while the third path is through BTB for branch target address computation. Table 6 shows access time for various structures in fetch unit for a 32 nm technology process using CACTI [19].

Table 6. Access time for branch predictor structures

	BTB	TAGE Component	SLB Tables Branch	SLB Tables T/NT	MUX
Access Time (ns)	0.458	0.22	0.203	0.187	0.041

- Access Time for TAGE Predictor (ns)
  = TAGE component delay + 4*(MUX delay)
  = 0.22 + 4*(0.041)
  = 0.384
- Access Time for SLB Predictor (ns)
  = Branch table delay + T/NT table delay
  = 0.203 + 0.187
  = 0.39

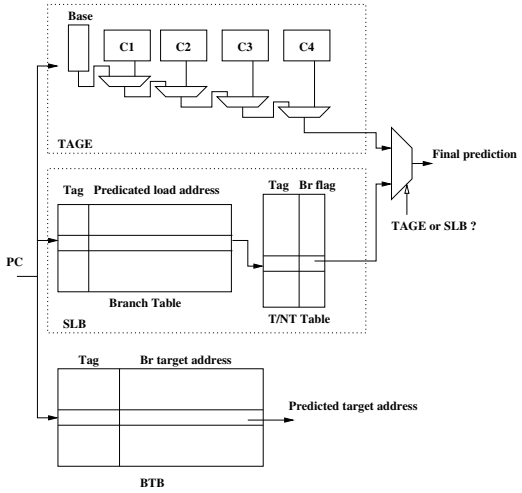

Figure 10. Comparing critical timing paths in fetch unit

- Access Time for BTB (ns)
  = BTB delay
  = 0.458

5.5.1 *Critical Timing Path in Fetch Unit*

Computing next instruction address after a branch instruction involves a) determining if branch is taken or not taken, i.e branch direction prediction, and b) if branch is indeed taken, computing branch target address. Therefore, critical path for fetch unit is the maximum delay between branch direction prediction and branch target address prediction, as shown below:

- Critical Timing Path in Fetch Unit (ns)
  = Max{Br. dir. pred. delay, Br. target pred. delay}
  = Max{Max{TAGE, SLB}+mux delay, BTB delay}
  = Max{Max{0.384, 0.39} + 0.041, 0.458}
  = Max{0.431, 0.458}
  = **0.458**

6 Related Work

Branch prediction research can be categorized into three classes: static branch prediction, dynamic branch prediction, and compiler-assisted dynamic branch prediction.

Static Branch Predictors: In static branch prediction, branch direction is predicted before program is executed,

Benchmarks	Br. Id	Br. Func	Dy. Count	Mispred Rate			MPKI		
				Bi-Mode	TAGE	SLB	Bi-Mode	TAGE	SLB
cache01	1	t_run_test	10k	46.2%	46.2%	0	2.65	2.65	0
cjpeg	1	forward_DCT	576k	48%	48%	0	3.68	3.66	0
	2	encode_mcu_huff	567k	12%	9%	x	0.91	0.67	x
	3	emit_bits	207k	29%	14%	x	0.81	0.39	x
	4	encode_mcu_huff	90k	40%	38%	x	0.48	0.45	x
	5	encode_mcu_huff	90k	40%	32%	x	0.48	0.38	x
canrdr01	1	t_run_test	2k	32%	32%	0	2.62	2.62	0.03
	2	t_run_test	2k	27%	22%	0	2.26	1.86	0.02
	3	t_run_test	2k	20%	24%	0	1.61	1.99	0.02
	4	t_run_test	0.5k	28%	18%	x	0.60	0.38	x
	5	t_run_test	0.3k	31%	20%	x	0.36	0.24	x
	6	t_run_test	0.3k	20%	24%	x	0.21	0.24	x
	7	t_run_test	0.3k	19%	13%	x	0.19	0.14	x
cmy	1	t_run_test	768k	24%	22%	0	8.81	8.29	0
	2	t_run_test	464k	11%	10%	0	2.45	2.33	0.14
coremark	1	core_state_transit	457k	52%	41%	32%	7.21	5.64	4.39
	2	core_state_transit	535k	12%	0	0	1.92	0	0
	3	core_state_transit	121k	23%	0	0	0.85	0	0
	4	core_state_transit	156k	16%	0	0	0.77	0	0
	5	core_state_transit	174k	13%	0	0	0.69	0	0
	6	core_bench_list	694k	2%	0	x	0.61	0	x
	7	core_state_transit	48k	34%	0	0	0.49	0	0
	8	core_list_mergeso	31k	42%	22%	x	0.40	0.21	x
	9	core_state_transit	560k	2%	0	x	0.33	0	x
	10	core_state_transit	154k	6%	4%	0	0.28	0.22	0
rotate01	1	rotateImage	648k	18%	17%	0	16.77	15.50	0
	2	rotateImage	726k	3%	2%	x	3.05	2.72	x
	3	rotateImage	34k	31%	18%	0	1.47	0.89	0
text01	1	parseRule	222k	19%	17%	x	4.57	4.15	x
	2	parseRule	45k	59%	57%	20%	2.87	2.74	0.99
	3	parseRule	93k	23%	12%	x	2.30	1.28	x
	4	strcmp	128k	12%	7%	x	1.75	1.07	x
	5	strncpy	45k	32%	22%	x	1.58	1.08	x
	6	strncpy	46k	26%	15%	x	1.30	0.78	x
	7	parseRule	26k	44%	41%	x	1.24	1.17	x
	8	parseRule	42k	24%	15%	x	1.09	0.67	x
	9	parseRule	47k	20%	16%	2%	1.01	0.80	0.14
	10	parseRule	24k	26%	19%	0	0.70	0.52	0
routelookup	1	pat_search	177k	41%	29%	0	40.57	28.50	0
	2	pat_search	177k	12%	1%	x	12.47	1.78	x

Table 7. Top mispredicting branches and their prediction accuracy for different predictors (EEMBC)

Benchmarks	Br. Id	Br. Func	Dy. Count	Mispred Rate			MPKI		
				Bi-Mode	TAGE	SLB	Bi-Mode	TAGE	SLB
252.eon	1	ggSpectrumSet	20m	7%	6%	0	1.46	1.33	0
	2	ggGridIterator	1789k	37%	35%	x	0.68	0.65	x
	3	mrMaterialshad	1232k	51%	51%	3%	0.65	0.65	0.04
	4	ggGridIterator	1537k	38%	36%	x	0.60	0.58	x
	5	mrGridviewingH	573k	59%	57%	1%	0.34	0.33	0
	6	mrGridviewingH	1113k	28%	2%	x	0.32	0.02	x
	7	mrGridshadowHi	634k	48%	46%	0	0.31	0.30	0
	8	mrMaterialboun	580k	49%	48%	x	0.29	0.28	x
	9	mrSurfaceListv	830k	34%	33%	x	0.29	0.28	x
	10	mrGridviewingH	950k	26%	20%	0	0.25	0.20	0
300.twolf	1	new_dbox_a	3144k	31%	24%	0	1.46	1.12	0
	2	new_dbox	2473k	29%	18%	0	1.10	0.69	0.01
	3	new_dbox_a	3175k	23%	15%	0	1.08	0.74	0
	4	new_dbox_a	3004k	20%	13%	0	0.91	0.58	0
	5	XPICK_INT	780k	62%	1%	x	0.73	0.01	x
	6	new_dbox_a	1140k	29%	21%	0	0.50	0.36	0
	7	new_dbox	1084k	26%	7%	x	0.42	0.12	x
	8	term_newpos	928k	25%	18%	4%	0.35	0.25	0.06
	9	add_penal	645k	30%	16%	x	0.29	0.16	x
	10	term_newpos_a	657k	30%	17%	0	0.29	0.16	0
473.astar	1	makebound2	424k	34%	34%	3%	1.20	1.21	0.11
	2	makebound2	399k	28%	24%	0	0.93	0.81	0
	3	makebound2	424k	25%	22%	12%	0.90	0.79	0.43
	4	makebound2	393k	26%	25%	10%	0.88	0.83	0.33
	5	makebound2	430k	21%	17%	3%	0.75	0.64	0.10
	6	makebound2	400k	22%	20%	6%	0.74	0.67	0.21
	7	makebound2	370k	22%	21%	3%	0.69	0.65	0.09
	8	makebound2	232k	35%	33%	0	0.68	0.64	0
	9	makebound2	390k	18%	16%	13%	0.59	0.53	0.44
	10	makebound2	213k	32%	30%	0	0.57	0.54	0
175.vpr_route	1	route_net	8365k	32%	36%	4%	5.19	5.84	0.75
	2	route_net	11m	14%	13%	0	3.26	2.91	0.13
	3	node_to_heap	4105k	39%	33%	x	3.05	2.63	x
	4	route_net	1247k	80%	3%	x	1.91	0.08	x
	5	node_to_heap	2982k	31%	28%	x	1.80	1.62	x
	6	route_net	5084k	16%	17%	0	1.64	1.66	0.07
	7	route_net	939k	40%	41%	x	0.73	0.75	x
	8	route_net	8880k	3%	3%	x	0.64	0.62	x
175.vpr_place	1	get_non_update	4088k	43%	38%	5%	1.76	1.59	0.22
	2	try_swap	3007k	43%	39%	2%	1.29	1.18	0.08
	3	get_non_update	2321k	44%	40%	10%	1.02	0.94	0.23
	4	get_non_update	2320k	44%	40%	10%	1.02	0.94	0.24
	5	try_swap	2381k	41%	36%	0	0.99	0.87	0
	6	try_swap	3446k	23%	16%	0	0.79	0.55	0
	7	update_bb	1814k	39%	33%	x	0.72	0.60	x
	8	try_swap	3082k	20%	16%	x	0.64	0.50	x
	9	get_bb_from_s	2801k	21%	18%	x	0.60	0.50	x
	10	get_bb_from_s	2784k	21%	17%	x	0.59	0.49	x

Table 8. Top mispredicting branches and their prediction accuracy for different predictors (SPECint)

and same prediction is used for all dynamic instances of that branch. Profiling is used in [13]. Simple heuristics such as *predict all backward branches taken*, and *predict all forward branch not-taken* were proposed in [33]. Machine learning was using in [5] to infer branch behavior of new program using existing programs.

Dynamic Branch Predictors: Dynamic branch prediction techniques propose hardware that attempts to learn branch behavior at run-time. While most dynamic schemes use branch taken/not-taken history information for training the branch predictor, few schemes have explored adding other information to the prediction process.

(i) **Short History Predictors:** James. E. Smith first presented bimodal branch prediction scheme [33]. Repeatedly taken branches will be predicted to be taken, and repeatedly not-taken branches will be predicted as not-taken. Two-level adaptive branch prediction scheme was presented in [37]. It is based on the observation that a branch can have multiple repetitive patterns. The two-level scheme differentiates among these patterns by keeping a record of direction taken by last *m* instances of each branch, and using it to index into a table of k-bit counter. Combining branch predictor was proposed in [24] which combines and takes advantage of different predictors types.

(ii) **Anti-aliasing Predictors:** Several schemes proposed different indexing mechanism for reducing branch aliasing effect. gselect predictor concatenates branch history and branch address, and uses it to index into counter table [27]. gshare predictor uses exclusive OR of branch address with branch history to index into counter table [24]. gskewed predictor uses multiple counter tables indexed by different hash functions [26]. Bi-Mode [21] and YAGS [11] predictors partitioned counter table into two halves – *taken* and *not-taken*. It reduces negative interference by keeping branches biased towards 'taken' direction in the *taken* array, and those biased towards 'not-taken' direction in the *not-taken* array. Agree predictor [35] updates prediction counter based on whether or not branch bias matches branch outcome, irrespective of branch direction.

(iii) **Long History Predictors:** Recent research has shown that prediction accuracy can further be improved by utilizing longer branch history, and using different history length for predicting different branches. Using perceptrons instead of two-bit saturating counter was proposed in [16]. It is based on the observation that not all branches in history are important. O-GEHL [29], PPM-like [25] and TAGE [32] use multiple predictor tables, each indexed by an increasing length of branch history. In O-GEHL, final prediction is computed by summing predictions read from each predictor table. In PPM-like and TAGE final prediction is provided by the longest table that matches the tag.

(iv) **Non-History Based Dynamic Branch Predictors:** Available Register Value Information (ARVI) predictor [7] hashes register values with branch pc to index into prediction table. Branch prediction through Value prediction was proposed in [15]. Address-Branch Correlation (ABC) predictor [14] observed that values inside a data structure tend to be stable, therefore, branch outcome can be correlated simply with address of data structure instead of value inside data structure. We argue that if values inside data structure are stable, multiple iterations over data structure should generate same branch outcome every time, and should be predictable with a history-based predictor. Correlating load address with branch outcome is also used in [1], however, unlike [14], they update predictor in case there are stores to those addresses.

Compiler-Assisted Dynamic Branch Predictors: Compiler-assisted dynamic branch prediction combines strengths of static and dynamic approaches – the low overhead of compiler-time analysis with the effectiveness of dynamic prediction. Wish branch was proposed in [18]. It combines strengths of conditional branches and predication. Compiler generate code for predicated execution, but leaves conditional branches intact. At run-time, if branch turns out to be an easy-to-predict, branch prediction is used, else predicated code is executed. In [23], compiler defines a *prediction function* for each branch, and inserts instructions for computing prediction function.

7 Conclusion and Future Work

Data-dependent branches are single biggest source of remaining branch mispredictions. These branches are commonly associated with program data structures such as arrays, linked lists, trees etc., and follow store-load-branch execution sequence. A set of memory locations is written while building and updating the data structure. During data structure traversal, these locations are read, and used for evaluating branch condition. Branch outcome depends on data values stored in the data structure, which, typically do not have repeatable pattern. Therefore, in addition to history-based predictors, we need a different kind of predictor for data-dependent branches.

Taking advantage of store-load-branch execution sequence, we propose a compiler-assisted Store-Load-Branch (SLB) predictor. For every data-dependent branch, compiler identifies all store instructions that modify the data structure associated with the branch. These store instructions are dynamically tracked, and stored values are used for computing branch flags ahead of time. These branch flags

are temporarily buffered in a hardware structure, and later used for making predictions. Section 5.2 shows that compared to standalone TAGE predictor, a hybrid TAGE+SLB predictor reduces branch MPKI by 21% for SPECint benchmark suite, and by 51% for EEMBC benchmark suite. Estimated power and area overhead of SLB predictor is 0.28% and 0.5% respectively (section 5.4), with no timing overhead (section 5.5).

Finally, we are working on a compiler implementation for identifying relevant load/store instructions using LLVM's Data Structure Analysis (DSA) [20]. In contrast to other alias analysis that operates on individual memory references, DSA operates at the level of entire instance of data structure, and provides context-sensitive mod/ref analysis.

References

[1] M. Al-Otoom, E. Forbes, and E. Rotenberg. EXACT: Explicit Dynamic-Branch Prediction with Active Updates. In *Proceedings of the 7th ACM international conference on Computing frontiers*, CF '10, pages 165–176, New York, NY, USA, 2010. ACM.

[2] ARM. ARM Architecture Reference Manual. `infocenter.arm.com`.

[3] J. L. Baer and T. F. Chen. An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, pages 176–186, New York, NY, USA, 1991. ACM.

[4] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz, and U. Weiser. Correlated Load-Address Predictors. In *Proceedings of the 26th annual international symposium on Computer architecture*, ISCA '99, pages 54–63, Washington, DC, USA, 1999. IEEE Computer Society.

[5] B. Calder, D. Grunwald, D. Lindsay, J. Martin, M. Mozer, and B. Zorn. Corpus-based Static Branch Prediction. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 79–92, New York, NY, USA, 1995. ACM.

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

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

[8] T. F. Chen and J. L. Baer. Effective Hardware-based Data Prefetching for High-Performance Processors. *IEEE Transactions on Computers*, 44:609–623, 1995.

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

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

[11] A. N. Eden and T. Mudge. The YAGS Branch Prediction Scheme. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, MICRO 31, pages 69–77, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[12] EEMBC. The Embedded Microprocessor Benchmark Consortium. `http://www.eembc.org`.

[13] J. A. Fisher and S. M. Freudenberger. Predicting Conditional Branch Directions From Previous Runs of a Program. *SIGPLAN Not.*, 27(9):85–95, Sept. 1992.

[14] H. Gao, Y. Ma, M. Dimitrov, and H. Zhou. Address-Branch Correlation: A Novel Locality for Long-Latency Hard-to-Predict Branches. In *HPCA*, pages 74–85. IEEE Computer Society, 2008.

[15] J. Gonzalez and A. Gonzalez. Control-flow Speculation through Value Prediction. *IEEE Transactions on Computers*, 50(12):1362–1376, 2001.

[16] D. A. Jiménez and C. Lin. Dynamic Branch Prediction with Perceptrons. In *HPCA-7*, page 197, 2001.

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

[18] H. Kim, O. Mutlu, J. Stark, and Y. Patt. Wish Branches: Combining Conditional Branching and Predication for Adaptive Predicated Execution. In *MICRO-38*, pages 12 pp.–54, Nov. 2005.

[19] H. Labs. CACTI 5.3. `http://quid.hpl.hp.com:9081/cacti/`.

[20] C. A. Lattner. *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, Champaign, IL, USA, 2005. AAI3182303.

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

[22] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 469–480, New York, NY, USA, 2009. ACM.

[23] S. Mahlke and B. Natarajan. Compiler Synthesized Dynamic Branch Prediction. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 29, pages 153–164, Washington, DC, USA, 1996. IEEE Computer Society.

[24] S. McFarling. Combining Branch Predictors. Technical Report TN-36, Digital Western Research Laboratory, 1993.

[25] P. Michaud. A PPM-like, Tag-based Branch Predictor. In *In Proceedings of the First Workshop on Championship Branch Prediction (in conjunction with MICRO-37*, 2004.

[26] P. Michaud, A. Seznec, and R. Uhlig. Trading Conflict and Capacity Aliasing in Conditional Branch Predictors. In *Proceedings of the 24th annual international symposium on Computer architecture*, ISCA '97, pages 292–303, New York, NY, USA, 1997. ACM.

[27] S.-T. Pan, K. So, and J. T. Rahmeh. Improving the Accuracy of Dynamic Branch Prediction using Branch Correlation. In *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, ASPLOS-V, pages 76–84, New York, NY, USA, 1992. ACM.

[28] RVCT. Realview Compilation Tools. `http://www.keil.com/arm/realview.asp`.

[29] A. Seznec. Genesis of the O-GEHL Branch Predictor. `http://www.jilp.org/vol7`, 2005.

[30] A. Seznec. The L-TAGE Branch Predictor. `http://www.jilp.org/vol9`, 2007.

[31] A. Seznec and P. Michaud. Championship Branch Prediction. `ftp://ftp.irisa.fr/local/caps/TAGE.tar.gz`.

[32] A. Seznec and P. Michaud. A Case for (Partially)-Tagged Geometric History Length Predictors. `http://www.jilp.org/vol7`, 2006.

[33] J. E. Smith. A Study of Branch Prediction Strategies. In *Proceedings of the 8th annual symposium on Computer Architecture*, ISCA '81, pages 135–148, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

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

[35] E. Sprangle, R. S. Chappell, M. Alsup, and Y. N. Patt. The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference. In *Proceedings of the 24th annual international symposium on Computer architecture*, ISCA '97, pages 284–291, New York, NY, USA, 1997. ACM.

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

[37] T.-Y. Yeh and Y. N. Patt. Alternative Implementations of Two-Level Adaptive Branch Prediction. In *IN PROCEEDINGS OF THE 19TH ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE*, pages 124–134, 1992.