

Branch Prediction via Load Value Prediction: A Case of BALL (Branch-ALU-Load-Load) Predictor

Jun Fan
junfan.me@gmail.com

Abstract

Load-data-dependent branches result in mis-predictions in the state-of-the-art history-based branch predictor TAGE-SC-L.

In this paper, **BALL (Branch-ALU-Load-Load)** predictor is proposed as a complementary predictor to TAGE-SC-L, targeting at load-data-dependent branches. When TAGE-SC-L mis-predicts, BALL constructs a Branch-ALU-Load-Load dependency chain. BALL first predicts the address of the load by constant stride pattern, branch path history or pointer-chasing pattern, then uses this address to read a data cache to obtain the predicted load value. The final predicted load value is compared with another predicted load value, or with zero to predict the dependent branch direction.

Evaluated on the CBP2025 training traces, the combined 64 KB TAGE-SC-L+128 KB BALL predictor achieves **3.542 MPKI** and **148.828 Cycles/WPPKI**, which is 5.57% and 2.43% lower than 64KB TAGE-SC-L, but 3.32% and 2.35% higher than 192 KB TAGE-SC-L, with 9.85% and 3.20% lower on floating-point workloads than 192 KB TAGE-SC-L.

1 Introduction

To motivate the BALL predictor, four examples from CBP2025 training traces are analyzed. The branches have the most mis-predictions by 64KB TAGE-SC-L in the traces. The instructions in the tables are oldest to youngest from the top to the bottom.

In Example 1, the branch (PC 0x2a3328) is dependent on three loads (PC 0x2a3304, 0x2a331c and 0x2a3320), whose address can be predicted by fixed address pattern, constant stride pattern and pointer chasing pattern. Then the load value can be predicted by reading a data cache using the predicted address.

After the load values are predicted, they can be compared to get the flag register value. If the operation of ALU is static, the operation can be trained and recorded, then predicted by its PC.

Finally, the branch can be predicted by the flag register value. If the relationship between flag register and branch direction is static, it can be trained and recorded, then predicted by its PC.

PC	Inst	SrcReg	DstReg	Pattern
0x2a3304	load	0x1	0x3	load address is fixed (for a period of access).
0x2a331c	load	0xf	0x2	load address is constant stride (stride = -0x8).
0x2a3320	load	0x2	0x4	load address is base + offset, base is the chaining load data, offset is constant (0x10).
0x2a3324	alu	0x4, 0x3	0x40	compare and set flag register (0x40).
0x2a3328	condBr	0x40		branch depends on flag register (0x40).

Table 1. Example 1 of int_21 trace

In Example 2, The branch (PC 0xffff0d8f288) is dependent on one load (PC 0xffff0d8f284), whose address can also be

predicted by constant stride but is wrapped between two address boundaries. No explicit ALU between the load and the branch, but there is an implicit comparison between the load value and zero.

PC	Inst	SrcReg	DstReg	Pattern
0xffff0d8f284	load	0xb, 0x9	0xc	load address is constant stride (stride = 0x8), wrap between 0xfffffd1e6c0 and 0xfffffd1e780.
0xffff0d8f288	condBr	0xc		branch depends on src reg value compared with zero.

Table 2. Example 2 of int_1 trace

In Example 3, the branch (PC 0x4bf194) is dependent on one load (PC 0x4bf134), whose address can also be predicted by constant stride. The fp operation compares the load value in floating-point format with zero and sets the flag register.

PC	Inst	SrcReg	DstReg	Pattern
0x4bf134	load	0x4, 0x1	0x32	load address is constant stride (stride = 0x4)
0x4bf188	fp	0x32	0x40	compare src reg value with zero and set flag register (0x40)
0x4bf194	condBr	0x40		branch depends on flag register (0x40).

Table 3. Example 3 of fp_8 trace

In Example 4, the branch (PC 0x402c04) is dependent on two load values (PC 0x402bf0 and 0x402bfc). Unlike Example 1-3, the load address is correlated with branch path history, which can be predicted using E-VTAGE [15], but to predict the address, not the value.

PC	Inst	SrcReg	DstReg	Pattern
0x402bf0	load	0x0, 0x2	0x2	load address is correlated with branch path history
0x402bfc	load	0x0, 0x3	0x3	load address is correlated with branch path history
0x402c00	alu	0x3, 0x2	0x40	compare and set flag register (0x40).
0x402c04	condBr	0x40		branch depends on flag register (0x40).

Table 4. Example 4 of fp_13 trace

To summarize, the key ideas of the BALL predictor are:

- Branch-ALU-Load-Load chain is constructed when TAGE-SC-L mis-predicts.
- Branch-required load values, ALU operation, flag register value and corresponding branch direction can be trained and predicted by their PC.
- Load value is predicted by first predicting its address, then using the address to access a data cache. Load address can be predicted by the following three patterns:
 - Constant stride pattern, including
 - ◆ fixed pattern: address sequence is [A, A, A, A, ...], can be treated as stride $s=0$.
 - ◆ increment pattern: address sequence is [A, A+s, A+2s, A+3s, ...], stride is s ($s \neq 0$).
 - ◆ wrap pattern: address is wrap between A and B, address sequence repeated by [A, A+s, A+2s, A+3s, ..., B-3s, B-2s, B-s, B], stride is s ($s \neq 0$).

Reference	Mechanism	Similarity	Difference
Branch precomputation. [1-6]	SSMT (Simultaneous Subordinate Microthreading) [1] constructs microthread by hand through profiling, and DP-SSMT (Difficult-path SSMT) [2] constructs microthread at run-time with branch path leading to the mis-predicted branch. DP-SSMT is pruning with constant and stride-based value and address predictor to shorten microthread computation. [3] constructs <i>speculative slices</i> by hand through profiling and can contain control-flow. Branch correlating hardware is proposed for conditionally-executed branch, mis-speculation recovery and late prediction. DDMT (Speculative Data-driven Multi-Threading) [4] forks a copy of critical instruction computation executed with main thread in parallel and can integrate the result directly into the main thread. Branch Runahead [5] constructs dependence chain between two consecutive H2P branches and precomputes in a dedicated dependence chain engine. TEA [6] can track long dependence chains across complex control flow and utilize existing backend execution resources and flush mechanism. SSMT [1], DP-SSMT [2], [3] and Branch Runahead [5] can override main branch predictor result at fetch or issue an early misprediction flush. DDMT [4] integrates precomputation results to main thread at rename stage. TEA [6] uses the precomputation result to issue early misprediction flush.	BALL builds the dependence chain and precompute branch.	BALL only constructs the dependence chain to a very specific pattern including at most one alu and four loads and must be terminated at loads (Section 4.6.1), while [1-6] are more general dependence chain construction and precomputation. In the CBP2025 simulator, BALL can only override main branch predictor result at fetch, cannot issue an early misprediction flush.
Branch precomputation with value prediction [7]	Value prediction for outputs of arithmetic and load instruction, or inputs of comparison instruction. And execute comparison according to the predicted inputs. Precompute branch according to the predicted arithmetic, load or comparison result.	BALL predicts load value in the dependence chain for branch precomputation.	BALL does not predict load value directly, but predicts the load address first, then access BP Data Cache to predict the load value (Section 4.6.3.1).
Branch precomputation with load value prediction by address prediction. [8-10]	[8] extracts <i>branch flow</i> where branch is predictable with arithmetic value or load address of constant stride pattern. And precomputes <i>branch flow</i> ahead of normal flow to override main branch predictor result at fetch or issue an early misprediction flush. [9] identifies 3D-Branch (Direct Data Dependent Branch) which only has one feeding load and only simple operations to compute the branch. And executes the feeding load or using E-VTAGE to predict the feeding load address, then use the load value to precompute branch outcome to issue early misprediction flush. LDBP (Load Driven Branch Predictor) [10] predicts constant load address ahead of time and computes the branch-load backward slice to override main branch predictor result at fetch.	BALL predicts load value by first predicting the address and using the load value to precompute.	BALL does not identify branches with only one feeding load, but with four feeding loads at maximum (Section 4.6.1). BALL predicts load addresses with constant stride, pointer-chasing pattern (Section 4.6.2.3) and with E-VTAGE (Section 4.5). BALL supports only three comparison operations (Section 4.6.2.2 and 4.6.3.2). [8-10] supports more types of alu operations. BALL does not count timeliness, but [8][10] triggers load address by a distance with address needed by current branch.
BP through register value correlation. [11]	ARVI (Available Register Value Information) correlates branch outcome with physical register value, logical register index and the depth along the branch data dependence chain. The load instruction is termination of dependence chain.	BALL builds dependence chain which is also terminated at load.	BALL does not correlate branch outcome with register value, register index or depth of dependence chain.
BP through load address correlation. [12-13]	ABC [12] (Address-Branch Correlation) correlates branch outcome with address of long-latency cache-missing load with observation that key data structures are stable. EXACT [13] (Explicit dynamic-branch prediction with ACTIVE updates) correlates branch outcome with ID of a distant prior branch (ID is a hash of branch PC and its producer load address). Store to the addresses on which a dynamic branch depends directly update the prediction.	BALL predicts branch using load information. BALL considers the impact by store.	BALL does not correlate the branch outcome with load address directly but predicts load address and then predicts load value using the load address (Section 4.6.3.1). BALL considers the impact by store through BP Data Cache (Section 4.3) and Store Resolve Queue (Section 4.4).
VP through address prediction. [14]	Load value is predicted through address prediction, and Stride Address Prediction (SAP) is to predict load address with constant stride pattern.	BALL predicts load value through address prediction.	BALL only predicts for load required by branch, and BALL extends a wrap sub-pattern (Section 4.6.2.3 and 4.6.3.1).
VP through branch path history. [15]	E-VTAGE uses global branch path history for value prediction.	BALL utilizes branch path history.	BALL uses E-VTAGE to predict load address, not load value (Section 4.5).

Table 5. Similarities and Differences between the BALL and Related Works (BP: Branch Prediction, VP: Value Prediction)

- Branch path history.
- Pointer-chasing pattern: using chaining load data as base address and plus a constant offset.
- When all necessary load values are predicted, the flag register value is predicted by comparison ALU with the load value. The comparison ALU includes
 - Compare two integer values.
 - Compare one integer value with zero.
 - Compare one floating point value with zero (treated as an ALU operation in this paper for unification).
- Finally, the branch direction is predicted using the predicted flag register value.

2 Related Work

Table 5 lists similarities and differences between the BALL predictor and related work.

3 High-Level Design Overview

As in Figure 1, the submission is a combined TAGE-SC-L+BALL predictor. The BALL and the TAGE-SC-L predict independently, final prediction is selected by a *ball_tage_sel* counter (training in Section 4.6.2.1, prediction in Section 4.6.3.3).

4 Predictor Operation

The BALL predictor consists of eight main components. Auxiliary components (Architecture Register File, Commit Queue, BP Data Cache and Store Resolve Queue) will be introduced first. The E-VTAGE [15] is followed with modifications for the BALL. Finally, the main tables CondBR Chain Table, ALU Pattern Table and Load Pattern Table are introduced, along with detail training and prediction mechanism.

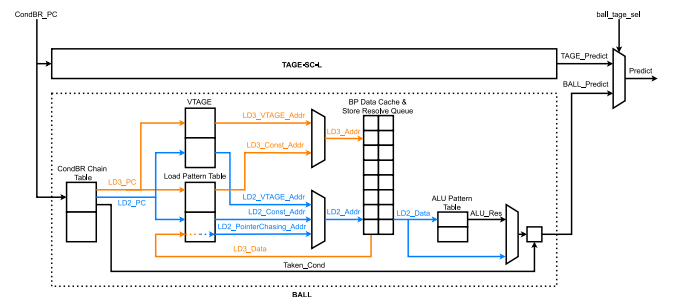


Figure 1. TAGE-SC-L+BALL

4.1 Architecture Register File

Since no interface to architecture register file is provided by the CBP2025 simulator, dedicated resources are used to track 32x64 bits integer registers r0-r31, 32x128 bits fp/simd registers r32-r63 and a flag register r64 at commit stage.

The Architecture Register File is for multiple purposes:

- To get store data and update BP Data Cache (Section 4.3).
- To build the relationship between flag values with branch direction (Section 4.6.2.1).
- To determine alu operation (Section 4.6.2.2).
- To calculate offset of a load (Section 4.6.2.3).

4.2 Commit Queue

Commit Queue is a 96-entry FIFO to record instruction class, PC, piece, source register and destination register at commit stage. The Commit Queue is only updated for instructions which update a destination register (so may impact branch direction).

4.3 BP Data Cache

Since no L1 data cache interface is provided by the CBP2025 simulator, the BP Data Cache is used to track load/store values, which is 128 sets, 11 ways with LRU replacement policy. The set/way number is chosen to meet the 192 KB cost requirements while making the BP Data Cache capacity as large as possible. In each 64-byte cacheline, 16 bits valid flag are for each 4-byte granule, because individual load/store instructions are tracked, not the whole cacheline read by fill buffer. The read data width is 8-byte and higher bits are masked to zero if load *mem_sz* is less.

4.4 Store Resolve Queue

The 8-entry Store Resolve Queue is used to track stores that are resolved but not committed. If a load has the same address as stores in the Store Resolve Queue, then the BP Data Cache lookup is miss. The Store Resolve Queue is pushed with address when store resolved and popped when the store committed. It cannot track all stores accurately with limited entries, so the ready-to-commit store may not find its address in the Store Resolve Queue, because it's already been popped by younger resolved stores.

4.5 E-VTAGE

E-VTAGE [15] is to predict load address that is correlated with branch path history. The E-VTAGE from 8KB EVES predictor is adopted with the modifications below:

- The original E-VTAGE predicts 64-bit value for all instructions. For the BALL predictor, only 48-bit branch-required load address is predicted.
- The original E-VTAGE allocates a new entry with a probability depending on the instruction type, latency and actual result. For the BALL predictor, E-VTAGE always allocates a new entry when misses and mis-prediction for the branch-required load address.

4.6 CondB Chain Table, ALU Pattern Table and Load Pattern Table

4.6.1 Allocation

When the mis-predicted branch is committed, the dependence chain will try to be established by searching for the RAW

relationship in the Commit Queue. The established dependence chain is recorded in CondB Chain Table, which is a 256-entry direct mapped table indexed by branch PC[9:2]. Each entry has one Branch Node (b_0_0), one ALU Node (alu_1_0) and four Load Nodes (load_2_0, load_2_1, load_3_0 and load_3_1). The four nodes are linked by the following rules:

- The root node is b_0_0, which is the mis-predicted branch. And the source register of b_0_0 is updated by alu_1_0 or load_2_0. If updated by load_2_0, then alu_1_0 is null.
- If alu_1_0 has two source registers, they are updated by load_2_0 and load_2_1. If alu_1_0 only has one source register, it is only updated by load_2_0, while load_2_1, and load_3_1 are null.
- The source register of load_2_0 is updated by load_3_0. Or load_2_0 is the leaf node and load_3_0 is null.
- The source register of load_2_1 is updated by load_3_1. Or load_2_1 is the leaf node and load_3_1 is null.
- The leaf node must be Load Node. Otherwise, the dependence chain failed to be established.

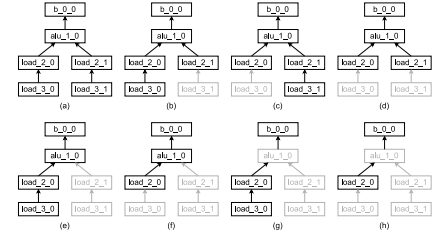


Figure 2. All forms of CondB Chain

Figure 2 lists all forms of dependence chain in the CondB Chain Table. The black node is valid, and the gray node is null. In Section 1, Example 1 in Table 1 is in the form of chain (b), Example 2 in Table 2 is chain (h), Example 3 in Table 3 is chain (f) and Example 4 in Table 4 is chain (d).

Then CondB Chain Table is updated as follows:

- b_0_0: tag with the branch PC[23:10], *taken*[16] is cleared and *ball_tag_sel* is set to half of the max value (128).
- alu_1_0: pc with the alu PC[12:2], and *src_1_valid* is set if the alu has two source registers.
- load_2_0, load_2_1, load_3_0, load_3_1: pc with the load PC[24:2], *piece* with the load piece[0].

When the CondB Chain Table is allocated, ALU Pattern Table and Load Pattern Table are also allocated.

ALU Pattern Table is a 512-entry direct mapped table indexed by alu PC[10:2] and tagged by PC[12:11].

Load Pattern Table is a 512-entry direct mapped table indexed by load {PC[9:3], PC[10], piece[0]} and tagged by {PC[24:11], PC[2]}. This trick is to prevent conflict between PC 0x4bf128 and 0x4cb528 in fp_8, if the Load Pattern Table is indexed by {PC[9:2], piece[0]}. And to prevent doubling the cost if indexed by {PC[10:2], piece[0]}. Piece[0] is to prevent address interference by different pieces from the same load.

4.6.2 Training

Unless otherwise noted, all confidence counters are saturation counters and increase by 1 under increment condition, otherwise decrease by 1. The corresponding pattern, e.g. *constant_stride*, is only updated when the confidence counter is 0.

4.6.2.1 Training of CondBR Chain Table

Conf Counter	Increase Condition
<i>ball_tage_sel</i> ∈ [0, 255]	Trained when a resolved branch hits the CondBR Chain Table. Increase if BALL hit-predicted, TAGE mis-predicted. Decrease if the BALL mis-predicted.
<i>taken[r64]</i> ∈ [-3, 3]	Trained when a committed branch hits the CondBR Chain Table. Increase if it is taken. Decrease if it is not-taken.

4.6.2.2 Training of ALU Pattern Table

ALU Pattern Table is trained when a committed ALU hits.

Conf Counter	Increase Condition
<i>cmp_conf</i> ∈ [0, 1]	If the ALU has two source registers, the calculated flag value (comparing the source registers) is the same as r64.
<i>fcmp_zero_conf</i> ∈ [0, 1]	If the ALU has one floating-point source register, the calculated flag value (comparing the source register with zero) is the same as r64.

4.6.2.3 Training of Load Pattern Table

Load Pattern Table is trained when a committed load hits.

First, load memory access size *mem_sz* is recorded with 2'b00 for 1-byte, 2'b01 for 2-byte, 2'b10 for 4-byte and 2'b11 for 8-byte.

Second, constant stride patterns, offset by pointer chasing pattern are trained. The *wrap_target_mem_va* and *wrap_mem_va* records only the lowest 12-bit of *mem_va*, and the full 48-bit *mem_va* is retrieved by padding *last_mem_va*[47:12].

Pattern	Conf Counter	Increase Condition
<i>constant_stride</i> ∈ [-4095, 4095]	<i>constant_stride_conf</i> ∈ [0, 7]	When current stride (difference between the <i>mem_va</i> of current load and the recorded <i>last_mem_va</i> when committed) is the same as <i>constant_stride</i> .
<i>wrap_target_mem_va</i> <i>wrap_mem_va</i>	<i>wrap_conf</i> ∈ [0, 7]	The sign of current stride is different from the sign of <i>constant_stride</i> , and current <i>mem_va</i> is the same as <i>wrap_target_mem_va</i> , <i>last_mem_va</i> is the same as <i>wrap_mem_va</i> .
<i>offset</i> ∈ [-31, 31]	<i>offset_conf</i> ∈ [0, 7]	When current load offset (by subtracting first source register value from the load <i>mem_va</i>) is the same as <i>offset</i> .

Third, the in-flight number of loads is recorded. When any fetch PC hits the Load Pattern Table, *in_flight_cnt* increases by 1, and when the load committed, *in_flight_cnt* decreases by 1. When the first fetch-hit load is committed, *in_flight_cnt_valid* is set, so *in_flight_cnt* becomes usable to predict constant stride address.

4.6.3 Prediction

When a fetch PC hits the CondBR Chain Table, load value, flag register value, branch direction are predicted sequentially.

4.6.3.1 Prediction of Load Value

Load address by branch path history is predicted when a fetch PC hits Load Pattern Table and E-VTAGE makes a prediction, then Load Pattern Table sets *vtage_predict_addr_valid* and updates *vtage_predict_addr*, which is used later when branch is at fetch.

Load address by constant stride pattern is predicted when a branch hits the CondBR Chain Table, the *pc* from valid Load Nodes hit Load Pattern Table and *constant_stride* is max value (7). The predicted address is calculated by *last_commit_mem_va* + *in_flight_cnt* * *constant_stride*. If the *wrap_conf* is max value (7), then the load is the wrap pattern, and when the predicted constant address exceeds *wrap_mem_va*, the address needs to be wrapped to start from *wrap_target_mem_va*.

Load address by pointer chasing pattern is predicted when the value of the dependent load can be predicted and *offset_conf* is max value (7). The predicted address is calculated by load data plus *offset*.

The final predicted address is selected by fixed priority: branch-path-history > wrap > fixed/increment > pointer-chasing.

The final predicted load value is obtained if lookup hit the BP Data Cache and miss the Store Resolve Queue.

4.6.3.2 Prediction of Flag Register Value

After all load values from valid Load Nodes are predicted. Flag register value is predicted if ALU Node of the hit CondBR Chain Table entry is valid, and *cmp_conf* or *fcmp_zero_conf* is their max value (1). The predicted load value for the ALU is compared with another load value (*cmp_conf* is 1) or treated as floating-point value and compared with zero (*fcmp_zero_conf* is 1) to predict the flag register value. Flag register value can also be predicted if ALU Node of the hit CondBR Chain Table entry is not valid, the load value from *load_2_0* is compared with zero to predict the flag register value.

4.6.3.3 Prediction of Branch Direction

If *taken[predicted_flag_register_value]* of the hit CondBR Chain Table entry is max value (3), then BALL predicts taken, if is min value (-3) then BALL predicts not-taken. BALL does not make a prediction when *taken[predicted_flag_register_value]* is not max/min value, or when any load values of valid Load Nodes or flag register value of valid ALU Node cannot be predicted.

If the BALL makes a prediction and *ball_tage_sel* of the hit CondBR Chain Table entry is not less than the half of the max value (between 128 and 255), then use the BALL prediction as the final prediction, otherwise TAGE prediction is used.

5 Experimental Results and Analysis

Experimental results are evaluated on the CBP2025 simulator with the training traces. The 64KB and 192KB TAGE-SC-L for comparison are provided in the CBP2025 simulator.

As in Table 6, the combined 64 KB TAGE-SC-L+128 KB BALL predictor achieves 3.542 MPKI and 148.828 CycWPPKI, which is 5.57% and 2.43% lower than 3.751 MPKI and 152.541 CycWPPKI of 64KB TAGE-SC-L predictor.

Compared with 64KB TAGE-SC-L, the combined 64 KB, TAGE-SC-L+128 KB BALL predictor shows 12.82% and 8.21% lower MPKI in fp and int workloads, but very little or no improvement in compress, infra, media and web workloads.

Workloads	64KB TAGE-SC-L		192KB TAGE-SC-L		64 KB TAGE-SC-L+128 KB BALL					
	MPKI	CycWPPKI	MPKI	CycWPPKI	MPKI	MPKI v.s. 64KB	MPKI v.s. 192KB	CycWPPKI	CycWPPKI v.s. 64KB	CycWPPKI v.s. 192KB
compress	2.765	84.778	2.724	83.925	2.761	-0.15%	1.36%	84.758	-0.02%	0.99%
fp	4.249	169.192	4.109	165.343	3.704	-12.82%	-9.85%	160.057	-5.40%	-3.20%
infra	2.473	210.344	2.402	208.146	2.477	0.16%	3.11%	210.702	0.17%	1.23%
int	4.657	172.436	4.182	162.109	4.275	-8.21%	2.21%	165.346	-4.11%	2.00%
media	0.890	29.192	0.818	27.862	0.891	0.12%	8.86%	29.244	0.18%	4.96%
web	3.722	119.518	3.236	109.311	3.714	-0.22%	14.76%	119.310	-0.17%	9.15%
all	3.751	152.541	3.428	145.411	3.542	-5.57%	3.32%	148.828	-2.43%	2.35%

Table 6. MPKI and CycWPPKI of Different Workload

Traces	64KB TAGE-SC-L		192KB TAGE-SC-L		64 KB TAGE-SC-L+128 KB BALL						128KB BALL	
	MPKI	CycWPPKI	MPKI	CycWPPKI	MPKI	MPKI v.s. 64KB	MPKI v.s. 192KB	CycWPPKI	CycWPPKI v.s. 64KB	CycWPPKI v.s. 192KB	Coverage	Accuracy
int_21	24.772	1018.848	24.294	997.407	15.865	-35.95%	-34.70%	842.573	-17.30%	-15.52%	35.02%	97.69%
fp_13	12.718	265.593	12.396	258.063	8.928	-29.80%	-27.98%	193.321	-27.21%	-25.09%	29.34%	98.02%
fp_8	11.112	249.653	10.542	239.136	7.548	-32.07%	-28.40%	199.706	-20.01%	-16.49%	33.57%	96.30%
int_1	11.565	469.670	11.326	463.769	9.109	-21.24%	-19.57%	428.044	-8.86%	-7.70%	20.95%	97.52%
int_2	10.829	457.997	10.561	451.805	8.576	-20.81%	-18.80%	418.969	-8.52%	-7.27%	20.21%	97.73%
fp_10	2.032	44.510	1.894	42.178	1.821	-10.38%	-3.84%	41.086	-7.69%	-2.59%	9.54%	99.80%
int_30	12.564	299.880	11.581	281.818	12.395	-1.35%	7.03%	297.289	-0.86%	5.49%	1.33%	98.24%
int_29	12.132	291.486	11.256	275.374	11.977	-1.28%	6.41%	289.200	-0.78%	5.02%	1.22%	98.89%
int_19	0.662	30.230	0.592	27.777	0.508	-23.27%	-14.14%	28.458	-5.86%	2.45%	23.54%	99.15%

Table 7. Traces Improved by BALL (in descending order by absolute value of MPKI reduction)

When the combined 64 KB TAGE-SC-L+128 KB BALL predictor is compared with an ISO area predictor, it's 3.32% and 2.35% higher than 3.428 MPKI and 145.411 CycWPPKI of 192 KB TAGE-SC-L. Still, 64 KB TAGE-SC-L+128 KB BALL predictor shows 9.85% and 3.20% lower MPKI and CycWPPKI in fp workloads, indicating fp workloads can benefit more from load-data based predictor than branch path history based predictor.

As in Table 7, there are in total 9 traces can be improved by the BALL predictor over 64KB TAGE-SC-L, the most MPKI improvement of TAGE-SC-L+BALL is int_21 in int workloads and fp_13 in fp workloads. When compared with 192 KB TAGE-SC-L, there are in total 6 traces can be improved by the BALL.

The BALL coverage, how many 64KB TAGE-SC-L mis-predictions can be eliminated by BALL, is calculated by

$$coverage = \frac{tage_mis_ball_hit - tage_hit_ball_mis}{tage_mis}$$

where *tage_mis_ball_hit*: total number that 64KB TAGE-SC-L mis-predicts but BALL hit-predicts, *tage_hit_ball_mis*: total number that 64KB TAGE-SC-L hit-predicts but BALL mis-predicts, *tage_mis*: total number that 64KB TAGE-SC-L mis-predicts.

The BALL accuracy, the proportion of BALL hit-predictions to the BALL total predictions, is calculated by

$$accuracy = \frac{ball_hit}{ball_hit + ball_mis}$$

where *ball_hit*: total number that BALL hit-predicts, *ball_mis*: total number that BALL mis-predicts.

The coverage varies largely between different traces (from 1.22% to 35.02%), but the accuracy is stable and high (from 96.30% to 99.80%).

As in Table 8, the most MPKI improvement of BALL are from load address prediction by constant stride pattern (fixed and increment) (lower MPKI by 4.45%), and from ALU prediction by comparing two integer values (lower MPKI by 3.20%).

Prediction Mechanism	MPKI	CycWPPKI
64KB TAGE-SC-L	3.751	152.541
LD and ALU All Enabled	3.542 (-5.57%)	148.828 (-2.43%)
LD Const Fix/Inc Disabled	3.709 (-1.12%)	151.734 (-0.53%)
ALU CMP Disabled	3.662 (-2.37%)	151.178 (-0.89%)
LD Pointer Chasing Disabled	3.626 (-3.33%)	150.500 (-1.34%)
ALU CMP Zero Disabled	3.595 (-4.16%)	149.687 (-1.87%)
LD E-VTAGE Disabled	3.576 (-4.67%)	149.490 (-2.00%)
ALU FCMP Zero Disabled	3.576 (-4.67%)	149.333 (-2.10%)
LD Const Wrap Disabled	3.544 (-5.52%)	148.832 (-2.43%)

Table 8. Different Prediction Mechanism Impacts

6 Conclusion

The BALL predictor can be a complementary predictor to TAGE-SC-L for load-data-dependent branches, by constructing Branch-ALU-Load-Load dependence chain for TAGE-SC-L mis-predicted branch, then predicting load value through address prediction, predicting ALU result using the load value and finally predicting branch direction using the ALU result.

The main source of prediction latency is from accessing the BP Data Cache for load value, which is doubled for pointer chasing pattern. To overcome latency issue, load address of constant stride patterns can look ahead as in [10] and override main branch predictor result at fetch. Load addresses predicted by branch path history cannot look ahead, but the BALL prediction can issue an early misprediction flush to minimize branch misprediction penalties as in [9], more investigation is needed to explore the impact of the latency on performance.

In the CBP2025 simulator, some CPU structures are not available. If L1 Data Cache can be accessed by the branch predictor, the BP Data Cache (97.45 KB) can be removed, and the BALL costs 25.16 KB. More investigation is needed, e.g. the data cache is accessed when there is no higher priority request or adding dedicated read port for the BALL.

References

- [1] R.S. Chappell, J. Stark, S.P. Kim, S.K. Reinhardt, and Y.N. Patt. 1999. Simultaneous subordinate microthreading (SSMT). In Proceedings of the 26th International Symposium on Computer Architecture (Cat. No.99CB36367). 186–195. doi:10.1109/ISCA.1999.765950
- [2] R.S. Chappell, F. Tseng, A. Yoaz, and Y.N. Patt. 2002. Difficult-path branch prediction using subordinate microthreads. In Proceedings 29th Annual International Symposium on Computer Architecture. 307–317. doi:10.1109/ISCA.2002.1003588
- [3] C. Zilles and G. Sohi. 2001. Execution-based prediction using speculative slices. In Proceedings 28th Annual International Symposium on Computer Architecture. 2–13. doi:10.1109/ISCA.2001.937426
- [4] A. Roth and G.S. Sohi. 2001. Speculative data-driven multithreading. In Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture. 37–48. doi:10.1109/HPCA.2001.903250
- [5] Stephen Pruett and Yale Patt. 2021. Branch Runahead: An Alternative to Branch Prediction for Impossible to Predict Branches. In MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (Virtual Event, Greece) (MICRO '21). Association for Computing Machinery, New York, NY, USA, 804–815. doi:10.1145/3466752.3480053
- [6] Aniket Deshmukh, Lingzhe Cai, and Yale N. Patt. 2024. Timely, Efficient, and Accurate Branch Precomputation. In 2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO). 480–492. doi:10.1109/MICRO61859.2024.00043
- [7] J. Gonzalez and A. Gonzalez. 1999. Control-flow speculation through value prediction for superscalar processors. In 1999 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00425). 57–65. doi:10.1109/PACT.1999.807406
- [8] A. Farcy, O. Temam, R. Espasa, and T. Juan. 1998. Dataflow analysis of branch mispredictions and its application to early resolution of branch outcomes. In Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture. 59–68. doi:10.1109/MICRO.1998.74276
- [9] Saurabh Gupta, Niranjan Soundararajan, Ragavendra Natarajan, and Sreenivas Subramoney. 2020. Opportunistic Early Pipeline Re-steering for Data-dependent Branches. In Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (Virtual Event, GA, USA) (PACT '20). Association for Computing Machinery, New York, NY, USA, 305–316. doi:10.1145/3410463.3414628
- [10] Akash Sridhar, Nursultan Kabytkas, and Jose Renau. 2020. Load Driven Branch Predictor (LDBP). arXiv:2009.09064 [cs.AR] <https://arxiv.org/abs/2009.09064>
- [11] Lei Chen, S. Dropsho, and D.H. Albonesi. 2003. Dynamic data dependence tracking and its application to branch prediction. In The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. 65–76. doi:10.1109/HPCA.2003.1183525
- [12] Hongliang Gao, Yi Ma, Martin Dimitrov, and Huiyang Zhou. 2008. Address-branch correlation: A novel locality for long-latency hard-to-predict branches. In 2008 IEEE 14th International Symposium on High Performance Computer Architecture. 74–85. doi:10.1109/HPCA.2008.4658629
- [13] Muawya Al-Otoom, Elliott Forbes, and Eric Rotenberg. 2010. EXACT: explicit dynamic-branch prediction with active updates. In Proceedings of the 7th ACM International Conference on Computing Frontiers (Bertinoro, Italy) (CF '10). Association for Computing Machinery, New York, NY, USA, 165–176. doi:10.1145/1787275.1787321
- [14] Rami Sheikh and Derek Hower. 2019. Efficient Load Value Prediction Using Multiple Predictors and Filters. In 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA). 454–465. doi:10.1109/HPCA.2019.00057
- [15] André Seznec. 2018. Exploring value prediction with the eves predictor. In CVP-1 2018-1st Championship Value Prediction. 1–6.

A Cost Analysis

#	Component	Details of each field of each entry, # entries, etc.	Cost
1	TAGE-SC-L	Unmodified.	524288 bits (64 KB)
2	Architecture Register File [reg_file]	low: 64 bits (integer register, lowest 64-bit fp/simd register, flag register, zero register), 66 entries high: 64 bits (highest 64-bit fp/simd register), 32 entries	6272 bits (0.765625 KB)
3	Commit Queue [commit_queue]	valid: 1 bit inst_class: 2 bits (only alu/load/fp/slowAlu) pc: 23 bits piece: 1 bit src_reg_0_valid: 1 bit src_reg_0: 7 bits src_reg_1_valid: 1 bit src_reg_1: 7 bits src_reg_size_more_than_two: 1 bit dst_reg_valid: 1 bit dst_reg: 7 bits 96 entries	4992 bits (0.609375 KB)
4	Store Resolve Queue [store_resolve_queue]	valid: 1 bit addr: 48 bits 8 entries	392bits (0.47852 KB)
5	CondBR Chain Table [b_0_0, alu_1_0, load_2_0, load_2_1, load_3_0, load_3_1]	B Node: valid: 1 bit tag: 14 bits taken: 48 bits ball_tag_sel: 8 bits ALU Node: valid: 1 bit src_1_valid: 1 bit pc: 11 bits Load Node: valid: 1 bit pc: 23 bits piece: 1 bit 256 entries, each entry has one B Node, one ALU Node and four Load Node.	47104bits (5.75 KB)
6	Load Pattern Table [load_pattern_table]	valid: 1 bit tag: 14 bits mem_sz: 2 bits last_mem_va: 48 bits const_stride: 13 bits const_stride_conf: 3 bits wrap_mem_va: 12 bits wrap_target_mem_va: 12 bits wrap_conf: 3 bits offset: 6 bits offset_conf: 3 bits vtage_predict_addr_valid: 1 bit vtage_predict_addr: 48 bits in_flight_cnt_valid: 1 bit	90624 bits (11.0625 KB)

		in_flight_cnt: 10 bit 512 entries	
7	ALU Pattern Table [alu_pattern_table]	valid: 1 bit tag: 2 bits cmp_conf: 1 bit fcmp_zero_conf: 1 bit 512 entries	2560 bits (0.3125 KB)
8	BP Data Cache [bp_data_cache]	valid: 16 bits tag: 35 bits data: 512 bits age: 4 bits 1408 entries (128 set, 11 way)	798336 bits (97.453125 KB)
9	E-VTAGE [15]	Adopted from 8KB version EVES, only includes E-VTAGE and excludes E-stride, the data value table is 41 bits because recording 48-bit address instead of 64-bit value (the remaining 7-bit is encoded by index). According to calculation method in Section 3.1 The 8KB EVES predictor in [15]: <ul style="list-style-type: none"> a 3-way skewed associative data value table with a total of 384 entries. Each entry is composed of a 41-bit value and a 2-bit useful counter: 16512 bits a 47 -bank interleaved VTAGE array with 32 entries on each bank. Each entry consists of a 9 bit hash-or-pointer, a 11-bit tag, a 2-bit useful counter and a 3-bit confidence counter, i.e., 25 bits per entry: 37600 bits A 10-bit counter TICK to manage the resetting of the useful counters and a 8-bit counter to date the last mis-prediction. 	54130 bits (6.607666 KB)
10	BALL Prediction Time History [ball_pred_time_histories]	tage_sc_1_predict: prediction made by 64KB TAGE-SC_L. use_ball_predict: choose to use BALL prediction instead of TAGE. ball_predict_valid: BALL make a prediction (unlike TAGE, BALL may not predict for every branch) ball_predict: prediction made by BALL Above prediction time information is recorded, and update TAGE and BALL when branch is resolved.	NOT counted into cost budget.
11	Load Prediction Time History [load_pred_time_histories]	load_pattern_table_hit: Whether hit Load Pattern Table when an instruction is at fetch. When the first hit	NOT counted into cost budget.

		<p>instruction is committed, Load Pattern Table set in_flight_cnt_valid, so the tracking in_flight_cnt can be used to predict constant stride address.</p> <p>make_prediction: Whether E-VTAGE make a prediction for load address at fetch stage.</p> <p>predicted_addr: The predicted load address by E-VTAGE at fetch stage.</p> <p>make_prediction and predicted_addr is for updating E-VTAGE when a load committed.</p>	
	TOTAL	1528698 bits (186.608643 KB)	