

Context-Base Computational Value Prediction with Value Compression

Yasuo Ishii
Arm
yasuo.ishii@arm.com

ABSTRACT

In this paper, we propose context-base computation value prediction and aggressive cost reduction techniques exploiting value locality. Context-base computational value prediction tracks both base value and its stride for corresponding context to cover context-dependent delta patterns. We also propose an aggressive value compression scheme using value compression cache. This cache tracks higher order bits of tracking values. To maximize the storage efficiency, the data can be cached only after the predicting value have a certain confidence.

We applied these proposals and dynamic coverage accuracy control on TAGE inspired value predictor, CBC-VTAGE. With 8KB storage budget, our optimized CBC-VTAGE achieves 17.1% speedup compared with processor without value prediction. The geomean of IPC for distributed 135 traces is 3.76.

1 Introduction

Modern microprocessors employ prediction mechanisms to improve their performance. However, some of known predictors were not widely implemented in commercial processors. A value predictor is one of such known unimplemented predictors. Recently, some academic papers [3,4] proposed new value prediction algorithms which demonstrated high performance. However, these predictors cannot improve performance if the detected value patterns rely on both delta from last value and its context. Moreover, the storage requirements of these value predictors are typically significant compared with the other prediction mechanisms. For example, VTAGE [3] requires a huge storage (e.g., 32KB) to realize noticeable performance improvement.

In this paper, we propose (1) context-base computational value prediction to correctly predictor certain value access patterns which could not be caught by existing value predictors and (2) aggressive data compression to reduce the storage requirement, and (3) dynamic coverage / accuracy control to maximize performance.

2 Context-Base Computational Value Prediction

Many existing value predictors capture value patterns by correlating the previously appeared value patterns and the corresponding context. One of representative value pattern is a stride pattern (or delta pattern). To track the stride, value

predictors hold the last value and its stride from second oldest value in the prediction storage. When the same instruction is executed with same context, the summation of the last value and the stride is produced as a predicted value. However, when one instruction processes multiple different data streams, existing value predictors cannot produce correct prediction efficiently.

Fig 1 shows the representative example. In this example, two different contexts (function A and function B) call one shared function (function C). One instruction in function C is the target of the value prediction. The produced value stream (2, 3, 4, 6, 6, and 9) on this instruction cannot be predicted by existing stride predictor because it correlates last value and stride with corresponding PC. However, if the value predictor can separate the stream for each caller function, the value becomes predictable. When the function was called from function A, the value pattern is 2, 4, and 6. If the function was called by function B, the value pattern is 3, 6, and 9.

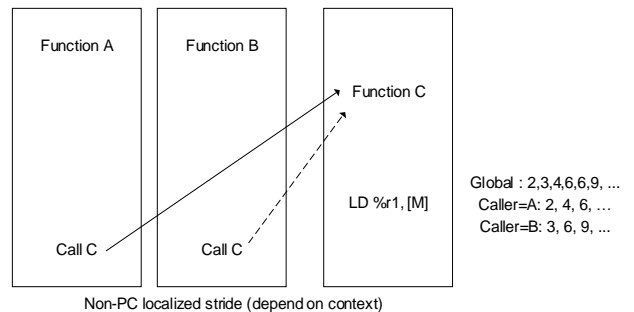


Fig 1: Context-base Computational Pattern

To predict this pattern, both the last value and the stride should be correlated with its context. However, recently proposed value predictors cannot track this type value patterns efficiently. For example, D-VTAGE [4] cannot predict the value pattern efficiently since D-VTAGE still uses PC indexed last value table to provide base value. To predict this pattern, D-VTAGE needs to allocate many entries to track each stride.

To support this value pattern, we propose context-base computational value prediction. With this prediction algorithm, one prediction entry tracks both the last value and its stride with correlated context. When the value predictor detected the same context, the value predictor starts to make prediction based on tracking last value and the stride. This prediction scheme was originally proposed as per-path stride, per-path

predictor [5]. However, the predictor capability was limited because the predictor employs only one pattern table.

To realize efficient context-base computational predictor, we applied this prediction algorithm on TAGE [6] inspired predictor, Context-Base Computational Value predictor TAGE (CBC-VTAGE). Fig 2 shows the overview of CBC-VTAGE. Like the other TAGE inspired predictors [3,4], CBC-VTAGE has multiple components correlated with different branch history length. A prediction entry in each component is partially tagged by hashed branch histories. The prediction entry can make a prediction only when the tag matched against hash value created from current context information. When multiple tags are matched, the component correlated with the longest branch history length can provide the final prediction. The value prediction is created only when the confidence counter of the longest matching entry is saturated.

The prediction part of each entry has two fields to track (1) a last value and (2) a stride of corresponding context. When the entry is updated, both the last value and the stride are updated. Unlike the other TAGE inspired predictors, CBC-VTAGE can update all tag matching entries if its strides are none-zero. This is necessary to maintain the correct last predicted value for each prediction entry while the other TAGE inspired predictors update only the longest matching prediction entry.

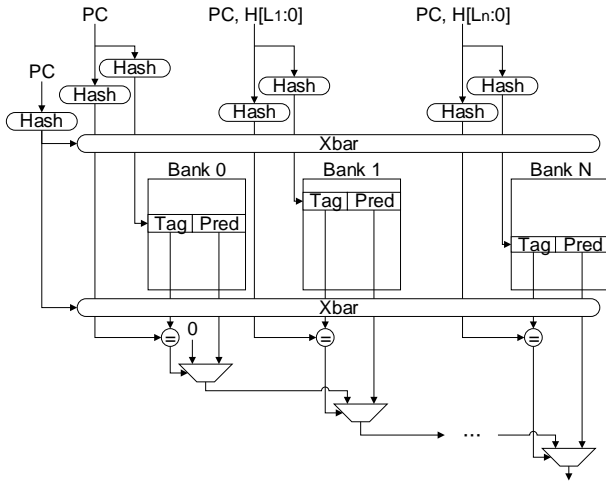


Fig 2: Context-base Computational Value Predictor TAGE

The other differences of CBC-VTAGE are also show in Fig 2. A CBC-VTAGE in this paper does not have tag less components. Base components are also tagged. Moreover, the tag width of all tables is same to support a bank interleaving mechanism [2] to maximize the storage efficiency. Each hash value created from the branch history is swapped based on the hashed PC. We also apply an aggressive value compression scheme described in next section to minimize the cost.

In most of existing value prediction papers, the predictor is updated after commit and CVP-1 also follows this design. For

such a processor core, the computational value predictor needs to maintain inflight information to reproduce speculative value. Since one CBC-VTAGE prediction entry tracks both last value and its stride. To reproduce the last value, the predictor needs to track only the number of instructions accessing the same prediction entry. If N inflight instruction used the entry, the speculative last value can be calculated by ‘committed last value + N * committed stride’. Unlike D-VTAGE, the speculative information tracking mechanism doesn’t need to track the speculative last value. This helps to mitigate the complexity and the cost of existing computational value predictors.

3 Aggressive Data Compression for Value Predictor

3.1 Value Compression Cache

When processor produced the integer value, the values tend to belong to subset of a certain value set (e.g., 0, 1, -1, ...). If the produced value is for an address, the produced values tend to point a similar address ranges since one application typically accesses to a small subset of huge virtual address space. These facts indicate that higher order bits of predicted values have some spatial locality. This means that multiple values can share same higher order bits.

To exploit this locality, we introduced value compression cache. This idea is basically derived from ITTAGE region cache [2]. In this study, we implement 255-entry full associative structure to compress higher order bits. For real hardware, 2-way or 4-way skewed associative cache should realize similar capability. To utilize this compression cache, each prediction entry needs to track 8-bit pointer instead of tracking raw data. When a prediction is created, the value predictor read this cache by that pointer to produce higher order bits of predicted value (Fig 3). The value compression cache entry tracks 54-bit value in this study. The value compression cache can also compress a stride value. When the stride value is compressed, 54-bit value tracked by the value compression cache is mapped to lower 54-bit of compressed stride. Higher order bits of stride value will be supplemented by sign extension. We also implement one reserved entry (256th entry) to produce ‘0’ from this table without consuming storage cost.

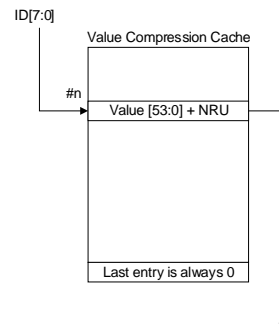


Fig 3: Value Compression Cache

3.2 Lazy Allocation for Value Compression Cache

The value compression cache exploits the spatial locality to save storage cost. We also found that non-negligible number of the allocated cache entries never be used for actual value prediction since the corresponding prediction entry is evicted before its confidence counter gets saturated.

For such entries, the confidence counters typically do not exceed a half of its saturated value ('3' for 3-bit forward probabilistic confidence counter) and only a few lower order bits are enough to qualify the confidence of the corresponding prediction entry. To exploit this locality, the allocation of value compression cache is delayed until the confidence counter of the corresponding value prediction entry reaches a certain threshold. In this paper, the predictor starts to allocate value compression cache entry only when the confidence counter reaches '3' for 3-bit confidence counter. The cache entry for stride value is allocated when the confidence value turns to '4' since full last value is necessary to calculate 54-bit stride value. Before the confidence value reaches '4', the value predictor qualifies only lower 8-bit of predicting value.

Fig 4 shows the overview of the lazy allocation for 3-bit forward probabilistic counter (FPC) [3]. When the prediction entry is allocated, the confidence counter is initialized to zero. When the correct prediction is qualified, the value is turned to 1. After confidence counter reaches 1, subsequent increment can happen only when the correct prediction was identified with the certain probability. When the counter turned to '3', the base value is allocated in the value compression cache. When the value turns to '4', the stride value is assigned on the value compression cache if necessary. Before the confidence counter reaches '4', only 8-bit of predicted value is checked at update timing since higher order bits might not be tracked by the predictor. After the confidence counter reaches 4, full value can be qualified. The predictor starts to make prediction after the confidence value turns to '7'.

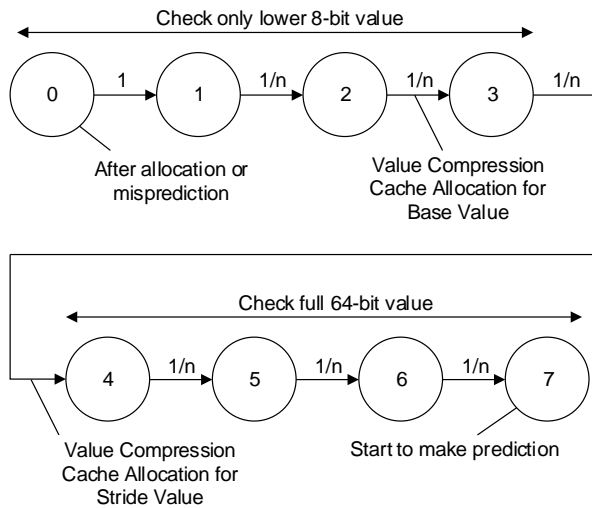


Fig 4: Lazy Allocation on Forward Probabilistic Counter

3.3 Value Compression Cache Replacement

The replacement policy of the value compression cache is CLOCK in this paper. Each entry has 1-bit NRU information and the cache maintains one shared CLOCK pointer. When the cache entry is replaced, the confidence counters using the victimized entry are reset to a defined value ('3' for 3-bit confidence counter in this study). In real hardware, this reset mechanism might not be necessary since its performance impact was acceptable (0.2% slowdown) in our submitted value predictor.

4 Dynamic Accuracy / Coverage Control

4.1 Adaptive Forward Probabilistic Counter

To reduce the storage cost of confidence counter, forward probabilistic counter (FPC) [3] is efficient because the accuracy and the coverage can be optimized by just changing forward progress probability. Setting right forward probability is very important to maximize a value predictor performance.

One problem of existing static FPC algorithm is that the probability is fixed at the design time. For CVP-1 distributed traces, each trace has its own 'best' forward probability. If the predictor can dynamically adjust its forward probability, it can improve the performance. To adjust forward probabilistic value dynamically, we propose adaptive forward probabilistic counter. This scheme employs small table, which is called accuracy tracking table, to track the accuracy of the recent prediction. The tracking is performed for some categories. If the accuracy is lower than pre-defined threshold, the forward probability is reduced to spend more time to learn the value.

In this submission, the accuracy tracking table employs multiple saturation counters to track the accuracy for certain categories which are generated by (1) instruction address, (2) instruction type (load instruction or not) and (3) actual latency since the instruction type can give us the resource confliction and the latency implies the benefit for a correct prediction. Using instruction type to classify the instruction is derived from old branch predictor work [1]. PC qualification helps to capture outlier instructions.

When a wrong value prediction is created, the counter of the accuracy tracking table entry is incremented. When a correct prediction is created, the counter is decremented by a certain probability. This probability can be defined by 'target accuracy'. If the value predictor is designed to realize more than 99.2% accuracy for the corresponding accuracy tracking type, the counter is decremented with 0.8% probability. When the counter is larger than thresholds, the predictor reduces the forward progress probability to improve accuracy.

When the accuracy of the corresponding category is lower than the target probability (e.g., 99.2%), the counter will be a large value. If the large value is detected, the value predictor starts to train the same value more by reducing the forward probability. This process increases the accuracy by reducing the coverage. On the other hand, when the counter indicates high accuracy, the counter is going to have small value. If this

case is detected, the value predictor starts to learn the value less by increasing the forward probability. This can improve prediction coverage.

In this submission, accuracy tracking table employs 32-entry 4-bit counters to track the accuracy of each category. 32-table entries are categorized into five classes (Table 1). Each class has 4 or 8 accuracy tracking entries and these entries are selected by hashed PC.

Table 1: Budget Counting for Submitted Predictor

	instruction type	actual latency	accuracy table entries
Category 1 (long latency load)	load	>64	8
Category 2 (medium latency load)	load	9-64	8
Category 3 (short latency load)	load	1-8	8
Category 4 (long latency arithmetic)	non-load	>1	4
Category 5 (short latency arithmetic)	non-load	1	4

4.2 Blacklist Filtering

When the predictor makes misprediction, small number of instructions often covers majority of incorrect predictions. This situation can happen even if the overall prediction accuracy is high. If the predictor can exclude such outliers not to make prediction, the overall performance can be improved.

To exclude these outliers, we propose blacklist filtering which is originally proposed for branch prediction [7]. The blacklist filter employs a set-associative partially tagged cache to track the recently miss-predicted instructions. The accuracy tracking is realized by forward probabilistic counter. The counter is incremented by misprediction, decremented by correct prediction with a certain probability. When the tracking accuracy is less than a target probability, the blacklist filter suppresses the prediction. Even if the blacklist filter prevents the prediction, the value predictor keeps learning. If the value predictor starts to generate correct prediction, the filter allows the predictor to make prediction again.

In this submission, the filter is implemented as 128-set, 2-way skewed associative cache. The replacement policy is NRU-like. Each filter entry has 4-bit counter to track the accuracy.

5 Value Predictor Configuration for Championship

For the championship, we implement all proposed features within 8KB budget. The main predictor is CBC-VTAGE which employs 8-tables. Each table has 128-entry. Each entry has 13-bit tag, 3-bit confidence counter, 2-bit usefulness counter, and 28-bit predict value information. Each prediction table entry is compressed by a value compression cache with lazy allocation.

To maximize the storage efficiency, one entry can have five different formats as shown in Fig 5. Fig 5 (0) is a format for training before confidence counter reaches 3. When a new CBC-VTAGE entry is allocated on the table, the prediction entry has this format. This format can track only lower 10-bit of last value. The value compression cache entry is not allocated for this format. Fig 5 (1) and (2) show the formats for last value

prediction. When the training entry reaches certain confidence (confidence counter turns to 3) and its stride is zero, the prediction entry switches to these formats. Fig 5 (3) and (4) show the formats for stride prediction.

For last value formats, the compression cache entry can be assigned only when the MSB shows complex bit patterns. If the MSB is just sign extension, the value is just supplemented by sign extension indication as shown in Fig 5 (1). For the stride formats, the base value is allocated on the value compression cache. The stride field has the value compression cache entry only when the stride cannot be represented by 8-bit value.

As shown in Fig 1, CBC-VTAGE tables are multi-banked and crossbars controlled by a hashed PC realizes bank-interleaving [2]. CBC-VTAGE uses a global counter tracks the difficulty of prediction table allocation as proposed in branch predictor [2]. If the allocation difficulty exceeds certain threshold, the predictor decrements usefulness bit from all prediction entries. On top of this base CBC-VTAGE predictor, we applied dynamic accuracy / coverage techniques proposed in previous section.

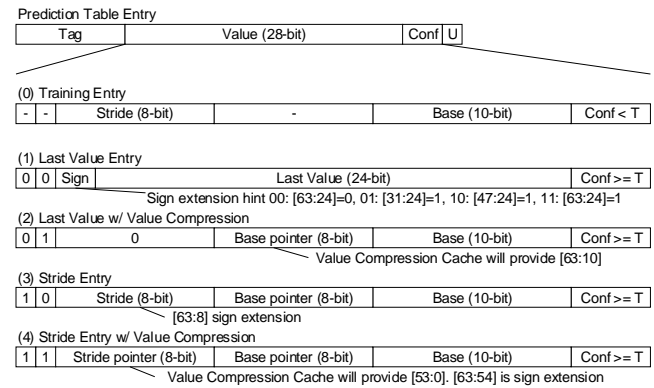


Fig 5: CBC-VTAGE Prediction Entry Format

6 Evaluation

Table 2 shows the total storage budget for the submitted predictor. The predictor consumes 65403 bits which satisfies the 8KB track constraint.

Table 2: Budget Counting for Submitted Predictor

Resource	Attribute	Bit per entry	# of entry	Total bits
CBC-VTAGE 1024 entry (8-tables x 128-entry)	Tag	13	1024	13312
	Confidence	3	1024	3072
	Usefulness	2	1024	2048
	Value	28	1024	28672
	Global TICK Counter	10	1	10
Value Compression Table (255-way full associative)	Value	54	255	13770
	Replacement info (NRU)	1	255	255
	CLOCK pointer	8	1	8
Accuracy tracking table (32-entry)		4	32	128
Blacklist filter (2-way, 128-entry/way)	tag	11	256	2816
	counter	4	256	1024
	Replacement info (NRU)	1	256	256
Miscellaneous	random value seed	32	1	32
Total				65403

This value predictor achieves 3.76 instruction per cycle for geometric mean of 135 distributed CVP-1 traces. This IPC is improved by 17.1% from no value prediction configuration. Fig 6 shows the performance comparison against representative value prediction algorithms. Each algorithm is evaluated with both small storage budget and reasonably large storage budget. The proposed value predictor not only outperforms existing predictors with similar storage budget, but also realize comparable performance with much bigger value predictors.

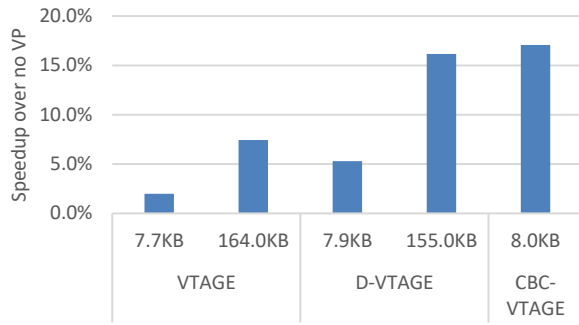


Fig 6: Speedup for CVP-1 distributed 135 traces

Fig 7 shows the sorted speedup over no value prediction. For 89 out of 135 traces, the submitted value predictor shows more than 1% speedup. For 13 out of 135 traces, the predictor caused more than 1% slowdown. The worst slowdown is observed on compute_fp_6 (-10.0%) and the biggest speedup is observed on compute_int_45 (1094.1%).



Fig 7: Sorted speedup. This chart excludes two traces which show 641% and 1094% speedup respectively

Fig 8 shows the breakdown of 17.1% speedup for the submitted predictor. In this chart, 19.4KB D-VTAGE is treated as the baseline. This predictor improves IPC by 6.2% from no value prediction. CBC-VTAGE without a compression (17.0KB) outperforms the baseline D-VTAGE by 5.3%. After we tuned forward probability parameters (e.g., dedicated static forward probability for each instruction class), 17.0KB CBC-VTAGE improves IPC by 4.2%. The value compression cache reduced storage budget from 17.0KB to 7.5KB with 0.3% slowdown. The adaptive forward probabilistic counter and the blacklist filter improve IPC by 0.1% and 1.6% respectively.

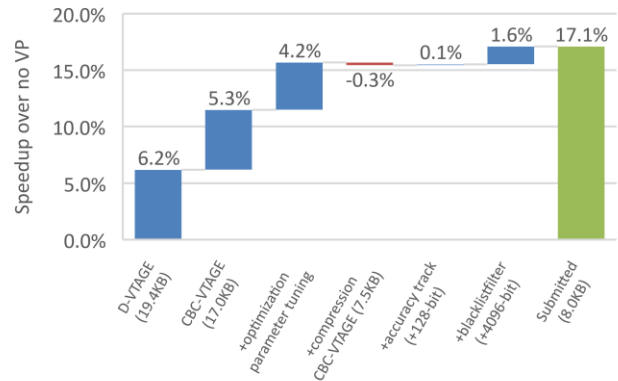


Fig 8: Speedup breakdown over 19.4KB D-VTAGE

7 Conclusion

In this paper, we proposed context-based computational value prediction to cover more value prediction opportunity. We also proposed value compression and dynamic accuracy and coverage control schemes to achieve high performance with limited resource.

Our context-based computation value prediction is a new efficient implementation to cover delta patterns. To maximize storage efficiency, we also proposed a value compression cache and lazy allocation. The value compression cache exploits value locality to minimize storage cost. Lazy allocation helps to eliminate useless cache entries by using progress of the confidence counter. We also proposed two different dynamic accuracy / coverage control schemes. One is controlling forward probability of forward probabilistic counter and the other filtered out incorrect predictions.

The value predictor supporting all proposed mechanisms achieves 17.1% speedup over no value prediction with 8KB budget.

REFERENCES

- [1] T.-Y. Yeh and Y. N. Patt, "A comparison of dynamic branch predictors that use two levels of branch history," in Proceedings of the 20th Annual International Symposium on Computer Architecture, ISCA '93
- [2] A. Seznec, "A 64-kbytes ITTAGE indirect branch predictor," in Third Championship Branch Prediction, JWAC-2, 2011.
- [3] A. Perais and A. Seznec, "Practical data value speculation for future high-end processors," in High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on, Feb 2014
- [4] A. Perais and A. Seznec, "BeBoP: A cost effective predictor infrastructure for superscalar value prediction," in 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), Feb 2015
- [5] T. Nakra, R. Gupta, M. L. Soffa, "Global Context -Based Value Prediction," in High Performance Computer Architecture (HPCA), 1999
- [6] A. Seznec and P. Michaud, "A case for (partially) TAgged GEometric history length branch prediction," Journal of Instruction Level Parallelism, vol. 8, pp. 1-23, 2006.
- [7] Y. Ishii, K. Kuroyanagi, T. Sawada, M. Inaba, and K. Hiraki. Revisiting local history for improving fused two-level branch predictor in Third Championship Branch Prediction, JWAC-2, 2011.

8 Appendix 1 (Prediction Resource)

Prediction Resource	Attribute	Bit per entry	# of entry	Total bits	How these are implemented?
CBC-VTAGE 1024 entry (8-tables x 128-entry)	Tag	13	1024	13312	ValuePredictor class in mypredictor.h covers this resource. predTable[] is the resource for this information. In this array MSB indicates table ID and lower 7-bit is actual index. Branch history length = [0, 0, 10, 17, 29, 45, 77, 126].
	Confidence	3	1024	3072	
	Usefulness	2	1024	2048	
	Value	28	1024	28672	
Value Compression Table (255-way full associative)	Global TICK Counter	10	1	10	member variable 'tick' in ValuePredictor class.
	Value	54	255	13770	ValueCompressCache in mypredictor.h covers this resource
	Replacement info (NRU)	1	255	255	ValuePredictor instantiates this storage
Accuracy tracking table (32-entry)	CLOCK pointer	8	1	8	'last_alloc' indicates CLOCK pointer
			4	32	128
Blacklist filter (2-way, 128-entry/way)	Tag	11	256	2816	BlackListFiterTable class in mypredictor.h cover this resource.
	Counter	4	256	1024	
	Replacement info (NRU)	1	256	256	
Miscellaneous	Random value seed	32	1	32	RandomNumberGenerator class has this resource.
Total				65403	

9 Appendix 2 (Inflight Resource)

Inflight resource	Attribute	Bit per entry	# of entry	total	How these are implemented?	
Inflight Buffer	Hashed PC	48	257	12336	Defined in inflightTable[] in InflightBuffer class	
	Pointer for GHR/PHR	11	257	2827	Track PC and GHR pointer to reproduce folded history	
	Load instruction type	1	257	257	'load' in InflightPredictionEntry. This is instruction type of corresponding inflight buffer entry	
	Eligibility	1	257	257	Eligibility (if the instruction can make value prediction) to update value predictor	
Inflight instruction count			9	1	9	Defined as 'size' in InflightBuffer class
Branch History	GHR, PHR	2048	2	4096	GHR and PHR tracks up to 2048-history within ring buffer. This buffer realizes 126-bit branch history.	
	Pointer	11	1	11	Pointer of the GHR/PHR ring buffer	

10 Appendix 3 (VTAGE/D-VTAGE config)

Predictor	Storage budget	Configuration detail
D-VTAGE	7.9KB	8-tables to track stride. Branch history length = [0, 0, 2, 4, 8, 16, 32, 64]. Each table entry contains 13-bit tag, 3-bit confidence, 2-bit usefulness counter. Each table entry can track up to 40-bit stride. Each table has 256-entry. Last value predictor is realized by 2-way skewed cache, each way contains 256-entry (total 512-entry for last value predictor). Each last value entry contains 64-bit last value, 13-bit tag, 1-bit NRU. Forward probability is 1/16.
	19.4KB	8-tables to track stride. Branch history length = [0, 0, 2, 4, 8, 16, 32, 64]. Each table entry contains 13-bit tag, 3-bit confidence, 2-bit usefulness counter. Each table entry can track up to 40-bit stride. Each table has 2048-entry. Last value predictor is realized by 2-way skewed cache, each way contains 2048-entry (total 4096-entry for last value predictor). Each last value entry contains 64-bit last value, 13-bit tag, 1-bit NRU. Forward probability is 1/16.
	155.0KB	8-tables to track stride. Branch history length = [0, 0, 2, 4, 8, 16, 32, 64]. Each table entry contains 13-bit tag, 3-bit confidence, 2-bit usefulness counter. Each table entry can track up to 40-bit stride. Each table has 2048-entry. Last value predictor is realized by 2-way skewed cache, each way contains 2048-entry (total 4096-entry for last value predictor). Each last value entry contains 64-bit last value, 13-bit tag, 1-bit NRU. Forward probability is 1/16.
VTAGE	7.7KB	3-tables to track last value. Branch history length = [0, 5, 13]. Each table entry contains 64-bit value, 13-bit tag, 3-bit confidence, 2-bit usefulness counter. Each table has 256-entry. Forward probability is 1/16.
	164.0KB	8-tables to track last value. Branch history length = [0, 0, 2, 4, 8, 16, 32, 64]. Each table entry contains 64-bit value, 13-bit tag, 3-bit confidence, 2-bit usefulness counter. Each table has 2048-entry. Forward probability is 1/16.