BeBoP: A Cost Effective Predictor Infrastructure for Superscalar Value Prediction

Arthur Perais, André Seznec

BeBoP: A Cost Effective Predictor Infrastructure for Superscalar Value Prediction

Arthur Perais André Seznec
IRISA/INRIA
Campus de Beaulieu
35042 Rennes, France
{arthur.perais,Andre.Seznec}@inria.fr

***Abstract*—Up to recently, it was considered that a performance-effective implementation of Value Prediction (VP) would add tremendous complexity and power consumption in the pipeline, especially in the Out-of-Order engine and the predictor infrastructure.**

Despite recent progress in the field of Value Prediction, this remains partially true. Indeed, if the recent EOLE architecture proposition suggests that the OoO engine need not be altered to accommodate VP, complexity in the predictor infrastructure itself is still problematic. First, multiple predictions must be generated each cycle, but multi-ported structures should be avoided. Second, the predictor should be small enough to be considered for implementation, yet coverage must remain high enough to increase performance.

To address these remaining concerns, we first propose a block-based value prediction scheme mimicking current instruction fetch mechanisms, BeBoP. It associates the predicted values with a fetch block rather than distinct instructions. Second, to remedy the storage issue, we present the *Differential* VTAGE predictor. This new tightly coupled hybrid predictor covers instructions predictable by both VTAGE and Stride-based value predictors, and its hardware cost and complexity can be made similar to those of a modern branch predictor. Third, we show that block-based value prediction allows to implement the checkpointing mechanism needed to provide D-VTAGE with last computed/predicted values at moderate cost.

Overall, we establish that EOLE with a 32.8KB block-based D-VTAGE predictor and a 4-issue OoO engine can significantly outperform a baseline 6-issue superscalar processor, by up to 62.2% and 11.2% on average (gmean), on our benchmark set.

I. Introduction & Motivations

Single thread performance is still an issue in general-purpose computing. In that context, architectural techniques that were proposed in the late 90's for high-end uniprocessors but were not implemented at that time could be worth revisiting; among these techniques is Value Prediction (VP), that was independently proposed by Gabbay et al. [22] and Lipasti et al. [20].

Value Prediction suffers from an asymmetry between the small average performance gains brought by a correct prediction and the high cost of recovering from a misprediction. This implies that to increase performance, VP must be very accurate, and/or the recovery mechanism must be very aggressive, e.g. *selective replay*. As a result, it was considered up to recently that VP would lead to a huge increase in complexity and power consumption in every stage of the pipeline.

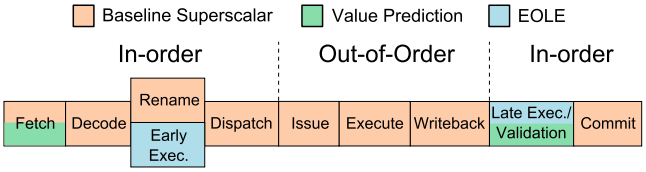


Fig. 1: Pipeline diagram of EOLE [25].

However, taking a step back, only two distinct components are mostly responsible for this additional complexity: 1) the Out-of-Order execution engine and 2) the intrinsic complexity of the value prediction data path.

The first source of complexity was studied by Perais and Seznec [25], [26]. They have shown that **the Out-of-Order engine does not need to be overhauled to support VP**. First, very high accuracy can be enforced at reasonable cost in coverage and minimal complexity [26]. Thus, both prediction validation and recovery by squashing can be done outside the Out-of-Order engine, at commit time instead of execution time. Moreover, they devised a new pipeline organization, EOLE[1] (depicted in Fig. 1), that leverages VP with validation at commit to execute many instructions outside the OoO core, in-order [25]. Those instructions are processed either as-soon-as-possible (*Early Execution* in parallel with *Rename*) using predicted/immediate operands or as-late-as-possible (*Late Execution* just before commit) if they are predicted. With EOLE, the issue-width can be reduced without sacrificing performance, mechanically decreasing the number of ports on the Physical Register File (PRF). Validation at commit time also allows to leverage PRF banking for prediction and validation, leading to an overall number of ports similar in EOLE with VP and in the baseline case without VP. That is, EOLE achieves VP-enabled performance while the overall hardware cost of the execution engine is actually *lower* than in a baseline processor without VP. In this paper, we address the remaining source of hardware complexity, which is associated with the value predictor infrastructure itself. Our contribution is threefold.

First, to handle the multiple value predictions needed each

[1] {Early — Out-of-order — Late} Execution

cycle in a wide issue processor, we propose **B**lock-**B**ased value **P**rediction (BBP or BeBoP). With this scheme, all the predictions associated with an instruction fetch block are put in a single predictor entry. The predictor is therefore accessed with the PC of the instruction fetch block and the whole group of predictions is retrieved in a single read. BeBoP accommodates currently implemented instruction fetch mechanisms. However, it only addresses complexity of operation, but not storage requirement.

As a result, in a second step, we propose a space-efficient hybrid predictor, D-VTAGE, by tightly coupling VTAGE [26] and a Stride-based predictor [7]. D-VTAGE is very space-efficient as it can use partial strides (e.g. 8- and 16-bit). Its storage cost can be made equivalent to that of the I-Cache or the branch predictor.

Third, we devise a cost-effective checkpoint-based implementation of the speculative last-value window. Such a window is required due to the presence of a Stride-based prediction scheme in D-VTAGE. Without it, instructions in loops that can fit several times inside the instruction window are unlikely to be correctly predicted.

The remainder of this paper is organized as follows: Section II describes the issues related to predicting several values per cycle and introduces BeBoP. Section III introduces the D-VTAGE predictor and the way it operates while Section IV discusses practical mechanisms to implement D-VTAGE on actual silicon. Section V reviews our evaluation framework. Section VI details the results of our experiments and gives insights on the impact of some predictor parameters on performance. Section VII describes related work. Finally, Section VIII provides concluding remarks.

II. Block-Based Value Prediction

A. Issues on Concurrent Multiple Value Predictions

Modern superscalar processors are able to fetch/decode several instructions per cycle. Ideally, the value predictor should be able to predict a value for all possible outcomes of all instructions fetched during that cycle.

For predictors that do not use any local value history to read tables (e.g. the Last Value Predictor, the Stride predictor or VTAGE), and when using a RISC ISA producing at most one result per instruction (e.g. Alpha), a straightforward implementation of the value predictor consists in mimicking the organization of the instruction fetch hardware in the different components of predictor. Predictions for contiguous instructions are stored in contiguous locations in the predictor components and the same interleaving structures are used in the instruction cache, the branch predictor and the value predictor. However, for predictors using local value history (e.g. FCM [32]), even in this very favorable scenario, the predictor components must be bank-interleaved at the instruction level and the banks individually accessed with different indexes.

Moreover, the most popular ISAs do not have the regularity property of Alpha. For x86 – that we consider in this study – some instructions may produce several results, and information such as the effective PC of instructions in the block[2] and the number of produced values are known after several cycles (after pre-decoding and after *Decode*, respectively). That is, **there is no natural way to associate a value predictor entry with a precise PC**: Smooth multiple-value prediction on a variable-length ISA remains a challenge in itself.

[2]Storing boundary bits in the I-cache can remedy this, but at high cost [1].

B. Block-based Value-Predictor accesses

We remedy this fundamental issue by proposing Block-Based value Prediction (BeBoP). To introduce this new access scheme and layout, we consider a VTAGE-like predictor, but note that block-based prediction can be generalized to any predictor.

The idea is to access the predictor using the *fetch-block PC* (i.e. the current PC right-shifted by $log_2(fetchBlockSize)$), as well as some extra global information as defined in VTAGE [26]. Instead of containing a single value, the entry that is accessed now consists in N_{pred} values ($N_{pred} > 0$). The m^{th} value in the predictor entry is associated with the m^{th} result in the fetch block, and not with its precise PC.

The coupling of the *Fetch*/*Decode* pipeline and the value predictor is illustrated in more details in Figure 2, assuming an x86-like ISA and a VTAGE-like value predictor. *Fetch* first retrieves a chunk of n bytes from the I-cache, then pre-decodes it to determine instruction boundaries, to finally put it in a decode queue. This queue feeds the x86 decoders in charge of generating μ-ops. At the same time, the value predictor is read using the fetch-block PC and global information to retrieve N_{pred} predictions. After *Decode*, the predictions are attributed to the μ-ops in the fetch block by matching instruction boundaries with small per-prediction tags. The rationale behind this tagging scheme is highlighted in the next paragraph. Finally, predictions proceeds to the PRF depending on the saturation of the confidence counter associated with each of them.

1) False sharing issues: Grouping predictions introduces the issue of false prediction sharing if one uses the most significant bits of the PC to index the value predictor (e.g. removing the 4 last bits of the address when fetching 16-byte instruction blocks). False sharing arises when an instruction block is fetched with two distinct entry points e.g. instructions I_1 and I_2. In that case, the traces of μ-ops generated for this block at *Decode* are different, leading to complete confusion in the predicted values.

We avoid this issue by tagging each prediction in the block with the last $log_2(fetchBlockSize)$ bits of the instruction PC to which the prediction was attributed the last time the block was fetched.

That is, to access the predictor, we use the block address, and once N_{pred} predictions have flowed out of the predictor, they are attributed to each μ-op by matching the indexes given by the boundary bits against the per-prediction tags, as described in Fig. 2. In this particular case, the first instruction of the block is I_2, while predictions correspond to the case where the block was entered through instruction I_1. If no specific action were taken, prediction P_1 would be attributed to *UOP1* of I_2. However, thanks to the use of tags to mask predictions, prediction P_2 (with tag T_2 equals 3) is attributed to *UOP1* of I_2 (with boundary 3 since I_2 starts at byte 3). That is, P_1 is not shared between I_1 and I_2.

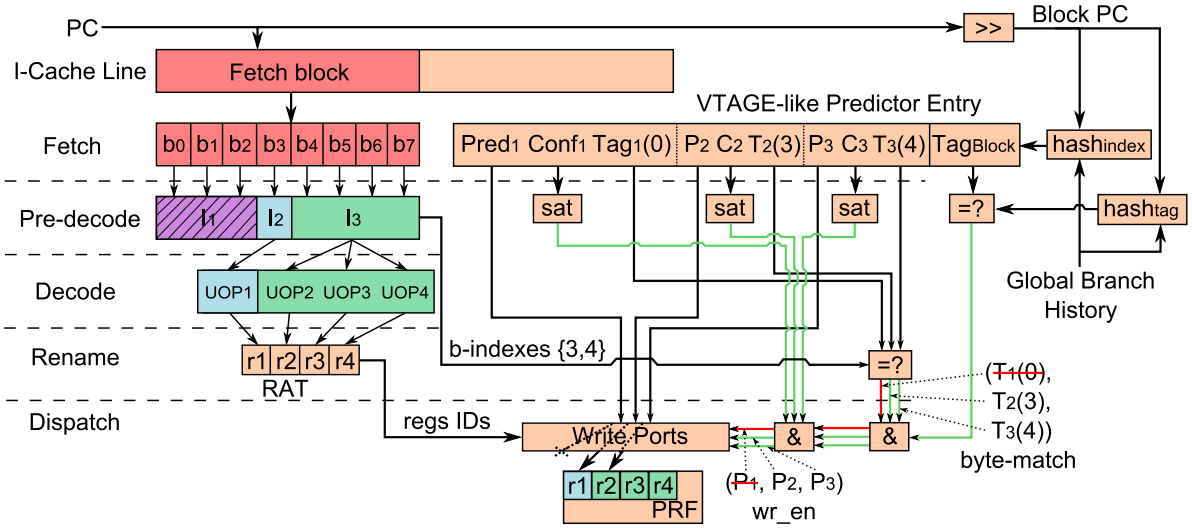


Fig. 2: Prediction attribution with BeBoP and 8-byte fetch blocks. The predictor is accessed with the currently fetched PC and predictions are attributed using byte indexes as tags.

Tags are modified when the predictor is updated with the following constraint: a greater tag never replaces a lesser tag, so that the entry can learn the "real" location of instructions/μ-ops in the block. For instance, if the block was entered through I_2 but has already been entered through I_1 before, then the tag associated with P_1 is the address of the first byte of I_1 (i.e. 0). The address of the first byte of I_2 is greater than that of I_1, so it will not replace the tag associated with P_1 in the predictor, even though dynamically, for that instance of the block, I_2 is the first instruction of the block. As a result, the pairing P_1/I_1 is preserved throughout execution. This constraint does not apply when the entry is allocated.

2) On the Number of Predictions in Each Entry: The number of μ-ops producing a register depends on the instructions in the block, but only the size of the instruction fetch block is known at design-time. Thus, N_{pred} should be chosen as a tradeoff between coverage (i.e. provision enough predictions in an entry for the whole fetch block) and wasted area (i.e. having too many predictions provisioned in the entry). For instance, in Figure 2, since I_1 and I_2 both consume a prediction, only *UOP2* of instruction I_3 can be predicted.

In Section VI-B, we will study the impact of varying N_{pred} while keeping the size of the predictor constant. Specifically, a too small N_{pred} means that potential is lost while a too big N_{pred} means that space is wasted as not all prediction slots in the predictor entry are used. Additionally, at constant predictor size, a smaller N_{pred} means more entries, hence less aliasing.

3) Free Load Immediate Prediction: Our implementation of Value Prediction features write ports available at dispatch time to write predictions to the PRF. Thus, it is not necessary to predict *load immediate* instructions[3] since their *actual* result is available in the front-end. The predictor need not be trained for these instructions, and they need not be validated or even dispatched to the IQ. They can be processed in the front-end even without the Early Execution stage of EOLE [25], by simply placing the decoded immediate in the PRF. Due to this optimization, the addition of a specific table that would only predict small constants [31] may become much less interesting.

4) Multiple Blocks per Cycle: In order to provide high instruction fetch bandwidth, several instruction blocks are fetched in parallel on wide-issue processors. For instance, on the EV8 [35], two instruction fetch blocks are retrieved each cycle. To support this parallel fetch, the instruction cache has to be either fully dual-ported or bank-interleaved. In the latter case, a single block is fetched on a conflict unless the same block appears twice. Since fully dual-porting is much more area- and power-consuming, bank interleaving is generally preferred.

In BeBoP, the (potential) number of accesses per cycle to the predictor tables is similar to the number of accesses made to the branch predictor tables, i.e. up to 3 accesses per fetch block: Read at fetch time, second read at commit time and write to update. This adds up to 6 accesses per cycle if two instruction blocks are fetched each cycle.

However, for value predictor or branch predictor components that are indexed using only the block PC (e.g. Last Value component of VTAGE and Stride predictors), one can replicate the same bank interleaving as the instruction cache in the general case. If the same block is accessed twice in a single cycle, Stride-based predictors must provision specific hardware to compute predictions for both blocks. This can be handled through computing both *(Last Value + stride)* and *(Last Value + stride1 + stride2)* using 3-input adders.

For value or branch predictor components that are indexed using the global branch history or the path history, such as VTAGE tagged components, one can rely on the interleaving scheme that was proposed to implement the branch predictor in the EV8 [35]. By forcing consecutive accesses to map to different banks, the 4-way banked predictor components are able to provide predictions for any two consecutively fetched blocks with a single read port per bank. Moreover, Seznec states that for the TAGE predictor [34], one can avoid

[3]With the exception of *load immediate* instructions that write to a partial register.

the second read at update. The same can be envisioned for VTAGE. Lastly, since they are less frequent than predictions, updates can be performed through cycle stealing. Therefore, tagged components for VTAGE and TAGE can be designed with single port RAM arrays.

That is, in practice, the global history components of a hybrid predictor using BeBoP such as VTAGE-Stride or D-VTAGE – that we present in the next Section – can be built with the same bank-interleaving and/or multi-ported structure as a multiple table global history branch predictor such as the EV8 branch predictor or the TAGE predictor. Only the Last Value Table of (D-)VTAGE must replicate the I-Cache organization.

III. The Differential Value TAGE Predictor

A. A Quick Refresher on VTAGE

The VTAGE predictor is a direct application of the TAGE [36] branch predictor to value prediction [26]. It consists of a single direct-mapped, untagged table as well several partially tagged tables accessed using a hash of the instruction PC, the global branch history and the path history (both indexes and tags are generated using this information). Each partially tagged table is accessed using a different number of bits of the branch history. The different lengths grow in a geometric fashion e.g. the first table will be accessed by hashing the PC and 2 bits of the global branch history, the second table with 4, the third with 8, and so on. In essence, the base table is a tagless *Last Value predictor* while each partially tagged table is a *gshare*-like value predictor.

An entry of VTAGE consists of a 64-bit prediction as well as a 3-bit confidence counter. The counter is reset on a wrong prediction, and incremented with a certain probability on a correct prediction. Predictions are used only when the counter is saturated. This scheme – *Forward Probabilistic Counters* – allows to reach very high accuracy ($> 99.5\%$) at low storage cost if low probabilities are used [26].

To predict, all components are accessed in parallel using the PC for the tagless one and different hashes for the partially tagged ones. The prediction comes from the hitting component using the longest global branch history. If no partially tagged component hits, the base predictor provides the prediction.

At update time, the providing component is updated and on an incorrect prediction, an entry is allocated in an "higher" component (i.e. using a larger portion of the branch history). The allocation policy is driven by an additional *useful* bit in each tagged component entry. The bit is set if the prediction was correct and no "lower" component has the same prediction. It is reset if the prediction was wrong or if a "lower" component already has the prediction. The component in which the entry is allocated is chosen randomly among those whose *useful* bit is 0. If all entries are useful, all corresponding *useful* bits are reset but no entry is allocated. Regardless, all *useful* bits are periodically reset to avoid entries remaining useful forever. We refer the reader to [26] for a more detailed description of VTAGE.

B. Motivations to Improve on VTAGE

Aside from performance, VTAGE [26] has several advantages over previously proposed value predictors. First, all predictions are independent, meaning that the predictor itself does not have to track in-flight predictions in the context of a wide instruction window processor. That is, prediction of a given dynamic instruction is not computed using a potentially inflight prediction of a previous instance. Second, and for the same reason, it does not suffer from a long prediction critical path and can handle back-to-back prediction of the same static instruction as is.

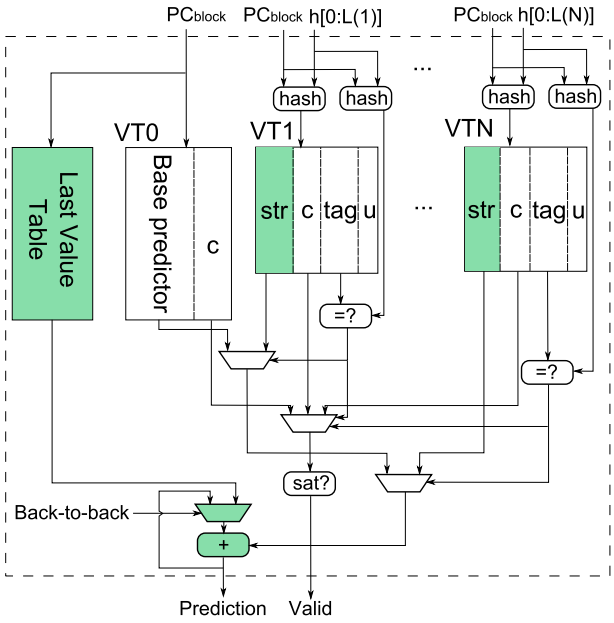


Fig. 3: $1 + n$-component Differential Value TAGE predictor.

However, VTAGE is not suited to handle instructions whose results series exhibit strided patterns (e.g. computation dependent on a loop induction variable). For those instructions, each dynamic instance will occupy its own predictor entry, while a single entry is required for the whole pattern in a Stride predictor. As such, the prediction of strided patterns by VTAGE is not only unlikely but also space inefficient.

To overcome this limitation, Perais and Seznec combine VTAGE with a 2-delta Stride predictor [26]. However, the resulting hybrid predictor is space inefficient since both components are trained for all instructions. As a result, there is a call for a more efficient association of the two prediction schemes.

C. Overview

Similarly to the *Differential* FCM predictor of Goeman et al. [13], we propose the *Differential* VTAGE predictor, D-VTAGE. Instead of storing whole values, the predictor stores – potentially much smaller – differences (strides) that will be added to last values in order to generate predictions.

In particular, VTAGE uses a tagless Last Value Predictor as its base component, while D-VTAGE uses a stride-based predictor (baseline Stride [11] in our case). This component can be further divided into two parts. First, the Last Value Table (LVT) contains only retired last values as well as byte-index tags (as previously discussed in II-B1). Second, the Base Predictor (VT0) contains the strides and a confidence

estimation mechanism. Both tables are direct-mapped but we use small tags (e.g. 5 bits) on the LVT to maximize accuracy.

A $1 + n$-component D-VTAGE is depicted in Fig. 3, assuming a single prediction per entry for clarity. Additional logic with regard to VTAGE is shaded on the figure. For each fetch-block, the last values will be read from the LVT. Then, the strides will be selected depending on whether a partially tagged component hits, following the regular VTAGE operation [26], [36]. If the same instruction block is fetched *twice in two consecutive cycles*, the predictions for the first block are bypassed to the input of the adders to be used as the last values for the second block.

If the same instruction block is fetched *twice in a single cycle*, both instances can be predicted by using 3-input adders (not shown in Fig. 3).

The advantages of D-VTAGE are twofold. First, it is able to predict control-flow dependent patterns, strided patterns and *control-flow dependent strided patterns*. Second, it has been shown that short strides could capture most of the coverage of full strides in a stride-based value predictor [13]. As such using D-VTAGE instead of VTAGE, for instance, would allow to greatly reduce storage requirements.

D. Implementation Issues

a) Speculative History: Because of its stride-based nature, D-VTAGE relies on the value produced by the most recent instance of an instruction to compute the prediction for the newly fetched instance. As many instances of the same instruction can coexist in the instruction window at the same time, the hardware should provide some support to grab the predicted value for the most recent speculative instance. Such a hardware structure could be envisioned as a chronologically ordered associative buffer whose size is roughly that of the ROB. For instruction-based VP, such a design should prove slow and power hungry, e.g. more than ROB-size entries at worst and 8 parallel associative searches each cycle for the simulation framework we are considering (8-wide, 6-issue superscalar).

Fortunately, BeBoP allows to greatly reduce the number of required entries as well as the number of parallel accesses each cycle since we group predictions per fetch block. We develop such a design of the speculative window in Section IV.

b) Impact of Block-Based Prediction: As for VTAGE and TAGE, the allocation policy in the tagged components of D-VTAGE is driven by the *usefulness* of a prediction [36]. The allocation of a new entry also depends on whether the prediction was correct or not. However, we consider a block-based predictor. Therefore, the allocation policy needs to be modified since there can be correct and incorrect predictions in a single entry, and some can be *useful* or not.

In our implementation, an entry is allocated if at least one prediction in the block is wrong. However, the confidence counter of predictions from the providing entry are propagated to the newly allocated entry. This allows the predictor to be efficiently trained (allocate on a wrong prediction) while preserving coverage since high confidence predictions are duplicated. The *usefulness* bit is kept per block and set if a single prediction of the block is useful as defined in [26], that is if it is correct and the prediction in the *alternate* component is not.

c) Prediction Validation: The EOLE processor model assumes that predictions are used only if they are very high confidence. Therefore, unused predictions must be stored to wait for validation in order to train the predictor. To that extent, we assume a *FIFO Update Queue* where prediction blocks are pushed at prediction time and popped at validation time. This structure can be read and written in the same cycle, but reads and writes are guaranteed not to conflict by construction. Specifically, this queue should be able to contain all the predictions potentially in flight after the cycle in which predictions become available. It would also be responsible for propagating any information visible at prediction time that might be needed at update time.

In essence, this structure is very similar to the speculative window that should be implemented for D-VTAGE. Indeed, it contains all the inflight predictions for each block, and it should be able to rollback to a correct state on a pipeline flush (we describe recovery in the context of BeBoP in the next Section). However, to maximize coverage, the FIFO update queue must be large enough so that prediction information is never lost due to a shortage of free entries. On the contrary, we will see that we can implement the speculative window with much less entries than the theoretical number of blocks that can be in flight at any given time. Moreover, the speculative window must be associatively searched every time a prediction block is generated while the FIFO does not need associative lookup (except potentially for rollbacks). As a result, if both structures essentially store the same data, it is still interesting to implement them as two separate items.

IV. Block-Based Speculative Window

In the context of superscalar processors where many instructions can be in flight, computational predictors such as D-VTAGE cannot only rely on their *Last Value Table*. Indeed, in the case where several instances of a loop body are live in the pipeline, the last value required by the predictor may not have been retired, or even computed. As a result, a speculative LVT is required so that the predictor can keep up with the processor. Given the fact that an instruction is allowed to use its prediction only after several tens of previous instances have been correctly predicted (because of confidence estimation), keeping the predictor synchronized with the pipeline is even more critical. As a result, our third and last contribution deals with the specific design of the *Speculative Window*.

An intuitive solution would consist in using an associative buffer that can store all the in-flight predictions. On each lookup, this buffer would be probed and provide the most recent prediction to be used as the last value, if any. Unfortunately, its size would be that of the ROB plus all instructions potentially in flight between prediction availability (*Rename*) and *Dispatch*. Associative structures of this size – such as the *Instruction Queue* – are known to be slower than their RAM counterparts as well as power hungry [8], [16], [24].

Nonetheless, thanks to BeBoP, the number of required entries is actually much smaller than with instruction-based VP, and the number of accesses is at most the number of

fetch blocks accessed in a given cycle. Furthermore, although this buffer behaves as a fully associative structure for *reads*, it acts as a simple circular buffer for *writes* because it is chronologically ordered. That is, when the predictor provides a new prediction block, it simply adds it at the *head* of the buffer, without having to match any tag. If the *head* overlaps with the *tail* (e.g. if the buffer was dimensioned with too few entries), both *head* and *tail* are advanced. Lastly, partial tags (e.g. 15 bits) can be used to match the blocks, as VP is speculative by nature (false positive is allowed).

To order the buffer and thus ensure that the most recent entry is providing the last values if multiple entries hit, we can simply use internal sequence numbers. In our experiments, we use the sequence number of the first instruction of the block. A block-diagram of our speculative window is shown in Fig. 4.

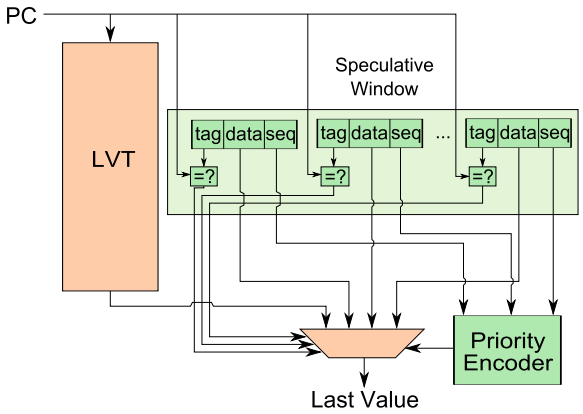


Fig. 4: N-way speculative window. The priority encoder controls the multiplexer by prioritizing the matching entry corresponding to the most recent sequence number.

A. Consistency of the Speculative History

Block-based VP entails intrinsic inconsistencies in the speculative window. For instance, if a branch is predicted as taken, some predictions are computed while the instructions corresponding to them have been discarded (the ones in the same block as the branch but located *after* the branch). Therefore the speculative history becomes inconsistent even if the branch is correctly predicted.

Moreover, pipeline squashing events may increase the impact of such inconsistencies in the speculative window and the FIFO update queue[4]. To illustrate why, let us denote the first instruction to be fetched after the pipeline flush as I_{new} and the instruction that triggered the flush as I_{flush}. Let us also denote the fetch block address of I_{new} as B_{new} and that of I_{flush} as B_{flush}.

On a pipeline flush, all the entries whose associated sequence number is strictly greater than that of I_{flush} are discarded in both the speculative window and the FIFO update queue. The block at the head of both structures therefore corresponds to B_{flush}. We must then consider if whether I_{new} belongs to the same fetch block as I_{flush} or not. If not, the predictor should operate as usual and provide a prediction block for the new block, B_{new}. If it does belong to the same block (i.e. B_{flush} equals B_{new}), several policies are available to us.

a) Do not Repredict and Reuse (DnRR): This policy assumes that all the predictions referring to B_{flush}/B_{new} are still valid after the pipeline flush. That is, I_{new} and all subsequent instructions belonging to B_{flush} will be attributed predictions from the prediction block that was generated when B_{flush}/B_{new} was first fetched. Those predictions are available at the head of the FIFO update queue. With this policy, the heads of the speculative history and FIFO update queue are not discarded.

b) Do not Repredict and do not Reuse (DnRDnR): This policy is similar to *DnRR* except that all newly fetched instruction belonging to B_{flush}/B_{new} will be forbidden to use their respective predictions. The reasoning behind this policy is that the case where B_{new} equals B_{flush} typically happens on a value misprediction, and if a prediction was wrong in the block, chances are that the subsequent ones will too.

c) Repredict (Repred): This policy squashes the heads of the speculative history and FIFO update queue (the blocks containing I_{flush}). Then, once I_{new} is fetched, a new prediction block for B_{flush}/B_{new} is generated. The idea is that some prediction blocks might have retired since I_{flush} was predicted, therefore, this new prediction block may not suffer from an inconsistency due to the block-based speculative history.

d) Keep Older, Predict Newer (Ideal): This idealistic policy is able to keep the predictions pertaining to instructions of block B_{flush} older than I_{flush} while generating new predictions for I_{new} and subsequent instructions belonging to B_{new}. In essence, this policy assumes a speculative window (resp. FIFO update queue) that tracks predictions at the instruction level, rather than the block level. As a result, the speculative window (resp. FIFO update queue) is always consistent.

V. Evaluation Methodology

A. Simulator

We use the framework used by Perais and Seznec in [25] to validate the EOLE architecture. That is, a modified[5] version of the *gem5* cycle-level simulator [3] implementing the x86_64 ISA. Note that contrarily to modern x86 implementations, *gem5* does not support *move elimination* [9], [15], [28], *μ-op fusion* [12] and does not implement a *stack-engine* [12].

Table I describes our model: A relatively aggressive 4GHz, 6-issue[6] superscalar pipeline. The fetch-to-commit latency is 20 cycles. The in-order front-end and in-order back-end are overdimensioned to treat up to 8 μ-ops per cycle. We model a deep front-end (15 cycles) coupled to a shallow back-end (5 cycles) to obtain a realistic branch/value misprediction penalty:

[4]In our implementation, each entry of the FIFO update queue is also tagged with the internal sequence number of the first instruction in the fetch-block to enable rollback.

[5]Modifications mostly lie with the ISA implementation. In particular, we implemented branches with a single μ-op instead of three and we removed some false dependencies existing between instructions due to the way flags are renamed/written.

[6]On our benchmark set and with our baseline simulator, an 8-issue machine achieves only marginal speedup over this baseline.

TABLE I: Simulator configuration overview. *not pipelined.

Front End	L1I 8-way 32KB, 1 cycle, Perfect TLB; 32B fetch buffer (two 16-byte blocks each cycle, potentially over one taken branch) w/ 8-wide fetch, 8-wide decode, 8-wide rename; TAGE 1+12 components 15K-entry total ($\simeq$ 32KB) [36], 20 cycles min. branch mis. penalty; 2-way 8K-entry BTB, 32-entry RAS.
Execution	192-entry ROB, 60-entry IQ unified, 72/48-entry LQ/SQ, 256/256 INT/FP 4-bank register files; 1K-SSID/LFST Store Sets [6]; 6-issue, 4ALU(1c), 1MulDiv(3c/25c*), 2FP(3c), 2FPMulDiv(5c/10c*), 2Ld/Str, 1Str; Full bypass; 8-wide WB, 8-wide VP validation, 8-wide retire.
Caches	L1D 8-way 32KB, 4 cycles, 64 MSHRs, 2 reads and 2 writes/cycle; Unified L2 16-way 1MB, 12 cycles, 64 MSHRs, no port constraints, Stride prefetcher, degree 8; All caches have 64B lines and LRU replacement.
Memory	Single channel DDR3-1600 (11-11-11), 2 ranks, 8 banks/rank, 8K row-buffer, tREFI 7.8us; Across a 64B bus; Min. Read Lat.: 75 cycles, Max. 185 cycles.

20/21 cycles minimum. We allow two 16-byte blocks worth of instructions to be fetched each cycle, potentially over a single taken branch. Table I describes the characteristics of the baseline pipeline we use in more details. In particular, the OoO scheduler is dimensioned with a unified centralized 60-entry IQ and a 192-entry ROB on par with the latest commercially available Intel microarchitecture. We refer to this model as the **Baseline_6_60** configuration (6-issue, 60-entry IQ).

As μ-ops are known at *Fetch* in *gem5*, all the widths given in Table I are in μ-ops. Independent memory instructions (as predicted by the Store Sets predictor [6]) are allowed to issue out-of-order. Entries in the IQ are released upon issue since *selective replay* is not needed on value mispredictions.

In the case where EOLE is used, we consider a 1-deep, 8-wide *Early Execution* stage and an 8-wide *Late Execution/Validation* stage limited to 4 read ports per register file bank [25]. Lastly, if EOLE and VP are not used, the Validation and Late Execution stage is removed, yielding a fetch-to-commit latency of 19 cycles. The minimum branch prediction penalty is kept at 20 cycles.

B. Value Predictor Operation

In its baseline version, the predictor makes a prediction at *Fetch* for every eligible μ-op (i.e. producing a 64-bit or less register that can be read by a subsequent μ-op, as defined by the ISA implementation). To index the predictor, we XOR the PC of the x86_64 instruction with the μ-op index inside that x86_64 instruction [25]. This avoids all μ-ops mapping to the same entry, but may create aliasing. We assume that the predictor can deliver *fetch-width* predictions each cycle. The predictor is updated in the cycle following retirement.

When using block-based prediction, the predictor is simply accessed with the PC of each fetch block and provides several predictions at once for those blocks. Similarly, update blocks are built after retirement and an entry is updated as soon as an instruction belonging to a block different than the one being built is retired. Update can thus be delayed for several cycles. To maximize accuracy, we tag the LVT with 5 higher bits of the fetch block PC.

We transpose the configuration of VTAGE in [25] to D-VTAGE: 8K-entry base component with 6 1K-entry partially tagged components. The partial tags are 13 bits for the first component, 14 for the second, and so on. Histories range from 2 to 64 bits in a geometric fashion. We also use 3-bit *Forward Probabilistic Counters* with probabilities $v = \{1, \frac{1}{16}, \frac{1}{16}, \frac{1}{16}, \frac{1}{16}, \frac{1}{32}, \frac{1}{32}\}$. Unless specified otherwise, we use 64-bit strides in all the predictors we consider.

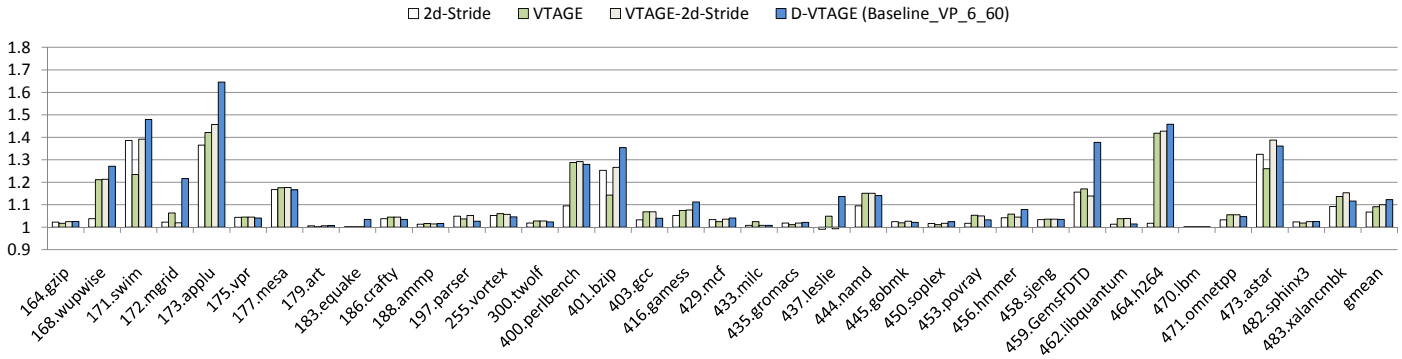
TABLE II: Benchmarks used for evaluation. Top: CPU2000, Bottom: CPU2006. INT: 18, FP: 18, Total: 36.

Program	Input	IPC
164.gzip (INT)	input.source 60	0.845
168.wupwise (FP)	wupwise.in	1.303
171.swim (FP)	swim.in	1.745
172.mgrid (FP)	mgrid.in	2.361
173.applu (FP)	applu.in	1.481
175.vpr (INT)	net.in arch.in place.out dum.out -nodisp -place_only -init_t 5 -exit_t 0.005 -alpha_t 0.9412 -inner_num 2	0.668
177.mesa (FP)	-frames 1000 -meshfile mesa.in -ppmfile mesa.ppm	1.021
179.art (FP)	-scanfile c756hel.in -trainfile1 a10.img -trainfile2 hc.img -stride 2 -startx 110 -starty 200 -endx 160 -endy 240 -objects 10	0.441
183.equake (FP)	inp.in	0.655
186.crafty (INT)	crafty.in	1.562
188.ammp (FP)	ammp.in	1.258
197.parser (INT)	ref.in 2.1.dict -batch	0.486
255.vortex (INT)	lendian1.raw	1.526
300.twolf (INT)	ref	0.282
400.perlbench (INT)	-I./lib checkspam.pl 2500 5 25 11 150 1 1 1 1	1.400
401.bzip2 (INT)	input.source 280	0.702
403.gcc (INT)	166.i	1.002
416.gamess (FP)	cytosine.2.config	1.694
429.mcf (INT)	inp.in	0.113
433.milc (FP)	su3imp.in	0.501
435.gromacs (FP)	-silent -deffnm gromacs -nice 0	0.753
437.leslie3d (FP)	leslie3d.in	2.151
444.namd (FP)	namd.input	1.781
445.gobmk (INT)	13x13.tst	0.733
450.soplex (FP)	-s1 -e -m45000 pds-50.mps	0.271
453.povray (FP)	SPEC-benchmark-ref.ini	1.465
456.hmmer (INT)	nph3.hmm	2.037
458.sjeng (INT)	ref.txt	1.182
459.GemsFDTD (FP)	/	1.146
462.libquantum (INT)	1397 8	0.459
464.h264ref (INT)	foreman_ref_encoder_baseline.cfg	1.008
470.lbm (FP)	reference.dat	0.380
471.omnetpp (INT)	omnetpp.ini	0.304
473.astar (INT)	BigLakes2048.cfg	1.165
482.sphinx3 (FP)	ctlfile . args.an4	0.803
483.xalancbmk (INT)	-v t5.xml xalanc.xsl	1.835

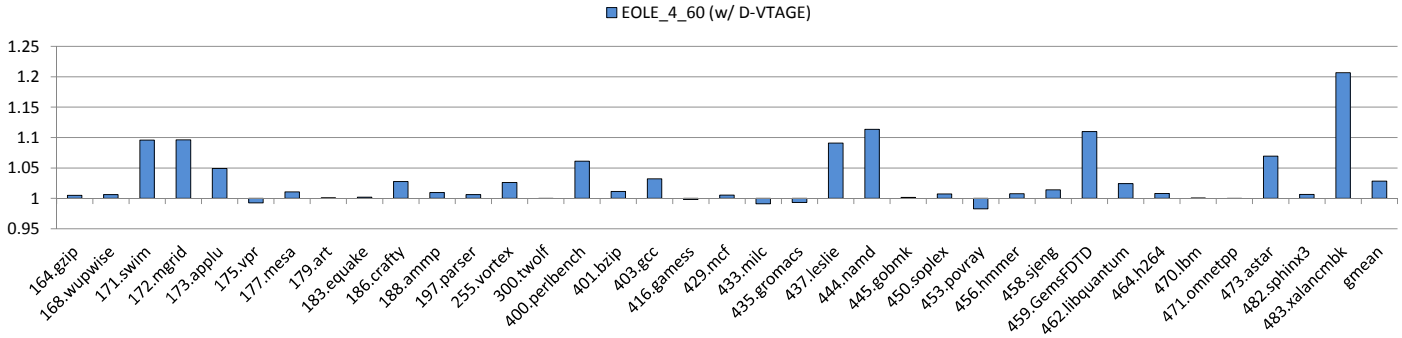
C. Benchmarks

We use a subset of the the SPEC'00 [37] and SPEC'06 [38] suites to evaluate our contributions as we focus on single-thread performance. Specifically, we use 18 integer benchmarks and 18 floating-point programs[7]. Table II summarizes the benchmarks we use as well as their input, which are part of the *reference* inputs provided in the SPEC software packages. To get relevant numbers, we identify a region of interest in the benchmark using Simpoint 3.2 [27]. We simulate the resulting slice in two steps: First, warm up all structures (caches, branch predictor and value predictor) for 50M instructions, then collect statistics for 100M instructions.

[7]We do not use the whole suites due to some missing system calls/x87 instructions in the *gem5-x86* version we use in this study.



(a) Speedup obtained with different value predictors over *Baseline_6_60*.



(b) Speedup of bank/port constrained *EOLE_4_60* [25] w/ D-VTAGE over *Baseline_VP_6_60*.

Fig. 5: Performance of D-VTAGE on a baseline 6-issue model and on a 4-issue EOLE pipeline.

D. A Remark on Power and Energy

In this work, we do not provide quantitative insights on power and energy. However, we would like to stress the following points.

Value Prediction as implemented in this paper decreases energy consumption because 1) Performance increases 2) The issue-width is reduced (thanks to EOLE). Moreover, the reduction in issue-width reduces power in the scheduler, which is a well-known hotspot [8]. Conversely, it increases energy consumption because of 1) The - simple - additional ALUs required by EOLE 2) The value predictor 3) The speculative value prediction window.

The value predictor itself is comparable in design (number of tables, ports, storage volume, pressure) - and therefore in power/energy consumption - to an aggressive branch predictor. An example of such an aggressive branch predictor – the *2bc-gskew* of the EV8 – required 44KB of storage [35]. In Section VI-C, we show that good performance is obtained with 16 to 32KB of storage only.

Regarding the associative speculative window, we argue in Section VI-B that 32 entries is a good tradeoff. This is roughly two times fewer entries than Haswell's scheduler. Assuming a baseline CAM-like scheduler and 6/8 results per cycle, then each entry of the scheduler must provision 12/16 comparators for wakeup, assuming 2 operands per entry (consider that AMD Bulldozer's actually has up to 4 per entry [14]). The speculative window only requires as many comparators per entry as there are blocks fetched per cycle (granted that the comparators are bigger since we match 15 bits instead of 6-8). That is, 64 15-bit comparators are needed as opposed to 720 8-bit comparators for a 60-entry, 2 operands per entry, 6 results per cycle, 8-bit identifier scheduler. As a result, the complexity and power consumption of the speculative window should be much lower than those of the scheduler. Moreover, it would always be possible to turn it off should the value predictor rarely hit in the speculative window (e.g. for a program with large loop bodies).

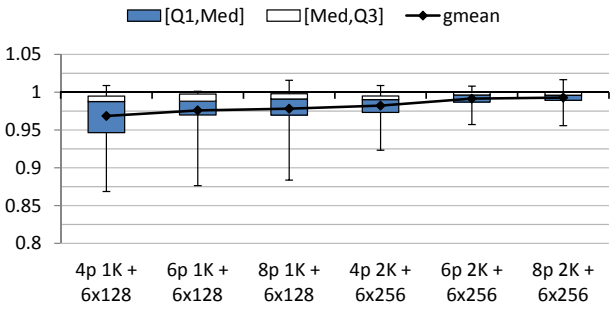
VI. Experimental Results

In further experiments, when we do not give speedup for each individual benchmark, we represent results using the geometric mean of the speedups on top of the $[Min, Max]$ box plot of the speedups.

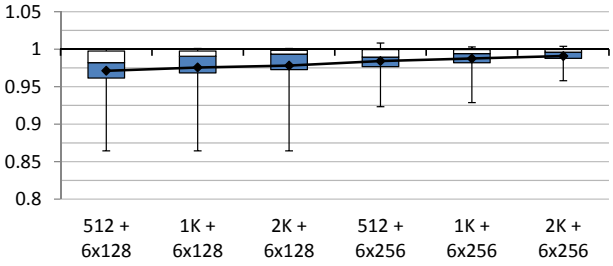
A. Baseline Value Prediction

In this first set of experiments, we study the potential of D-VTAGE **without** using BeBoP or EOLE in order to confirm that it is a good candidate for a hybrid between VTAGE and Stride. We compare it to other similarly sized predictors (e.g. 8K-entry 2-delta Stride predictor and similarly laid out VTAGE).

Fig. 5 (a) demonstrates the potential of D-VTAGE on the baseline model. First, no slowdown is observed with D-VTAGE. Second, D-VTAGE often performs better than a naive VTAGE-Stride hybrid [26] e.g. *wupwise, swim, mgrid, applu, bzip, gamess, leslie* and *GemsFDTD*. It is generally on-par with said hybrid except in *parser, gcc, astar* and *xalancbmk*, although the difference in speedup remains limited

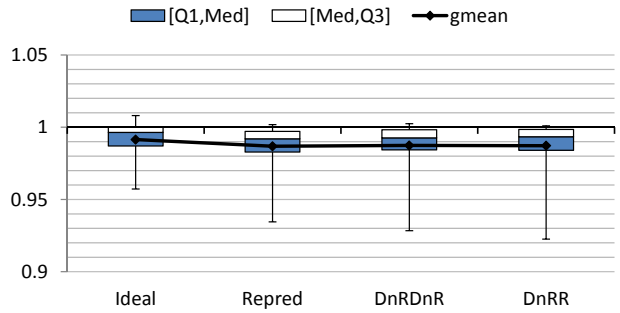


(a) Impact of the number of predictions per entry in D-VTAGE.

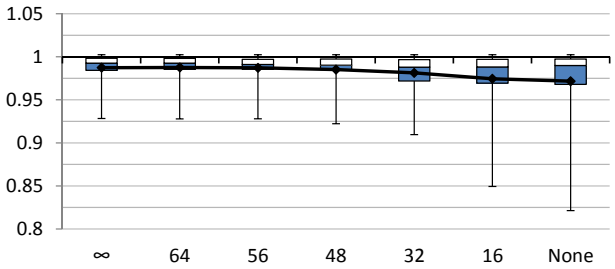


(b) Impact of the key structures sizes for a 6 predictions per entry D-VTAGE.

Fig. 6: Performance of D-VTAGE with BeBoP. Speedup over *EOLE_4_60*.



(a) Impact of the recovery policy of the speculative window and FIFO update queue on speedup over *EOLE_4_60*.



(b) Impact of the size of the speculative window on speedup over *EOLE_4_60*. The recovery policy is *DnRDnR*.

Fig. 7: Impact of the speculative window on performance for D-VTAGE with BeBoP. Speedup over *EOLE_4_60*.

(4% at most in *xalancbmk*). As a result, we consider D-VTAGE to be a serious candidate for implementation, and in further experiments, we focus on D-VTAGE only. In particular, we refer to the *Baseline_6_60* configuration to which D-VTAGE is added as the **Baseline_VP_6_60** (6-issue, 60-entry IQ) configuration.

Fig. 5 (b) depicts the speedup of a 4-issue, 60-entry IQ, 4-bank PRF, 12-read ports per bank EOLE architecture [25] featuring D-VTAGE over *Baseline_VP_6_60*. We reproduce the results of Perais and Seznec in the sense that very little slowdown is observed by scaling down the issue width from 6 to 4 (at worst 0.982 in *povray*). We refer to this configuration as **EOLE_4_60** and use it as our baseline in further experiments.

B. Block-Based Value Prediction

Fig. 6 (a) shows the impact of using BeBoP with D-VTAGE on top of *EOLE_4_60*. We respectively use 4, 6 and 8 predictions per entry in the predictor, while keeping the size of the predictor roughly constant. For each configuration, we study a predictor with either a 2K-entry base predictor and six 256-entry tagged components or a 1K-entry base predictor and six 128-entry tagged components. The *Ideal* policy is used to manage the infinite speculative window.

We first make the observation that 6 predictions per 16-byte fetch block appear sufficient. Second, we note that reducing the size of the structures plays a key role in performance. For instance, maximum slowdown is 0.876 for {*6p 1K + 6x128*}, but only 0.957 for {*6p 2K + 2x256*}.

To gain further insight, in Fig. 6 (b), we focus on a D-VTAGE predictor with 6 predictions per entry and we vary the number of entries in the base component while keeping the number of entries in the tagged components constant, and vice-versa. The results hint that decreasing the sizes of the tagged components from 256 entries to 128 has more impact than reducing the number of entries of the base predictor.

In further experiments, we consider an optimistic predictor configuration with a 2K-entry base predictor and six 256-entry tagged components. Each entry contains six predictions. Such an infrastructure has an average slowdown over *EOLE_4_60* of 0.991 and a maximum slowdown of 0.957.

a) Partial Strides: To predict, D-VTAGE adds a constant, the *stride* to the last value produced by a given instruction. In general, said constant is not very large. As most of the entries of D-VTAGE only contain strides and not last values, reducing the size of strides from 64 bits to 16 or 8 bits would provide tremendous savings in storage.

Using the baseline D-VTAGE configuration, but varying the size of the strides stored used to predict, we found that average speedup (gmean) ranges from 0.991 (64-bit strides) to 0.985 (8-bit strides), while maximum slowdown increases from 0.957 (64-bit) to 0.927 (8-bit). However, results for 16-, 32- and 64-bit strides are very similar. In other words, performance is almost entirely conserved even though the size of the predictor has been reduced from around 290KB (64-bit) to respectively 203KB (32-bit), 160KB (16-bit) and 138KB (8-bit). That is, it makes little doubt that by combining partial strides and a reduction in the number of entries of the base predictor, good performance can be attained with a storage budget that is comparable to that of the L1 D/I-Cache or the branch predictor.

b) Recovery Policy: Fig. 7 (a) gives some insight on the impact of the speculative window recovery policy

used for D-VTAGE with BeBoP. We assume an infinite speculative window. In general the differences between the realistic policies are marginal, and on average, they behave equivalently. As a result, we only consider *DnRDnR* as it reduces the number of predictor accesses versus *Repred* and it marginally outperforms *DnRR*.

c) Speculative Window Size: Fig. 7 (b) illustrates the impact of the speculative window size on D-VTAGE. When last values are not speculatively made available, some benchmarks are not accelerated as much as in the infinite window case e.g. *wupwise* (0.914 vs. 0.984), *applu* (0.866 vs. 0.996), *bzip* (0.820 vs. 0.998) and *xalancbmk* (0.923 vs. 0.973). Having 56 entries in the window, however, provides roughly the same level of performance as an infinite number of entries, while using only 32 entries appears as a good tradeoff (average performance is a slowdown of 0.980 for 32-entry and 0.988 for ∞).

C. Putting it All Together

In previous experiments, we used a baseline *EOLE_4_60* model having a D-VTAGE predictor with 6 predictions per entry, a 2K-entry base component and six 256-entry tagged components. Because it also uses 64-bit strides, it requires roughly 290KB, not even counting the speculative window. Fortunately, we saw that the size of the base predictor could be reduced without too much impact on performance. Moreover, a speculative window with only a small number of entries performs well enough. Therefore, in this Section, we devise three predictor configurations based on the results of previous experiments as well as the observation that partial strides can be used in D-VTAGE [13]. To obtain a storage budget we consider reasonable, we use 6 predictions per entry and six 128/256-entry tagged components. We then vary the size of the base predictor, the speculative window, and the stride length. Table III reports the resulting configurations.

In particular, for *Small* ($\simeq$ 16KB), we also consider a version with 4 predictions per entry but a base predictor twice as big (*Small_4p*). For *Medium* ($\simeq$ 32KB), we found that both tradeoffs have similar performance on average; for *Large* ($\simeq$ 64KB), 4 prediction per entry perform worse than 6 on average, even with a 1K-entry base predictor. Thus, we do not report results for the hypothetical *Medium_4p* and *Large_4p* for the sake of clarity.

TABLE III: Final predictor configurations. In all cases, the *DnRDnR* (i.e. realistic) recovery policy is used for the speculative window.

Predictor	#Base Ent.	#Tagged	#Spec. Win.	Str. len.	Size (KB)
Small_4p	256 (5b tag)	6 × 128	32 (15b tag)	8 bits	**17.26KB**
Small_6p	128 (5b tag)	6 × 128	32 (15b tag)	8 bits	**17.18KB**
Medium	256 (5b tag)	6 × 256	32 (15b tag)	8 bits	**32.76KB**
Large	512 (5b tag)	6 × 256	56 (15b tag)	16 bits	**61.65KB**

To summarize, the implementation cost – i.e. storage budget, but also RAM structure (see Section II-B4) – is in the same range as that of the TAGE branch predictor considered in the study. In particular, even if we omit the – marginal – cost of sequence numbers and that of the FIFO update queue (around 5.6KB for 116 blocks in flight in our configuration, at worst), the sizes reported in Table III have to be contrasted with the 44KB of the EV8 branch predictor [35].

Fig. 8 reports the speedups obtained with these four predictor configurations on top of *EOLE_4_60*, over *Baseline_6_60*. We also show performance for *Baseline_VP_6_60* (see Fig. 5 (a) in VI-A) and *EOLE_4_60* with an instruction-based D-VTAGE (see Fig. 5 (b) in VI-A). We observe that even with around 32KB of storage and practical mechanisms, we manage to preserve most of the speedup associated with an idealistic implementation of VP. Maximum speedup decreases from 1.726 (*applu*) for *EOLE_4_60* to 1.622 (*swim*) for *Medium* and 1.710 (*applu*) for *Large*. Average speedup goes from 1.154 for *EOLE_4_60* down to 1.112 for *Medium* and 1.130 for *Large*. A noteworthy data point is *GemsFDTD*, where *Medium* and *Large* perform slightly better than *EOLE_4_60*. This is due to accuracy being slightly higher in *Medium* and *Large*, even if coverage is lower.

Allocating around 17KB of storage to the predictor, however, hurts performance noticeably: Average speedup over *Baseline_6_60* is only 1.088 for *Small_6p* (although maximum speedup is 1.622 in *swim*). Yet, it still represents a noticeable improvement. Interestingly, for this storage budget, using only 4 predictions per entry but a bigger base predictor (*Small_4p*) provides better performance on average: 1.095. This is because a smaller base predictor hinders coverage in *wupwise*, *applu*, *namd* and *GemsFDTD*.

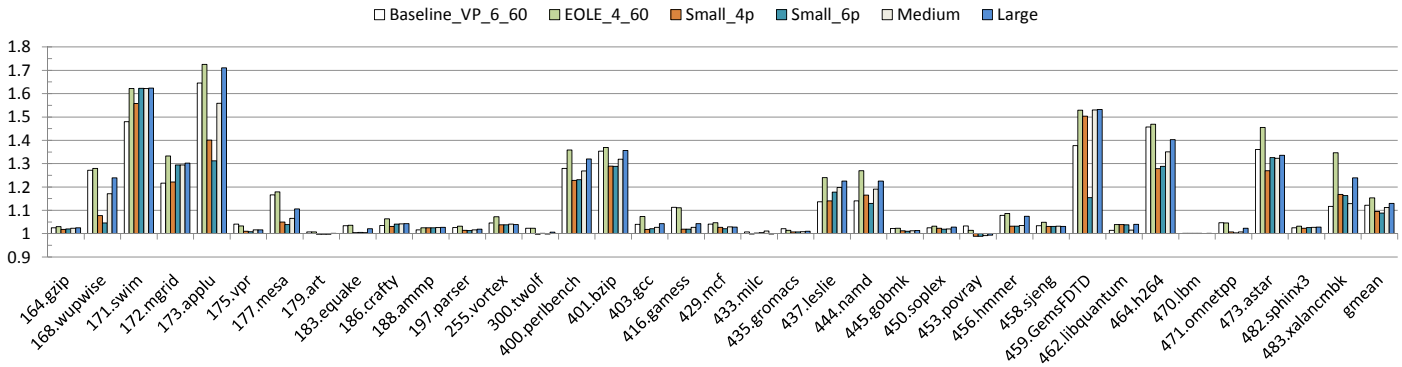


Fig. 8: Speedup brought by different D-VTAGE configurations with BeBoP on an *EOLE_4_60* pipeline over *Baseline_6_60*.

As a result, we claim that BeBoP combined to D-VTAGE is an appealing possibility for an actual implementation of VP, even at reasonable storage budget (e.g. 16/32KB).

VII. Related Work

A. Value Prediction

Lipasti et al. [19], [20] and Gabbay et al. [10], [11], [22] independently identify *Value Locality* and introduce *Value Prediction*. Sazeides et al. [32] define two complementary classes of value predictors: *Computational* and *Context-based*.

Computational predictors generate a prediction by applying a function to the value(s) produced by the previous instance(s) of the instruction, such as the addition of a constant (Stride predictor). *Context-based* predictors rely on patterns in the value history of a given static instruction to generate predictions. The main representatives of this category are n^{th} order *Finite Context Method* predictors (FCM) [32]. Such predictors are usually implemented as two-level structures. The first level (*Value History Table* or VHT) records a n-long value history – possibly compressed – and is accessed using the instruction address. The history is then hashed to form the index of the second level (*Value Prediction Table* or VPT), which contains the actual prediction [4], [33]. The major hurdle of those predictors is the two-steps lookup. The prediction critical path is too long to predict several instances of a static instruction in a short amount of time (e.g. tight loops). This is because the index computation of the 2^{nd} level for the second instance has to wait for the first prediction [26].

Another avatar of – global – context-based value predictor is the VTAGE predictor introduced by Perais and Seznec in [26]. Here, the context consists of the global branch history and the path history rather than local values. As a result, VTAGE does not require a speculative window to provide coherent predictions, contrarily to most existing predictors. VTAGE has been shown to perform better than a similarly sized order 4 FCM [33], assuming around 256KB of storage. It can also be combined to a Stride-based predictor in order to maximize coverage. Finally, VTAGE avoids the major shortcoming of FCM-like predictors: a long prediction critical path. Therefore, it is able to seamlessly provide predictions for instructions inside tight loops.

B. Hybrid Value Predictor Design

As computational and context-based predictors are complementary, many hybrid predictors have been developed. The simplest way to design a hybrid is to put two or more predictors alongside and add a metapredictor to arbitrate. Since value predictors use confidence estimation, the metapredictor can be very simple e.g. never predict if both predictors are confident but disagree, otherwise use the prediction from the confident predictor. Yet, if components are complementary, there often is overlap: storage is wasted if an instruction predictable by all components has an entry in every component. Similarly, if an instruction is not predictable by one component, it is not efficient to let it occupy an entry in said component. Rychlik et al. propose to use a classifier to attribute an instruction to one component at most [29]. This addresses space-efficiency, but not complexity.

Goeman et al. [13] propose the *Differential* FCM predictor (D-FCM). They argue that storing strides instead of values in the VHT and the VPT of FCM predictors allows to increase the coverage of the predictor while also improving table usage efficiency. Such a predictor is a hybrid between a Stride predictor and a baseline FCM predictor in the regular sense, but it actually combines the two prediction functions, allowing it to capture strided patterns that depend on the value history of instructions. Unfortunately, D-FCM suffers from the same long prediction critical path as FCM due to the two-level lookup and may not be fit for practical implementation [26].

Another hybrid is the *Per-Path Stride* predictor (PS) of Nakra et al. [23]. In PS, the instruction address is used to access the last value in the *Value History Table* (VHT), while the stride is selected in the *Stride History Table* (SHT) using a hash of the global branch history and the PC. Then, both values are summed to form the prediction. This legitimizes the use of the global branch history to predict instruction results.

C. Predictor Storage & Complexity Reduction

To further improve space-efficiency, Sato et al. propose to use two tables for the Last Value Predictor (*2-mode scheme*) [30]. One contains full-width values and the other 8- or 16-bit values. At prediction time, both tables are accessed and the one with a tag match provides the prediction. Using 0/1-bit values is also a very space-efficient alternative [31].

Loh extends the 2-mode scheme by implementing several tables of differing width [21]. The width of the result is predicted by a simple classifier at prediction time. By serializing width-prediction and table access, a single prediction table has to be accessed in order to predict. In both cases, executing *load immediate* instructions for free in the front-end overlaps with these propositions.

Burtscher et al. reduce the size of a Last n value predictor by noting that most of the high-order bits are shared between recent values [5]. Hence, they only keep one full-width value and the low-order bits of the $n-1$ previous ones. Moreover, they remark that if $n > 1$, then a stride can be computed on the fly, meaning that stride prediction can be done without additional storage.

Finally, Lee et al. leverage trace processors to reduce complexity [17]. They decouple the predictor from the *Fetch* stage: Predictions are attributed to traces by the fill unit. Thus, they solve the access latency and port arbitration problems on the predictor. They also propose some form of block-based prediction by storing predictions in the I-cache. [18]. However, a highly ported centralized structure is still required to build predictions at retire-time. To address this, Bhargava et al. replace the value predictor by a *Prediction Trace Queue* that requires a single read port and a single write port [2]. Lastly, Gabbay and Mendelson also devise a hardware structure to handle multiple prediction per cycle by using highly interleaved tables accessed with addresses of instructions in the trace cache [10]. Unfortunately, the scope of all these studies except [18] is limited to trace processors.

VIII. Conclusion

Value Prediction is a very attractive technique to enhance sequential performance in a context where power efficiency

and cycle time entail that the instruction window cannot scale easily. However, many implementation details make VP hard to imagine on real silicon. In recent work, Perais and Seznec showed that a slow – but realistic – recovery mechanism can still yield speedup [26]. They also presented a more practical predictor, VTAGE, that unfortunately cannot efficiently capture strided patterns. Then, by executing some simple ready/predicted instructions early or late outside the OoO engine, they showed that issue width can be reduced, meaning that an implementation of VP with as many PRF ports as a baseline superscalar can be envisioned. Said implementation would have a simpler Out-of-Order execution engine compared to a baseline superscalar processor [25].

In this work, we first described BeBoP, a block-based prediction scheme adapted to complex variable-length ISAs such as x86 as well as usually implemented fetch mechanism. BeBoP contributes to reducing the number of ports required on the predictor by allowing several instructions to be predicted in a single access. Then, to reduce the footprint of the value predictor, we proposed a space-efficient hybrid able to capture both strided patterns and control-flow dependent patterns, D-VTAGE. We provided solutions for an actual implementation of D-VTAGE. In particular, a small speculative window to handle in-flight instructions that only requires an associative search on *read* operations, and a reduction in size thanks to partial strides and smaller tables. In a nutshell, the hardware complexity of D-VTAGE is similar to that of an aggressive TAGE branch predictor.

As a result, this paper complements recent work and addresses the remaining complexity involved with Value Prediction. That is, the body of work consisting of [25], [26] and this paper provides one possible implementation of VP that requires less than 64KB of additional storage. The issue width is even reduced when compared to the baseline 6-issue model. Nonetheless, this implementation of VP is able to speed up execution by up to 62.2% with an average speedup of 11.2% while requiring only around 32KB of storage.

ACKNOWLEDGMENTS

This work was partially supported by the European Research Council Advanced Grant DAL No. 267175.

REFERENCES

[1] Advanced Micro Devices, "AMD K6-III Processor Data Sheet," pp. 11–12, 1998.

[2] R. Bhargava and L. K. John, "Latency and energy aware value prediction for high-frequency processors," in *Proceedings of the International Conference on Supercomputing*, 2002, pp. 45–56.

[3] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[4] M. Burtscher, "Improving context-based load value prediction," Ph.D. dissertation, University of Colorado, 2000.

[5] M. Burtscher and B. G. Zorn, "Hybridizing and coalescing load value predictors," in *Proceedings of the International Conference on Computer Design*, 2000, pp. 81–92.

[6] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," in *Proceedings of the International Symposium on Computer Architecture*, 1998, pp. 142–153.

[7] R. Eickemeyer and S. Vassiliadis, "A load-instruction unit for pipelined processors," *IBM Journal of Research and Development*, vol. 37, no. 4, pp. 547–564, 1993.

[8] D. Ernst and T. Austin, "Efficient dynamic scheduling through tag elimination," in *Proceedings of the International Symposium on Computer Architecture*, 2002, pp. 37–46.

[9] B. Fahs, T. Rafacz, S. J. Patel, and S. S. Lumetta, "Continuous optimization," in *Proceedings of the International Symposium on Computer Architecture*, 2005, pp. 86–97.

[10] F. Gabbay and A. Mendelson, "The effect of instruction fetch bandwidth on value prediction," in *Proceedings of The International Symposium on Computer Architecture*, 1998, pp. 272 –281.

[11] F. Gabbay and A. Mendelson, "Using value prediction to increase the power of speculative execution hardware," *ACM Trans. Comput. Syst.*, vol. 16, no. 3, pp. 234–270, Aug. 1998.

[12] S. Gochman, R. Ronen, I. Anati, A. Berkovits, T. Kurts, A. Naveh, A. Saeed, Z. Sperber, and R. C. Valentine, "The Intel Pentium M processor: Microarchitecture and performance," *Intel Technology Journal*, vol. 7, May 2003.

[13] B. Goeman, H. Vandierendonck, and K. De Bosschere, "Differential FCM: Increasing value prediction accuracy by improving table usage efficiency," in *Proceedings of the International Conference on High-Performance Computer Architecture*, 2001, pp. 207–216.

[14] M. Golden, S. Arekapudi, and J. Vinh, "40-entry unified out-of-order scheduler and integer execution unit for the AMD Bulldozer x86_64 core," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2011, pp. 80–82.

[15] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz, "A novel renaming scheme to exploit value temporal locality through physical register reuse and unification," in *Proceedings of the International Symposium on Microarchitecture*, 1998, pp. 216–225.

[16] I. Kim and M. H. Lipasti, "Half-price architecture," in *Proceedings of the International Symposium on Computer Architecture*, 2003, pp. 28–38.

[17] S.-J. Lee, Y. Wang, and P.-C. Yew, "Decoupled value prediction on trace processors," in *Proceedings of the International Symposium on High-Performance Computer Architecture*, 2000, pp. 231–240.

[18] S.-J. Lee and P.-C. Yew, "On table bandwidth and its update delay for value prediction on wide-issue ilp processors," *Computers, IEEE Transactions on*, vol. 50, no. 8, pp. 847–852, 2001.

[19] M. H. Lipasti and J. P. Shen, "Exceeding the dataflow limit via value prediction," in *Proceedings of the International Symposium on Microarchitecture*, 1996, pp. 226–237.

[20] M. Lipasti, C. Wilkerson, and J. Shen, "Value locality and load value prediction," *ASPLOS-VII*, 1996.

[21] G. H. Loh, "Width prediction for reducing value predictor size and power," *First Value Prediction Workshop, ISCA*, 2003.

[22] A. Mendelson and F. Gabbay, "Speculative execution based on value prediction," Technion-Israel Institute of Technology, Tech. Rep. TR1080, 1997.

[23] T. Nakra, R. Gupta, and M. Soffa, "Global context-based value prediction," in *Proceedings of the International Symposium On High-Performance Computer Architecture*, 1999, pp. 4–12.

[24] S. Palacharla, N. Jouppi, and J. Smith, "Complexity-effective superscalar processors," in *Proceedings of the International Symposium on Computer Architecture*, 1997, pp. 206–218.

[25] A. Perais and A. Seznec, "EOLE: Paving the way for an effective implementation of value prediction," in *Proceedings of the International Symposium on Computer Architecture*, 2014.

[26] A. Perais and A. Seznec, "Practical data value speculation for future high-end processors," in *Proceedings of the International Symposium on High-Performance Computer Architecture*, 2014.

[27] E. Perelman, G. Hamerly, and B. Calder, "Picking statistically valid and early simulation points," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2003, pp. 244–.

[28] V. Petric, T. Sha, and A. Roth, "Reno: a rename-based instruction optimizer," in *Proceedings of the International Symposium on Computer Architecture*, 2005, pp. 98–109.

[29] B. Rychlik, J. Faistl, B. Krug, A. Kurland, J. Sung, M. Velev, and J. Shen, "Efficient and accurate value prediction using dynamic classification," *Carnegie Mellon University, CMμART-1998-01*, 1998.

[30] T. Sato and I. Arita, "Table size reduction for data value predictors by exploiting narrow width values," in *Proceedings of the International Conference on Supercomputing*, 2000, pp. 196–205.

[31] T. Sato and I. Arita, "Low-cost value predictors using frequent value locality," in *High Performance Computing*, 2002, pp. 106–119.

[32] Y. Sazeides and J. Smith, "The predictability of data values," in *Proceedings of the International Symposium on Microarchitecture*, 1997, pp. 248–258.

[33] Y. Sazeides and J. Smith, "Implementations of context based value predictors," Department of Electrical and Computer Engineering, University of Wisconsin-Madison, Tech. Rep. ECE97-8, 1998.

[34] A. Seznec, "A new case for the TAGE branch predictor," in *Proceedings of the International Symposium on Microarchitecture*, 2011, pp. 117–127.

[35] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides, "Design tradeoffs for the alpha EV8 conditional branch predictor," in *Proceedings of the Interational Symoposiumy on Computer Architecture*, 2002, pp. 295–306.

[36] 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.

[37] Standard Performance Evaluation Corporation. CPU2000. Available: http://www.spec.org/cpu2000/

[38] Standard Performance Evaluation Corporation. CPU2006. Available: http://www.spec.org/cpu2006/