Effective Context-Sensitive Memory Dependence Prediction

Sebastian S. Kim and Alberto Ros
University of Murcia
sebastiansumin.kim@um.es, aros@ditec.um.es

***Abstract*—Memory dependence prediction is a fundamental technique to increase instruction- and memory-level parallelism in out-of-order processors, which are crucial for high performance. However, over the years, the performance gap of state-of-the-art memory dependence predictors with respect to an ideal predictor has grown due to the increase of the pipeline width, reaching up to 6% for modern architectures. State-of-the-art predictors brace context sensitivity, however, not-well-adjusted history lengths lead to loss of accuracy and high storage requirements.**

This work proposes PHAST, a novel context-sensitive memory dependence predictor that identifies for each load the minimum history length necessary to provide precise predictions. Our key observation is that for each load, it suffices to identify the youngest conflicting store and the path between them. This observation is proven empirically using an unlimited budget version of PHAST, which performs close to an ideal predictor with a 0.47% gap.

Through cycle-accurate simulation of the SPEC CPU 2017 suite, we show that a 14.5KB implementation of PHAST falls 1.50% behind an ideal predictor. Compared to the top-performing state-of-the-art predictors, PHAST achieves average speedups of 5.05% (up to 39.7%), 1.29% (up to 22.0%), and 3.04% (up to 38.2%) with respect to an 18.5KB StoreSets, a 19KB NoSQ, and a 38.6 MDP-TAGE, respectively. This stems from a considerable misprediction reduction, ranging between 62.5% and 70.0%, on average.

I. Introduction

High-performance out-of-order processors can execute loads speculatively before previous (older) stores. Still, to safeguard sequential semantics, loads need to obtain the data written by previous stores in case both the load and the store target the same memory location, i.e., they are *conflicting* or *memory dependent*. To this end, when executing, loads search the store queue (SQ) looking for older conflicting stores. The SQ holds, in program order, the stores that have been dispatched but not written to memory yet. In case of dependence, loads obtain the required data from the store through a mechanism called store-to-load forwarding. However, if the target address of previous stores has not been computed when a load executes (i.e., the load overtakes the store) the memory dependence cannot be disambiguated at that time without a prediction.

Memory dependence prediction (MDP) [24], [25] is crucial for performance in modern out-of-order processors as it predicts if a load is dependent on a previous unresolved store. In case a dependence is predicted, the load waits at the issue stage until the conflicting store computes its target address, and after that, the load can be resolved through store-to-load forwarding. When a no-dependence is predicted, the load can execute speculatively, retrieving the data either from a previously already-resolved conflicting store or from the cache hierarchy. When stores compute their target addresses, they search the load queue (LQ), which tracks loads from dispatch to commit in program order, looking for younger dependent speculative loads. In case of a match, a memory order violation is detected and the speculative load has to be re-executed.

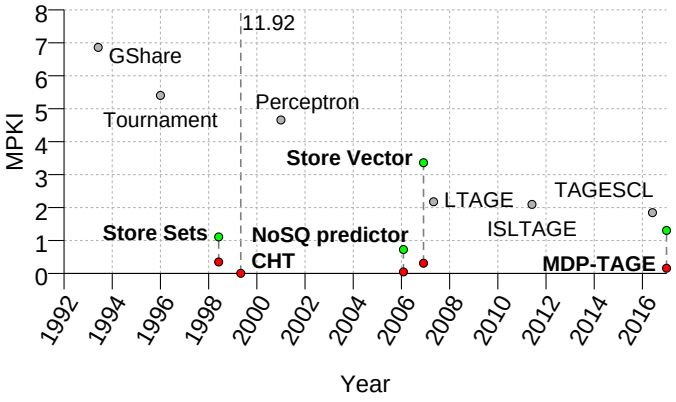
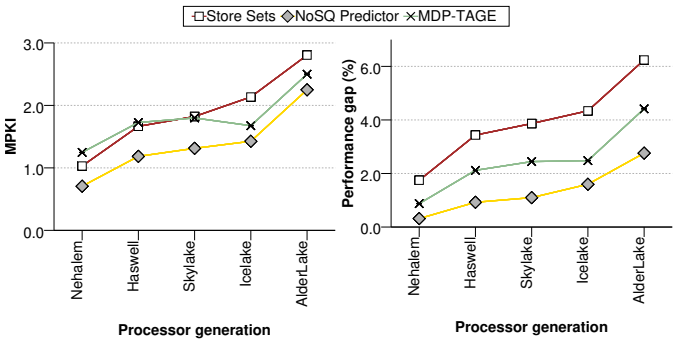


Fig. 1: Average MPKI for SPEC CPU 2017 of branch (gray) and memory dependence (red-green) predictors proposed over the past 30 years (x axis). For MDP, we show the MPKI reported by a Nehalem-like processor [13], released in 2008

While branch prediction has attracted a lot of interest over the last decades [14], [16], [17], [22], [23], [32], [33], [35]–[38], [45], MDP has received less attention, with just a few notable proposals, namely, Store Sets [7], CHT [44], Store Vector [41], the predictor employed in NoSQ [39], and MDP-TAGE [28]. On one hand, branch prediction is a more critical problem as it drives instruction fetch. On the other hand, on machines contemporary with those works, early memory dependence predictors already achieved lower mispredictions per kilo-instructions (MPKI) than branch predictors. This is depicted in Figure 1, which shows the average MPKI for the SPEC CPU 2017 benchmark suite [40] considering both branch predictors (gray circles) and memory dependence predictors (red and green circles and names in bold) proposed over the past 30 years (see Section V for a detailed description of the methodology employed). For memory dependence predictors, we show both MPKI that cause memory dependence violations (false negatives) with a red circle and MPKI due to false dependencies (false positives) with a dotted line towards a



(a) Average MPKI (lower is better)

(b) Performance gap over ideal prediction (lower is better)

Fig. 2: Trends in MDP for successive processor generations and the five state-of-the-art memory dependence predictors evaluated in this work: Store Sets, Store Vectors, the NoSQ predictor, and MDP-TAGE

green circle. While the first class of mispredictions is more critical, as they result in a pipeline squashing, the second class can also severely limit performance when it becomes frequent, as they incur unnecessary stalls.

However, the number of in-flight instructions is increasing and so is the MPKI of memory dependence predictors. The reason is that (i) there are more unresolved in-flight stores since the SQ is larger and (ii) more loads execute out-of-order with respect to previous stores due to a higher execution width. Figure 2a shows the MPKI of the memory dependence predictors considered in this work for various processor generations[1]. Store Sets, with around 1 MPKI in a Nehalem-like processor [13], suffers around 2 MPKI in the more recent AlderLake-like processor [30]. Besides, since the number of branch prediction squashes is practically independent of the processor size, memory dependence mispredictions account for a larger fraction of the squashes in modern processors. Additionally, squashes become more costly since more work is discarded when a misprediction occurs. As a result, memory dependence predictors are farther from an ideal scenario in current processors. As shown in Figure 2b, the performance gap of the mainstream Store Sets memory dependence predictor to an ideal predictor grows from 1.8% in a Nehalem-like processor to 6.0% in an AlderLake-like processor. This motivates us to revisit the problem of memory dependence prediction.

The main limitations of early state-of-the-art memory dependence predictors, e.g., Store Sets, CHT, and Store Vectors, are that (i) they link loads with a set of store instructions and (ii) they do not explicitly leverage context information, such as branch history, in their predictions. Both limitations lead to extra unnecessary load stalls due to false positives.[2] On the other hand, recent predictors, as NoSQ [39] and MDP-TAGE [28], track a single store (in particular, the *store distance* [44], i.e., the number of stores older than the load but younger than the conflicting store) and leverage context information to detect loads conflicting with stores from different paths. Nevertheless, these predictors are trained without using context-dependent history lengths. As we show in Section III, this training either provides sub-optimal performance (for shorter histories than necessary) or dramatically increases the number of tracked histories (i.e. larger histories than necessary).

This work makes two main observations. First, each time a load executes, it depends on at most one store, because even in the presence of several older stores targeting the same address, loads need to be squashed only if they overtake the younger store (the one that forwards the latest value). The only exception to this rule are loads that read from memory locations written by two or more previous stores (Section III-A). Second, the minimum context information required for precise MDP is the execution path from the store to the dependent load. In other words, adding information about branches much older than the conflicting store as in prior works does not increase the accuracy of the predictor, but pollutes the prediction tables (Section III-B).

Based on the previous observations, we propose PatH-Aware STore-distance (PHAST), a memory dependence predictor that, upon detecting a conflict, trains the predictor with (i) the information about the path taken from the store to its dependent load and (ii) the store distance. The path is defined as the global history of divergent branches, that is, any branch that can take different paths on different executions. Those branches include both conditional branches and indirect branches. The predictor is trained with a history length representative of that dependence, namely $N+1$ divergent branches older than the load, where N is the number of divergent branches between the load and the store. For prediction, loads perform multiple accesses using a set of history lengths, and on a match with one of the lengths, the corresponding store distance is provided (Section IV).

This work makes the following contributions:

- We prove that the path followed from the conflicting store to its dependent load is the only context information required to provide near-ideal accuracy (within 0.47% performance degradation).
- We design PHAST, a cost-effective predictor using only the aforementioned context information. On a conflict, PHAST considerably reduces the number of entries required for prediction compared to predetermined-history-length training schemes, by precisely detecting the minimum necessary history length for each dependence.
- Through cycle-accurate simulation, we show that PHAST achieves high accuracy with an average MPKI of 0.766. This represents a reduction of 62.0% with respect to the top-performing predictor, NoSQ. Performance-wise, with a storage budget of 14.5KB, PHAST outperforms state-of-the-art larger predictors such as 19KB NoSQ and 38.6KB MDP-TAGE with a mean speedup of 1.29% (up to 22.0%) and 3.04% (up to 38.2%), respectively.

[1]Results for CHT and Store Vector are not shown further since they underperform the mainstream Store-Sets.

[2]Store Sets does not show a high false-positive MPKI in Figure 1 because they are replaced by artificial store-store dependencies (see Section VII).

II. Background

In this section, we describe the three top-performing state-of-the-art memory dependence predictors, which we use for comparison to PHAST.

A. Store Sets

Proposed by Chrysos and Emer [7], the Store Sets predictor groups all conflicting stores of a load in a set. Each set is defined by a unique identifier, named the Store Set Identifier (SSID), which is created upon a memory order violation and assigned to both the load and the store.

Store Sets is implemented using two tagless tables. The first table, known as Store Sets Identification Table (SSIT), contains the SSID of every active set and a valid bit. SSIT is accessed by every load and store using their instruction address, or program counter (PC). If the entry in the SSIT is valid, the SSID is used to access the second table, Last Fetched Store Table (LFST). LFST includes a valid bit and the id of the last store of the set that was fetched.

Whenever a load retrieves a valid SSID, it will check the LFST to get the id of the last fetched store belonging to the set and, if it is valid, a dependency will be established between the load and the store.

Since loads must wait on all the stores in the set, stores that access the LFST and find a valid id will set a dependency on that store before updating the table. By doing so, every store in the set will be serialized and the load will execute once all the stores belonging to the set and older than the store with a dependence on the load have executed.

Given that each load and store can have one valid SSID at most, the authors proposed a rule for merging two sets in one and allowing multiple loads to depend on the same store.

The main disadvantage of the merging rule is that the predictor may end up converging several sets into one, forcing the execution of all memory instructions in order. To tackle this problem, the tables are reset periodically.

B. NoSQ predictor

Sha et al. [39] proposed a store-load bypassing predictor to map each dynamic load to the dynamic store from which it will receive the value. This predictor consists of two load-indexed tables structured as set-associative caches. Each entry contains a partial tag, a distance field that marks the dependency of the load, and an n-bit counter that indicates if the prediction should be considered.

One of the tables is for path-insensitive loads and it only requires the PC of the load to index it. The other table captures path-sensitive loads by hashing the PC of the load with a fixed number of bits of the branch history of conditional branches (1 taken/not-taken bit per branch) and calls (2 bits of the PC per call). When a memory order violation occurs, an entry is allocated in both tables. Similarly, loads will access both tables to check for a dependency and, in case of a match in both of them, the path-sensitive prediction is used.

The path-sensitive table is accessed using a history length of 8 bits. When larger history is needed to correctly predict dependencies, the number of false positives will increase. On the other hand, dependencies that require fewer branches for correct prediction will allocate in the path-sensitive table more entries than necessary.

C. MDP-TAGE

Perais and Seznec [27], [28] modified the TAGE branch predictor [34] to also predict memory dependencies using store distances. A TAGE entry consists of a partial tag, a *useful* bit (u), and a 3-bit saturating counter. When memory dependencies are predicted, only the u bit is used to control the prediction: if there is a tag match and the u bit is not zero, then the prediction is used. The 3-bit saturating counter is used to record the store distance. A value of 111b in the saturated counter is reserved to mark the load as dependent on all older stores. This way, branches, and memory dependences can use the same *Omnipredictor*. In our evaluation, we use MDP-TAGE as a standalone predictor and we increase the saturating-counter field to 7 bits to be able to track distances to all in-flight stores.

When a memory order violation is detected, a TAGE entry is allocated. If the faulting load had no prediction, the smaller history length is used to allocate an entry, setting the u bit to 1 and the counter value to the distance between the load and the store. If there was already an incorrect prediction, an entry with a larger history than the original prediction is allocated. On a prediction, the match with a larger history length is taken.

In order to forget predictions, TAGE resets the u counters periodically. This frequency has to be tuned for memory dependence prediction, which requires a higher reset frequency. To this end, when a false dependency is detected the entry can also be reset with a probability of $\frac{1}{256}$.

MDP-TAGE has three main limitations. Firstly, the predictor is trained using a sort of brute force. It starts allocating entries on conflicts using a short history length (e.g., 6 branches) and, if the used history length is not enough to offer correct predictions, it will allocate entries using larger history lengths following a geometric sequence until it reaches the length that suits the dependence. Hence, the a particular dependence can cause allocation of several entries with different history lengths.

Secondly and similar to the NoSQ predictor, when the history length used is larger than needed, MDP-TAGE will need to learn all combinations of the history, leading up to 2^n -1 additional entries (being n the number of extra history bits used). For example, assuming that a single bit per branch is tracked by the history, if a load requires 4 branches to capture the dependence and MDP-TAGE uses 6 branches, up to 4 entries will be required in the 6-bit history table to learn the dependence. This causes the same dependence to be scattered in multiple entries across the same table.

Lastly, MDP-TAGE generates many false dependencies either when a memory dependence occurs infrequently, since it needs to wait until the entry gets reset to forget the prediction, or when a short-history-length entry is allocated for a larger dependence, since dependence-free executions having the same short history will be predicted as dependent.

III. Motivation

This section elaborates on the two observations that drive our memory dependence predictor. First, loads depend commonly on a single store. Second, the context information required for precise dependence predictions is the path from the store to its dependent load.

A. Store sets versus single store

Early techniques for MDP enforce dependencies between a load and a set of previous stores [7], [41], [44]. Although those techniques can keep the MPKI that results in memory order violations very low, the MPKI because of false dependencies increases, which unnecessarily delays the execution of loads (see Section VI). In practice, however, a single dependent store is commonly found per load. Even when several previous stores target the same address as the load, the load is indeed only dependent on the youngest store among them.

Figure 3 shows several scenarios that motivate the single-store dependence in the presence of two stores targeting the load address *a*. In case (a), the load executes after the stores, and therefore the forwarding logic will provide the data written by the second store. In case (b), the load executes before the second store but after the first store, getting the data forwarded from the first store. When the second store resolves its target address, a dependency with the load is detected, and the load is correctly squashed. In case (c), the load executes after the second store, getting the data forwarded from it, but before the first store. When the first store resolves the target address, the load should not be squashed, as the loaded value is correct (see Section IV-A1 for details). Finally, in case (d), the load overtakes both stores. When the stores are resolved they conflict with the load, thus forcing its re-execution. However, the predictor should only learn the actual dependence with the second store. Indeed, in all cases where the load waits for the second store, the load does not squash, so learning just the correct store distance suffices for accurate MDP.

In some cases, the data required by a load is written by two or more previous stores. For example, in *525.x264_3*, an 8-byte load operation has 8 previous 1-byte dependent stores. In those cases, one could argue that several stores should be predicted as dependent. However, our analysis (Figure 4) shows that (1) the percentage of loads that depend on multiple stores is low (0.04% of executed loads, on average) and (2) the multiple stores commonly execute in order among other reasons because their target addresses depend on the same physical register (70%, on average). The benchmark with more loads depending on several stores is *503_bwaves*, with still a very low percentage of such loads (0.25%) which are executing in order, while fourteen benchmarks do not have any of such loads. As a result, just tracking the youngest conflicting store and waiting for it to be resolved ensures that the load executes in order with respect to all previous dependent stores. Hence, we conclude that predicting a group of stores is not necessary for achieving accurate predictions. When the load depends on different stores on different paths, context information should determine the dependence precisely.

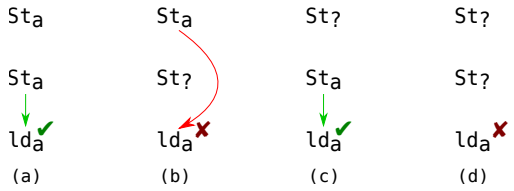


Fig. 3: Examples of two stores targeting the same address as a subsequent load. Stores with a question mark as a subscript indicate that they have not yet computed their target address. Arrows indicate forwarding (if red, incorrect forwarding). The red *x* indicates that the load will be squashed when the store computes its target address

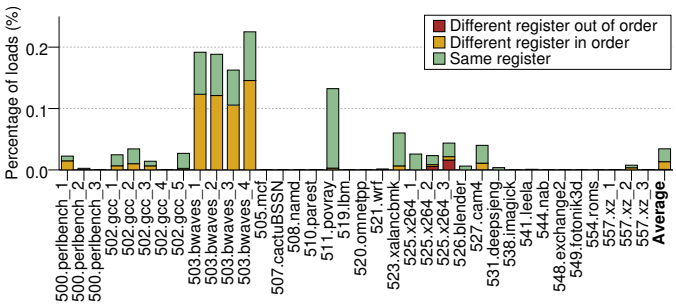


Fig. 4: Percentage of loads that depend on multiple stores

B. Context information

Store Vectors attempted to include context information in their predictions (e.g., global branch direction history) with limited success [41]. Other predictors, such as the NoSQ predictor and MDP-TAGE, leverage global branch history with a better outcome. The key aspect that brings benefits when using context information is predicting a single store distance (as in NoSQ and MDP-TAGE).

Nevertheless, NoSQ and MDP-TAGE blindly train the predictor using predetermined history lengths. NoSQ uses an 8-branch history length while MDP-TAGE uses several geometrically increasing history lengths. We argue that training the predictor with predetermined history lengths is a sub-optimal decision: when the selected history length is shorter than necessary, many false positives can be introduced; and when it is much larger than necessary, the number of entries allocated suffers an exponential explosion, which results in high pollution of the prediction structures. In this paper, we present a mechanism for determining the training history length based on the context of the conflict.

Current predictors that use context information are based on designs originally proposed for branch prediction, where, a priori, there is no notion of optimal history length. For instance, as stated in Section II, MDP-TAGE needs to perform a brute-force-like exploration to find the suitable history length for a given prediction. In contrast, we make the key observation that *for MDP the path between the conflicting store and its dependent load corresponds to the history prefix that effectively*

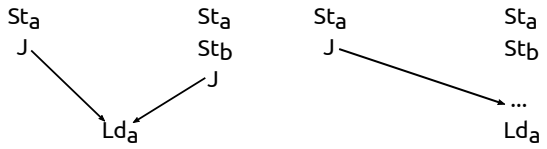


Fig. 5: Two scenarios in which information about where the conflicting store is located in the code is required for path disambiguation

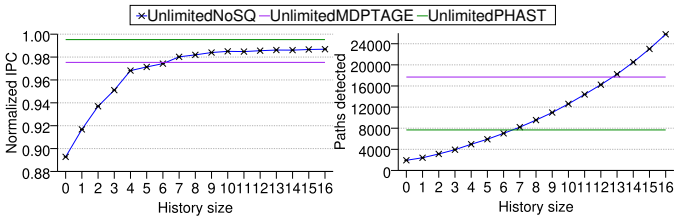
determines the context in which a conflict may happen. The rationale behind this affirmation is that if the exact path repeats from the store to the load, there is a high chance that the dependence will repeat too. Consequently, PHAST uses the history length of the store-to-load path to train the predictor. This is the key contribution of our work.

In particular, we claim that the history should consider only divergent branches, as this offers the predictor the unique path from the store to the load. Divergent branches are those that are conditional and/or indirect. For conditional branches, the necessary information is the taken/not taken outcome. For indirect branches, the information required is the target of the branch. Furthermore, to precisely determine the path of the conflict the address where the divergent branch previous to the store jumps (even in the case of conditional branches) is also included in the history. This way, $N+1$ conditional/indirect branches are collected, where N is the distance in terms of those branches from the store to the load.

Figure 5 motivates through two scenarios why the $N+1$ conditional/indirect branch is needed. In both cases, the store distance of the conflicting store on the left path is 0 (i.e. no stores appear in between St_a and Ld_a), while the store distance of the conflicting store on the right path is 1. The code executed between the load and the store contains only non-divergent branches, so no history would be necessary when tracking only branches between the load and the store. As a consequence, the same history will report conflicts with a store distance of 0 or 1 depending on the previous execution path. As mentioned, our solution to differentiate those paths is adding to the context information the target address of the divergent branch previous to the conflicting store.

C. Analysis of unconstrained predictors

Figure 6 shows a study conducted to demonstrate the claims made in this section. First, we executed unlimited versions of both the NoSQ predictor (blue line), considering a history length ranging from 1 to 16 branches (x-axis), and MDP-TAGE (purple line). The history tracks only the taken/not taken behaviour of conditional branches and either the PC of the calls for NoSQ or targets of indirect branches for MDP-TAGE, it ends with the PC of the dependent load, and it is not compressed, meaning that all previous information is present in the history. No aliasing is possible in these predictors. We report both the IPC normalized to an ideal predictor (Figure 6a) and the average number of paths tracked per application to perform the predictions (Figure 6b).



(a) IPC normalized to ideal MDP (higher is better)

(b) Average number of paths (lower is better)

Fig. 6: IPC and average number of paths detected for UnlimitedNoSQ, using different history sizes (x axis), for UnlimitedMDPTAGE, and for UnlimitedPHAST

For NoSQ, the increase in IPC is marginal when including more than 9 branches in the history, whereas the number of tracked paths increases exponentially as more branches are considered (Figure 6b). On the other hand, MDP-TAGE shows a higher IPC than the 6-branch NoSQ predictor, but lower than the 7-branch NoSQ predictor because MDP-TAGE ideal performance is biased towards the first (smallest) history used to track dependencies (6 bits). The use of larger histories (allocated when the smaller history misses dependent stores) can ideally offer extra performance, but this small uplift in performance would come with a considerable number of tracked paths (more than 16000, on average, Figure 6b), which will not help prediction when the branches are older than the one previous to the conflicting store, as claimed in this work.

In contrast, unlimitedPHAST (green line) is trained using the history length that effectively determines the context of the conflict, so it improves over NoSQ by using the optimal length for each conflict and over MDP-TAGE by always training (or allocating) the entry that corresponds to the optimal length and updating the same prediction counter. In other words, if the path between the load and the store repeats, then PHAST will allocate only one entry, whereas in MDP-TAGE the same path can be scattered among many entries and counters will only be updated for the entry used for prediction.

An example of the advantages of PHAST can be seen in *511.povray*, where a load can conflict with three different stores separated from the load by a single indirect branch. MDP-TAGE uses an initial history length of 6 and it suffers from extra memory order violations until it registers all possible path combinations. PHAST, however, suffers a single violation per store by using a 2-branch history.

Finally, the improvements of unlimitedPHAST are achieved with less than a third of the paths detected by the 16-branch NoSQ and half of the detected by MDP-TAGE. This reduced number of paths will derive in either less aliasing or less storage when using a limited version of the predictor. The performance gap between UnlimitedPHAST and an ideal predictor is just 0.47%, confirming that the proposed selection of history length is effective for MDP.

IV. PHAST

This section presents PHAST, a novel memory dependence predictor based on (i) the fact that each executed load commonly depends on at most one store and (ii) the use of the optimal history length for training the predictor on conflicts, which corresponds to the minimum length able to identify unequivocally the path between the store and the load. First, we describe the behaviour of PHAST. Then, we present our cost-effective implementation.

A. Predictor behaviour

The following aspects define a memory dependence predictor: detecting dependencies, updating the predictor, predicting dependencies, and propagating the dependencies to the scheduler.

1) Detecting dependencies: Before starting with updating the predictor it is important to describe the two main techniques employed to squash mispeculated instructions: eager squash and lazy squash. Eager squash acts as soon as the mispeculation is detected, and therefore, it can squash instructions that are part of the wrong path. Lazy squash waits until the commit stage, thus only performing squashes when actually required.

The follow-up question is when to update the predictor. There are two main choices here, too. The predictor may be updated when the mispeculation is detected, thus training it fast, but potentially with dependencies that may not exist in the execution of the program. Otherwise, the predictor may be updated at commit, when the detected mispeculation is guaranteed to actually happen.

Our evaluation is conducted using lazy squash for memory conflicts. We performed an analysis of both updating the predictors at mispeculation and at commit. We found that all state-of-the-art predictors performed better when updating at mispeculation, with the exception of NoSQ, which had a negligible difference. However, PHAST benefits from performing the update at commit, as it avoids learning long paths that are not leading to actual dependencies. It is important to note that updating the predictor at commit is also possible with eager squash. For this, it would be necessary to track the information of the mispeculated instructions in a buffer, including instruction order (e.g., an increasing *sequence number* [31]) and the information necessary to update the predictor.

We turn back our attention now to Figure 3 (d). An update when the mispeculation is detected may lead the predictor to learn the first store incorrectly as dependent if that store executes first. However, when waiting until commit, the predictor can be updated when all previous stores are executed, thus avoiding updating the predictor with unnecessary dependencies.

On the other hand, when a load receives the data through forwarding, it should not be squashed by stores older than the one forwarding the data, as depicted in Figure 3 (c). While this observation seems intuitive, these squashes happen in accurate research simulators such as Gem5 [19], and can significantly impact the predictor behaviour as shown later in Fig. 12. To avoid such squashes, the load can track the sequence number of the forwarding store. Stores searching the LQ can compare its sequence number with the sequence number of the forwarding store, ignoring the memory dependence violation if the forwarder is younger. NoSQ, not having an SQ in their proposal used a different mechanism to achieve a similar purpose. They filter squashes by making use of the Store Vulnerability Window concept (see Section VII for details).

2) Updating the predictor: PHAST is updated on the detection of a true dependence. To this end, we need information about the store distance and the path from the store to the load. The distance can be obtained in a similar way as in Store Vectors by calculating the difference between the store queue indexes of the store previous to the load and the store involved in the conflict [41]. In order to calculate the length of the history between the store and the load, a global register tracks the number of conditional and indirect branches that are decoded. When a load or store is decoded, it receives a copy of the register value. Upon a conflict, the length of the history can be obtained by calculating the difference between both values. The register should be large enough to account for wraparounds of the counter, usually to track the maximum expected number of divergent branches in the pipeline plus one extra bit [5].

For the update, when a load is about to be squashed at the commit stage, the predictor receives the PC of the load and the history length from the store to the load. Then, the history is collected from a global history register at commit.[3] The global history register (both at decode and commit) needs to track per divergent branch: a bit indicating the type of branch (conditional or indirect), a bit indicating if the branch is taken or not, and a few bits (e.g., 5 least significant bits) of the actual destination taken by the branch (the branch target if taken). Having all the history entries the same length makes the history easily to be processed in parallel.

The history is formed with the branch outcome (taken/not taken) bit for conditional branches, the destination for indirect branches, and the destination of the divergent branch previous to the store. The history is combined, using a hash function, with the PC of the load to generate the index and tag of the predictor caches.

Then, the information is stored in a prediction cache that contains a tag, a store distance field, and an n-bit confidence counter. The confidence counter is used to disable aliased dependencies that cause mispredictions. On a new entry, the store distance is stored and the confidence counter is set to the maximum. If an entry already existed with non-zero confidence, then the entry is replaced.

When a load with a predicted distance commits, it updates the confidence. If the load waited for the correct store, the counter is reset to the maximum value. Otherwise, the counter is decremented.

3) Predicting dependencies: Each load performs a MDP at the decode stage. The prediction is done for a set of history

[3]Alternatively, we could access the history at decode since we know the number of branches that the load is ahead of.

lengths considered by the predictor. The more lengths, the more accurate the prediction will be, but the more searches in the prediction tables are necessary. The histories and PC are hashed in the same way as when updating the predictor. In case of a match for any of the history lengths with a confidence greater than zero, the store distance is retrieved. If several matches are found, the larger history length is selected. If there is no match in the table or the confidence is zero, we predict no dependence.

Although this process requires a set of searches with different histories, memory dependence prediction is not as critical as branch prediction and can take several cycles to provide a prediction without hurting performance. Prediction can start as soon as the load is decoded, and the outcome is not used at least until the load is allocated in the issue queue. Still, high-performance branch predictors, e.g., TAGE, also do several searches with different history lengths.

4) Propagating dependencies to the scheduler: If a dependence is predicted between a load and a store, then the store distance must be converted into a dependence for the load to wait until the previous store executes. To propagate the dependence, PHAST employs a similar mechanism as in the NoSQ microarchitecture and MDP-TAGE. In particular, when the load is allocated in the LQ, the conflicting store is detected by subtracting the predicted distance from the index of the most recent store added in the SQ. It is worth noting that PHAST, like other store-distance predictors, is independent of the synchronization method. We opted for using the index of the SQ as register tag for the store-distance predictors evaluated in this work.

B. A cost-effective implementation

Our implementation of PHAST consists of a table for each of the possible history lengths. Tables are searched in parallel on each prediction, similar to the structure of a TAGE branch prediction, already adopted in commercial designs [9].

The first decision is the set of selected history lengths. Tracking the whole range of history lengths is not feasible from the scalability and lookups point of view. After analyzing the performance of UnlimitedPHAST with several maximum history lengths (Section VI-A) we concluded that a maximum length of 32 suffices for highly accurate prediction. With that in mind, and inspired by TAGE, we set up a geometric-like sequence of eight history lengths: (0, 2, 4, 6, 8, 12, 16, 32). Histories not covered by this sequence are truncated. For example, all dependencies with a history length of 9, 10, and 11 branches use the 8 branches closer to the load. Note that the optimal history lengths for MDP differ from the ones for branch prediction, which implies that an Omnipredictor [28] cannot be tuned for both types of prediction.

The predictor is accessed with a compressed form of the history. First, we found out by performing a sensitivity analysis that taking the five least significant bits of the branch targets suffices for avoiding most aliasing scenarios. The resulting history is then folded until $S+T$ bits remain, where S denotes the number of bits necessary for indexing our prediction table and T is the number of bits used for the tag. For the index, the PC of the load is hashed in the following manner: $(PC \bigoplus (PC >> 2) \bigoplus (PC >> 5))$, while for the tag the PC is offset by 3 and 7. The hashed PCs and the folded history are then combined with an exclusive OR.[4]

TABLE I: System configuration

4-core Alder Lake Processor [8]	
Front-end width	6-wide fetch and decode
Branch predictor	TAGE-SC-L [35]
Back-end width	12 execution ports and commit width
ROB/IQ/LQ/SB	512/204/192/114 entries
Memory hierarchy	
L1I (private)	32KB, 8 ways [42], 4-cycle hit latency, pipelined, 64 MSHRs
L1D (private)	48KB 12 ways [42], 5-cycle hit latency, pipelined, 64 MSHRs
L1D prefetcher	IP-stride with a prefetch degree of 3
L2 (private)	1.25MB, 10 ways [42], 14-cycle hit latency, 64 MSHRs
L3 (shared)	3MB/bank (4 banks), 12 ways [42], 36-cycle hit latency, 64 MSHRs
Memory	4GB, 100-cycle access latency

Each table is four-way associative, and stores entries containing a 16-bit tag, a 7-bit store distance field, a 4-bit confidence counter, and a 2-bit field for the less-recently-used (LRU) replacement policy. The confidence counter is used to discard predictions with low confidence due to aliasing. A size for each table of 128 sets (512 entries) achieves a good performance-area trade-off and requires just 14.5KB.

V. Experimental Methodology

Our simulation infrastructure consists of a cycle-accurate in-house simulator modeling in detail an out-of-order processor with an x86 instruction set architecture. The core is fed with an instruction flow (split into micro-operations at decode) generated by Sniper [6]. The memory hierarchy is modeled with Gems [21], using its embedded GARNET interconnect network model [3]. Wrong-path execution is modeled similarly as in the Scarab simulator [1]. The energy consumption of the memory dependence predictor is computed with Cacti-P [18] using a 7nm process technology [2] (see Table II).

The simulated core resembles an Intel Alder Lake microarchitecture [30]. The main system parameters are summed up in Table I. The pipeline has 3 ports for load execution and 2 ports for store execution. The 2-ported LQ and the 3-ported SB are searched associatively and in parallel with the L1D access, incurring the same latency as the L1D [10], but allowing 2 and 3 new searches, respectively, each cycle (pipelining).

Stores are issued once both the address and the data registers are ready. We perform eager squash on branch misprediction for a fast recovery, but lazy-squash for the less frequent memory order violations. Delaying memory dependence squashes until commit simplifies the design of PHAST and, as shown in the results, the number of mis-speculations is very low in all state-of-the-art MDPs, so this decision does not cause noticeable effects in performance.

[4]This hashes are indeed used by all predictors evaluated in this work as it improves their performance.

TABLE II: Configuration of the state-of-the-art predictors

Predictor	Tables	Total entries	Fields per entry	Energy per access (pJ)	Size (KB)
Store Sets	SSIT	8K	valid bit 12 bit SSID	0.2403	**18.5**
	LFST	4K	valid bit 10 bit St ID	0.1026	
NoSQ	2	4K	22 bit tag 7 bit counter 7 bit distance 2 bit lru	0.3721	**19**
MDP-TAGE	12	16K	7-15 bit tag 7 bit distance 1 bit *u*	1.3103	**38.625**
MDP-TAGE-S	8	4K	16 bit tag 7 bit distance 2 bit lru 1 bit *u*	0.4421	**13**
PHAST	8	4K	16 bit tag 4 bit counter 7 bit distance 2 bit lru	0.4856	**14.5**

We compare PHAST to the state-of-the-art memory dependence predictors described in Section II: Store Sets [7], the predictor employed by NoSQ [39], and a standalone MDP-TAGE [28] (i.e., only used for memory dependence prediction) with 7-bits to track the store distance in order to be able to track all store distances. The MDP-TAGE features 12 components that use the (6, 2000) geometric history lengths [34], but we also evaluate it using the same table and history lengths configuration as PHAST (labeled as MDP-TAGE-S, for Shorter history lengths) in order to demonstrate that our improvements are due to the accurate selection of the history length for training. Table II details the configuration and storage of the best performance-storage trade-off version of each predictor, which is used in the evaluation (the complete performance-storage analysis is presented in the next section).

Predictors are evaluated with the SPEC CPU 2017 benchmark rate suite [40]. The applications supporting multiple inputs have been relabeled with an increasing counter (each counter meaning a different input). For each pair of application/input, we generated a set of intervals using Simpoints [29]. For each interval, we simulate 100M instructions.

VI. Results

This section, first, analyses the potential of the unlimited version of PHAST. Then, we show the effects of filtering memory order violations when store-to-load forwarding takes place. Finally, we compare the top-performing state-of-the-art predictors against a cost-effective implementation of PHAST.

A. Potential of PHAST and analysis

In this section, we detail the per-application performance gap of our unlimited version with regard to a perfect memory dependence predictor. Figure 7 shows the IPC obtained with UnlimitedPHAST normalized to an ideal scenario. The geometric mean shows that UnlimitedPHAST is 0.47% behind the ideal predictor. The applications that are farther from the ideal scenario are *502.gcc_1*, *502.gcc_2*, *510.parest* and *541.leela*.

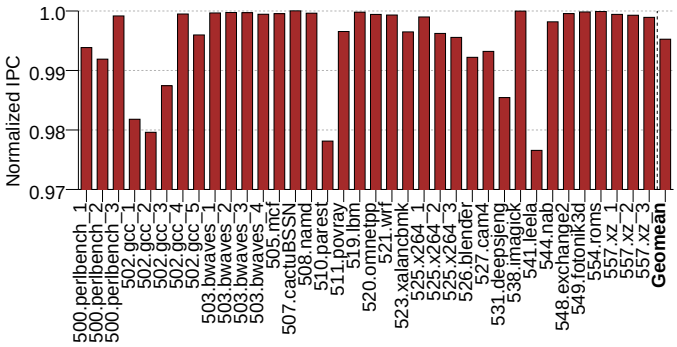


Fig. 7: IPC of the UnlimitedPHAST predictor normalized to a perfect memory dependence predictor (higher is better)

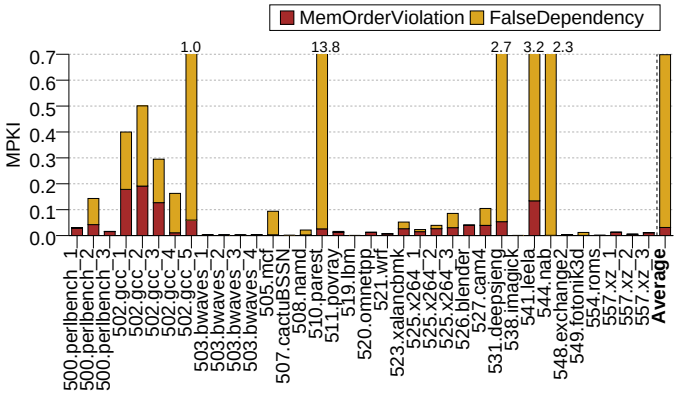


Fig. 8: MPKI of the UnlimitedPHAST predictor (lower is better)

The *502.gcc* applications exhibited the highest number of paths among all applications as well as many occasional dependencies that were not path dependent. On the other hand, *541.leela* had a total number of paths below average but suffered a similar level of memory order violations as *502.gcc* while having a higher amount of false dependencies.

Figure 8 presents the MPKI of UnlimitedPHAST. The vast majority of memory order violations are due to cold misses at the start of the simulation. Although applications *502.gcc_1* and *502.gcc_3* show a high number of memory order violations, they are cold misses. Both applications showed an abnormally high amount of paths to track as already mentioned above.

Regarding false dependencies, some applications such as *510.parest*, *531.deepsjeng*, *541.leela*, and *544.nab* present the highest amount of this type of misprediction, followed by all variants of *502.gcc*. Since our unlimited predictor tracks the full path from the branch before the conflicting store to the load, aliasing is discarded as a reason. In these cases, the problem is due to load-store pairs that are not path-independent but data-dependent, so they conflict occasionally. Finally, it is worth noting that false dependencies only impact the performance if the loads are on the critical path.

Figure 9 shows the number of paths detected per application for UnlimitedPHAST. Most applications present fewer than

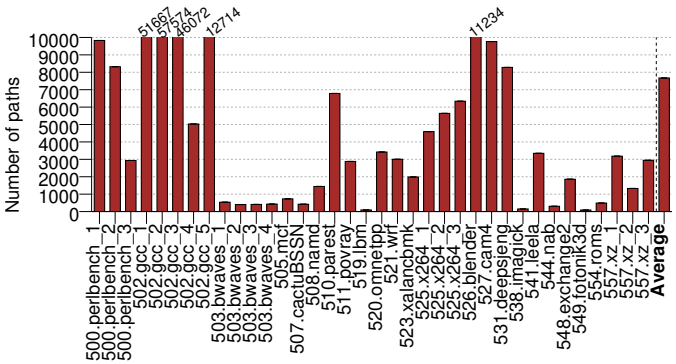


Fig. 9: Number of paths registered per application with UnlimitedPHAST

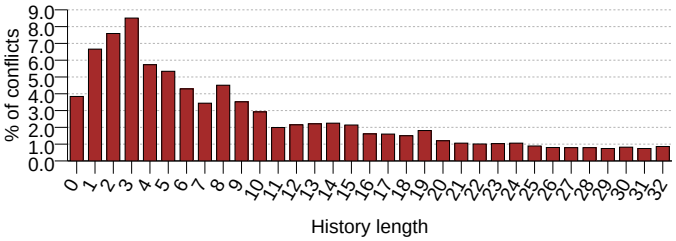


Fig. 10: Percentage of conflicts detected at each history length

five thousand different paths. Exceptions to this are *500.perlbench_2*, *502.gcc_1, 2, 3 and 5*, *526.blender*, *527.cam4*, and *531.deepsjeng*. Nonetheless, for these applications, most of the paths are used just a few times, especially the longest ones. Because of this, we believe that these long histories can be obviated without much consequences to performance.

Figure 10 depicts the percentage of unique conflicts detected with UnlimitedPHAST at each history length up to 32 branches (85.4% of all unique conflicts). Most conflicts appear in the range of $[0-19]$ branches (73.6% of all unique conflicts). In addition, dependencies with long history lengths are unlikely to show at run time, so they have less impact in execution time. Figure 11 shows how the normalized IPC of the UnlimitedPHAST is affected when limiting the history length. The figure reveals that tracking a maximum history length of 32 branches is enough to achieve the performance of unlimited histories, since most conflicts occur with shorter history lengths. It is worth noting that for most benchmarks it is enough to track up to 16 branches, but for a small subset, greater histories offer performance improvements.

B. Effect of avoiding squashes on forwarding

Figure 12 shows, for our evaluated predictors, the geometric mean of the IPC normalized with respect to an ideal predictor. The predictors are both run when the optimization to avoid squashing for forwarded loads described in Section IV-A1 is off (No FWD) and on (FWD). This optimization, despite not being present in state-of-the-art simulators is fundamental for performance when predicting a single dependent store. In all the paper, except in this sub-section, our evaluation uses FWD.

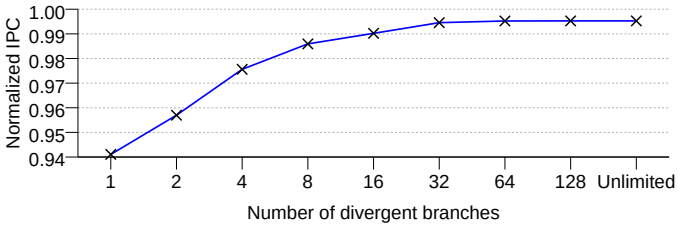


Fig. 11: Normalized IPC of UnlimitedPHAST at several maximum history lengths (higher is better)

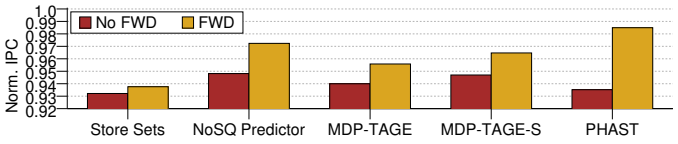


Fig. 12: IPC of memory dependence predictors against a perfect predictor with (FWD) and without filtering through forwarding (higher is better)

Store Sets shows less than a 1% increase in IPC when enabling the forwarding optimization. The reason for the variation is the merging of sets. When a store has to be added to a new set, it will cause the merging of both sets, predicting false positives in two loads.

On the other hand, the NoSQ predictor and MDP-TAGE show an increment of around 2% in performance. Since loads only wait for one store, filtering re-executions helps eliminate the incorrect memory order violations depicted in Fig 3(c).

PHAST is by large the most benefited predictor, with an increase of 5%. The reason is that when several stores match the load address (as depicted in Figure 3), the filtering allows PHAST to only learn the dependence between the load and the most recent store. If PHAST learns older *incorrect* dependences (Figure 3(c)), there is a high chance that they have longer histories, making PHAST select them over the correct dependence. In those cases, PHAST will predict wrong distances until the saturated counter of that entry reaches zero. At that point, although PHAST will give the correct distance, the memory order violation with the *incorrect* store may re-trigger. Although NoSQ and MDP-TAGE are also susceptible to this case, they show a higher IPC without FWD because they reduce the memory order violations by increasing false dependencies. In the case of NoSQ, once the saturated counter is below the threshold, the path-independent table will provide the wrong prediction until its counter is reduced to below the threshold, while MDP-TAGE will only reset the entry with a probability of 1/256 or after 512K accesses.

C. Comparison to state-of-the-art predictors

We start our comparison by showing the performance-storage trade-off of the evaluated predictors in Figure 13. PHAST is able to outperform all the state-of-the-art predictors while requiring less storage. Both PHAST and MDP-TAGE-S use a similar prediction structure, being the key difference the fact

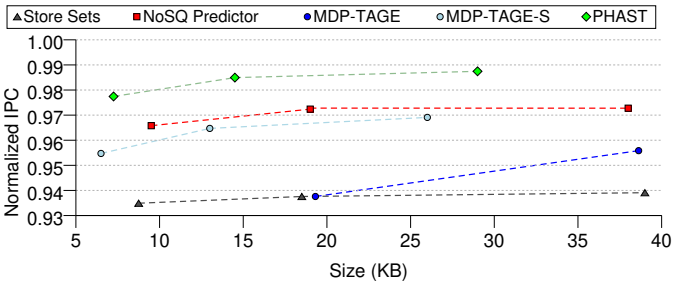


Fig. 13: Performance (higher is better) versus storage of Store Sets, NoSQ predictor, MDP-TAGE, MDP-TAGE with PHAST configuration and PHAST at different budgets compared against an ideal MDP

that PHAST chooses the minimum effective history length for training, while MDP-TAGE starts the training at the lower predetermined length and increases it on mispredictions. Even with a budget of 7.25 KB, PHAST still shows a higher IPC (97.74% compared to the ideal) than the state-of-the-art predictors. Store Sets and the NoSQ predictor show practically no improvement after doubling their storage. For the rest of the evaluation, we use for each predictor the size that achieves better performance-storage trade-off, detailed in Table II.

The MPKI for each application is depicted in Figure 14. Store Sets includes an implicit path sensitivity since the dependence will only be predicted if the instance of the store is present [39]. Nonetheless, sharing the entries of the SSIT between loads and stores is a double-edged sword: on the one hand, it helps to reduce false dependencies by not making the load stall if an instance of the dependent store is not present. On the other hand, merging sets increase false dependencies. Adding explicit path information to the memory dependence predictor helps in reducing mispeculations in NoSQ and MPD-TAGE. MDP-TAGE-S is able to reduce false negatives with respect to MDP-TAGE thanks to the use of shorter histories. This supports our claim that an Omnipredictor cannot be tuned for both MDP and branch prediction. However, MDP-TAGE-S has the highest MPKI due to false dependencies due to the large number of tables with short histories, which causes a load to require several mispredictions until it finds its right history length. PHAST presents the lower MPKI for both false negatives and false positives.

Our unlimited implementation of PHAST showed low performance in applications *502.gcc_1* to *502.gcc_3*, and *541.leela*. However, in the case of *502.gcc*, PHAST is able to greatly decrease the MPKI compared to the rest of the state-of-the-art predictors. While the MPKI for false negatives that PHAST predicted is on par with NoSQ and MDP-TAGE-S, the false positives are reduced to half.

Regarding *541.leela*, it did not have many path-dependent conflicts, which increased the false positive mispredictions of PHAST. Once the saturated counter was decreased to zero, the next time the conflict is presented, PHAST would not predict it, leading to a memory order violation. NoSQ is able to reduce the false negative MPKI with the use of the path-independent table, which in turn increases the false positive MPKI.

On the other hand, PHAST exhibits great performance in applications such as *500.perlbench_1*, *511.povray*, and *531.deepsjeng*, where both types of MPKI are significantly decreased. *511.povray*, is an application where memory dependencies are tightly connected to branch history. This has been corroborated by Perais and Seznec in their MDP-TAGE [27], as it was able to improve the performance of this application. However, the use of not-well-adjusted history lengths leads to a loss of accuracy.

Figure 15 shows the IPC for all applications normalized to an ideal predictor. PHAST is the closest to the ideal MDP with a gap of 1.5%, improving over Store Sets by 5.05% (up to 39.7%), the NoSQ predictor by 1.29% (up to 22.0%), MDP-TAGE by 3.04% (up to 38.2%) and MDP-TAGE-S by 2.10% (up to 17.6%). These results come as a consequence of the MPKI reduction shown from each application in Figure 14.

Another issue regarding Store Sets is that when multiple instances of a store are in-flight, the load will be always made dependent on the youngest of those instances while also forcing all instances to execute in-order. NoSQ and PHAST are able to overcome this problem by using explicit path information. This can be seen in applications such as *500.perlbench_3*, where NosQ, PHAST, and even MDP-TAGE achieve at least a 95% of the IPC compared to an ideal predictor, while Store Sets falls behind.

MDP-TAGE performs slightly better than Store Sets because it always marks a load to depend on a unique store, at most. However, it has a higher amount of memory order violations compared to NoSQ and PHAST because when predicting for a single store, accuracy is vital. MDP-TAGE trains blindly on a set of geometric history lengths, which accounts for many mispredictions that either result in unnecessary stalls or in squashes. Although the NoSQ predictor uses a fixed length, it avoids many of the squashes with the path-insensitive table at the cost of increasing the false dependencies.

Regarding NoSQ, our PHAST implementation has a speedup of 1.29%, achieving the same or better performance in all applications with the exception of *525.x264* and *541.leela*. Overall, both predictors present a low MPKI related to memory order violations but, PHAST has fewer MPKI due to false dependencies, which makes a total reduction in MPKI of 62% (20% less MPKI in memory order violations and 65% less MPKI in false dependencies). The main reason for this improvement is using only the path information comprehended between the conflicting store and the dependent load. NoSQ, by using a fixed history length, can either lead to an explosion of paths or extra false positives.

Finally, regarding energy consumption, it is key to highlight that PHAST reduces the number of re-executed instructions by 2% with respect to the NoSQ predictor and 8% with respect to Store Sets, which implies important reductions in the energy consumption of the entire core. Figure 16 shows the energy consumption of the predictors, broken down into reads and writes. The main observation is that the consumption of

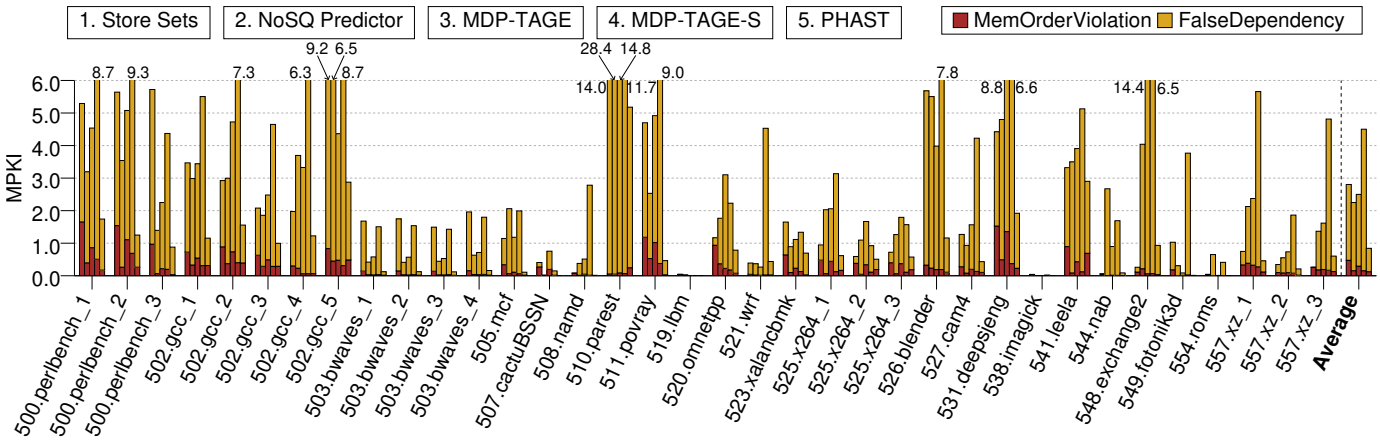


Fig. 14: MPKI of the evaluated memory dependence predictors with the exception of CHT (lower is better)

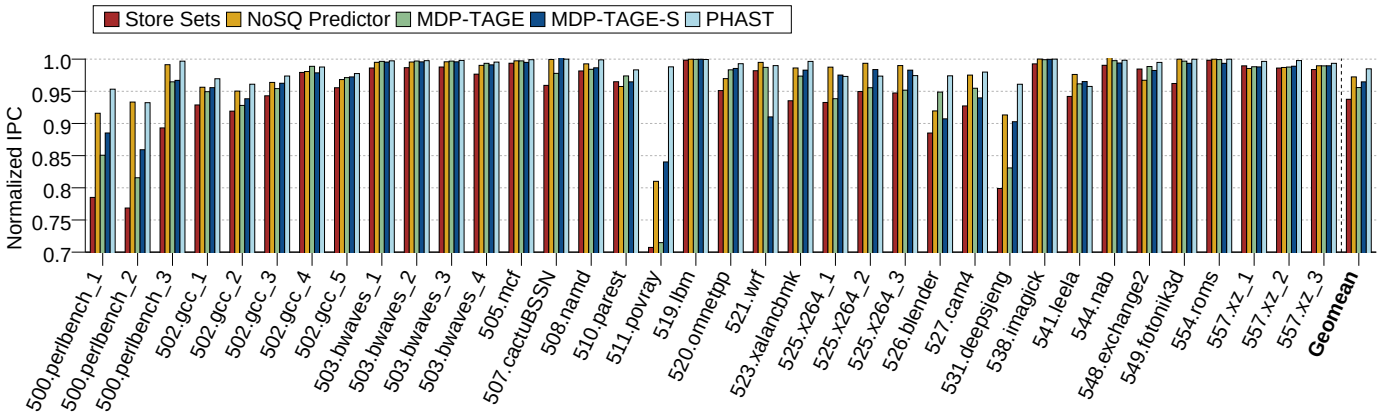


Fig. 15: IPC of the memory dependence predictors per application normalized to the perfect MDP (higher is better)

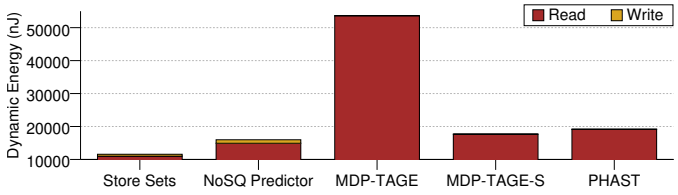


Fig. 16: Energy consumption (nJ) of the evaluated predictors

standard TAGE-like predictors is much higher than the other memory dependence predictors. Hence, compared to the vast structures (branch predictors and BTBs) required to steer fetch accurately, the consumption of the memory dependent predictor represents a small fraction.

VII. Related work

Önder and Gupta [26] analyzed the main drawbacks of the Store Sets predictor: the serialization of the stores of the same set and the inability to distinguish dependencies among multiple instances of the same store. If Store Sets allows out-of-order issuing of the stores of a set, *false memory order violations* may appear between the load and stores from the set other than the correct provider. To tackle this problem, they moved the memory disambiguation to commit stage by comparing the value obtained by the load with the value of the store. If the values are the same, the memory order violation is ignored. In our case, the filtering done with the forwarding (Section IV-A1) can achieve a similar result (except for the case of silent stores) without having to do an extra check on commit.

NoSQ [39] performs speculative memory bypassing (SMB) [25], [43] without an SQ. It makes use of the Store Vulnerability Window (SVW) described by Roth et al. [31]. The SVW filters re-execution with the help of a Store Sequence Bloom Filter (SSBF), which is a small untagged direct mapped address-indexed table that tracks the sequence number of the youngest committed store that wrote in each address. When a load executes, it records the sequence number of the last store that committed. If a store forwards the data to the load, that sequence number will be overwritten with the one corresponding to the forwarding store. Later, prior to commit, the load will access the SSBF with its target address and check if the sequence number contained is younger than the one it recorded. The rule to skip re-execution is different for bypassing and non-bypassing loads. The latter will perform an inequality

test, skipping re-execution if the sequence number recorded by the load is less than or equal to the sequence number written in the entry of the SSBF. For bypassing loads, re-execution can only be skipped if the sequence numbers are the same. NoSQ upgraded the SSBF to a tagged set-associative table managed in a FIFO fashion, which improves the filtering of squashes.Again, PHAST is also able to filter squashes when older stores than a forwarding one conflict with the load (Section IV-A1).

Later, Jin and Önder [15] proposed improvements to the NoSQ mechanism. When NoSQ has low confidence for a predicted memory dependence, instead of bypassing, it forces the load to wait until the store is committed and the cache is updated. To eliminate the need of delaying these low confidence loads, the authors proposed to perform predication and take both the cache data and the store data. If the prediction was right, the store data is kept; otherwise, the cache value is used.

Alves et al. [4] proposed a mechanism to filter the L1/TLB probes by using a store-queue/buffer/cache (S/QBC). This adds a third logical partition to the SQ/SB that maintains data that has already been written back. A memory dependence predictor based on store distance [44] is used to predict a hit or miss in the S/QBC. Hits predicted correctly reduce energy consumption since the L1/TLB is not probed, while correctly predicted misses reduce the latency by letting the load probe both the S/QBC and L1/TLB in parallel.

Lustig et al. [20] make use of memory dependence prediction and memory disambiguation to achieve high-performance forwarding. Memory dependence prediction is used to anticipate the same-physical address dependencies between stores and loads. Later, the memory disambiguation will assert the prediction. If correct, the pairing of the store and the load ensures that synonyms can be detected while maintaining the TLB off the critical path.

Huang et al. [12] propose the use of software assistance to identify loads that will not conflict with older unresolved stores and prevent them from competing for the LQ and SQ resources. The idea is to analyze the binary with software and annotate it. Later, the hardware can make use of the annotations to know what set of dynamic memory operations needs memory disambiguation. These instructions, guaranteed not to overlap with older unresolved stores, do not need to check the SQ when they execute, nor do they need an entry in the LQ.

Hasan [11] proposes a perceptron-based memory dependence predictor designed for energy-constrained devices. The scheme is based on the application of perceptron to branch prediction and uses a history vector containing the results of the past *n* loads retired and whether or not they caused a violation. The resulting memory dependence predictor was able to gain almost as much IPC speedup as the Store Sets.

VIII. Conclusion

We have presented PHAST, a memory dependence predictor that is trained with the execution path between the conflicting store and its dependent load and predicts the exact distance of the conflicting store on that path. We have shown that using that path information, an unlimited predictor can practically reach the performance of an ideal predictor while keeping the number of tracked paths reduced.

With a budget of just 14.5KB PHAST outperforms all the state-of-the-art predictors evaluated in this work (speedups of 5.05% over Store Sets, 3.04% over MDP-TAGE and 1.29% over the NoSQ predictor, with improvements of up to 22% compared against NoSQ) and with a performance gap of 1.5% with respect to an ideal predictor.

Acknowledgments

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 819134), from the MCIN/AEI/10.13039/501100011033/ and the "ERDF A way of making Europe", EU (grant PID2022-136315OB-I00), and from the MCIN/AEI/10.13039/501100011033/ and the European Union NextGenerationEU/PRTR (grant TED2021-130233B-C33). Sebastian S. Kim is a PhD student funded by the Fundación Séneca, Región of Murcia (21456/FPI/20).

References

[1] "Scarab Simulator," https://github.com/hpsresearchgroup/scarab.
[2] "PCACTI," https://sportlab.usc.edu/downloads/packages, 2021.
[3] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2009, pp. 33–42.
[4] R. Alves, A. Ros, D. Black-Schaffer, and S. Kaxiras, "Filter caching for free: The untapped potential of the store buffer," in *46th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2019, pp. 436–448.
[5] A. Buyuktosunoglu, A. El-Moursy, and D. H. Albonesi, "An oldest-first selection logic implementation for non-compacting issue queues," in *15th Annual Int'l ASIC/SOC Conference*, Sep. 2002, pp. 31–35.
[6] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations," in *Conf. on Supercomputing (SC)*, Nov. 2011, pp. 52:1–52:12.
[7] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," in *25th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 1998, pp. 142–153.
[8] clamchowder, "Popping the hood on golden cove," https://chipsandcheese.com/2021/12/02/popping-the-hood-on-golden-cove, Dec. 2021.
[9] M. Evers, "AMD next generation "zen 3" core," in *33rd HotChips Symp.*, Aug. 2021.
[10] A. Fog, "The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers," https://www.agner.org/optimize/microarchitecture.pdf, Technical University of Denmark, Nov. 2022.
[11] K. M. Hasan, "Memory dependence prediction for energy constrained devices," Master's thesis, University of Toronto (Canada), 2021.
[12] R. Huang, A. Garg, and M. C. Huang, "Software-hardware cooperative memory disambiguation," in *12th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2006, pp. 244–253.
[13] *First the Tick, Now the Tock: Next Generation Intel Microarchitecture (Nehalem)*, Intel Corporation, White paper, Apr. 2009.
[14] D. A. Jiménez and C. Lin, "Dynamic branch prediction with perceptrons," in *7th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Jan. 2001, pp. 197–206.
[15] Z. Jin and S. Önder, "Dynamic memory dependence predication," in *45th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2018, pp. 235–246.
[16] T. A. Khan, M. Ugur, K. Nathella, D. Sunwoo, H. Litz, D. A. Jiménez, and B. Kasikci, "Whisper: Profile-guided branch misprediction elimination for data center applications," in *55th Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2022, pp. 19–34.
[17] J. K. F. Lee and A. J. Smith, "Analysis of branch prediction strategies and branch target buffer design," EECS Department, University of California, Berkeley, Tech. Rep. UCB/CSD-83-121, Aug. 1983.

[18] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques," in *2011 Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov. 2011, pp. 694–701.

[19] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, M. D. S. Boris Shingarov, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and Éder F. Zulian, "The gem5 simulator: Version 20.0+," *arXiv preprint arXiv:2007.03152*, 2020.

[20] D. Lustig, G. Sethi, M. Martonosi, and A. Bhattacharjee, "Coatcheck: Verifying memory ordering at the hardware-os interface," in *21st Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Apr. 2016, pp. 233–247.

[21] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, Sep. 2005.

[22] S. McFarling, "Combining branch predictors," Digital Western Research Laboratory, Technical report TN-36, Jun. 1993.

[23] P. Michaud, "An alternative TAGE-like conditional branch predictor," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 15, no. 3, pp. 30:1–30:24, Oct. 2018.

[24] A. Moshovos, "Memory dependence prediction," Ph.D. dissertation, University of Wisconsin-Madison, 1998.

[25] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi, "Dynamic speculation and synchronization of data dependences," in *1997 Int'l Symp. on Computer Architecture (ISCA)*, Jun. 1997, pp. 181–193.

[26] S. Önder and R. Gupta, "Dynamic memory disambiguation in the presence of out-of-order store issuing," in *32nd Int'l Symp. on Microarchitecture (MICRO)*, Dec. 1999, pp. 170–176.

[27] A. Perais and A. Seznec, "Storage-free memory dependency prediction," *IEEE Computer Architecture Letters*, vol. 16, no. 2, pp. 149–152, Jul. 2017.

[28] A. Perais and A. Seznec, "Cost effective speculation with the omnipredictor," in *27th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Nov. 2018, pp. 25:1–25:13.

[29] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 1, pp. 318–319, Jun. 2003.

[30] E. Rotem, A. Yoaz, L. Rappoport, S. J. Robinson, J. Y. Mandelblat, A. Gihon, E. Weissmann, R. Chabukswar, V. Basin, R. Fenger, M. Gupta, and A. Yasin, "Intel Alder Lake CPU architectures," *IEEE Micro*, vol. 42, no. 3, pp. 13–19, Mar. 2022.

[31] A. Roth, "Store vulnerability window (SVW): Re-execution filtering for enhanced load optimization," in *32nd Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2005, pp. 458–468.

[32] A. Seznec, "The L-TAGE branch predictor," *The Journal of Instruction-Level Parallelism*, vol. 9, pp. 1–13, May 2007.

[33] A. Seznec, "A 64-Kbytes ITTAGE indirect branch predictor," in *2nd JILP Workshop on Computer Architecture Competitions (JWAC-2): Championship Branch Prediction*, Jun. 2011.

[34] A. Seznec, "A new case for the tage branch predictor," in *44th Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2011, pp. 117–127.

[35] A. Seznec, "TAGE-SC-L branch predictors again," in *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, Jun. 2016.

[36] A. Seznec and P. Michaud, "De-aliased hybrid branch predictors," INRIA, Research report RR-3618, 1999.

[37] A. Seznec and P. Michaud, "A case for (partially) tagged geometric history length branch prediction," *The Journal of Instruction-Level Parallelism*, vol. 8, 2006.

[38] A. Seznec, J. S. Miguel, and J. Albericio, "The inner most loop iteration counter: A new dimension in branch history," in *48th Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2015, pp. 347–357.

[39] T. Sha, M. M. K. Martin, and A. Roth, "NoSQ: Store-load communication without a store queue," in *39th Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2006, pp. 285–296.

[40] Standard Performance Evaluation Corporation, "SPEC CPU2017," 2017. [Online]. Available: http://www.spec.org/cpu2017

[41] S. Subramaniam and G. H. Loh, "Store vectors for scalable memory dependence prediction and scheduling," in *12th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2006, pp. 65–76.

[42] A. Syed, "Intel 12th gen alder lake golden cove-gracemont cache configuration detailed," Jul. 2021, https://www.hardwaretimes.com/intel-12th-gen-alder-lake-golden-cove-gracemont-cache-configuration-detailed/.

[43] G. S. Tyson and T. M. Austin, "Improving the accuracy and performance of memory communication through renaming," in *30th Int'l Symp. on Microarchitecture (MICRO)*, Dec. 1997, pp. 218–227.

[44] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan, "Speculation techniques for improving load related instruction scheduling," in *26th Int'l Symp. on Computer Architecture (ISCA)*, May 1999, pp. 42–53.

[45] S. Zangeneh, S. Pruett, S. Lym, and Y. N. Patt, "Branchnet: A convolutional neural network to predict hard-to-predict branches," in *53rd Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2020, pp. 118–130.