AVPP: Address-first Value-next Predictor with Value Prefetching for Improving the Efficiency of Load Value Prediction

LOIS OROSA, University of Campinas (UNICAMP) and ETH Zürich
RODOLFO AZEVEDO, University of Campinas (UNICAMP)
ONUR MUTLU, ETH Zürich

Value prediction improves instruction level parallelism in superscalar processors by breaking true data dependencies. Although this technique can significantly improve overall performance, most of the state-of-the-art value prediction approaches require high hardware cost, which is the main obstacle for its wide adoption in current processors. To tackle this issue, we revisit *load* value prediction as an efficient alternative to the classical approaches that predict *all* instructions. By speculating only on loads, the pressure over shared resources (e.g., the Physical Register File) and the predictor size can be substantially reduced (e.g., more than 90% reduction compared to recent works). We observe that existing value predictors cannot achieve very high performance when speculating only on load instructions. To solve this problem, we propose a new, accurate and low-cost mechanism for predicting the values of load instructions: the Address-first Value-next Predictor with Value Prefetching (AVPP). The key idea of our predictor is to predict the load address first (which, we find, is much more predictable than the value) and to use a small non-speculative Value Table (VT)—indexed by the predicted address—to predict the value next. To increase the coverage of AVPP, we aim to increase the hit rate of the VT by predicting also the load address of a *future instance* of the same load instruction and prefetching its value in the VT. We show that AVPP is relatively easy to implement, requiring only 2.5% of the area of a 32KB L1 data cache. We compare our mechanism with five state-of-the-art value prediction techniques, evaluated within the context of load value prediction, in a relatively narrow out-of-order processor. On average, our AVPP predictor achieves 11.2% speedup and 3.7% of energy savings over the baseline processor, outperforming *all* the state-of-the-art predictors in 16 of the 23 benchmarks we evaluate. We evaluate AVPP implemented together with different prefetching techniques, showing additive performance gains (20% average speedup). In addition, we propose a new taxonomy to classify different value predictor *policies* regarding predictor update, predictor availability, and in-flight pending updates. We evaluate these policies in detail.

CCS Concepts: • **Computer systems organization → Superscalar architectures**; *Complex instruction set computing*; *Multicore architectures*;

Additional Key Words and Phrases: Value prediction, load value prediction, instruction level parallelism, speculative execution, address prediction, prefetching

This work was supported by FAPESP (fellowships 2013/08293-7, 2014/03840-2, and 2016/18929-4), CNPq, and CAPES.
Authors' addresses: L. Orosa and R. Azevedo, Institute of Computing, University of Campinas, Av. Albert Einstein 1251, 13083-852 Campinas, Brazil; emails: {lois.orosa, rodolfo}@ic.unicamp.br; L. Orosa and O. Mutlu, Department of Computer Science, ETH Zürich, Universitätstr. 6, 8092 Zürich, Switzerland; emails: lois.orosa.nogueira@gmail.com, onur.mutlu@inf.ethz.ch.

ACM 1544-3566/2018/12-ART49
https://doi.org/10.1145/3239567

ACM Reference format:
Lois Orosa, Rodolfo Azevedo, and Onur Mutlu. 2018. AVPP: Address-first Value-next Predictor with Value Prefetching for Improving the Efficiency of Load Value Prediction. *ACM Trans. Archit. Code Optim.* 15, 4, Article 49 (December 2018), 30 pages.
https://doi.org/10.1145/3239567

1 INTRODUCTION

Improving single thread performance is critical to accelerate modern applications. In many cases, these applications are difficult to parallelize and scale with a large number of threads [8, 24, 36]. Also, many parallel applications spend a large amount of time in serialized code portions [3, 29, 60], which limits their overall performance.

Value prediction is a technique that aims to improve single thread performance in out-of-order processors by increasing Instruction Level Parallelism (ILP). The key idea is to break true data dependencies by predicting the output value of an instruction and execute dependent instructions speculatively. The technique was simultaneously proposed by Lipasti et al. [34, 35] and Mendelson and Gabbay [37]. Many subsequent works proposed new value prediction techniques [9, 20, 22, 44, 52, 55, 62, 63, 66].

More recently, Perais and Seznec revisited the topic [16, 46–49] with the main goal of reducing complexity, which is a major reason as to why value prediction is still not implemented in current processors. They simplified value prediction with four different proposals.

First, in Reference [48], the authors propose a confidence estimation mechanism that improves value prediction accuracy. This allows the use of a much simpler mechanism to recover from value mispredictions (pipeline squashing). This work also proposes the VTAGE predictor, which outperforms all prior state-of-the-art predictors.

Second, EOLE [47] tackles the problem of extra ports required in the Physical Register File (PRF) for value prediction. EOLE modifies the architecture to enable the execution of single-cycle instructions in both the in-order front-end and the in-order back-end. The operands of such single-cycle instructions are not read from the PRF. EOLE reduces the complexity of the out-of-order engine but increases the complexity of the in-order front-end and back-end (by adding extra ALUs and logic). As a result, EOLE allows a narrower out-of-order instruction window without causing performance loss.

Third, BeBoP [49] tackles the problem of high number of ports required by the value predictor to provide several predictions and updates per cycle. BeBoP uses a single entry to place all the predictions associated with a single cache block, instead of using a single entry per instruction. This work also proposes Differential VTAGE, which provides better performance and lower cost than VTAGE, but it requires additional hardware to predict tight loops correctly.

Fourth, in Reference [46], the authors propose a technique that reuses registers with values that are predicted to be the same as the output of the current instruction. They use Instruction Distance (IDist) to identify which physical registers contain values that can be used as predictions. They propose an Inflight Shared Registers Buffer (RSET) for tracking reuse opportunities. RSET provides 5% average speedup with realistic 10.1KB structures, lower than previous value prediction schemes [47, 48].

Even though these previous works have proposed excellent alternatives to reduce complexity, the cost of value prediction is, unfortunately, still a concern, especially in energy-constrained processors. In Reference [48] and EOLE [47], the predictors are large (8,192 entries), and they have several ports to satisfy the demand (up to eight predictions per cycle in an eight-issue core). BeBop [49] proposes a solution to the multi-port problem, but it still requires predictors with many (2,048) entries (one entry per fetch block). Finally, in Reference [46], Perais and Seznec reduce the predictor structure size, but this comes at the cost of lower performance.

Our goal in this article is to reduce value prediction complexity while preserving its performance benefits. Our contributions are based on three observations: First, predicting only load values provides speedup very close to that of predicting all instruction's values. Second, load instructions comprise only 25% of all instructions, which allow us to (1) use small predictors with fewer ports and (2) reduce the back pressure over the PRF. Third, load addresses are more predictable than load values. Based on these observations, we design a new load value predictor, the Address-first Value-next Predictor with Value Prefetching (AVPP).

The key idea of our AVPP is to leverage the better predictability of a load instruction's effective address to more accurately estimate the value of a load instruction. The AVPP predictor is divided into two consecutive parts; the first part is a classical predictor indexed by the Program Counter (PC), which returns the predicted address for the current instance of a given load instruction. The second part of the predictor is a non-speculative Value Table (VT) indexed by the predicted address, which returns the value predicted for the load instruction. To increase the coverage of AVPP, we increase the hit rate in the VT by using an adaptive algorithm to prefetch future predicted addresses on time in the VT. Our AVPP predictor outperforms five state-of-the-art predictors [15, 22, 34, 48, 49], which we evaluate in the context of load value prediction on a relatively narrow out-of-order processor. AVPP reduces the overall hardware overhead dedicated to value prediction, as we leverage, as much as possible, existing structures for the implementation (i.e., the load queue).

We make the following three **key contributions**:

(1) We introduce a new load value predictor that exploits predictability of load addresses to provide accurate load value prediction, the AVPP predictor (Section 4). This predictor outperforms five state-of-the-art predictors and it significantly improves the performance of several benchmarks that do *not* benefit from any of the other predictors. The size of our AVPP predictor is only 2.5% that of a 32KB L1 data cache size; it has only 1 read port, 1 write port and 512 entries. AVPP provides an average speedup of 11.2% (up to 53%) and average energy savings of 3.7% (up to 15%), outperforming *all* the state-of-the-art predictors in 16 of 23 benchmarks. We also show that our predictor is complementary to conventional prefetchers.

(2) We propose a new taxonomy for value prediction policies (Section 5) based on different design choices that can be made to implement value prediction. Although most of the policies were already used by previous works, they were *not* formally classified, compared or evaluated. We divide the policies into three categories: predictor updates, predictor availability and in-flight pending updates.

(3) We propose specialized microarchitecture optimizations for load value prediction (Section 6) that simplify the required pipeline changes and make load value prediction easier to implement.

2 MOTIVATION

The potential gain of value prediction depends on two factors. First, the data dependency patterns between the instructions being executed. The more dependencies between instructions, the better the potential for speculation. Second, the number of in-flight instructions that can be maintained simultaneously in the instruction window (i.e., the aggressiveness of the processor). The bigger the instruction window, the more the opportunities for value prediction to issue speculative instructions.

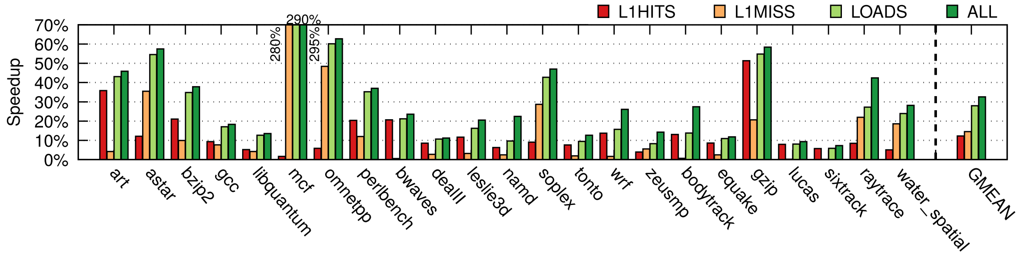


Fig. 1. Speedup when speculating with perfect value prediction on only L1 hit loads (L1HITS); only L1 miss loads (L1MISS); all LOADS; and ALL instructions.

In this work, we focus on narrow, energy-constrained microarchitectures. Many processors have relatively small instruction windows (128 entries) due to power considerations, as is the case with the ARM Cortex-A72 [4] or the Intel Westmere microarchitecture [19]. In this section, we show the potential of value prediction in a four-issue processor (configuration details in Section 7) by evaluating an *oracle* value predictor that always provides correct value predictions. All the correctly predicted values are consumed by the dependent instructions, thereby breaking the true data dependencies. We run four experiments where an oracle predictor predicts the following four categories of instructions: (1) only loads that hit in L1 cache (L1HITS), (2) only loads that miss in L1 (L1MISS), (3) all load instructions (LOADS) and (4) all instructions (ALL).

Figure 1 shows the speedup resulting from these experiments on the benchmarks tested in our evaluation. We make two key observations. First, in most cases, correctly speculating only on load instructions provides almost the same potential speedup (28% on average) as speculating on all the instructions (32% on average), which provides the theoretical maximum speedup for a particular architecture. Second, speculating on L1 hit loads provides more performance than speculating on L1 miss loads on 16 of 23 benchmarks. The reason is that, even though L1 hits provide lower gain per instruction, they are also much more frequent than L1 misses, which results in a better overall speedup. We will show in our evaluation (Section 7) that, in practice, most of the correct predictions are L1 hits.

Although load instructions are the primary cause of pipeline stalls [1, 41, 43], they represent, on average, only 25% of all dynamic instructions in our experiments, which implies that the performance gain per speculated load instruction is more significant than with non-load instructions.

In the benchmarks used in our evaluation, the average performance gain resulting from speculating on (1) all instructions is 0.19 cycles per instruction, (2) only the non-load instructions is 0.07 cycles per instruction and (3) only the load instructions is 0.6 cycles per instruction. Thus, the per-instruction benefit of speculating on load instructions is $\approx$10$\times$ higher than that of speculating on non-load instructions. Based on these observations, we argue that predicting the values of only load instructions is likely a more efficient way of implementing value prediction (i.e., the way that likely provides the best performance/cost ratio).

Another alternative to maximize prediction efficiency is to use mechanisms to find the most critical instructions for speculation. Calder et al. [11] propose to speculate only on important instructions that are expected to be on the critical path. Similarly, Tune et al. [64] dynamically identify instructions likely to be on the critical path. However, determining the critical path in a program is a difficult problem [18, 29, 30], which requires additional complexity and sophisticated hardware structures.

The current research trend in value prediction is to (1) predict *all* instructions to achieve very high performance and 2) try to reduce the hardware cost as much as possible [46–49]. We argue that such approaches are limited by the very large number of instructions these approaches have to predict.

In this work, we revisit the idea of *load value prediction*, i.e., predicting the values of *only* load instructions. The first value predictors were also focused on predicting the values of only load instructions [35], but they did *not* focus on reducing the hardware cost and complexity. We propose an efficient and low-cost load value prediction scheme that uses fewer resources than the best state-of-the-art approaches, without sacrificing performance. We achieve our goals by (1) leveraging existing structures in conventional out-of-order processors and (2) using a new load value predictor that outperforms all the state-of-the-art value predictors, achieving speedups close to the oracle in L1HITS (11.2% vs. 12.3%).

3 STATE-OF-THE-ART VALUE PREDICTORS

In this section, we describe the most important state-of-the-art value predictors, five of which are later used in our evaluations (Section 7).

The Last Value Predictor **LVP** [34, 35] predicts that the next value will be the same as the last one. The predictor latency can be one cycle for such a simple predictor (because it is implemented as a single direct-mapped array).

The **Stride** predictor [37] is able to detect stride patterns between consecutive values. It predicts that the next value will be the sum of the last value plus the calculated stride. When the predictor is updated, the stride is calculated by subtracting the last value from the new value, and the last value is updated with the new value. It is implemented as a tagged cache that contains the last value and the stride. The predictor latency can be two cycles (one cycle for reading the table, another cycle for calculating the predicted value).

The 2-Delta Stride (**2D-Stride**) predictor [15] is an optimization of the Stride predictor that is designed to minimize the number of mispredictions in regular stride patterns that eventually have breaks in their sequence. For example, in a loop processing an array, when the array reaches its end and starts again to iterate in the same or in another array, the 2D-Stride predictor mispredicts only once. The 2D-Stride predictor latency is the same as the Stride predictor (two cycles).

The Finite Context Method (**FCM**) predictor [54] is a context-based predictor. It is implemented with two tables: The Value History Table (VHT) is indexed by the PC and it keeps the history of the last values accessed by the instruction. The Value Prediction Table (VPT) keeps the actual prediction and is indexed by the hashed history from the VHT. The predictor latency is two cycles, one cycle to read the VHT and another cycle to read the VPT. The predictor update requires one additional cycle to compute the hash.

The Differential FCM (**DFCM**) predictor [22] stores strides instead of values in the VHT. This technique improves the FCM predictor's accuracy significantly. The predictor latency is one more cycle than FCM to sum the value and the stride (three cycles in total).

The **VTAGE** predictor [48] calculates its predictions based on the control flow history. VTAGE is an adaptation of the ITTAGE indirect branch predictor [57], which exploits correlations on the global branch history. VTAGE predictor performs well when the instruction values depend on the control flow. VTAGE uses several tables, each one composed of the value, a confidence counter, a tag and a bit used for the replacement policy. Each table is indexed by an increasing number of bits from the global branch history, hashed with the PC. There is also a base predictor that is accessed only by the PC, and it is implemented as a tagless last-value predictor (with a confidence counter). The VTAGE predictor latency is two cycles (one cycle to hash the PC and the global history, and one cycle to read the tables).

Finally, similarly to the DFCM predictor, the Differential VTAGE (**DVTAGE**) predictor [49] is an improvement over the VTAGE predictor that stores strides instead of full values. The predictor latency is one more cycle than VTAGE to perform the stride addition (three cycles in total).

4 THE AVPP PREDICTOR: ADDRESS-FIRST VALUE-NEXT PREDICTOR WITH VALUE PREFETCHING

The load address patterns of many applications are more predictable than their value access patterns [23]. This is common, for example, in a loop that accesses an array of values: the addresses follow a regular pattern (stride), whereas the values themselves could be completely random.

To leverage this observation, we create a new load value predictor that relies on address prediction. Our AVPP predictor (Address-first Value-next Predictor with Value Prefetching) achieves high coverage and accuracy, which leads to better performance than the best state-of-the-art value predictors (as we show in Section 7).

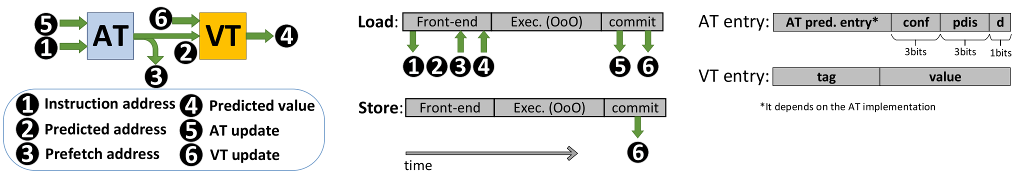
The key idea of AVPP is to predict the address first, and use it to index a small non-speculative table to obtain the predicted value next. AVPP is composed of two main parts. First, the value prediction part (called AVPP prediction), which predicts the output value of a load instruction and allows dependent instructions to execute speculatively (Section 4.1). Second, the value prefetching part (called value prefetching), which allows improving the coverage of the value prediction part (i.e., the first part) by increasing the predictor hit rate (Section 4.2). As a positive side effect, value prefetching brings data to L1, which could also be beneficial for reducing the load access latency of a regular request (as in conventional prefetching techniques). However, as we show in our evaluation (Section 7.3), this prefetching effect has much less influence on the overall performance improvement provided by AVPP. Most of the performance benefits come from the ability of AVPP to *accurately predict load values* in a timely manner.

4.1 AVPP Prediction

To facilitate the description of our mechanism, we define the four types of addresses involved in a load instruction: (1) the *instruction address* is the PC of the load instruction, (2) the *load address* is the address accessed by the load instruction, (3) the *predicted address* is the address that AVPP predicts to be accessed by the *current* instance of the load instruction (or the *load address* prediction), (4) the *prefetch address* is an address that AVPP predicts will be accessed by a *future* instance of the load instruction.

Figure 2(a) shows the high-level description of the two tables that compose the AVPP predictor. The Address Table (AT) is indexed by the *instruction address* ❶, and it returns the *predicted address* ❷. The Value Table (VT) is indexed by the *predicted address*, and it returns the corresponding predicted value ❹, which has to be prefetched in advance ❸. Figure 2(b) shows when the key actions of the predictor happen in the pipeline. Figure 2(c) shows the AT and VT entry format. AT can be implemented with any of the state-of-the-art predictors (with some restrictions, as we will see in Section 4.2), and VT is a tagged direct mapped array (in our implementation). A confidence counter (conf) is maintained in each AT entry to improve the accuracy of the predictor.

Although VT is a small cache, we do not leverage the L1 cache for the same purpose. The main reason is that prediction and load access are performed in different places in the pipeline. The prediction has to be performed in the front-end, whereas the load access is performed later in the Out of Order (OoO) execution engine. Even assuming that we can overcome this issue, we would have two other issues if we leverage the L1 cache to store the VT values. First, the L1 cache is designed with a number of ports appropriate to support the processor read and write requests. The value predictor would *increase* the number of requests significantly, degrading the performance of



(a) High-level overview. (b) Load (LD) and Store (ST) pipeline. (c) AT and VT entries.

Fig. 2. The AVPP predictor: (a) high-level overview, (b) timeline of load (LD) and store (ST) execution, and (c) AT and VT entries.

the overall system. To solve this problem, the L1 cache could be designed with a larger number of ports, increasing its area and energy overheads. Second, the prediction latency would increase because of the contention and conflict misses caused by demand requests.

Predictions are made in two steps at the early stages of each load instruction. First, the AT is accessed with the *instruction address* ❶. If the AT hits and the confidence counter is saturated (a.k.a. the address prediction is reliable), then the AT generates a *predicted address* ❷ and AVPP moves on to the second step. Otherwise, AVPP does *not* generate a load value prediction. Second, the VT is indexed by the *predicted address* generated by the AT in the first step. If the *predicted address* hits in the VT, then the VT returns the predicted value ❹, and the processor feeds this value to the dependent instructions, which are executed speculatively. Otherwise, the processor does *not* speculate.

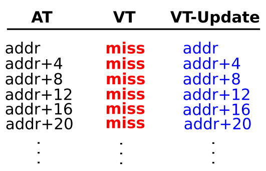
In a single-core system, AVPP predicts the load value correctly if the *predicted address* is correct. In case the address prediction is wrong, the value prediction would also likely be wrong (unless the memory location at the wrong *predicted address* has the same value as the memory location of the correct *load address)*. If the value prediction is wrong, then the confidence counter is reset. In a single-core system, we keep the contents of VT coherent with memory by updating the VT with the new values generated by store instructions. In Section 4.3, we explain how our mechanism works in a multi-core system.

Predictor Updates are made at commit time. The AT is updated with the *load address* generated by the instruction ❺, which at this point of the pipeline is already known. The VT is updated ❻ with (1) the values prefetched by our mechanism and (2) the values of each store instruction (to keep the VT coherent with memory).

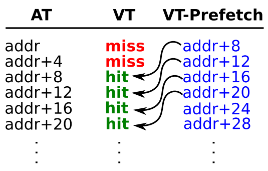
4.2 Value Prefetching: Increasing the Hit Rate in the VT

To improve the hit rate in the VT, AVPP uses the *prefetch address* to prefetch data into the VT that is predicted to be accessed by future instances of the same load instruction. The AVPP *prefetch address* is generated at address prediction time, and it is used to prefetch data into the VT only when the *predicted address* generated in the AT is reliable.

Figure 3 illustrates why the VT needs a prefetching mechanism. The AT column shows the *predicted address* for the load instruction and the VT column shows if the *predicted address* hits or misses in the VT. The load instruction represented in the figure follows an address pattern with a stride-4 pattern. Figure 3(a) and Figure 3(b) show six consecutive predictions. We assume that all the address predictions are correct in AT (a.k.a., the *predicted address* is equal to the *load address*). Figure 3(a) shows a naive VT update policy, where the VT updates are made with the values of the load addresses that are accessed by the load instructions. The VT-Update column shows the address whose value updates the VT at the *commit time* of the load instruction. With this policy, a simple stride pattern in the load address will *always miss* in the VT (i.e., not obtaining any valid value prediction).



(a) VT update *without* prefetching. The load instruction follows a stride-4 pattern, and the VT is updated with the committed value of the current load address.



(b) VT update *with* value prefetching. The load instruction follows a stride-4 access pattern. The mechanism prefetches into the VT the value of the load address from *two iterations ahead* of the current load access.

Fig. 3. Examples of (a) VT update without prefetching and (b) VT update with prefetching.

Figure 3(b) shows the prefetching-based VT update policy, which updates the VT with the value of the *prefetch address* calculated by our value prefetching mechanism (VT-Prefetch). The prefetch address is predicted to be used some iterations ahead of the current instance of the load instruction. In the first occurrence of the load instruction, the *predicted address* misses in VT, and the mechanism prefetches the value of *addr+8*. After two occurrences, the *predicted address* starts hitting the VT.

The ideal prefetch distance depends on three main factors. First, the frequency with which the load instruction is executed. Second, the latency to access the memory location requested by the load instruction. Third, the probability that other load instructions evict the value from VT. However, a mechanism that predicts these factors would increase the complexity of the predictor.

We make two observations that simplify the calculation of the prefetch distance in our mechanism. First, if the prefetch is performed too early, the corresponding VT entry has a higher probability of being evicted from VT. Second, if the prefetch is performed too late, the *predicted address* has a high probability of missing in the value table. Considering these observations, we design an algorithm to calculate the prefetch distance of each load instruction dynamically. In each occurrence of a load instruction, AVPP decides between keeping the current distance, increasing it, or decreasing it. To implement our approach, we add N extra bits (*pdis* in Figure 2(c)) per AT entry to indicate the prefetch distance. In our experiments, we found that a maximum distance of 8 (3 bits) is enough to provide good results.

Inspired by the forward probabilistic counters (FPC) used by Perais and Seznec [48], the AVPP updates *pdis* with a certain probability (*probUp*), which emulates a small confidence counter. Finally, each entry of the AT also requires a direction bit (*d* in Figure 2(c)) to indicate the direction of the *pdis* update (keep, increase or decrease).

For each value prediction, if the *predicted address* in the AT is reliable, AVPP calculates the *prefetch address* with distance *pdis* and it prefetches the *prefetch address* value into VT.

The *prefetch address* and the *predicted address* are calculated at the same time in the AT, and they should be generated with low latency and low-cost. Therefore, not all the classical predictors are suitable for the prefetching part of our AVPP predictor. The main requirement is that the prefetch address predictor implementation has to be simple. For example, in a Stride predictor, predicting an address with prefetch distance of 8 is as easy as left-shifting the stride by 3 bits and adding it to the last value. In other predictors, such as the FCM predictor, the complexity and latency to calculate the *prefetch address* can be very high [22, 54]. For this reason, in our evaluation (Section 7) we only consider the Stride and the DVTAGE predictors for implementing the AT. Unlike the *predicted address*, the *prefetch address* is propagated into the memory access stage in the pipeline (in the OoO engine). Our prefetch mechanism allocates an entry in the load queue for the *prefetch address* (which we call value prefetch entry). Only if the *prefetch address* and the *load address* are the same,

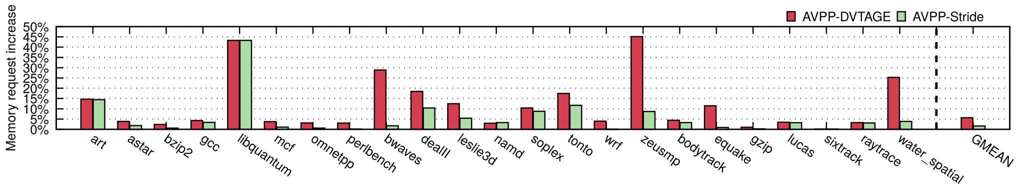


Fig. 4. Increase in memory requests due to value prefetches.

they coalesce into a unique value prefetch entry. The load queue needs an extra bit to indicate if the entry corresponds to a value prefetch or a demand load.

Figure 4 shows the increase in the memory requests caused by the value prefetch requests, with the AT implemented either as a DVTAGE predictor or as a Stride predictor (Section 3). In both cases, the typical increase is under 20%, and the geometric mean on a variety of benchmarks is under 5%. We show in our evaluation (Section 7) that we can keep the load queue the same size as the baseline without degrading performance.

Our mechanism adjusts the predictor distance (*pdis*) dynamically with the goal of prefetching at the right time to increase the VT hit rate. AVPP detects the reason of the VT miss and acts accordingly. The two reasons why the *predicted address* can miss the VT are (1) the value prefetching is late or (2) the value prefetching is too early (and it is replaced by some other value in the VT). The mechanism is triggered when the *predicted address* **misses in the VT**, and it works as follows:

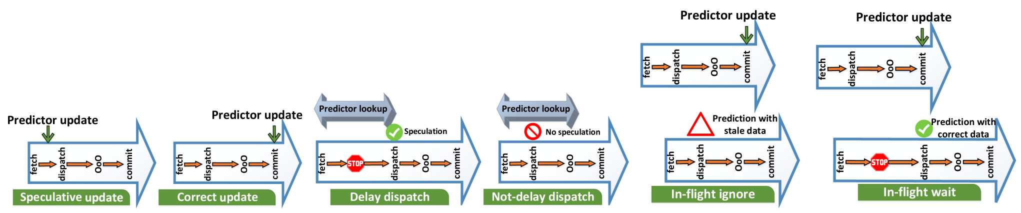
- If the load queue has a value prefetch entry for the *predicted address*, then the prefetched memory access is not completed yet. This indicates that the current prefetch distance is too short, and the distance (*pdis*) should be increased. Therefore:
  - If *d* is 1 (increase direction), then *pdis* is increased with probability *probUp* (if it is not saturated).
  - If *d* is 0 (decrease direction), then we change the direction of the updates by setting the *d* bit to 1 (increase direction). *pdis* is *not* updated.
- If the load queue does not have any value prefetch entry for the *predicted address*, then the prefetch memory access request must have already been served and also evicted from the VT. This indicates that the current prefetch distance is too large, and the distance (*pdis*) should be decreased. Therefore:
  - If *d* is 0 (decrease direction), then *pdis* is decreased with probability *probUp* (if it is not zero).
  - If *d* is 1 (increase direction), then we change the direction of the updates by setting the *d* bit to 0 (decrease direction). *pdis* is not updated.

If the *predicted address* **hits in the VT**, then *pdis* is not updated.

4.3 Coherence in Multi-core Processors

In multi-core processors, different cores could have different versions of the same data in their private caches. Most modern multi-core processors keep the data coherent between cores by implementing a cache coherence protocol in hardware.

The VT is coherent with memory in a single-core processor, but to keep the data in VT coherent in a multi-core processor, we would need a VT coherence mechanism. To keep the design simple, our approach does *not* keep the VT coherent with memory, which might cause some additional mispredictions. These mispredictions do *not* affect the correctness of execution. In our evaluation



(a) Predictor update. (b) Prediction availability. (c) In-flight Pending updates.

Fig. 5. Different policies for predictor update, predictor availability and in-flight pending predictions.

with multi-core processors (Section 7.11) we do *not* observe any mispredictions caused by incoherent data in VT.

5 TAXONOMY OF VALUE PREDICTION POLICIES

A variety of value prediction proposals do not explain or pay much attention to the description of the predictor policies used in their mechanisms, which makes some of these proposals difficult to reproduce and compare with. In an attempt to normalize these design choices, we introduce a *new taxonomy* of different policies for three fundamental aspects of value prediction: predictor update, predictor availability, and in-flight pending updates. Our taxonomy is general: It can be applied to load value prediction or value prediction for all instructions. Previous taxonomies [53] cover several aspects of the microarchitecture details, but ours is the first covering these three prediction-level considerations. We evaluate the design space of our taxonomy in Section 7. Figure 5 summarizes all the policies that we discuss in the following sections.

5.1 Predictor Update Policies

The value predictor can be updated in different places in the pipeline. In our taxonomy, as illustrated in Figure 5(a), we include two policies. First, the ***correct update*** policy, in which the predictor is updated at the commit stage. The correct value is known at this point, so the predictor is updated with the correct value.

Second, the ***speculative update*** policy, in which the predictor is updated just after making the prediction at the fetch stage, and before dispatch time. As the correct value is *not* known at this point, the predictor is updated with the predicted value. With the *speculative update* policy, a prediction is based on previous speculative updates. When a misprediction is detected (at commit time), the predictor is updated with the correct value, by either resetting the entry or by restoring the old state before the speculative update [27]. In this case, the complexity of restoring the old state of the prediction entry depends on the predictor. A simple last value predictor only requires overwriting the speculative value with the correct value.

We can also implement a hybrid approach that only updates the predictor speculatively at fetch time if the prediction is reliable. Otherwise, the predictor is updated with the correct value at commit time. In both cases, the prediction is validated at commit time. To implement this hybrid policy, we need an additional bit in each entry to indicate if the predictor is speculatively updated at fetch time, to avoid duplicate updates at commit time.

5.2 Prediction Availability Policies

To speculate using a predicted value, the predicted value has to be ready at dispatch time. However, this does not always happen, as the prediction can be delayed for different reasons (predictor contention, waiting for previous predictions, etc.). The system can be designed in two different ways to deal with an instruction that doesn't have the prediction ready at dispatch time. First, the

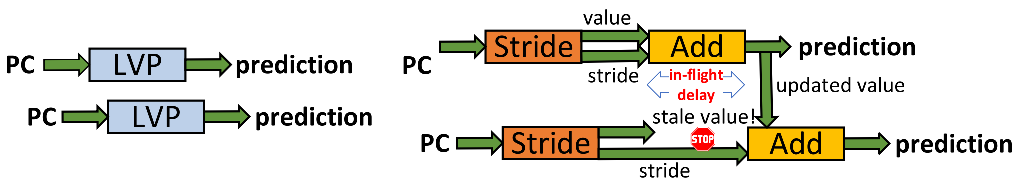


Fig. 6. Back-to-back prediction in an LVP and in a Stride predictor.

pipeline can be stalled waiting for the prediction (***delay dispatch*** policy). Second, the prediction can be discarded (there is no value speculation), not delaying the dispatch time (***not-delay dispatch*** policy). In this case, the predictor is still updated at commit time for training. Figure 5(b) shows the behavior of both policies when the prediction latency is too long.

5.3 In-flight Pending Update Policies

If two instances of the same instruction are executed very close in time, then the older instance of the instruction could read the predictor before the younger instance updates it. We define *in-flight pending update* as the predictor update that an in-flight instruction (a.k.a., an instruction being executed) will eventually perform at the back-end of the pipeline at commit time. We can manage in-flight pending updates of different instances of the same instruction with two different policies. The first policy ignores the in-flight occurrences of the same instruction, and it gets the prediction without waiting for the updates of the in-flight instructions, which can produce wrong value predictions (***in-flight ignore*** policy). The second policy stalls the pipeline, waiting for all the in-flight instructions to update the predictor (***in-flight wait*** policy). We define ***in-flight delay*** as the time that a back-to-back prediction is delayed waiting for the predictor update from the previous instance of the same instruction. The *in-flight wait* policy is a reasonable option for load value prediction, as the number of in-flight load instructions is small. Figure 5(c) shows the behavior of both policies when executing two instances of the same instruction back-to-back.

The in-flight pending update policies are not applicable to predictors that do not base their predictions on values generated by previous instances of the same instruction (i.e., they do not have *in-flight delay*). This is the case for LVP or VTAGE predictors but not for Stride or FCM preditors. Figure 6 shows an example of a back-to-back prediction with two different predictors. First, the LVP predictor, despite its name, does not need to wait for the previous prediction to correctly predict the current instance of the instruction as long as the predictor is trained (i.e., the *in-flight delay* is zero). Second, the Stride predictor needs to wait for the update from the previous instance of the same instruction to make a correct prediction. Otherwise, it would calculate the prediction based on a stale value. In this example, the mechanism updates the predictor using the *speculative update* policy from Section 5.1. The FCM predictor has an even longer *in-flight delay* [48]. In general, a predictor with a long *in-flight delay* could have performance problems with the *in-flight wait* policy (i.e., in tight loops that could execute several instances of the same instruction very close in time).

We can implement the *in-flight wait* policy by adding an extra bit in each predictor entry to indicate if there is another instance of the same instruction in the pipeline.

5.4 Putting It All Together

The previous three types of policies may have correlations between each other. We analyze these correlations in pairs of policy types.

First, the *predictor update policies* and *in-flight pending update policies*: the time at which the predictor update is done in the pipeline influences the duration of pending updates. For example,

Table 1. Area of Each Value Predictor as a Fraction of the Area of a 32KB 8-way L1 Data Cache ($0.33mm^2$)

LVP	2D-Stride	DFCM	DVTAGE	VTAGE	**AVPP-DVTAGE**	**AVPP-Stride**
1.7%	2.6%	3.6%	2.6%	3.6%	2.8%	2.8%

if the predictor update is done in the front-end (*speculative update* policy), the duration of the pending predictions is short. In this case, waiting for the prediction (*in-flight wait* policy) may provide the best performance, as it barely stalls the pipeline and increases the predictor coverage.

Second, the *predictor availability policies* and the *in-flight pending update policies*: whether or not to wait for the prediction at dispatch time influences the duration of pending updates. For example, if the pipeline stalls waiting for a prediction at dispatch time (delay dispatch policy), the pending update time also increases.

Third, the *predictor update policies* and the *predictor availability policies* are not correlated.

The best combination of policies and the best implementation depends on the target microarchitecture. Section 7.4 evaluates all possible combinations of the different policies from the three policy types.

6 REDUCING THE COMPLEXITY OF VALUE PREDICTION

Value prediction has not been implemented in real processors yet mainly due to its complexity issues. We have already discussed some recent works that focus on reducing complexity [46–49] in Section 1. We go one step further and analyze some problems and complexities of predicting values for all instructions, and we study and propose new low-cost organization alternatives and new policies to implement load value prediction.

6.1 Predictor Area Footprint

One of the advantages of speculating on load instructions is that the predictor can be smaller, as it has to handle only a fraction of the total instructions.

Table 1 shows the area footprint of the predictors we evaluate as the fraction of the area of a 32KB 8-way L1 data cache. We calculate the area with McPAT [33] configured with a Westmere OoO architecture with a frequency of 2.4GHz and 22nm technology. The total area of the core (including L1 and L2 caches) is $11.4mm^2$, whereas the L1 data cache is $0.33mm^2$. The predictors we use are configured in the same way as in the evaluation (Section 7).

We make two main observations from the table. First, the area of the load value predictors is very small compared to the data cache. Second, our predictors (AVPP-DVTAGE and AVPP-Stride) occupy a similar area as the state-of-the-art predictors.

6.2 Register Port Pressure

The physical register file (PRF) consumes an important portion of the energy in an OoO engine [71], and value prediction puts even more pressure on it. Figure 7 shows the reads and writes in the PRF during the instruction flow in the front-end and the back-end in-order pipeline stages. Figure 7(a) represents a processor that does *not* implement value prediction. The instruction reads the two input registers (requiring two read ports), and it writes to the output register at commit time (requiring one write port).

Figure 7(b) shows a classical scheme proposed in the state-of-the-art value prediction schemes [47, 48]. It requires reading two input registers and writing the prediction into the destination register before dispatch. The prediction is also written into a FIFO queue to be validated at commit time. After the OoO execution, the resulting data is written into the PRF (like in an

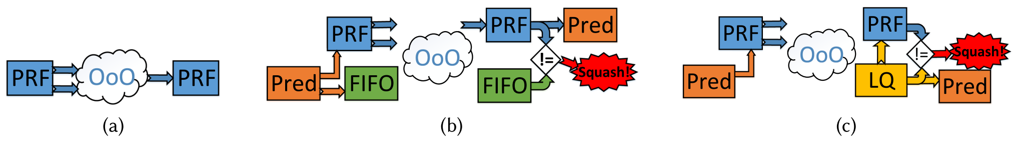


Fig. 7. PRF port pressure in (a) a processor without prediction, (b) a processor with value prediction, and (c) our proposal for load value prediction.

unmodified processor), and finally, the prediction is validated by comparing the real value in the PRF with the predicted value in the FIFO queue. This scheme requires one additional read port and one additional write port (two write ports in total), which increases the complexity of the PRF substantially. The extra write port causes the growth of the power consumption by 50% [48, 71] (although, this extra power consumption could be reduced by limiting the number of predictions per cycle, allocating physical registers in different register file banks for consecutive instructions [48], or using other optimization techniques [11, 18, 64]).

To alleviate the PRF port pressure, we propose a new specific organization for load value prediction. Figure 7(c) shows a brief summary of our approach. As in existing value prediction approaches, the two input registers are read and the output register is written before dispatch. We leverage the Load Queue (LQ), which is part of modern OoO processors, to keep the correct version of the data when received from memory, so our scheme does not need an additional FIFO queue. When the value prediction is reliable, the speculative value is written into the PRF, and the value coming from memory is written in the LQ. At commit time, the validation is done by comparing the values in the PRF and in the LQ. The LQ does *not* need any extra read/write ports, as the prediction validation and/or the PRF writing are done in the same pipeline stage, requiring just a single access to the LQ.

Compared to performing value prediction for all instructions, our approach reduces the port pressure in two main ways. First, when a prediction is correct, the value in the PRF is already the correct value at commit time, so the PRF does *not* need to be updated, reducing the write pressure. Implementing this optimization requires only a small modification to disable the writing of the value from the LQ to the PRF in case of a correct load value prediction. The probability of mispredicting is very low, as a simple confidence estimation mechanism can increase the accuracy to more than 99% (see Section 7). Second, the extra reads and writes in the PRF are performed only for load instructions, which are a fraction (≈25%) of all instructions. Notice also that all the optimizations for reducing the register port pressure in value prediction [11, 18, 64] are also applicable to our proposal.

Another approach to eliminating the FIFO queue is to carry the predicted value as part of the payload of the load instruction. The additional payload bits require additional FFs/bitcells between the dispatch and execution stages, but it is more desirable than adding read ports to the already expensive PRF (area, energy, cycle time). Alternatively, we can eliminate the FIFO queue by just checking (not updating) the prediction after the load execution (e.g., writeback) and setting a flag in the ROB for a load with a mispredicted value.

6.3 Thrashing and Bandwidth

Performing value prediction for all instructions can pollute the predictor because of the large number of lookups and updates. As a consequence, thrashing could be a problem if the prediction table is not *large* enough. Performing value prediction *only* for load instructions lowers pollution, because the reads and updates are significantly reduced. Therefore, we can implement smaller tables. The number of unique static load instructions is on the order of only a few thousand in our evaluated benchmarks.

Previous work [48] uses a predictor with 8k entries, and BeBoP [49] reduces this number to only 2k, because it uses a predictor that works with blocks of instructions, not with individual instructions. In our evaluation of load value prediction, we have good results with a predictor of only 512 entries, which is a 75% reduction compared with BeBoP [49], with good speedup results (see Section 7). The predictor misses are less than 10% in most of the benchmarks for a predictor of this small size.

Performing value prediction for all instructions requires potentially serving several prediction requests per cycle. Although the predictor has several cycles to provide the predicted value, if the processor issue width is too large, it could create a bottleneck in the predictor. A practical way to alleviate this problem is to use value prediction with processors that have a *moderate* issue-width. In this work, we demonstrate that predicting only loads has significant performance benefits in these relatively modest processors (4-issue processor in our case) at low hardware cost.

6.4 Microinstructions with the Same PC

In microarchitectures that implement ISAs with complex instructions, e.g., x86, each instruction is potentially decoded into several *microinstructions* (μops). Thus, several microinstructions could be associated with the same instruction address, and one unique entry in the predictor could correspond to several microinstructions (leading to aliasing). In our tests, the fraction of x86 instructions that are decoded into more than one μop is up to 35% in some cases, which is not negligible.

There are several ways to solve this problem when performing value prediction for all instructions. We describe two of them. First, by hashing the μop sequence number to the index used for indexing the predictor. This solution increases the design complexity. Second, by using a larger granularity than a single instruction. For example, BeBoP [49] uses a block-based value predictor, in which each entry is associated with a fetch block and not *with* individual instructions.

Fortunately, this is not a problem if we perform value prediction only for load instructions. Instructions that contain more than one load microinstruction are *rare* in the x86 ISA: Only the instruction for comparing string operands produce two load microinstructions (CMPSB, CMPSW, CMPSD, CMPSQ). The presence of these instructions in our benchmarks is negligible: only the $h264ref$ benchmark executes 1 per 5 billion instructions, and the *perlbench* benchmark executes one per 400.000 instructions.

6.5 Identification of Load Instructions

Load value prediction has the disadvantage that, unlike value prediction for all instructions, it needs to identify instructions that execute at least one load. To do so, the prediction is done when this information is available at the pre-decode stage, which can potentially delay the prediction by 1 cycle. We take this delay into account in our evaluations.

7 EVALUATION

In this section, we compare our AVPP load value predictor with five state-of-the-art load value predictors in single-core and multi-core systems, in terms of both performance and energy efficiency. We also compare AVPP with prefetching and memory dependence prediction techniques.

Our evaluation mainly considers load value prediction, as we have already shown that predicting all instructions provides incremental speedup benefits compared to load value prediction (Figure 1) while requiring much higher cost (Section 6).

7.1 Experimental Setup

Our evaluation uses a customized version of the ZSIM open-source simulator [51] with support for load value prediction. ZSIM resembles a four-issue Westmere microarchitecture that is validated

Table 2. Configuration of Our Baseline System

core	Westmere x86-64bit, 4-issue TAGE branch pred., 4 tag tables (32KB) 128-entry ROB, 32-entry load queue 32-entry store queue
L1I Cache	32KB 4-way, LRU, 3-cycle latency
L1D Cache	32KB 8-way, LRU, 4-cycle latency
L2 Cache	256KB, 16-way, LRU, 12-cycle latency 16 MSHRs
Prefetcher	L2 prefetcher, 16 stream buffers, 64-line buffers
Memory	DDR3-1333-CL10, 4 ranks per channel, 8 banks per rank
Policies	CNW: *Correct update*, *Not-delay dispatch* and *in-flight Wait*

Table 3. Predictors Considered in Our Evaluation. Our Proposed New Models are Bold-Faced

Type	Entries	Latency	*In-flight delay*
LVP	512	1 cycle	–
2D-Stride	512	2 cycles	1 cycle
DFCM	512	3 cycles	2 cycles
DVTAGE	512	3 cycles	1 cycle
VTAGE	512	2 cycles	–
AVPP-DVTAGE	512+64	4 cycles	1 cycle
AVPP-Stride	512+64	3 cycles	1 cycle

against a real Westmere machine, reporting an IPC average error of 9.2%/11.2% for single/multi-thread applications [51]. Table 2 shows our baseline configuration, which includes a 32KB 8-way data cache with 4-cycle latency, a 32KB 4-way instruction cache with 3-cycle latency, a 16-way L2 cache with 12-cycle latency and a Stream prefetcher. The baseline load value prediction policies are *correct update*, *not-delay dispatch* and *in-flight wait.*

Our ZSIM Westmere model implements load-store ordering, load forwarding, fences, and TSO. Each load is implemented as an address computation μop and a memory read μop, and it does not issue until all prior store addresses have been resolved. the processor can fetch 16B/cycle (the limit is the predecoder), and up to 4 μops/cycle. The processor is composed of 6 execution ports, 19 integer functional subunits, eight floating-point subunits and 36 reservation station entries. Branch mispredictions are resolved at commit time. They cancel all in-flight data misses that are in the wrong path. Branch misprediction penalties are simulated according to a measured misprediction penalty of at least 17 clock cycles in the real Westmere architecture. The same mechanism is used in our implementation for recovering from value mispredictions, which leads to a misprediction penalty that is in the same order as the branch misprediction penalty. Long latency loads can increase this penalty (up to 45 cycles in our tests), but for most of the benchmarks we tested, the average value misprediction penalty is similar to the branch misprediction penalty.

We run a subset of the SPEC2006 and SPEC2000 benchmark suites with their reference inputs, and a subset of benchmarks from PARSEC [7] and SPLASH2 [67] with their native inputs. We select a diverse set of workloads, considering both good and bad candidates for load value prediction. We use the Pinpoint [50] methodology to find representative simulation points, and Pinplay [45] to log these regions and deterministically replay them. We generate the single-core PARSEC and SPLASH Pinpoints by running the application with a single thread. Our multi-core evaluation (Section 7.11) is performed with 10 billion instructions of the region of interest, without using Pinpoints.

We evaluate the behavior of different value predictors for load value speculation. Table 3 shows the number of entries, the latency (in cycles), and the *in-flight delay* (see Section 5.3) of the predictors evaluated, namely LVP, 2D-Stride predictor, DFCM, DVTAGE and VTAGE, as a diverse set of the best state-of-the-art predictors, and the AVPP-DVTAGE and AVPP-Stride as our new contributions (Section 4). The AVPP predictors are named depending on the predictor used in the AT. We choose candidates with and without *in-flight delay* to show the influence of the in-flight pending update policies (Section 5.3) in the performance. Latencies are calculated according to Section 3 and Section 4.

All predictors use 3-bit FPC counters [48] for prediction confidence estimation, with the vector $\{1, \frac{1}{16}, \frac{1}{16}, \frac{1}{16}, \frac{1}{16}, \frac{1}{32}, \frac{1}{32}, \frac{1}{32}\}$ (that resembles a 7-bit counter). When the prediction is correct, the counter is increased with the probability defined by the vector. The counter is reset on a misprediction, and the prediction is used only when the counter is saturated. We validate predictions at commit time, and we implement pipeline flushing for recovering from mispredictions.

We set up all predictors with 512 entries as our baseline system for maintaining a good trade-off between area and performance. In the AVPP predictors, AT has 512 entries and VT has only 64 entries (Section 7.5 justifies this decision). Each entry in AT has a 3-bit *pdis*, and 1-bit *d* (Section 4). All tagged tables of the predictors are implemented as a direct-mapped cache.

We implement a four order DFCM in our evaluation. In the VHT, each new stride is encoded into 16 bits with a hash function based on XORs [54], and the VPT is indexed by this compressed context (64 bits total). Furthermore, we add a counter to each VPT entry to limit replacements due to interference [48]. The counter is increased if the value matches the one already stored, and it is decreased otherwise. An entry is replaced only when the counter is zero.

The implemented DVTAGE predictor [49], in addition to the base predictor and the last value table (512 entries each), is composed of 6 tagged tables (64 entries each). It uses a *useful* bit in the tagged tables, and tags of 12+*rank* bits, with *rank* going from 1 to 6. The minimum history length is 2, and the maximum is 64. The VTAGE predictor has the same configuration, but without the last value table, and saving values (64 bits) instead of strides.

All the predictors based on strides use only 16 bits to store the stride. We found out that strides are usually small, and the impact of using only 16 bits instead of 64 bits is negligible in terms of performance. Furthermore, AVPP predictors have 32-bit fields in the AT for storing the addresses.

We also implement a Hybrid predictor composed of two predictors: (1) AVPP-DVTAGE predictor and (2) a 2D-Stride predictor. For choosing between predictors, we add an FPC confidence counter per predictor, that indicates the reliability of each predictor. The prediction is only used in case the counter is saturated, and in case the counters for both predictor are saturated, we choose the AVPP predictor by default.

We heavily modified the ZSIM simulator to support load value prediction. We took into account the latencies in Table 3, for reading and updating the predictor, as well as the appropriate order and timing of the lookups and updates (depending on the policies), and the critical paths. Each predictor has one read port and one write port to reduce complexity. We simulate port contention. Additionally, we include a 1-cycle delay to identify loads in our simulator (see Section 6.5).

All the predictors are pipelined, implying that they can accept a new lookup/update request each cycle. Predictor updates have the same latencies as the lookups in almost all cases (DFCM has one extra cycle). We take into account all predictor latencies and contention in our simulations.

From the described policies in Section 5, we choose for our baseline the CNW (*Correct update*, *Not-delay dispatch*, and *in-flight Wait*) policy as it is one of the best performing in our tests (see Section 7.4).

Energy results are obtained with a customized version of McPAT [33] to take into account different value predictors, additional operations in the register file and the load queue, value mispredictions, and extra prefetch memory requests.

7.2 Results in a Single-core Processor

To show the benefits of AVPP, we simulate the system described in Table 2 implementing the five value predictors described in Table 3, running the benchmarks in Table 4.

Figure 8 shows the speedup of a processor with eight different load value predictors compared to the baseline (no value prediction). We make three main observations. First, AVPP outperforms all state-of-the-art predictors in 16 of the 23 benchmarks. The average speedup of AVPP is 11.2%

Table 4. Single-core Benchmarks

SPEC2000	art, gzip, lucas, sixtrack, equake
SPEC2006	astar, bzip2, gcc, libquantum, mcf, omnetpp, perlbench, bwaves, dealII, leslie3D, namd, soplex, tonto, wrf, zeusmp
PARSEC/SPLASH2 (1 thread)	bodytrack, raytrace, water_spatial

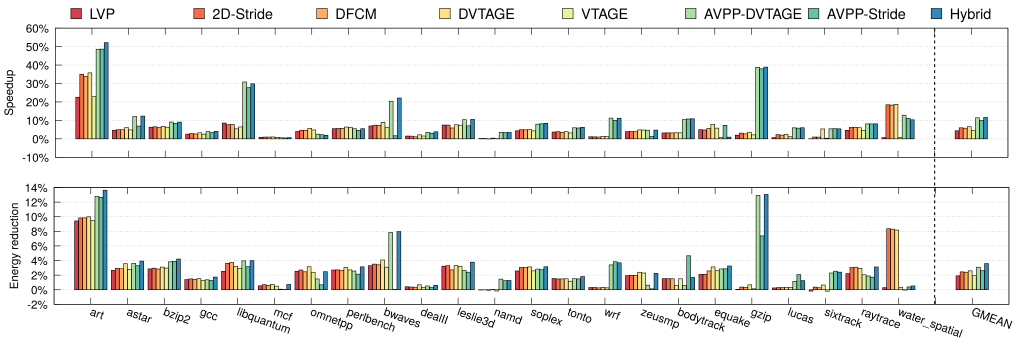


Fig. 8. Speedup and energy comparison of different value predictors.

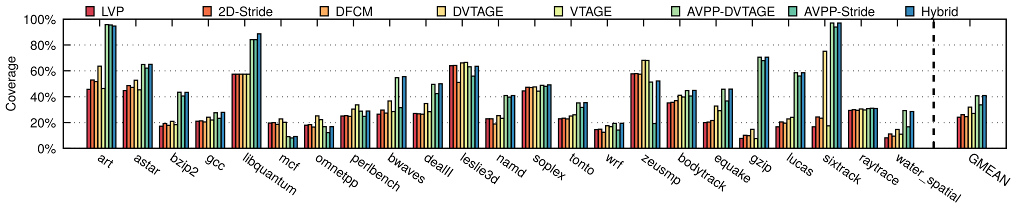


Fig. 9. Coverage of the evaluated predictors.

versus 5.9% of the best state-of-the-art predictor. Second, the address predictability leveraged by AVPP is reflected in the performance improvements of the benchmarks *art*, *libquantum*, *bwaves*, and *gzip*, where none of the state-of-the-art predictors achieve performance similar to AVPP. Third, the Hybrid predictor does not have much better performance than AVPP alone, as AVPP alone covers most of the accurate predictions made by the 2D-Stride predictor. We conclude that (1) AVPP is very effective and (2) a Hybrid predictor is not worth the hardware cost compared to AVPP alone.

Figure 8 (bottom graph) shows the energy reduction provided by the different value predictors. The main observation is that almost all the predictors provide energy savings and that these savings are directly correlated with the speedup the predictor provides. AVPP's energy savings is 3.7% versus 2.6% of the best previous predictor.

To better show the performance of the load value predictors, we define coverage and accuracy: (1) coverage of a predictor is the percentage of loads to which a prediction is used for speculation, and (2) accuracy is the percentage of predictions used for speculation that are correct. Figure 9 shows the coverage, and Figure 10 shows the accuracy of the evaluated load value predictors. The main observation from these figures is that our AVPP predictor achieves better coverage than with other predictors in most of the benchmarks while maintaining a very high level of accuracy. On average, our best AVPP predictor (AVPP-DVTAGE) has better coverage (44%) than DVTAGE,

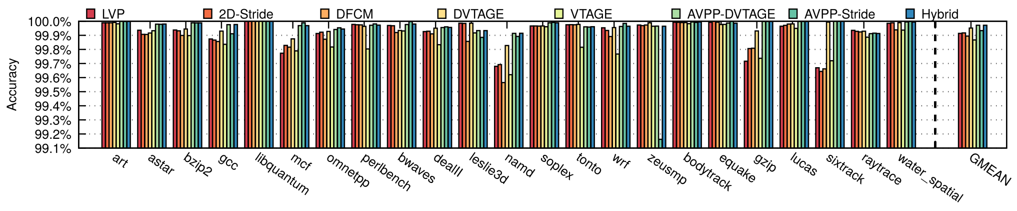


Fig. 10. Accuracy of the evaluated predictors.

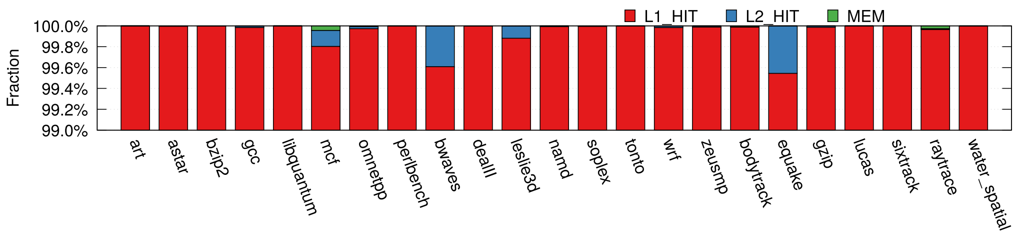


Fig. 11. Cache hit distribution among the predicted loads (the ones covered in Figure 9) for the AVPP-DVTAGE predictor.

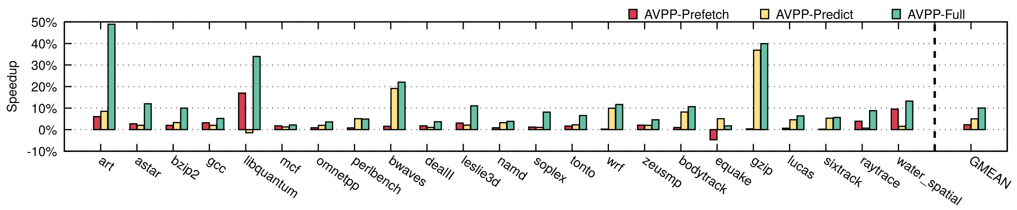


Fig. 12. Speedup of AVPP-DVTAGE when only the prefetch is active (AVPP-Prefetch), when only the prediction is active (AVPP-Predict), and when both are active (AVPP-Full).

the best state-of-the-art predictor (36%). Because of the use of FPC confidence counters, AVPP's accuracy is very high, which minimizes the total squash penalties.

Figure 11 shows the cache hit distribution of the predicted (i.e., covered) load instructions for the AVPP predictor. The results are similar for the other predictors. The main observation is that most of the predicted loads are L1 cache hits. This observation suggests that, for the tested workloads and system configurations, load value prediction should be complementary to prefetching. We will demonstrate this in Section 7.9 and Section 7.10.

7.3 Where Are the Benefits Coming From?

We analyze whether the AVPP benefits are coming from AVPP prediction or AVPP prefetching (Section 4). Figure 12 shows the performance of our AVPP only with prefetching (AVPP-Prefetch), only with load value prediction (AVPP-Predict) and the complete AVPP with both prefetching and load value prediction (AVPP-Full).

We make three main observations. First, the benefit of AVPP prefetching is low in general (1.8% speedup on average). This is because in many cases the prefetch requests are L1 hits, as we show in Figure 13. Second, AVPP value prediction has better performance than value prefetching (4.8% speedup on average). Third, when the complete AVPP predictor (with both prediction and prefetching) is used, the performance improvement is significant (10.6% speedup in average). The main reason for the large speedup jump when full AVPP is used is that the prefetches in VT highly

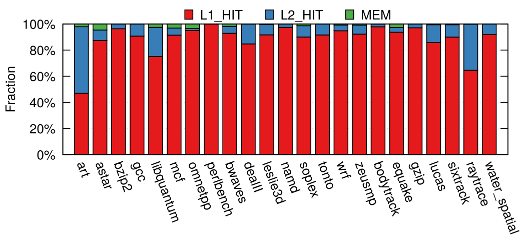


Fig. 13. Distribution of the AVPP-DVTAGE prefetch requests that hit in L1, L2 and main memory.

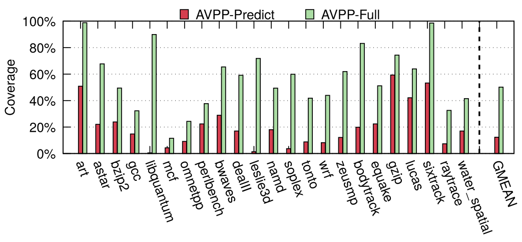


Fig. 14. Coverage of AVPP without prefetching (AVPP-Predict), and the full AVPP including prefetching (AVPP-Full).

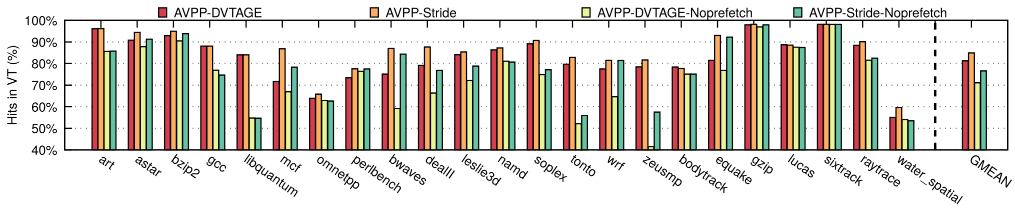


Fig. 15. VT hit rate with and without prefetching.

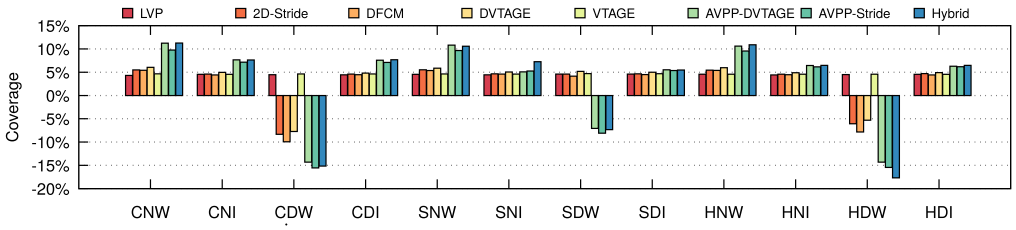


Fig. 16. Impact of the implemented policies (Section 5.3).

increase its hit rate, and consequently, the prefetching coverage also increases. Figure 14 shows the coverage of the full AVPP load value predictor (AVPP-Full) and the AVPP predictor without prefetching (AVPP-Predict). The average coverage of AVPP without prefetching is 12.3% (99.7% accuracy), whereas in the full AVPP is 50.2% (99.9% accuracy).

Figure 15 shows the hit rate of the VT table in our AVPP predictor with prefetching (AVPP-DVTAGE and AVPP-Stride) and without prefetching (AVPP-DVTAGE-Noprefetch and AVPP-Stride-Noprefetch). The prefetching schemes use dynamic distance with a 5% *probUp*, and non prefetching implementations update the VT with the value of the current instruction. The main observation is that dynamic prefetching improves the VT hit ratio in almost all benchmarks.

7.4 Impact of the Value Prediction Policies

Figure 16 shows the average speedup of the combination of the policies described in Section 5. The first letter indicates *Correct update* (C), *Speculative update* (S), or *Hybrid update* (H) (Section 5.1). The second letter indicates *Delay dispatch* (D) or *Not-delay dispatch* (N) (Section 5.2). The third letter indicates *in-flight Ignore* (I) or *in-flight Wait* (W) (Section 5.3).

We make four main observations. First, the combination of the *Delay dispatch* (D) and *in-flight Wait* (W) policies generally have poor performance, because they create a lot of contention together. With these policies, the predictor is on the critical path, and it can generate stalls in the

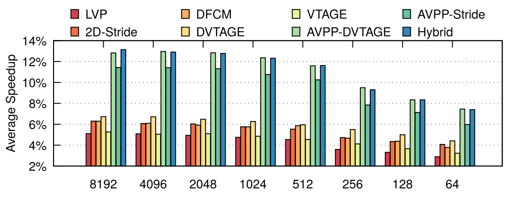


Fig. 17. Impact of predictor size on speedup. For this study, the AT and the VT have the same size in the AVPP predictors, to emphasize the effect of AT size.

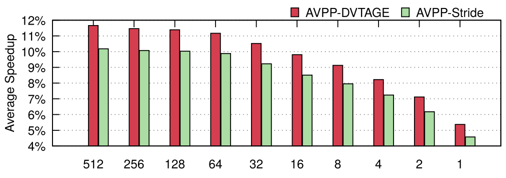


Fig. 18. Impact of VT table size for an AVPP predictor with an AT of 512 entries.

pipeline. Second, *Correct update* (C) and *Hybrid update* (H) policies have similar behavior, which indicates that the hybrid policy provides almost no benefit. Third, *in-flight Wait* (W) has better results than *in-flight Ignore* (I) if it is *not* combined with *Delay dispatch* (D). This shows the benefits of making the prediction using the updated value from the previous in-flight instruction, provided that waiting for the updated value does not delay the dispatch time. Fourth, LVP and VTAGE predictors, which do not have *in-flight delay* (Section 5.3), always provide performance improvements compared to the baseline, independently of the policy.

These observations are valid for the tested microarchitecture, and they can not be easily generalized or formalized. The study of these policies in microarchitectures with different specifications and design choices is out of the scope of this work.

7.5 Impact of the Predictor Size

Figure 17 shows the impact of the predictor size on system performance. Unlike our baseline system, in this experiment we over-sized the VT (i.e., with the same size as AT) to emphasize the impact of the AT size in the AVPP predictors. We make three main observations. First, a large predictor table does not provide a significant performance improvement. Increasing the number of entries from 512 to 8,192 does *not* affect performance significantly. Second, the most significant performance jump is produced when we increase the size of the predictor from 256 to 512 entries. This is the reason why we establish 512 entries as our baseline.

Figure 18 shows the impact of the VT size of the AVPP predictor. The figure shows the speedup of the AVPP-DVTAGE and the AVPP-Stride predictors when changing the VT size from 512 to 1. In this experiment the AT has a fixed size of 512 entries. The main observation is that the performance benefits of having 512 entries in the VT are very close to those of having 64 entries, which is the reason why we choose 64 as the VT size for our baseline.

7.6 Impact of the Prefetch Dynamic Distance Mechanism

AVPP calculates the prefetch distance dynamically by observing the VT misses (Section 4.2). Figure 19 shows the impact of the probability of updating the prefetch distance (*probUp*). We make two observations. First, different values of *probUp* can significantly change performance for specific applications in AVPP-DVTAGE (e.g., moving from *probUp=1%* to *probUp=5%*, improves the speedup by 8% for *art* and by 6% for *water_spatial*). In AVPP-Stride we do not observe large differences. Second, the AVPP-DVTAGE predictor achieves the best average speedup with *probUp=5%*, and the AVPP-Stride predictor achieves the best average speedup with *probUp=1%*. We chose *probUp=5%* for our baseline as AVPP-Stride performs better with *probUp=5%* than AVPP-DVTAGE performs with *probUp=1%*.

Figure 20 shows the predictor distance (*pdis*) distribution when *probUp* is 5%. We make two observations. First, as the common case is hitting in L1 (low latency), most of the prefetch

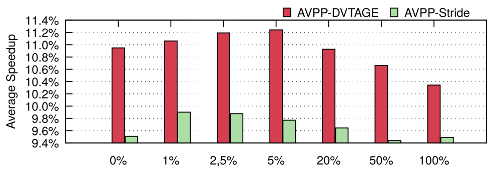


Fig. 19. Effect of the probability of updating the prefetch distance (*probUp*) on AVPP speedup.

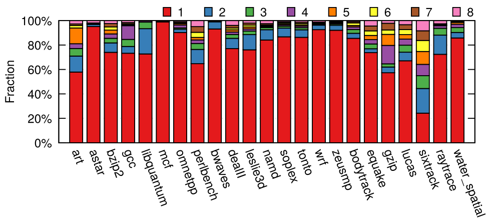


Fig. 20. AVPP prefetch distance distribution. The common case is distance=1 (i.e., to prefetch the next address), but other distances are also significant in some benchmarks.

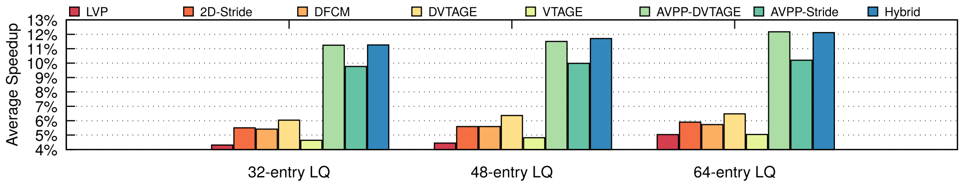


Fig. 21. Impact of the LQ size on the predictor performance. The load queue is the same size in the baseline system and in the value prediction system.

distances are one (next value). Second, some benchmarks (e.g., *art*) have a significant number of distances larger than one, which contributes to significant performance improvements. In the case of *sixtrack*, although most of the prefetches have a distance greater than one, we do not see large performance benefits as *sixtrack* does *not* exhibit many data dependencies and consequently, it cannot benefit much from value prediction.

7.7 Impact of the Load Queue Size

We quantify the impact of the load queue size on the performance of a system implementing AVPP, with the goal of measuring the pollution caused by the additional value prefetch requests in the load queue. Figure 21 shows the speedup provided by several load value prediction mechanisms compared to our baseline. Both baseline and load value prediction based systems have the same load queue size. Our main observation is that the size of the load queue does *not* have much influence on the performance of our AVPP predictors. We conclude that value prefetching of AVPP does not add much pressure to the load queue.

7.8 Sensitivity to the Cache Hierarchy

We study the sensitivity of load value prediction to the cache hierarchy. Table 5 shows the different configurations we evaluate, including 1) our baseline described in Section 7.1 (Baseline), 2) our baseline augmented with a 2MB L3 cache (Baseline+L3), and 3) a larger system with double-size L1, L2, L3 caches (Baseline+L3 Large). Figure 22 shows the speedup results (plotted with color bars), as well as the Instructions per Cycle (IPC, plotted with hoops) for the three configurations.

We make two observations. First, as expected, IPC increases when the cache size is larger. Second, the speedup of value prediction does *not* depend much on the cache configuration, because most of the value predictions are for load instructions that hit in L1 (see Section 7.2). We conclude that neither AVPP nor other load value predictors are largely influenced by the cache hierarchy configuration, which suggests that load value prediction could be a low-cost performance accelerator for processors with various cache sizes.

Table 5. Configurations for the Cache Sensitivity Analysis

Baseline	L1D&I: 32KB, L2: 256KB
Baseline+L3	L1D&I: 32KB, L2: 256KB, L3: 2MB
Baseline+L3 large	L1D&I: 64KB, L2: 512KB, L3: 4MB

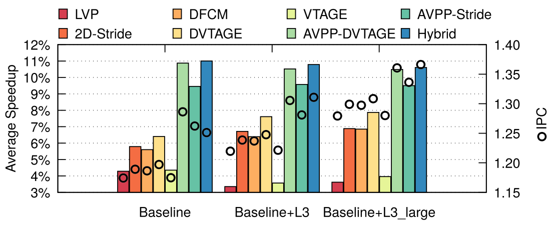


Fig. 22. Speedup of load value prediction with the three cache configurations shown in Table 5.

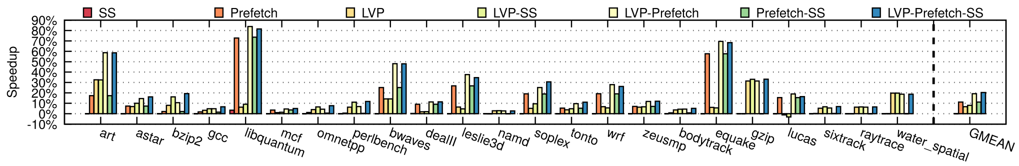


Fig. 23. Interaction between store set prediction (SS), L2 prefetching (Prefetch), and Load Value Prediction (LVP).

7.9 Comparison with Prefetching and Memory Dependency Prediction

There are other techniques than load value prediction to hide load latencies and to break true data dependencies. In this section, we analyze two of them, data prefetching and memory dependence prediction, which are already present in most commodity processors, and we study their interactions with load value prediction.

Memory dependence prediction allows dispatching loads before the previous store addresses have been resolved, in case the prediction is that they are accessing different addresses. If the speculation was wrong, then the pipeline has to be flushed, similarly to value prediction or branch prediction.

Store Set Prediction [14] is an easy way to implement memory dependence prediction. The scheme identifies the set of stores on which a load depends (the store set) and communicates that information to the instruction scheduler. We implement this technique with a Store Set ID table (SSIT) (indexed by the instruction PC), which maintains the store sets identifier (SSID), and with a Last Fetched Store Table (LFST) (indexed by the SSID), that maintains dynamic information about the most recent store in the store set. In our evaluation, we use store sets with an SSIT of 4096 entries, and an LFST of 128 entries.

Our baseline is the same as in Table 2 but without any prefetcher. Figure 23 shows the speedup of the system with the store set predictor (SS), the L2 prefetcher (Prefetch), load value prediction with an AVPP-DVTAGE predictor (LVP), and the combination of all of them.

We make three main observations. First, the performance improvement of SS over our baseline is negligible (the bars are not visible in almost all the benchmarks). We even tested a perfect memory dependence predictor (an oracle) and find similar negligible performance improvements. The cause is the relatively narrow 4-issue processor, which does not experience that many loads to wait for store resolution. Furthermore, in the evaluated processor, store address calculation and the store data are two different microinstructions, which reduces the load-store scheduling problem even more. Second, the L2 prefetcher and our AVPP load value predictor achieve the best speedups (11% and 8% on average). Third, L2 prefetcher and our AVPP are complementary and, if implemented together, their speedups are additive (20% average speedup). We conclude that AVPP

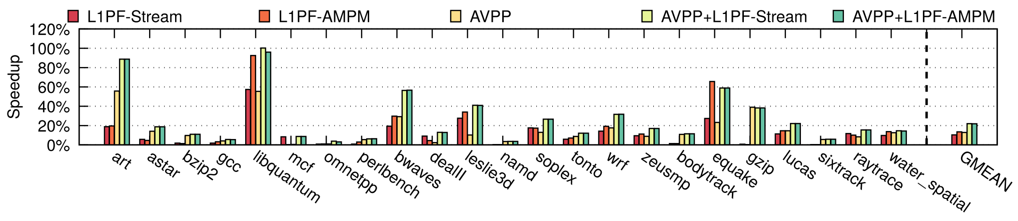


Fig. 24. Performance effect of L1 prefetching vs. AVPP.

is complementary to prefetching and memory dependency prediction. The three mechanisms can be implemented together in the same system with additive performance gains.

7.10 Comparison with L1 Prefetching

AVPP does prefetching into the VT to increase the predictor coverage and accuracy, but also brings the data to L1. In this section, we compare our AVPP predictor with two state-of-the-art L1 prefetching approaches, with the goal of showing the relationship between both. We implemented an L1 Stream prefetcher (similar to the one described in Table 2) and the Access Map Pattern Matching (AMPM) prefetcher [28]. Both prefetchers request their prefetches through the load queue.

To analyze the effects of L1 prefetching, we disable L2 prefetching. Other than that, we maintain the configuration of the baseline (Table 2). Figure 24 compares the two L1 prefetchers (L1PF-Stream and L1PF-AMPM) with the AVPP alone (AVPP) and with the AVPP working with the two L1 prefetchers (AVPP+L1PF-Stream and AVPP+L1PF-AMPM). In this section, we consider the full AVPP (with both AVPP prediction and AVPP prefetching).

We make three main observations. First, performance improvements provided by the L1 prefetchers (especially AMPM) are significant, as expected (because there is no L2 prefetching). Second, AVPP provides approximately the same performance benefits as the L1 prefetchers. Third, when AVPP and the L1 prefetchers are used together, the performance is additive in many benchmarks. The main reason is that most of the benefits are coming from value prediction (e.g., breaking true data dependencies), not from prefetching. We conclude that AVPP and L1 prefetching can effectively coexist in the same system. They are complementary and provide additive performance benefits.

7.11 Evaluation in a Muti-core Processor

In this section, we evaluate our AVPP predictor in a multi-core processor. The baseline is a 4-core processor with 32KB L1D&I private caches, 256KB private L2 caches and 2MB (per core) shared L3 cache (Table 6). We evaluate two different types of workloads (Table 6). First, we evaluate 6 multithreaded applications from PARSEC and SPLASH benchmarks. Second, we evaluate 50 randomly assembled workloads, each comprising 4 benchmarks from Table 4 (Table 6 shows 6 of these workload mixes). We run all the benchmarks for 10 billion instructions in the region of interest.

7.11.1 Multithreaded. Figure 25 shows the speedup of a system with load value prediction running multithreaded applications in a multi-core system. Two observations are in order. First, the AVPP predictors perform better than other predictors in five of six benchmarks. Second, in the two benchmarks which we also use on single-core evaluation with only one thread (*bodytrack* and *raytrace*), the speedups we observe here are on the same order of those we observe for the single-core system.

Table 6. Configuration of the Multi-core Baseline Processor, Multithreaded Benchmarks, and a Subset of Multiprogrammed Benchmarks with Detailed Results

Processor Configuration		Multithreaded Benchmarks		Example Multiprogrammed Benchmarks
Cores:	4 cores		bodytrack,	MIX1: art, art, libquantum, gzip
	from Table 2	PARSEC Bench.:	freqmine,	MIX2: leslie3d, soplex, libquantum, dealII
			streamcluster	MIX3: mcf, gzip, art, bodytrack
	L1D&I: 32KB		barnes, ocean_cp,	MIX4: wrf, bodytrack, libquantum, bodytrack
Caches:	L2: 256KB,	SPLASH2 Bench.:	raytrace	MIX5: water_spatial, libquantum, zeusmp, dealII
	L3: 8MB		water_spatial	MIX6: raytrace, gzip, art, wrf

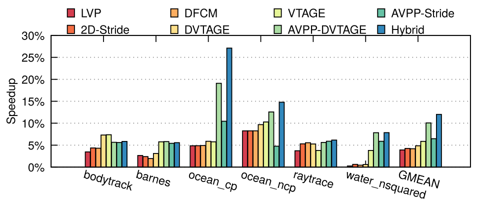


Fig. 25. Speedup of 6 multithreaded workloads.

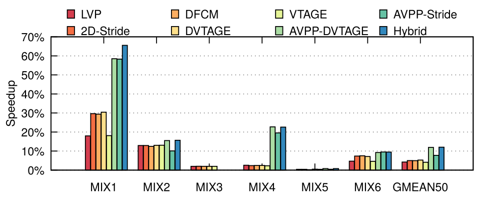


Fig. 26. Speedup of 50 multiprogram mixes of workloads.

As pointed out in Section 4.3, VT is not coherent with memory, which could cause additional mispredictions. In our experiments, we do not observe any misprediction due to this cause because of three main reasons. First, the percentage of invalidations in L1 caches caused by other cores is very small (the number of invalidation requests is less than 0.002% of the read requests). Second, the VT is much smaller than the L1 cache, which causes its entries to be evicted and refreshed with updated values much more frequently than L1, reducing the probability of having stale values. Third, AVPP does *not* speculate until the prediction is confident, which requires 128 consecutive correct predictions in our setup. In regions of code where invalidations are common, the prediction is not confident and consequently, the probability of misprediction is very low. We conclude that AVPP outperforms all previous predictors in most of the multithreaded benchmarks without requiring any coherence mechanisms in the VT.

7.11.2 Multiprogrammed. Figure 26 shows the weighted speedup [17] of a system with load value prediction running different combinations of single-threaded programs. The figure shows the speedup on 6 mixes out of 50, and the geometric mean speedup across all 50 mixes. We make two observations. First, the AVPP predictor outperforms the previous predictors in 40 of the mixes. Second, the geometric mean speedup across the 50 mixes has similar gains we do observed for our single-core evaluation (Section 7.2). We conclude that AVPP outperforms all previous predictors in most of the multiprogrammed workload mixes.

8 RELATED WORK

In this work, we propose a novel load value prediction scheme that (1) is based on address prediction and (2) uses prefetching to increase the prediction coverage and accuracy. AVPP is complementary to conventional prefetching (Section 7.3). We have already discussed and evaluated other state-of-the-art value prediction mechanisms [15, 16, 22, 34, 35, 37, 46–49, 54].

Mechanisms that have Similarities with AVPP. AVPP has a similar philosophy with store instructions as the EXACT [2] branch predictor. EXACT updates the branch predictor information when a store is performed to an address which dynamic branch prediction depends on, in a similar way than AVPP updates the VT when a store is performed. Otherwise, AVPP and EXACT are totally different mechanisms.

APDP [23] is a stride value predictor for load and store instructions that prefetches the next load address. It consists of a single 2,048-entry Memory Prefetching Table (MPT), each MPT entry composed by several fields including the last effective address, the current stride, and the prefetched value. It also implements a 1,024-entry Prefetching Validation Vable (PVT) that maintains the state of the prefetched values in the MPT. When a store instruction overwrites a load address present in the MTV, APDP invalidates such entry. APDP is *not* designed for being a simple mechanism, and as a consequence it has a high implementation cost. APDP has four main differences with our AVPP. First, AVPP is *simple* and *easy to integrate* into current microarchitectures: it does not require selective re-execution on mispredictions and it leverages existing microarchitectural structures (e.g., load queue) instead of incorporating new hardware. Second, AVPP predicts *only* load instructions to improve the performance/cost ratio. Third, AVPP *decouples* the address and the value tables for enabling a smaller and simpler predictor, that it is also more accurate. Fourth, AVPP uses a prefetching mechanism that is *dynamic* and more accurate.

Decoupled Load Value Prediction (DLVP) [58] is a context-based load value predictor that predicts the load address in the fetch stage, requests this address from memory, and stores the prefetched value into a non-speculative value table (that resembles the AVPP value table) before dispatch time. DLVP has at least three major differences from AVPP. First, DLVP prefetches the value of the predicted address for the current load instruction. As DLVP relies on accessing the L1 cache, the prediction fails when the predicted address misses in L1, or when the memory system is busy serving a demand memory request, which limits the coverage of the predictor. AVPP uses a dynamic algorithm that predicts the address of the current and future instances of the load instruction with the goal of increasing coverage and hiding the memory latency. Second, DLVP uses a prediction mechanism that relies on issuing a memory request to the L1 cache (placed far away from the front-end) and retrieve the value back to the value table before the current load instruction is issued. The authors assume very low latencies that could not be implementable in other architectures (e.g., a latency of 2 cycles to access L1 data cache), like the one used in AVPP, limiting the generality of the approach. Third, DLVP is a context-based predictor that has limited performance when predicting back to back, in a similar way to other context-based predictors, such as FCM (see Section 5.3). The article does not discuss this issue, which could limit the performance of some workloads. AVPP is a high-performance predictor that takes into account this and other issues (Section 7.4).

Prefetching Mechanisms. AVPP, although it gets most of its benefits from value prediction (Section 7.3), it also performs prefetching into the VT to increase coverage and accuracy of value prediction.

Prefetching is one of the most effective techniques to reduce the effective load access time [5, 13]. Some proposals focus on LLC misses [12, 32, 59], as they are a major source of pipeline stalls. Other works prefetch into the L1 cache [31, 42, 43].

Runahead [42, 43] is a prefetching mechanism that speculatively executes independent instructions when the instruction window is full, with the goal of generating cache misses. Runahead can achieve very accurate predictions, as it uses branch prediction information. More advanced versions of runahead achieve better coverage [25, 26].

B-Fetch [31] is a prefetching mechanism driven by branch prediction and address prediction. B-Fetch operates in two steps. First, B-Fetch predicts the future path of execution with a mechanism based on branch prediction. Second, B-Fetch predicts and prefetches the effective address of the load instructions along the previously predicted path. B-Fetch uses the register content at earlier branch instructions to predict the effective address.

Zhou et.al. [70] propose a recovery-free value prediction mechanism that aims to increase memory level parallelism. The mechanism speculatively breaks true data dependencies of dependent load instructions with the sole purpose of prefetching data into L1.

Approximate Load Value Prediction. In applications that are resilient to errors, load value prediction can be relaxed to improve performance and efficiency at the cost of potentially lower prediction accuracy [38, 61, 68, 69]. RFVP [61, 68] predicts the requested value of the load instructions that are safe-to-approximate, and it never checks or recovers from mispredictions, thereby avoiding pipeline flushes. The programmer, however, has to annotate the code to identify which loads are safe-to-approximate.

Other Related Techniques. In addition to the previous alternatives to speculate on load instructions, dependence prediction [21, 39, 56] and memory renaming [40, 65] have been proposed. However, it has been shown that load value prediction is, by far, the most effective of them all [10].

Address prediction is another technique that resembles AVPP. This technique is used for speculating on the address of a load during the early stages of the pipeline, with the goal of hiding the load latency [6]. Unlike AVPP, address prediction does not break true data dependencies and has limited performance gains.

9 CONCLUSION

Value prediction can significantly improve instruction level parallelism at the cost of introducing additional hardware. Despite recent advances, complexity is still a barrier that prevents value prediction to be widely adopted. The goal of this work is to go a step further and reduce complexity and hardware cost while maintaining most of the performance benefits of value prediction. To this end, we revisit load value prediction (i.e., predicting the values of *only* load instructions) as an efficient alternative to predicting the values of all instructions.

We propose a new, low-cost load value predictor that leverages address predictability to achieve high coverage and accuracy: the *Address-first Value-next Predictor with Value Prefetching (AVPP)*. The key idea of AVPP is to predict the load address first, which is used to index a small non-speculative Value Table (VT) to get the predicted value next. To increase the predictor coverage, AVPP also predicts and prefetches the load address of a *future instance* of the same load instruction into the VT, which increases the VT hit rate of future load instructions. Our extensive evaluation shows that a system with AVPP is, on average, 11.2% faster and 3.7% more energy efficient than the baseline (outperforming all the previous state-of-the-art predictors in 16 of the 23 benchmarks we evaluate), and its area is only 2.5% of the area of a 32KB L1 data cache. We show that both AVPP and L1/L2 prefetchers implemented together achieve additive performance improvements.

We propose, analyze, and evaluate a taxonomy of value prediction policies based on different design choices that can be made to implement value prediction. These policies were *not* formally classified, compared or evaluated by previous works. This taxonomy will help to better define, understand and reproduce future value prediction works.

We propose microarchitectural optimizations that make load value prediction a better approach than value prediction for all instructions in terms of complexity: We reduce the register port pressure and we leverage existing hardware instead of introducing specialized elements for value prediction. Also, we reduce the size of the predictor by 90% compared to the state-of-the-art approaches that predict all instructions.

We conclude that predicting only load instructions enables the implementation of AVPP, a simple, low-cost and high-performance value predictor that requires minimal modifications to the architecture of existing processors. AVPP addresses the key challenges against enabling wide adoption of value prediction in current processors.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable suggestions. We thank the SAFARI Group Members for feedback and the stimulating research environment that they provide.

REFERENCES

[1] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. 1999. DBMSs on a modern processor: Where does time go? In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB'99)*. 266–277.

[2] Muawya Al-Otoom, Elliott Forbes, and Eric Rotenberg. 2010. EXACT: Explicit dynamic-branch prediction with active updates. In *Proceedings of the 7th ACM International Conference on Computing Frontiers (CF'10)*. 165–176.

[3] Gene M. Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the Spring Joint Computer Conference (AFIPS'67 (Spring))*. 483–485.

[4] ARM. 2016. ARM Cortex-A72 MPCore Processor Technical Reference Manual. Retrieved from http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.100095_0003_06_en/index.html.

[5] Todd M Austin and Gurindar S Sohi. 1995. Zero-cycle loads: Microarchitecture support for reducing load latency. In *Proceedings of the 28th Annual International Symposium on Microarchitecture (MICRO'95)*. 82–92.

[6] Michael Bekerman, Stephan Jourdan, Ronny Ronen, Gilad Kirshenboim, Lihu Rappoport, Adi Yoaz, and Uri Weiser. 1999. Correlated load-address predictors. In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA'99)*. 54–63.

[7] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*. 72–81.

[8] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2010. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. 1–16.

[9] Martin Burtscher and Benjamin G. Zorn. 1999. Exploring last N value prediction. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*. 66–76.

[10] Brad Calder and Glenn Reinman. 2000. A comparative survey of load speculation architectures. *J. Instruct.-Level Parallel.* 2 (2000), 1–39.

[11] Brad Calder, Glenn Reinman, and Dean M. Tullsen. 1999. Selective value prediction. In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA'99)*. 64–74.

[12] Luis Ceze, Karin Strauss, James Tuck, Josep Torrellas, and Jose Renau. 2006. CAVA: Using checkpoint-assisted value prediction to hide L2 misses. *ACM Trans. Archit. Code Optim.* 3, 2 (Jun. 2006), 182–208.

[13] Tien-Fu Chen and Jean-Loup Baer. 1995. Effective hardware-based data prefetching for high-performance processors. *IEEE Trans. Comput.* 44, 5 (1995), 609–623.

[14] George Z. Chrysos and Joel S. Emer. 1998. Memory dependence prediction using store sets. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA'98)*. 142–153.

[15] Richard J. Eickemeyer and Stamatis Vassiliadis. 1993. A load-instruction unit for pipelined processors. *IBM J. Res. Dev.* 37, 4 (1993), 547–564.

[16] Fernando A. Endo, Arthur Perais, and André Seznec. 2017. On the interactions between value prediction and compiler optimizations in the context of EOLE. *ACM Trans. Archit. Code Optim.* 14, 2, Article 18 (Jul. 2017), 24 pages.

[17] Stijn Eyerman and Lieven Eeckhout. 2008. System-level performance metrics for multiprogram workloads. *IEEE Micro* 28, 3 (May 2008), 42–53.

[18] Brian Fields, Shai Rubin, and Rastislav Bodík. 2001. Focusing processor policies via critical-path prediction. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA'01)*. 74–85.

[19] Agner Fog. 2017. Lists of Instruction Latencies, Throughputs and Micro-Operation Breakdowns for Intel, AMD and VIA CPUs. Retrieved from http://www.agner.org/optimize/.

[20] Chao-Ying Fu, Matthew D. Jennings, Sergei Y. Larin, and Thomas M. Conte. 1998. Value speculation scheduling for high performance processors. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)*. 262–271.

[21] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. 1991. Two techniques to enhance the performance of memory consistency models. In *Proceedings of the 1991 International Conference on Parallel Processing*. 355–364.

[22] Bart Goeman, Hans Vandierendonck, and Koen de Bosschere. 2001. Differential FCM: Increasing value prediction accuracy by improving table usage efficiency. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA'01)*. 207–216.

[23] José González and Antonio González. 1997. Speculative execution via address prediction and data prefetching. In *Proceedings of the 11th International Conference on Supercomputing (ICS'97)*. 196–203.

[24] Vishal Gupta, Hyesoon Kim, and Karsten Schwan. 2012. *Evaluating Scalability of Multi-Threaded Applications on a Many-Core Platform*. Technical Report. Georgia Institute of Technology.

[25] Milad Hashemi, Onur Mutlu, and Yale N. Patt. 2016. Continuous runahead: Transparent hardware acceleration for memory intensive workloads. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. 1–12.

[26] Milad Hashemi and Yale N Patt. 2015. Filtered runahead execution with a runahead buffer. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO'15)*. 358–369.

[27] Timothy H. Heil, Zak Smith, and J. E. Smith. 1999. Improving branch predictors by correlating on data values. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO'99)*. 28–37.

[28] Yasuo Ishii, Mary Inaba, and Kei Hiraki. 2009. Access map pattern matching for data cache prefetch. In *Proceedings of the 23rd International Conference on Supercomputing (ICS'09)*. 499–500.

[29] José A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt. 2012. Bottleneck identification and scheduling in multithreaded applications. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*. 223–234.

[30] José A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt. 2013. Utility-based acceleration of multithreaded applications on asymmetric CMPs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*. 154–165.

[31] D. Kadjo, J. Kim, P. Sharma, R. Panda, P. Gratz, and D. Jimenez. 2014. B-fetch: Branch prediction directed prefetching for chip-multiprocessors. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'14)*. 623–634.

[32] Nevin Kirman, Meyrem Kirman, Mainak Chaudhuri, and Jose F. Martinez. 2005. Checkpointed early load retirement. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA'05)*. 16–27.

[33] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*. 469–480.

[34] Mikko H. Lipasti and John Paul Shen. 1996. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO'96)*. 226–237.

[35] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. 1996. Value locality and load value prediction. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'96)*. 138–147.

[36] Paul E. McKenney. 2017. Is parallel programming hard, and, if so, what can you do about it?(v2017. 01.02 a). *arXiv Preprint arXiv:1701.00854* (2017).

[37] A. Mendelson and F. Gabbay. 1996. *Speculative Execution based on Value Prediction*. Technical Report. EE Department TR 1080, Technion, Israel Institue of Technology.

[38] Joshua San Miguel, Mario Badr, and Natalie Enright Jerger. 2014. Load value approximation. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'14)*. 127–139.

[39] Andreas Moshovos, Scott E. Breach, T. N. Vijaykumar, and Gurindar S. Sohi. 1997. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA'97)*. 181–193.

[40] Andreas Moshovos and Gurindar S. Sohi. 1997. Streamlining inter-operation memory communication via data dependence prediction. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO'97)*. 235–245.

[41] Onur Mutlu, Hyesoon Kim, and Y. N. Patt. 2005. Techniques for efficient processing in runahead execution engines. In *32nd International Symposium on Computer Architecture (ISCA'05)*. 370–381.

[42] Onur Mutlu, Hyesoon Kim, and Yale N. Patt. 2006. Efficient runahead execution: Power-efficient memory latency tolerance. *IEEE Micro* 26, 1 (Jan. 2006), 10–20.

[43] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. 2003. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA'03)*. 129–140.

[44] Tarun Nakra, Rajiv Gupta, and Mary L. Soffa. 1999. Global context-based value prediction. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture (HPCA'99)*. 4–12.

[45] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. 2010. PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'10)*. 2–11.

[46] Arthur Perais, Fernando A. Endo, and André Seznec. 2016. Register sharing for equality prediction. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. 1–12.

[47] Arthur Perais and André Seznec. 2014. EOLE: Paving the way for an effective implementation of value prediction. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA'14)*. 481–492.

[48] Arthur Perais and André Seznec. 2014. Practical data value speculation for future high-end processors. In *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture, (HPCA'14)*. 428–439.

[49] Arthur Perais and André Seznec. 2015. BeBoP: A cost effective predictor infrastructure for superscalar value prediction. In *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA'15)*. 13–25.

[50] Ered Perelman, Greg Hamerly, and Brad Calder. 2003. Picking statistically valid and early simulation points. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT'03)*. 244–255.

[51] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*. 475–486.

[52] Toshinori Sato and Itsujiro Arita. 2002. Low-cost value predictors using frequent value locality. In *High Performance Computing*. Springer, 106–119.

[53] Yiannakis Sazeides. 2002. Modeling value speculation. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA'02)*. 211–222.

[54] Yiannakis Sazeides and James E Smith. 1997. *Implementations of Context Based Value Predictors*. Technical Report. Citeseer.

[55] Yiannakis Sazeides and James E. Smith. 1997. The predictability of data values. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO'97)*. 248–258.

[56] Yiannakis Sazeides, Stamatis Vassiliadis, and James E. Smith. 1996. The performance potential of data dependence speculation and collapsing. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO'96)*. 238–247.

[57] André Seznec and Pierre Michaud. 2006. A case for (partially) TAgged GEometric history length branch prediction. *J. Instruct. Level Parallel.* 8 (2006), 1–23.

[58] Rami Sheikh, Harold W. Cain, and Raguram Damodaran. 2017. Load value prediction via path-based address prediction: Avoiding mispredictions due to conflicting stores. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'17)*. 423–435.

[59] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. 2007. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA'07)*. 63–74.

[60] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. 2009. Accelerating critical section execution with asymmetric multi-core architectures. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*. 253–264.

[61] Bradley Thwaites, Gennady Pekhimenko, Hadi Esmaeilzadeh, Amir Yazdanbakhsh, Onur Mutlu, Jongse Park, Girish Mururu, and Todd Mowry. 2014. Rollback-free value prediction with approximate loads. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT'14)*. 493–494.

[62] Nathan Tuck and Dean M Tullsen. 2005. Multithreaded value prediction. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA'05)*. IEEE, 5–15.

[63] Dean M. Tullsen and John S. Seng. 1999. Storageless value prediction using prior register values. In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA'99)*. 270–279.

[64] Eric Tune, Dean M. Tullsen, and Brad Calder. 2002. Quantifying instruction criticality. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques (PACT'02)*. 104.

[65] Gary S. Tyson and Todd M. Austin. 1997. Improving the accuracy and performance of memory communication through renaming. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO'97)*. 218–227.

[66] Kai Wang and Manoj Franklin. 1997. Highly accurate data value prediction using hybrid predictors. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO'97)*. 281–290.

[67] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)*. 24–36.

[68] Amir Yazdanbakhsh, Gennady Pekhimenko, Bradley Thwaites, Hadi Esmaeilzadeh, Onur Mutlu, and Todd C. Mowry. 2016. RFVP: Rollback-free value prediction with safe-to-approximate loads. *ACM Trans. Archit. Code Optim.* 12, 4, Article 62 (Jan. 2016), 26 pages.

[69] Amir Yazdanbakhsh, Bradley Thwaites, Hadi Esmaeilzadeh, Gennady Pekhimenko, Onur Mutlu, and Todd C. Mowry. 2016. Mitigating the memory bottleneck with approximate load value prediction. *IEEE Des. Test* 33, 1 (Feb. 2016), 32–42.

[70] Huiyang Zhou and Thomas M. Conte. 2005. Enhancing memory-level parallelism via recovery-free value prediction. *IEEE Trans. Comput.* 54, 7 (July 2005), 897–912.

[71] Victor Zyuban and Peter Kogge. 1998. The energy complexity of register files. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design (ISLPED'98).* 305–310.

Received May 2017; revised June 2018; accepted July 2018