Load Value Prediction via Path-based Address Prediction: Avoiding Mispredictions due to Conflicting Stores

Rami Sheikh[†], Harold W. Cain[‡], and Raguram Damodaran[†]

[†]Qualcomm Technologies, Inc.

[‡]Qualcomm Datacenter Technologies, Inc.
`{ralsheik, tcain, raguramd}@qti.qualcomm.com`

ABSTRACT

Current flagship processors excel at extracting instruction-level-parallelism (ILP) by forming large instruction windows. Even then, extracting ILP is inherently limited by true data dependencies. Value prediction was proposed to address this limitation. Many challenges face value prediction, in this work we focus on two of them. *Challenge #1*: store instructions change the values in memory, rendering the values in the value predictor stale, and resulting in value mispredictions and a retraining penalty. *Challenge #2*: value mispredictions trigger costly pipeline flushes. To minimize the number of pipeline flushes, value predictors employ stringent, yet necessary, high confidence requirements to guarantee high prediction accuracy. Such requirements can negatively impact training time and coverage.

In this work, we propose *Decoupled Load Value Prediction* (DLVP), a technique that targets the value prediction challenges for load instructions. DLVP mitigates the stale state caused by stores by replacing value prediction with memory address prediction. Then, it opportunistically probes the data cache to retrieve the value(s) corresponding to the predicted address(es) early enough so value prediction can take place. Since the values captured in the data cache mirror the current program data (except for in-flight stores), this addresses the first challenge. Regarding the second challenge, DLVP reduces pipeline flushes by using a new context-based address prediction scheme that leverages load-path history to deliver high address prediction accuracy (over 99%) with relaxed confidence requirements. We call this address prediction scheme Path-based Address Prediction (PAP). With a modest 8KB prediction table, DLVP improves performance by up to 71%, and 4.8% on average, without increasing the core energy consumption.

CCS CONCEPTS

• **Computer systems organization** → **Superscalar architectures**; *Pipeline computing*; Reduced instruction set computing;

KEYWORDS

Microarchitecture, Value Prediction, Address Prediction, Path-based Predictor

ACM Reference format:
Rami Sheikh[†], Harold W. Cain[‡], and Raguram Damodaran[†] [†]Qualcomm Technologies, Inc. [‡]Qualcomm Datacenter Technologies, Inc. `{ralsheik, tcain, raguramd}@qti.qualcomm.com`. 2017. Load Value Prediction via Path-based Address Prediction: Avoiding Mispredictions due to Conflicting Stores. In *Proceedings of MICRO-50, Cambridge, MA, USA, October 14–18, 2017,* 13 pages.
https://doi.org/10.1145/3123939.3123951

1 INTRODUCTION

Single-thread (ST) performance is critical for both single-threaded and multi-threaded applications [16]. Current flagship processors excel at extracting instruction-level-parallelism (ILP) by forming large instruction windows. Unfortunately, extracting ILP is inherently limited by true data dependencies. Value prediction was proposed to address this limitation [12, 20]. By predicting the value(s) produced by an instruction (*producer*), instructions that consume the value(s) (*consumers*) can speculatively execute before the producer has executed. The prediction is later confirmed when the producer is executed. If the predicted value did not match the computed value, recovery actions take place. In this work we assume a flush-based recovery microarchitecture, similar to the work of Perais and Seznec [28, 29].

The recently proposed state-of-art value predictor VTAGE [28, 29] addresses some of the key practical challenges facing value prediction. In this work, we focus on two of the remaining challenges:

(1) A value predictor's history tables attempt to capture the current program state. Store instructions can change the program state by modifying the values in memory. This change can render the values in the value predictor stale, resulting in value mispredictions and then retraining the predictor. To demonstrate the severity of this problem, we profiled the load-store sequences in our workloads (discussed in Section 4.1). Figure 1 shows the fraction of dynamic load instructions that exhibit the following sequences:
   (a) *Load* → *Store* → *Load*: two dynamic instances of the same static load read the same memory location with an interleaving store that modifies the value in that memory location.

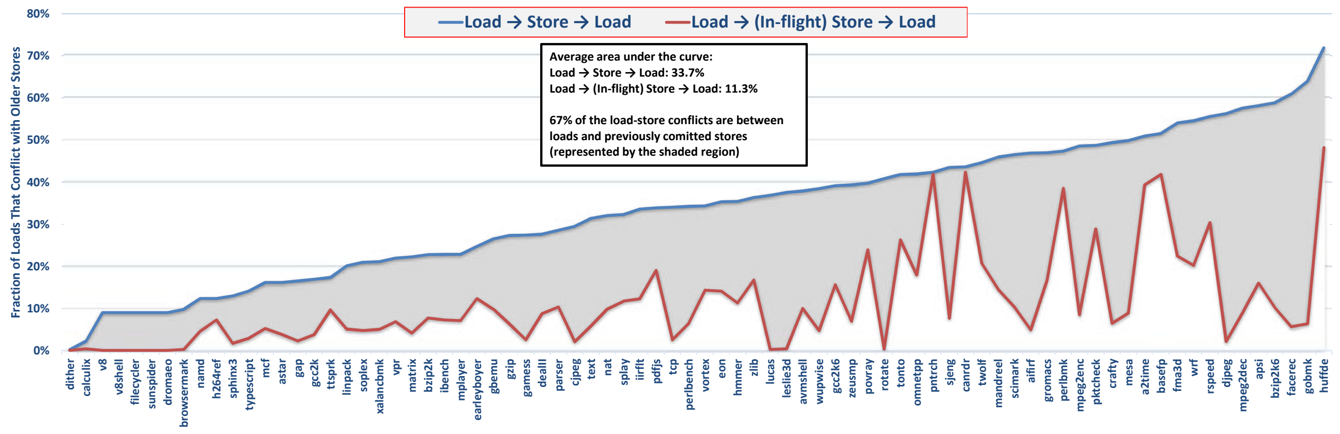


Figure 1: Fraction of dynamic loads that consume a value that is produced by a store that occurs since the prior dynamic instance of that load. X-axis shows different workloads.

(b) *Load* → "In-flight" *Store* → *Load*: similar to the earlier sequence except that by the time the latter load is fetched, the conflicting store is not committed yet.

For the loads that belong to the first sequence, a conventional value predictor (e.g., Last-Value-Predictor, LVP [20]) might mispredict the second load's value because the value has been changed by the interleaving store. The shaded region represents value mispredictions that can be avoided if the value prediction table can be pro-actively updated when stores commit. Unfortunately, doing so requires a complicated machinery for identifying which predictor entries are impacted by the committing stores. We refer the readers to the EXACT branch predictor [1] in which a similar machinery is employed to pro-actively update the branch predictor on stores.

(2) Due to the high performance cost of value mispredictions (they trigger pipeline flushes), value predictors employ stringent, yet necessary, high confidence requirements to guarantee very high prediction accuracy. For example, VTAGE [28] establishes confidence after encountering the value 64 or 128 times. The high confidence requirements negatively impact training time and coverage.

In this work, we target the value prediction challenges for one class of critical instructions: load instructions.

Given that data cache contents mirror the current program state, except for in-flight stores, we argue that by replacing value prediction with memory address prediction, we can potentially eliminate most of the negative interactions between stores and value prediction. As shown in Figure 1, 67% of the load-store conflicts are between loads and previously committed stores. This can effectively mitigate *Challenge #1*.

Similar to early work [3, 9, 13], we observed that load memory addresses exhibit similar *temporal* locality properties as loaded values. This observation is illustrated in Figure 2, which shows the breakdown of dynamic load instructions averaged across all of our workloads. The x-axis shows how often an address or value repeats,

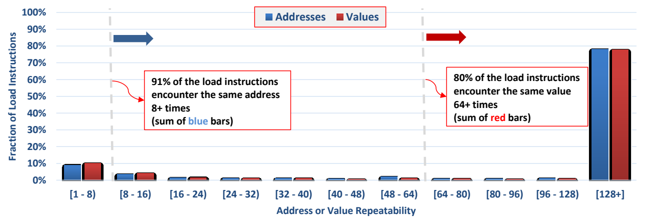


Figure 2: Breakdown of dynamic load instructions according to the repeatability of observed addresses or values.

and the y-axis shows the fraction of load instructions. On average, not shown in the figure, loaded values are encountered 4% more than memory addresses. I.e., a particular load encounters the same address less often than it encounters the same value. This introduces an interesting challenge for address prediction. To deliver coverage equivalent to that of value prediction, address prediction needs to establish confidence sooner. To put it another way, an address predictor needs to establish confidence after encountering fewer address occurrences, compared to the number of value occurrences needed to establish confidence in a value predictor.

Regarding *Challenge #2*, ensuring high prediction accuracy, we propose a new context-based address prediction scheme that leverages a unique context information that we refer to as *load-path history*, to enable high address prediction accuracy with *less stringent confidence requirements*. In contrast to branch-path history [26, 37], load-path history is less compact but allows the predictor to distinguish among multiple loads in the same basic block. We call the proposed address prediction scheme Path-based Address Prediction (PAP). Experimentally, we found that an address needs to be observed only 8 times to establish high confidence in PAP, as opposed to observing a value 64 or 128 times in VTAGE [28]. Interestingly, by inspecting the corresponding points on Figure 2 (i.e., points 8 and 64 on the x-axis), we see that the fraction of loads with addresses that repeat 8 or more times is 91%, while the fraction of loads with values that repeat 64 or more times is 80%. This 11% difference suggests that PAP can potentially deliver higher coverage of load instructions. Our results, presented in Section 5, confirm this observation.

In this paper, we propose Decoupled Load Value Prediction (*DLVP*), a novel approach to value prediction through address prediction. DLVP works as follows: using PAP, we predict the load memory addresses early in the pipeline (in our studies we assume the first stage of fetch.) Then, we communicate the predicted memory addresses to the out-of-order engine (OoO), where they are used to speculatively probe the data cache. If the data is present in the cache, we retrieve it and use it to perform value prediction. If the data is not present in the cache, we can generate a prefetch request. We discuss the details of DLVP in Section 3.2.2.

This paper makes the following contributions:

- We identify ISA-specific efficiency and accuracy challenges with conventional value predictors, and propose a simple workaround to mitigate these challenges.
- We propose PAP, a new address prediction scheme that is capable of delivering high accuracy predictions with relaxed confidence requirements.
- We propose DLVP, a microarchitecture that leverages PAP to perform data value prediction and prefetching. With a modest 8KB prediction table, DLVP improves performance by up to 71%, and 4.8% on average, without increasing the core energy consumption.
- We present comprehensive analysis of our solution and contrast it against prior art in value prediction (e.g., VTAGE [28]) and address prediction (e.g., CAP [3]).

The paper is organized as follows. In Section 2, we discuss related work in address prediction and value prediction. In Section 3, we present our address prediction scheme (PAP) and the microarchitecture that uses it for value prediction (DLVP). In Section 4, we discuss our methodology and describe our evaluation framework and baseline. In Section 5, we present a thorough evaluation of our solution. We conclude the paper in Section 6.

2 RELATED WORK

2.1 Value Prediction

Since the introduction of value prediction [12, 20], there has been a plethora of work on this subject. In general, value predictors can be classified into two broad categories.

Computation-based Predictors. In this class of predictors, predicted values are generated by applying a function to the value(s) produced by previous instance(s) of the instruction. Stride predictors [10, 12] are good examples of this class: the prediction is generated by adding a constant (*stride*) to the previous value.

Context-based Predictors. This class of predictors rely on identifying patterns in the history of a given static instruction to predict the value. Finite Context Method predictors (FCM) [32, 33] are good examples of this class. Typically, such predictors use two structures: one captures the history for the instruction. This history is used to index the second structure, which captures the values.

Most, if not all, value predictors employ confidence mechanisms to ensure high prediction accuracy.

The recently proposed state-of-art value predictors VTAGE [28] and D-VTAGE [29] are context-based predictors. VTAGE uses several tagged prediction tables that are indexed using a hash of instruction PC and different number of bits from the global branch history (*context*). These tables are backed up by a PC indexed, tagless last-value predictor (LVP). D-VTAGE augments VTAGE with a last-value-table (LVT) that is located before the first VTAGE table (VT0). LVT stores the last value (per instruction), while the VTAGE tables store the strides/deltas. D-VTAGE introduces additional complexity as it requires an addition on the prediction critical path, moreover, it requires maintaining a speculative window to track in-flight last values.

In our evaluation, we use VTAGE as a representative of context-based value predictors. We use the best performing VTAGE configuration we identified through extensive design space exploration: we did a full sweep of the following predictor dimensions: number of tables, table parameters (tag width, history length, and associativity), hash functions, and forward probabilistic counter (FPC) vector. Interestingly, we found that using tags with the LVP table is crucial for delivering the expected high prediction accuracy. A quantitative comparison against VTAGE is presented in Section 5.2.3. Arguably, VTAGE might indirectly workaround the conflicting store problem by using long branch history. If that is the case, then it is already included in our evaluation. We show that directly addressing the conflicting store problem through address prediction is a more optimal solution.

Our proposed value prediction solution differs from conventional value prediction in many ways:

Functionality. First, we do not directly predict the value, instead we predict the memory address. Second, predicted values are acquired by probing the data cache using the predicted memory address. In a way, the data cache acts as the data-store in our solution. This can potentially eliminate most of the negative interactions between stores and value prediction. Finally, by virtue of using address prediction, our solution is capable of generating highly accurate prefetches.

Storage efficiency. By virtue of predicting memory addresses (32-bit for ARMv7 or 49-bit for ARMv8) as opposed to 64-bit values, DLVP is more storage efficient in that respect. Moreover, some ISAs (e.g., ARM) support load instructions that have two or more destination registers. E.g., load-pair (LDP) has two destination registers, and load-multiple (LDM) has up to sixteen destination registers. With DLVP (i.e., address prediction), only one address needs to be predicted (the base address), all destination registers are loaded from relative or consecutive memory locations starting at the predicted address. Therefore, only one address predictor entry is required per instruction. Meanwhile, with value prediction, the value of each destination register needs to be predicted, requiring up to sixteen predictor entries (i.e., one value predictor entry per destination register). Overall, DLVP is more storage efficient.

Coverage. DLVP can predict loads only, while conventional value predictors can potentially predict all instructions (including loads). Therefore, conventional predictors *can* provide higher coverage. But, given that it is very common for non-load instructions to directly or indirectly depend on loads, this may indirectly expose them to the load-store vulnerability discussed in Section 1. Therefore, it is possible that the higher coverage achieved by predicting all instructions can translate into higher vulnerability. Additionally, our evaluation (in Section 5.2.2) confirms that given a modest predictor budget, conventional value predictors get most benefit when they target load instructions only [5].

Complexity. DLVP requires probing the data cache to retrieve the predicted values. This is a necessary evil to work around the negative interactions between stores and value prediction (discussed in Section 1). To retrieve the predicted values, conventional value predictors do not require accessing the data cache, but this makes them more vulnerable to the negative interaction between stores and value prediction. In our baseline, the L1 prefetcher generates prefetch requests that check the L1 cache before they are propagated down the memory hierarchy. Our work (DLVP) leverages the same path to probe the data cache. To help mitigate the energy cost incurred by speculatively probing the data cache, we augment DLVP with way prediction. Therefore, only one cache way is checked instead of the entire cache set.

2.2 Address Prediction

Address predictors can also be categorized into: Computation-based Predictors [10], and Context-based Predictors [3]. Address prediction has been mostly used for hiding the multi-cycle access latency of the memory hierarchy [2, 3, 9, 10, 34], and to prefetch the data to be accessed by the next invocation in strided memory accesses [13]. Similar to other prediction schemes, address predictors employ confidence mechanisms to ensure high prediction accuracy.

Correlated Address Predictor (CAP) [3] is a context-based predictor that uses previous load memory addresses as context. A dedicated structure, called the Load Buffer table, is used to capture the memory address history per static load. Load's PC is used to probe the load buffer table, and the memory address history read is used to probe a second structure, called the Link table, which stores the predicted memory addresses. CAP is capable of capturing *stride* and *non-stride* memory addresses. Therefore, we believe it is a good representative of address predictors in the literature.

Like CAP, our proposed path-based address prediction scheme (PAP) is context-based. Unlike CAP, PAP uses global program context (load-path history) as opposed to history per-static load. Therefore, it does not require a dedicated table for storing context information. PAP captures context using a single history register, which simplifies the management of the predictor's speculative state: speculative history update and restoration on a misprediction. For instance, with PAP, we take a snapshot of the history register after each speculative history update. Upon recovery from a misprediction, we simply restore the snapshot associated with the value mispredicted load. Meanwhile, for other prediction schemes that use per static instruction history (e.g., CAP), the management of the predictor's speculatively updated history is more complicated: we take a snapshot of the per static instruction history after each speculative history update. Upon recovery from a misprediction, we walk the structure that stores the snapshots in reverse program order, and we restore the snapshots corresponding to squashed instructions. This is a serial process. In Section 5.1, we present a quantitative comparison of PAP against CAP predictor.

2.3 Memory Dependence Prediction

Data dependences between instructions can be explicit (via registers) or implicit (via memory). Memory Dependence Predictors (MDPs) [7, 18, 24] predict dependencies between memory operations (e.g.,loads and stores). One common application of MDP is to prevent memory ordering violations: by predicting store-load dependencies, and then delaying the dependent load until it is safe to execute.

By speculatively reading the data cache using the predicted load memory address (well before the actual load can execute), DLVP can incur a value misprediction if an in-flight, conflicting store exists in the pipeline. This is a form of memory ordering violation. Our baseline MDP [18] (described in section 4.2) can be potentially leveraged to detect and prevent such violations. Unfortunately, since the MDP is tightly coupled with the processor's back-end, it is not feasible to use it for this purpose. To workaround this issue, DLVP introduces a small (4-entry) table, called LSCD, shown in Figure 3 and described in section 3.2.2. LSCD acts as a simple, special-purpose MDP that prevents such violations.

Memory Renaming (MR) [25, 41] is another application that leverages MDP to predict loads and stores that are dependent, and speculatively remove them from the *DEF-store-load-USE* chains containing them. I.e., it speculatively forwards the data from the store data producer (DEF) to the load value consumer (USE). Effectively, transforming these chains into *DEF-USE* chains.

MR can partially address the first challenge (described in Section 1), namely, the negative interaction between stores and value prediction. In particular, it can effectively detect the load-store conflict and predict the loaded value given that both instructions are in the pipeline. Per this definition, MR is profitable when the instructions of the *DEF-store-load-USE* chain co-exist in the pipeline. Meanwhile, DLVP targets the scenario in which the conflicting store has left the pipeline. As a matter of fact, LSCD is introduced to prevent predicting loads that depend on in-flight stores. Therefore, DLVP and MR are orthogonal and can complement one another.

3 DLVP: DECOUPLED LOAD VALUE PREDICTION

In this section, we discuss our proposed address prediction scheme (PAP), and then we present DLVP, a microarchitecture that leverages PAP to perform value prediction.

Figure 3 shows our processor pipeline. Throughout this section, we will reference this figure and highlight the changes required to support value prediction, in general, and DLVP, in particular.

3.1 PAP: Path-based Address Prediction

We propose a context-based address prediction scheme. We investigated using various context information (e.g., local or global branch history [37], branch-path history [26] ...etc.) Our investigation revealed a new, unique context that showed strong correlation with load memory addresses. We call this context load-path history, and we call our proposed prediction scheme path-based address prediction (PAP).

Load-path history is constructed by shifting the least significant, non-zero bit from each load PC (i.e., bit-2, the third bit, because most instructions are 4 bytes) into a new load-path history register.

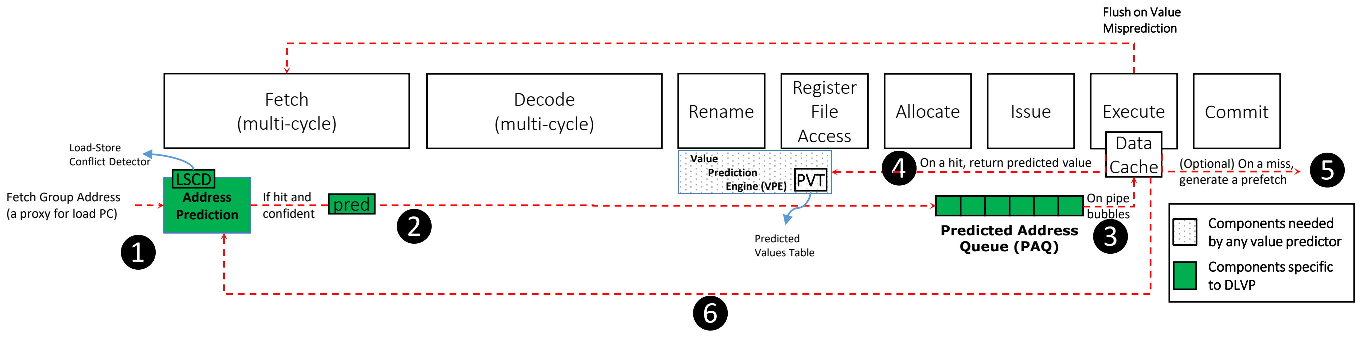

Flush on Value Misprediction
Fetch (multi-cycle)
Decode (multi-cycle)
Rename
Register File Access
Allocate
Issue
Execute
Commit
Data Cache
Load-Store Conflict Detector
LSCD
Address Prediction
Fetch Group Address (a proxy for load PC)
If hit and confident
pred
Value Prediction Engine (VPE)
PVT
Predicted Values Table
On a hit, return predicted value
(Optional) On a miss, generate a prefetch
On pipe bubbles
Predicted Address Queue (PAQ)
Components needed by any value predictor
Components specific to DLVP
1
2
3
4
5
6

Figure 3: Pipeline with support for value prediction and DLVP.

Tag	Memory Address	Confidence	Size	Cache Way (Optional)
14 bits	32 bits (for ARMv7), or 49 bits (for ARMv8)	2 bits	2 bits	2 bits

Table 1: Fields of our address predictor entry.

This load-path history forms a global context of the path by which a current load was reached. By virtue of using global program context rather than per static instruction context, the management of the predictor's speculatively updated history register is simple (discussed in Section 2.2).

Next, we discuss prediction and training PAP predictor.

3.1.1 Prediction. We introduce a partially tagged, direct-mapped structure called the Address Prediction Table (APT). It resides in the processor front-end, inside the address prediction block in Figure 3.

Table 1 shows the fields of an APT entry:

- *Tag* field is 14 bits, and it is computed using an XOR of the low order bits of load PC and load-path history.
- *Memory Address* field is 32-bit or 49-bit, and it stores a predicted memory address.
- *Confidence* is a 2-bit FPC [31]. An FPC is different than a conventional counter in that each forward transition is only triggered with a certain probability. We use the following probability vector in our design $\{1, {}^1/_2, {}^1/_4\}$.
- *Size* field is 2-bit, and it encodes the number of bytes to be read (e.g., 0 means 4 bytes, 1 means 8 bytes ..etc.)
- *Cache Way* is an optional field, it stores the cache way at which the cache block is expected to be present.

We index the APT in the first stage of fetch using the fetch group address (FGA). Experimentally, we found that the FGA is a good proxy for the load PC. Therefore, in our design, FGA is used. For simplicity, we will refer to this proxy PC as load PC in the text.

APT is indexed and tagged using an XOR of the low order bits from the load PC and folded load-path history. On a tag mismatch (*APT miss*), no prediction happens. On a tag match (*APT hit*), we read the contents of the hitting entry. If the confidence counter is saturated, signifying the predictor is confident, we predict using the memory address field. Otherwise, the predictor is still training and no prediction is made.

Predicting Multiple Load Instructions. In our workloads, over 98% of the fetch groups contain at most two load instructions. In our design, we use load PC and load PC plus one (*aka*, fetch group PC and fetch group PC plus one) to predict up to two load addresses in any given cycle. For fetch groups that contain more than two load instructions (less than 2% of the fetch groups), only the first two load instructions are predicted.

3.1.2 Training. APT is trained when a load executes. Training takes place for both scenarios: APT hit, and APT miss.

Training on an APT Miss. We considered two allocation policies. *Policy-1*, a new entry is always allocated and replaces the probed entry. *Policy-2*, a new entry is allocated if the confidence of the probed entry is zero. If the confidence of the probed entry is greater than zero, we decrement it.

Our experiments (not included due to limited space) show that Policy-2 is superior. This is expected since entries with high confidence can survive eviction. Over time, confident but less frequently accessed entries will get reallocated. We adopt Policy-2 in our design.

On allocation, the confidence is initialized to zero, and the remaining entry fields (tag, memory address, and cache way) are initialized with the executed load information.

Training on an APT Hit. If the predicted address matches the computed address, we increment the confidence of the hitting entry. Otherwise, we reset the confidence and reallocate the entry. Similarly, the allocated entry is initialized with the executed load information.

3.2 Value Prediction

First, we discuss the hardware support needed for communicating predicted values from producers to consumers. Such hardware support is common to all value prediction schemes. Then, we discuss the proposed DLVP microarchitecture.

3.2.1 Value Prediction Engine. In a conventional design, values are communicated between instructions through the physical register file (PRF). *Producer* instructions write computed values to the PRF (using write ports), after execution (*Execute* stage). *Consumer* instructions read their source values from the PRF (using read ports) after rename, at *Register File Access* stage. Our baseline, shown in Table 4, has a 4-wide in-order front-end and an 8-wide out-of-order execution engine. Therefore, the PRF has 8 read ports (2 read ports per instruction: one per source register), and 8 write ports (one per execution lane.)

With value prediction, a mechanism is needed to communicate the predicted values from the value-predicted producers to their consumers. Timely predicted values are usually available before Register File Access stage. Their consumers however can be any instruction(s) younger than the value-predicted producer. Therefore, predicted values need to be captured in anticipation of future consumption.

In order to leverage the existing value communication machinery (through the PRF) to communicate predicted values to their consumers, we envision two possible designs:

(1) *Design #1*: arbitrating on PRF write ports between executed and value-predicted instructions. This design requires minimal changes to the PRF and has little or no impact on its area and energy-per-access. The main disadvantage of this design is: if all execution lanes are heavily utilized, PRF write ports can become a bottleneck, disrupting the seamless flow of instructions and degrading performance. Such design may not be compelling for high performance cores.

(2) *Design #2*: increasing number of PRF write ports to accommodate writing the predicted values. Since the PRF is a large, multi-ported structure, increasing the number of write ports can have significant implications on the area and energy-per-access for the PRF, yielding this option unattractive as well [42].

Our profiling shows that, at any given point in time, a relatively small subset of the registers are value predicted. This is especially true since we only target load instructions in this work. We advocate for using a new, dedicated structure that captures the predicted values, called the *Predicted Values Table* (PVT). By virtue of using a dedicated structure, we no longer require the PRF write port arbitration (needed for design #1), and we do not increase the number of PRF write ports (needed for design #2).

PVT is simply a small cache for predicted values (only 32 entries): it uses the destination register identifier (i.e., physical register number) of the predicted instruction as a tag, and stores the predicted value in the payload. A PVT entry is allocated for each value-predicted register. If the PVT is full, a value prediction is treated as no prediction (i.e., no entry is allocated). In our evaluation, this scenario is almost never encountered. PVT entries are deallocated when the value predicted instructions execute and validate the predictions (now, the *actual* values can be retrieved from the PRF instead.)

In this work, we assume that up to two value predictions can be made every cycle. Therefore, we design the PVT with two write ports. To reduce the PVT read port requirements, we augment each rename-map-table (RMT) entry with one bit, called the *predicted bit*, that signals whether the corresponding register is value predicted or not: predicted values are read from PVT, and non-predicted values are read from PRF. Consequently, PVT will only be probed for predicted values. For our workloads, using a PVT with two read ports is sufficient. When the value predicted instruction executes, it writes the computed value to the PRF, it clears the corresponding predicted bit in the RMT (if the register mapping has not changed), and it deallocates the corresponding PVT entry. On a pipeline flush, we deallocate the PVT entries corresponding to squashed instructions, we restore the RMT from a checkpoint, and we reset all predicted bits.

	PVT (2rd/2wr ports)	**Design #1** (PRF with 8rd/8wr ports)	**Design #2** (PRF with 8rd/10wr ports)	**Design #3** (Design #1 plus PVT)
Area	0.06	1.00	1.16	1.06
Read energy	0.10	1.00	1.10	0.80
Write energy	0.07	1.00	1.51	1.07

Table 2: Area and energy normalized to design #1.

We call the machinery that manages the PVT, the Value Prediction Engine (VPE), shown in Figure 3. A PVT-based design requires a MUX to allow PVT values to get selected instead of PRF values, which can add some delay to the critical path. We refer to this approach as: *Design #3*.

Using the area and energy model described in Section 4.2, we estimate the area, read energy, and write energy for the three designs. Table 2 shows the normalized area and energy (assuming that 30% of the register values read/written are predicted.) Since the PVT is small, design #3 incurs a modest area increase compared to design #1. Additionally, PVT has fewer ports than PRF, and its read energy is smaller. This results in design #3 having: a lower read energy (by replacing some of the PRF reads with PVT reads), and a higher write energy (because predicted values are written to PVT). Each one of the three designs has its own merits, it is up to the designers to decide amongst the three design options. We adopt design #3.

In Figure 3, we show *Register File Access* stage immediately after *Rename* stage. An alternative core design might have the Register File Access after *Issue* stage [21]. The VPE operations described earlier work similarly in both designs, except that in the latter design, values read from PVT need to be captured with the instructions in Issue stage.

3.2.2 DLVP Microarchitecture. DLVP is a technique that performs value prediction through address prediction. It works as follows (refer to Figure 3):

- Leverage the PAP address predictor to generate high accuracy predictions early in the pipeline (❶). In our studies we assume that address prediction takes place in the first stage of fetch.
- Communicate the predicted memory addresses to the out-of-order engine (❷), where they get deposited in a newly introduced FIFO queue, called *Predicted Address Queue* (PAQ).
- On load-store execution lane bubbles, opportunistically probe the data cache using the predicted memory addresses from PAQ (❸, *first cache access*).
  - On a cache hit (❹), retrieve the predicted values from the cache data array and communicate them to the *Value Prediction Engine*, at rename stage.
  - On a cache miss, a prefetch request *can* be generated (❺).
- Update the address predictor and confirm the value prediction, when the load instruction executes (❻, *second cache access*).
  - We always update the address predictor as described in Section 3.1.2.
  - If the *predicted* value does not match the *loaded* value, a value misprediction happened, we flush the pipeline. We assume a 1-cycle penalty for checking and confirming the correctness of the predicted value. This penalty is exposed only on a misprediction.

Next, we discuss important aspects specific to DLVP.

Generating Timely Value Predictions.

In conventional value prediction, the predicted values are read from the prediction tables, and delivered to VPE, in a timely manner. With DLVP, the predicted values are read from the data cache. For a timely value prediction, the predicted value(s) must be delivered to the VPE by the time the predicted load reaches rename stage. This introduces an interesting challenge to DLVP: *the timely delivery of predicted values to the VPE*.

We propose to drop an PAQ entry after a fixed, predefined number of cycles (*N*) from its allocation. N corresponds to the *guaranteed* minimum number of cycles available for retrieving the values from the data cache before the load reaches *Rename*. Therefore, it's value is influenced by the pipeline depth, wiring delays, and cache access latency. In our model, the value of N is determined with the following latencies in mind: 1-cycle for load address prediction, 1-cycle for sending the request or data from the front-end to the back-end and vice versa, and 1-cycle for reading the data cache (facilitated by way prediction.) In a pipeline similar to Cortex-A72 [11], the *Fetch* and *Decode* stages take 5 and 3 cycles, respectively. Therefore, N equals 4. Pipeline stalls can allow for a larger N, but, for design simplicity we adopt this definition. Our evaluation shows less than 0.1% of the PAQ entries are dropped. Note that in our design, a request can bypass the PAQ if empty.

Avoiding Value Mispredictions Due to Conflicts with In-flight Stores.

By virtue of predicting memory addresses, not values, DLVP eliminates most of the negative interactions between stores and value prediction. As shown in Figure 1, two thirds of the load-store conflicts are eliminated in our workloads. The only exception is the older *in-flight* stores that modify the memory content read speculatively by DLVP. Under these circumstances, even a correctly predicted memory address can result in an incorrect value prediction.

To prevent such stores from hurting DLVP, we augment our address predictor with a 4-entry filter called the *Load-Store Conflict Detector* (LSCD). LSCD is used to capture the PCs of address predicted loads that suffered conflicts with older, in-flight stores. A load PC is inserted into the filter when its address is predicted correctly but the predicted value was incorrect. This signifies that an in-flight store updated the content of the memory location after the value prediction was made. LSCD prevents future instances of the captured loads from getting predicted or updating the prediction table. The entries allocated for these loads will be naturally evicted from the APT as new entries get allocated. This optimization eliminates the value mispredictions that are caused by in-flight, conflicting stores. An alternative to LSCD is to leverage the existing memory dependence predictor (MDP) [7, 18] to detect and prevent the conflicts. In our baseline, the MDP is tightly coupled with the processor's back-end, therefore, it is not feasible to use it for this purpose. LSCD functions as a simple, special-purpose MDP.

Power Optimization.

Due to the non-negligible energy cost of probing the data cache speculatively (using predicted memory addresses), we propose augmenting our address predictor with cache way prediction, shown in Table 1. The width of the cache way field is log2(cache-associativity) bits. This optimization can help mitigate the aforementioned energy cost. A way misprediction is possible, and it can happen if the cache block is evicted and then reinserted into the cache, but at a different cache way. Our evaluation shows way mispredictions almost never happen.

Memory Consistency.

ARM's relaxed consistency model allows for reordering most memory operations with one exception: dependent loads are not allowed to be reordered [6, 27]. Value prediction can violate this rule. To avoid violating the memory consistency model, we employ a technique similar to the work of Martin *et al.* [22]. Also, address prediction is not used with memory ordering instructions, atomic and exclusive memory accesses.

4 METHODOLOGY AND EVALUATION ENVIRONMENT

4.1 Methodology

We cast a wide net to expose as many load address and value occurrence patterns as possible. We use benchmarks from the following benchmark suites: SPEC2K [38], SPEC2K6 [39], and EEMBC [30]. Moreover, we enriched our benchmark pool with other popular applications: Linpack [19], media player [23], browser benchmark [4], and various Javascript benchmarks [8, 14, 15, 17, 35, 40].

Table 3 shows a list of our benchmarks. We use 100-million instruction simpoints, except for short-running benchmarks (i.e., EEMBC), we simulate the first 100 million instructions, or until the benchmark completes.

Benchmark Suite	Applications
EEMBC (ARMv7)	a2time, aifirf, basefp, canrdr, cjpeg, dither, djpeg, fbital, huffde, iirflt, matrix, mp4encode, mpeg2dec, mpeg2enc, nat, pktcheck, pntrch, rotate, routelookup, rspeed, tcp, text, ttsprk, typescript
Miscellaneous (ARMv7)	browsermark, ibench, linpack, mplayer
Octane/Javascript (ARMv7)	avmshell, dromaeo, earleyboyer, filecycler, gbemu, mandreel, pdfjs, scimark, splay, sunspider, v8, v8shell, zlib
SPEC2K (ARMv8)	apsi, bzip2k, crafty, eon, facerec, fma3d, gap, gcc2k, gzip, lucas, mcf, mesa, perlbmk, twolf, vortex, vpr, wupwise
SPEC2K6 (ARMv8)	astar, bzip2k6, calculix, dealII, gamess, gcc2k6, gobmk, gromacs, h264ref, hmmer, leslie3d, namd, omnetpp, parser, perlbench, povray, sjeng, soplex, sphinx3, tonto, wrf, xalancbmk, zeusmp

Table 3: Applications used in our evaluation.

4.2 Evaluation Environment

The microarchitecture presented in Section 3 is faithfully modeled in our internally-developed, cycle-accurate, RTL-validated, industry simulator. The simulator runs ARM ISA binaries (v7 and v8). It is used by our CPU research and development organization. Because the simulator is specific to our proprietary custom ARM CPU design, we don't release it, and there is nothing publicly available that we can cite. Similarly, we use an in-house, RTL-PTPX validated area and energy model. We assume 28nm technology.

Section 4.1 lists the benchmark suites used. All benchmarks are compiled to the ARM ISA using gcc with -O3 level optimization: SPEC2K and SPEC2K6 are compiled for ARMv8 (*aarch64*), and the remaining benchmarks are compiled for ARMv7.

The parameters of our baseline core are configured as close as possible to those of Intel's Skylake core [21]. The baseline core uses state-of-art TAGE and ITTAGE branch predictors [36, 37], and an MDP similar to Alpha 21264 [18]. We use a fetch-to-execute latency of 13 cycles. Table 4 shows the baseline core configuration.

Branch Prediction	**BP**: state-of-art 32KB TAGE predictor and 32KB ITTAGE predictor **RAS**: 16 entries
Memory Hierarchy	**Block size**: 64B (L1), 128B (L2 and L3) **L1**: split, 64KB each, 4-way set-associative, 1-cycle/2-cycle (I/D) access latency **L2**: unified, private, 512KB, 8-way set-associative, 16-cycle access latency **L3**: unified, shared, 8MB, 16-way set-associative, 32-cycle access latency **Memory**: 200-cycle access latency **Stride-based prefetchers**
TLB	512-entry, 8-way set-associative
Fetch through Rename Width	4 instr./cycle
Issue through Commit Width	8 instr./cycle (8 *execution lanes: 2 support load-store operations, and 6 generic*)
ROB/IQ/LDQ/STQ	224/97/72/56 (modeled after Intel Skylake)
Fetch-to-Execute Latency	13-cycle
Physical RF	348
DLVP	1k-entry, direct-mapped, use 16-bit load-path history. Total budget = 1k x (50 or 67) = 50k bits (ARMv7) or 67k bits (ARMv8) 32-entry predicted address queue (PAQ)
CAP	2 tables, 1k-entry each, direct-mapped. Each load buffer entry uses 14-bit tag, 2-bit confidence, 8-bit offset, and 16-bit history. Each link entry uses 14-bit tag and 24-bit or 41-bit link. Total budget = 78k bits (ARMv7) or 95k bits (ARMv8) 32-entry predicted address queue (PAQ)
VTAGE	3 tables, 256-entry each, direct-mapped, use global branch histories of {0, 5, 13}. Each entry uses 16-bit tag, 64-bit value and 3-bit confidence. Total budget = 3 x 256 x 83 = 62.3k bits

Table 4: Baseline core configuration.

5 RESULTS AND ANALYSIS

In this section, we start by demonstrating the strengths of our PAP address predictor (i.e., its superior coverage and accuracy), by comparing it to related prior work, the CAP address predictor [3]. Then, we transition into value prediction and present a detailed evaluation of our proposed technique, DLVP. In an effort to compare DLVP to prior art in value prediction, we implemented and evaluated state-of-art value predictor VTAGE [28]. Our evaluation uncovered ISA-specific efficiency and accuracy challenges with conventional value predictors (including VTAGE) that prevent them from delivering their full potential. We propose a simple workaround to mitigate these challenges. After that, we compare DLVP to two value predictors: VTAGE, and CAP (this one is simply, DLVP but uses CAP address predictor instead of PAP). Finally, we evaluate the performance impact of using replay instead of pipeline flush as a misprediction recovery mechanism.

5.1 Address Prediction

In this section, we quantitatively compare PAP and CAP [3] address predictors. A qualitative comparison between the two predictors is presented in Section 2.2. It is worth noting that as the accuracy and coverage of address prediction improve, the benefits of DLVP improve accordingly.

Through design space exploration, we identified that a confidence of 8 suffices for PAP to deliver high prediction accuracy (greater than 99%.) In [3], CAP used a confidence of 3. Figure 4 shows the coverage and accuracy of PAP and CAP predictors when used as standalone address predictors: PAP is evaluated with a confidence of 8, and CAP is evaluated with multiple confidence levels: 3 through 64.

When the same confidence requirements are used (i.e., Confidence = 8), data shows a clear advantage for PAP over CAP, in terms of both: coverage (37% vs. 29.5%) and accuracy (99.1% vs. 97.7%)[1].

[1]Coverage is defined as the number of predicted dynamic loads divided by the number of dynamic loads, and accuracy is defined as the number of correctly predicted dynamic loads divided by the number of predicted dynamic loads.

CAP accuracy improves as we increase the confidence level, but that comes with a price: a non-negligible reduction in coverage. Interestingly, for CAP to deliver an accuracy on par with PAP, it needs to use a confidence of 64. At that design point, CAP's coverage is reduced to 24%. This clearly demonstrates the high accuracy and low confidence requirements of PAP predictor.

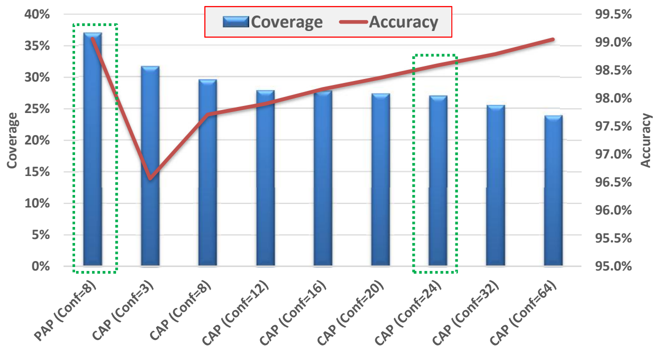


Figure 4: Address prediction accuracy and coverage.

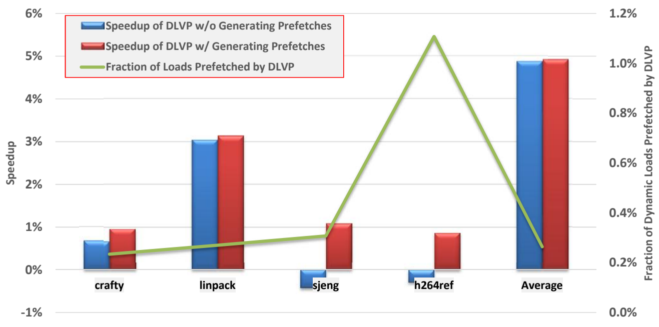


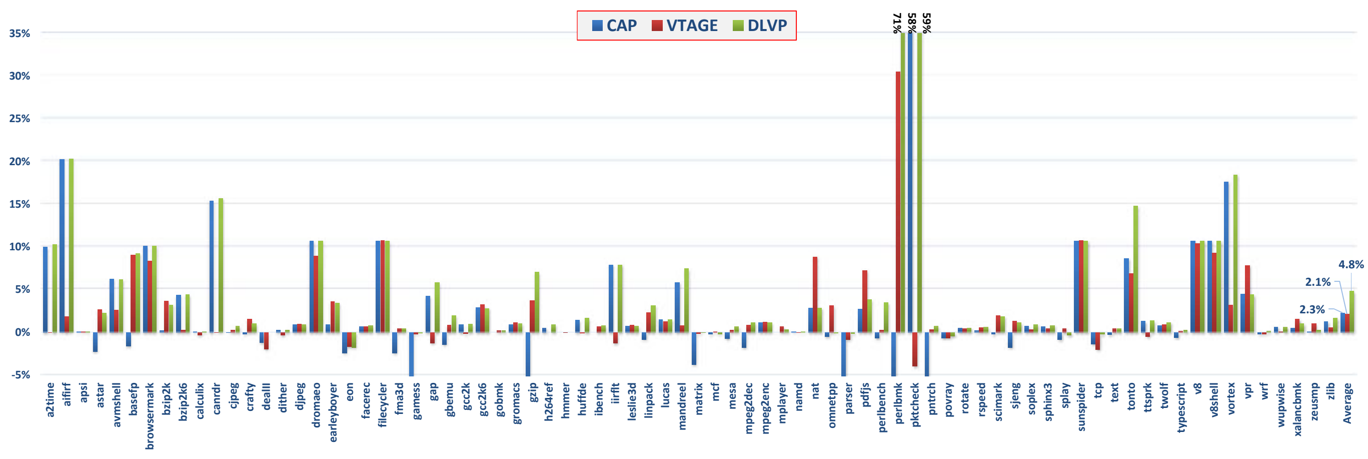
Figure 5: Benefits of DLVP generated prefetches.

5.2 Value Prediction

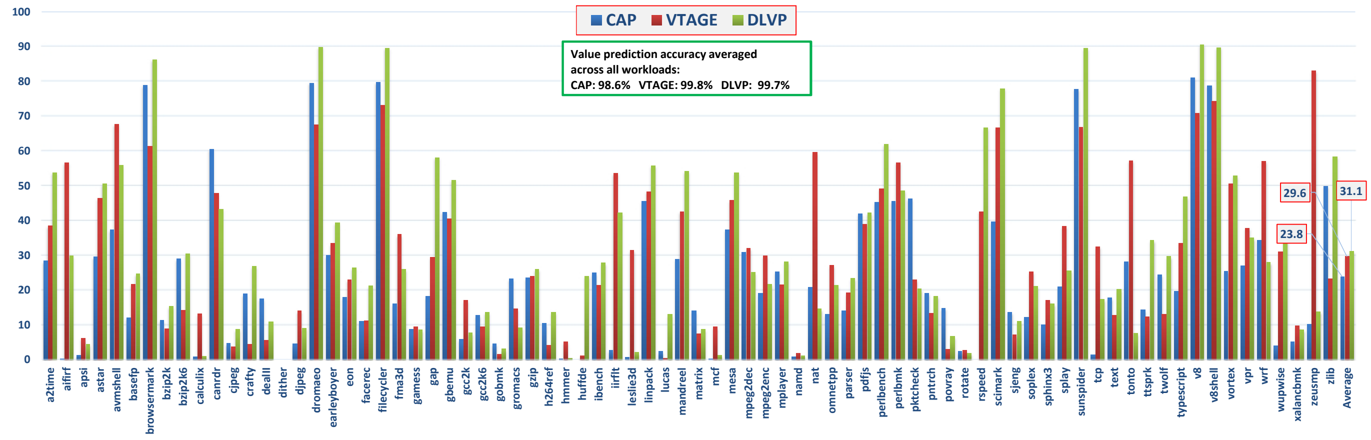
In this section, we present: a detailed evaluation of DLVP, an analysis of state-of-art VTAGE predictor, a comparison between VTAGE, CAP and DLVP, and the integration of DLVP and VTAGE. Finally, we evaluate the benefits of DLVP, CAP and VTAGE when replay is used as the misprediction recovery mechanism.

5.2.1 DLVP. Figure 6a shows the speedup of DLVP (amongst other schemes that we discuss in subsequent sections) on our workloads. DLVP improves 45 out of the 78 workloads by more than 1%, with maximum speedup of 71% on perlbmk, and an average speedup of 4.8%.

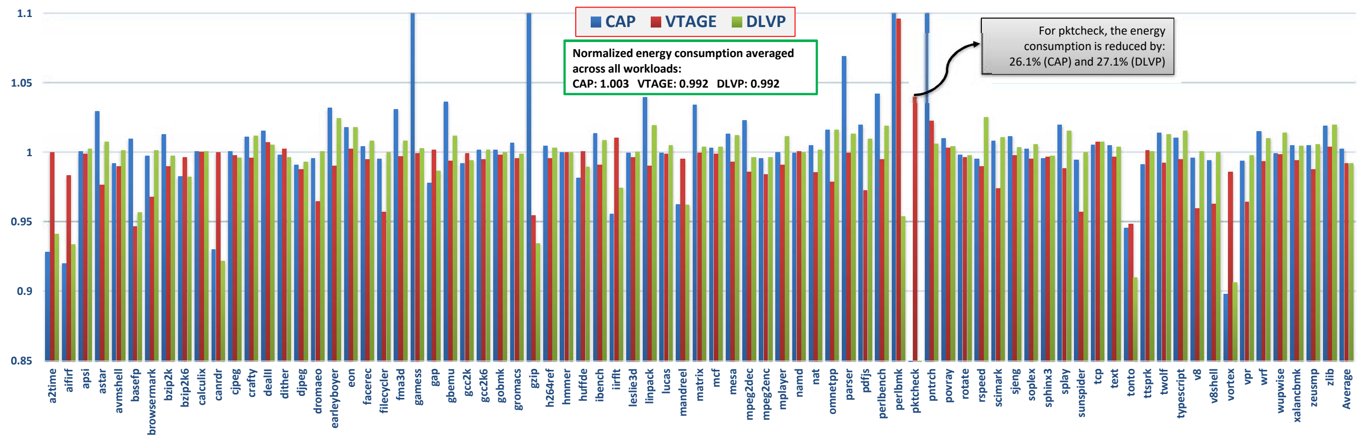
By virtue of predicting memory addresses, unlike conventional value predictors, DLVP is capable of generating high accuracy prefetch requests. We evaluate the impact of enabling or disabling this feature of DLVP in Figure 5. Bars show speedup on a subset of the benchmarks along with the average across all benchmarks. The curve shows the fraction of loads for which DLVP generated a prefetch. Observe that the fraction of loads prefetched by DLVP



(a) Speedup



(b) Coverage



(c) Total Core Energy Normalized to Baseline (No Value Prediction)

	PAP	CAP	VTAGE
Area	1.00	1.75	0.87
Read energy	1.00	1.53	1.91
Write energy	1.00	1.41	0.41

(d) Prediction Table(s) Area and Energy Normalized to PAP (with 2rd/2wr ports)

Figure 6: Comparison of the three prediction schemes: CAP, VTAGE, and DLVP.

is quite low (less than 1.1% for h264ref, and 0.3% on average), which explains the small average performance improvement when this feature is enabled (only 0.1%.)

5.2.2 VTAGE Implementation. We evaluated state-of-art, context-based value predictor VTAGE, on our workloads. Our initial evaluation showed speedups and accuracies far lower than what is reported by Perais and Seznec [28]. We conducted a thorough investigation to uncover the reasons behind this. Our investigation revealed the following interesting interaction between conventional value predictors (including VTAGE) and the ISA. Recall that our benchmarks are compiled for the ARM ISA, while in [28] x86 binaries are used.

The ARM ISA supports many load instructions that have multiple destination registers: for example, load-pair instructions (LDP) have two destination registers, and load-multiple instructions (LDM) can load any subset of the 16 general-purpose registers. With conventional value prediction, the value of each destination register needs to be predicted, requiring up to sixteen predictor entries in the case of LDM (i.e., one predictor entry per destination register). Moreover, vector-load instructions (VLD) load a 128-bit value, which typically gets stored into two 64-bit chunks in the value predictor (consuming two entries.) This exposes a *storage inefficiency* issue with conventional value predictors.

Our experiments with VTAGE predictor show suboptimal accuracy and coverage for the aforementioned types of load instructions, namely: LDP, LDM, and VLD. We adjusted VTAGE so it can handle such loads gracefully, by concatenating the number of destination registers to the PC (which is then hashed with history). E.g., if a load has 2 destination registers, the load PC is concatenated with a 0 (to predict the first destination register) and a 1 (to predict the second destination register). Unfortunately, that did not eliminate the destructive aliasing problem introduced by the significant increase in prediction table pressure. The bottom line is: mispredicting any of the many predicted values will trigger a pipeline flush.

We evaluate three flavors of VTAGE:

(1) *Vanilla VTAGE*: unmodified VTAGE, as described in [28].
(2) *VTAGE augmented with a dynamic opcode filter*: vanilla VTAGE augmented with a table that tracks the prediction accuracy for the different instructions types. Instruction types that exhibit low prediction accuracy (less than 95%) are prevented from getting predicted or updating the prediction tables.
(3) *VTAGE augmented with a static opcode filter*: vanilla VTAGE augmented with a static filter that is preloaded with the instruction types that exhibit low prediction accuracy, namely: LDP, LDM, and VLD. No filter training is required in this case.

We considered using VTAGE to predict load instructions only, or all instructions. Figure 7 shows the speedup, coverage and accuracy of the three VTAGE designs. The results clearly show two things. First, the performance of Vanilla VTAGE improves significantly when a dynamic/static filter is used. Interestingly, using a static filter beats using a dynamic filter. This is due to the cost of training the dynamic filter as it takes time (and mispredictions) to detect the load types that exhibit low prediction accuracy. Second, predicting only load instructions is more rewarding than predicting all instructions, especially when the predictor budget is modest (e.g., 8KB in our evaluation.) The criticality of the predicted instructions comes into play. Load instructions are widely known to be very critical. Therefore, it is not a surprise that under these circumstances (i.e., modest budget), predicting only load instructions is superior to predicting all instructions.

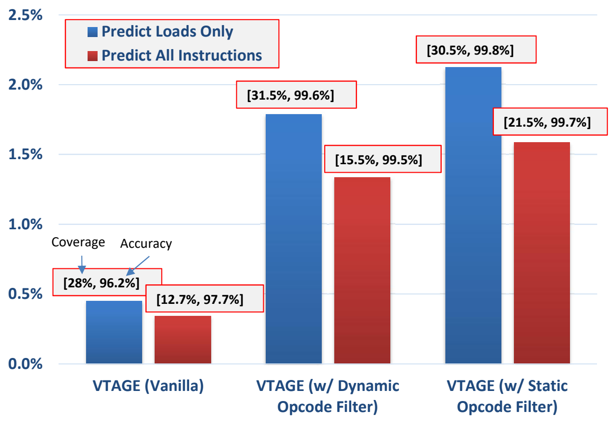


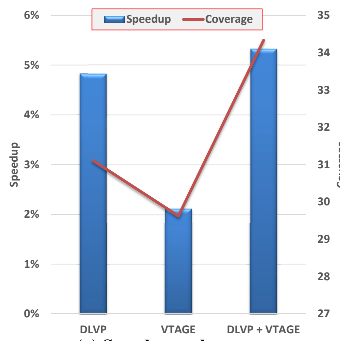
Figure 7: VTAGE evaluation: y-axis shows average speedup across our workloads, x-axis shows the evaluated configurations.

In all of our subsequent evaluations, we use VTAGE with a static filter to predict load instructions only.

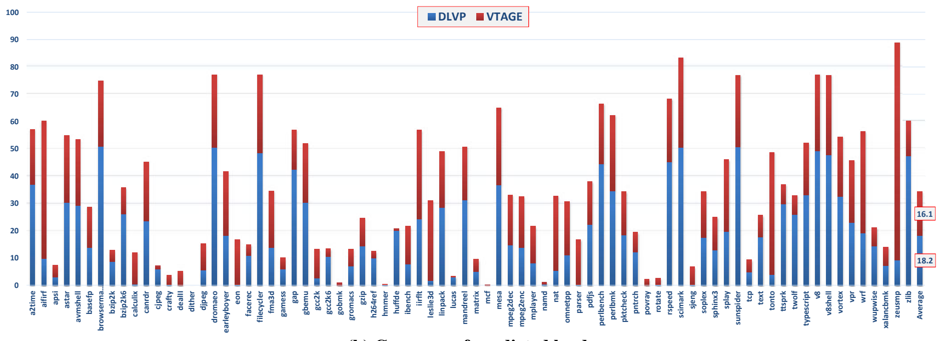
5.2.3 Comparing value predictors. Figures 6a and 6b show the speedup and coverage of the three value predictors: VTAGE, DLVP (uses our proposed PAP predictor), and CAP (just like DLVP except CAP address predictor is used). We swept the confidence level parameter for CAP, found that a confidence of 24 delivers the best average speedup. Therefore, in our evaluation, we use CAP with confidence of 24. For some workloads, CAP delivers a relatively low prediction accuracy (in the low 90s) resulting in some performance degradation. Despite these degradations, CAP delivers an average speedup of 2.3%. Compared to VTAGE, DLVP delivers a higher coverage (31.1% vs. 29.6%), and a significantly higher speedup (4.8% vs. 2.1%). Interestingly, the figure shows benchmarks that favor VTAGE and others that favor DLVP. E.g., *aifirf* favors DLVP, while *nat* favors VTAGE. Observe that DLVP's coverage (31.1%) is lower than PAP's coverage (37%, shown in Figure 4) due to LSCD filtering out loads the did conflict with older, in-flight stores. The same observation applies to CAP, where coverage went from 27% to 23.8%.

Perlbmk presents an interesting case in which the criticality of the predicted values influence the observed speedup. Despite DLVP's lower coverage compared to VTAGE, the speedup of DLVP is much higher due to the positive interaction between value prediction and branch prediction. In DLVP, the value predicted loads facilitate the early resolution of mispredicted branches, thus lowering the branch misprediction cost and magnifying the benefits of value prediction.

Since DLVP predicted loads probe the L1 data cache twice, it's energy consumption increases. The speedups achieved by DLVP allow the core to complete execution sooner, and can reduce the total core energy. Figure 6c shows the total core energy (includes L1 cache



(a) Speedup and coverage



(b) Coverage of predicted loads

Figure 8: Combining DLVP and VTAGE.

and prediction tables) normalized to our baseline (with no value prediction). The data shows that DLVP's speedup more than offset the energy increase due to additional core activity. Interestingly, the average core energy is on par with that of VTAGE, in which no extra cache activity is needed. Figure 6d shows the area, read and write energy of the three predictors, normalized to PAP predictor.

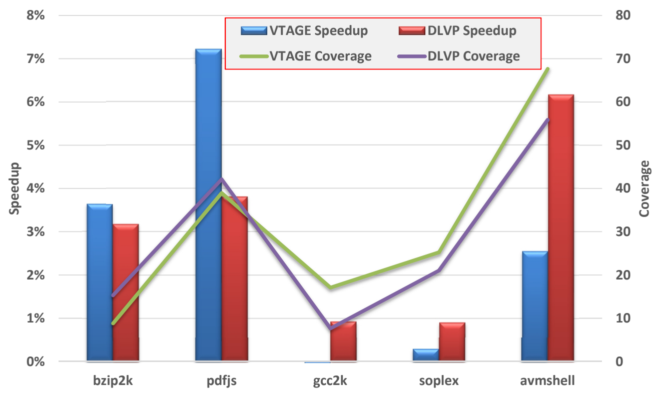


Figure 9: Speedup and coverage of DLVP and VTAGE on selected benchmarks that show no correlation between the two metrics.

Figure 9 shows the speedup and coverage of VTAGE and DLVP for a subset of the benchmarks (namely: *bzip2*, *pdfjs*, *gcc*, *soplex*, and *avmshell*). These benchmarks show an interesting property: the speedup does not seem to correlate with coverage. Upon a closer look, we uncovered the following:

In bzip, DLVP suffers a higher TLB miss rate. This is caused by a second order effect induced by probing the data cache twice for address predicted loads (one time using the predicted address and another time using the computed address). The opposite scenario happens in avmshell, where VTAGE suffers a higher TLB miss rate. In this case, probing the data cache two times (in DLVP) results in better TLB behavior. In pdfjs, the accuracy of VTAGE (100%) is higher than DLVP (99.7%). The opposite happens in gcc and soplex, DLVP accuracy (99.9%) is higher than VTAGE (99.3%).

Combining VTAGE and DLVP.

We evaluate integrating DLVP and VTAGE as tournament predictors: both predictors run concurrently, and a chooser table decides which predictor makes the final prediction. The chooser is PC indexed, and uses 2-bit counters to track which predictor performs better. Figure 8a shows the average speedup and coverage when each predictor is used alone and when combined (tournament predictor). The small increase in coverage, when combining the predictors, clearly show that there is a significant overlap between the loads captured by both schemes. Perhaps a more intelligent chooser can reduce this overlap by partitioning the loads amongst the two predictors. This is an interesting observation and warrants further investigation. Figure 8b shows the breakdown of predicted loads, with respect to the predictor that made the final prediction. On average, DLVP delivers more predictions (18.2%) than VTAGE (16.1%).

5.2.4 Value Misprediction Recovery. Due to the complexity of replay-based value misprediction recovery schemes, we assumed a flush-based recovery microarchitecture, similar to the work of Perais and Seznec [28].

In this section, we entertain the thought of using replay as the recovery mechanism. Instead of throwing away all instructions following a value mispredicted load, one can replay only the load dependents. This change has implications on some design aspects. E.g., consumers of predicted values can no longer leave the instruction queue because they have the potential to replay. This could take away from the benefits of value prediction.

We approximate an oracle replay mechanism in our model, by simply treating mispredictions as no-predictions. i.e., we use oracle information to treat value mispredictions as if the load was never predicted in the first place. Figure 10 shows the speedup of CAP, DLVP and VTAGE with flush and oracle replay as the recovery mechanism. Observe that CAP predictor's performance improves significantly, from 2.3% to 4.2%, with oracle replay. This is expected, because as we stated in Section 5.2.3, CAP delivered relatively low prediction accuracy for some workloads: replacing excessive flushes with less costly replays helps. Meanwhile, the speedup of VTAGE and DLVP predictors improve by 0.7% and 0.8%, respectively. This is expected as well, because both predictors have very high prediction accuracy across all workloads (greater than 99%), and therefore, rarely flush or replay. Note that if we lessen the accuracy requirements for

VTAGE and DLVP, their coverage is likely to increase, and their performance benefits, in a replay-based design, can improve.

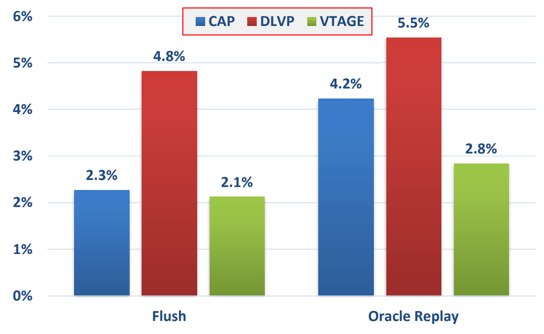


Figure 10: Average speedup across our workloads for two misprediction recovery schemes: flush and oracle replay.

To truly harvest the benefits of replay as a recovery mechanism, one can trade accuracy for higher coverage, and then, identify the sweet spot at which maximum performance can be achieved. We leave this exercise as future work.

6 CONCLUSION

In this paper, we address some of the challenges facing value prediction, namely: the negative interaction between stores and value prediction, and the high cost of value mispredictions which forces value predictors to employ stringent, yet necessary, high confidence requirements.

We proposed Decoupled Load Value Prediction (DLVP), a microarchitecture that targets the challenges of value prediction for load instructions. The basic idea is to replace value prediction with memory address prediction then, rely on the data cache to deliver the predicted values early enough so value prediction can take place. Since the values captured in the data cache mirror the current program state (except for in-flight stores), this can eliminate most of the negative interactions between stores and value prediction. Moreover, we proposed Path-based Address Prediction (PAP), a new context-based address prediction scheme that leverages load-path history to enable high address prediction accuracy with less stringent confidence requirements.

We demonstrate the benefits of DLVP using a wide range of workloads, and show how it compares favorably to other address and value prediction schemes. DLVP delivers an average speedup of 4.8%, which is more than twice the speedup of state-of-art value predictor VTAGE, without increasing the core energy consumption.

7 ACKNOWLEDGEMENTS

We thank Arthur Perais, Mikko Lipasti, Derek Hower, Eric Rotenberg, Anil Krishna, Vignyan Reddy, David Palframan, and the anonymous reviewers for comments that greatly improved the manuscript. This research was supported by Qualcomm Technologies, Inc. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of Qualcomm Technologies, Inc.

REFERENCES

[1] 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)*. ACM, New York, NY, USA, 165–176. https://doi.org/10.1145/1787275.1787321

[2] T. M. Austin and G. S. Sohi. 1995. Zero-cycle loads: microarchitecture support for reducing load latency. In *Microarchitecture, 1995., Proceedings of the 28th Annual International Symposium on*. 82–92. https://doi.org/10.1109/MICRO.1995.476815

[3] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz, and U. Weiser. 1999. Correlated load-address predictors. In *Computer Architecture, 1999. Proceedings of the 26th International Symposium on*. 54–63. https://doi.org/10.1109/ISCA.1999.765939

[4] Browser benchmark. [n. d.]. Browsermark. In *http://web.basemark.com*.

[5] 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)*. IEEE Computer Society, Washington, DC, USA, 64–74.

[6] Nathan Chong and Samin Ishtiaq. 2008. Reasoning About the ARM Weakly Consistent Memory Model. In *Proceedings of the 2008 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness: Held in Conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08) (MSPC '08)*. ACM, New York, NY, USA, 16–19. https://doi.org/10.1145/1353522.1353528

[7] G. Z. Chrysos and J. S. Emer. 1998. Memory dependence prediction using store sets. In *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No.98CB36235)*. 142–153. https://doi.org/10.1109/ISCA.1998.694770

[8] Dromaeo: Javascript Performance Testing. [n. d.]. Dromaeo. In *http://dromaeo.com*.

[9] R. J. Eickemeyer and S. Vassiliadis. 1993. A load-instruction unit for pipelined processors. *IBM Journal of Research and Development* 37, 4 (July 1993), 547–564. https://doi.org/10.1147/rd.374.0547

[10] R. J. Eickemeyer and S. Vassiliadis. 1993. A load-instruction unit for pipelined processors. *IBM Journal of Research and Development* 37, 4 (July 1993), 547–564. https://doi.org/10.1147/rd.374.0547

[11] Ian Forsyth. 2015. The ARM Cortex-A72 processor: Delivering high efficiency for Server, Networking and HPC. Presented at ARM TechDay, Londom.

[12] Freddy Gabbay. 1996. *Speculative Execution based on Value Prediction*. Technical Report. EE Department TR 1080, Technion - Israel Institue of Technology.

[13] 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)*. ACM, New York, NY, USA, 196–203. https://doi.org/10.1145/263580.263631

[14] Google Octane Benchmark. [n. d.]. Octane. In *https://developers.google.com/octane*.

[15] Google V8 Benchmarks. [n. d.]. V8. In *http://code.google.com/apis/v8/benchmarks.html*.

[16] M. D. Hill and M. R. Marty. 2008. Amdahl's Law in the Multicore Era. *Computer* 41, 7 (July 2008), 33–38. https://doi.org/10.1109/MC.2008.209

[17] iBench Benchmark. [n. d.]. iBench. In *http://ibench.sourceforge.net*.

[18] R. E. Kessler. 1999. The Alpha 21264 microprocessor. *IEEE Micro* 19, 2 (Mar 1999), 24–36. https://doi.org/10.1109/40.755465

[19] Linpack Benchmark. [n. d.]. Linpack. In *http://www.netlib.org/benchmark/hpl*.

[20] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. 1996. Value Locality and Load Value Prediction. *SIGPLAN Not.* 31, 9 (Sept. 1996), 138–147. https://doi.org/10.1145/248209.237173

[21] Julius Mandelblat. 2015. Technology Insight: Intel's Next Generation Microarchitecture Code Name Skylake. Presented at Intel Developer Forum, San Francisco.

[22] M. M. K. Martin, D. J. Sorin, H. W. Cain, M. D. Hill, and M. H. Lipasti. 2001. Correctly implementing value prediction in microprocessors that support multithreading or multiprocessing. In *Microarchitecture, 2001. MICRO-34. Proceedings. 34th ACM/IEEE International Symposium on*. 328–337. https://doi.org/10.1109/MICRO.2001.991130

[23] Media Player Benchmark. [n. d.]. MPlayer. In *http://mplayerhq.hu/design7/dload.html*.

[24] 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)*. ACM, New York, NY, USA, 181–193. https://doi.org/10.1145/264107.264189

[25] A. Moshovos and G. S. Sohi. 1997. Streamlining inter-operation memory communication via data dependence prediction. In *Proceedings of 30th Annual International Symposium on Microarchitecture*. 235–245. https://doi.org/10.1109/MICRO.1997.645814

[26] Ravi Nair. 1995. Dynamic Path-based Branch Correlation. In *Proceedings of the 28th Annual International Symposium on Microarchitecture (MICRO 28)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 15–23.

[27] Paul E. McKenney. 2010. Memory Barriers: a Hardware View for Software Hackers. In *Linux Technology Center, IBM Beaverton*.

[28] A. Perais and A. Seznec. 2014. Practical data value speculation for future high-end processors. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. 428–439. https://doi.org/10.1109/HPCA.2014.6835952

[29] A. Perais and A. Seznec. 2015. BeBoP: A cost effective predictor infrastructure for superscalar value prediction. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 13–25. https://doi.org/10.1109/HPCA.2015.7056018

[30] J. A. Poovey, T. M. Conte, M. Levy, and S. Gal-On. 2009. A Benchmark Characterization of the EEMBC Benchmark Suite. *IEEE Micro* 29, 5 (Sept 2009), 18–29. https://doi.org/10.1109/MM.2009.74

[31] N. Riley and C. Zilles. 2006. Probabilistic counter updates for predictor hysteresis and stratification. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*. 110–120. https://doi.org/10.1109/HPCA.2006.1598118

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

[33] Y. Sazeides and J. E. Smith. 1997. The predictability of data values. In *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on*. 248–258. https://doi.org/10.1109/MICRO.1997.645815

[34] Yiannakis Sazeides, Stamatis Vassiliadis, and James E. Smith. 1996. The Performance Potential of Data Dependence Speculation & Collapsing. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 29)*. IEEE Computer Society, Washington, DC, USA, 238–247.

[35] SciMark: Java benchmark for scientific and numerical computing. [n. d.]. SciMark. In *http://math.nist.gov/scimark2/*.

[36] André Seznec. 2011. A 64-Kbytes ITTAGE indirect branch predictor. In *Third Championship Branch Prediction (JWAC-2)*.

[37] André Seznec. 2011. A New Case for the TAGE Branch Predictor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. ACM, New York, NY, USA, 117–127. https://doi.org/10.1145/2155620.2155635

[38] Standard Performance Evaluation Corporation. [n. d.]. The SPEC CPU 2000 Benchmark Suite. In *http://www.spec.org*.

[39] Standard Performance Evaluation Corporation. [n. d.]. The SPEC CPU 2006 Benchmark Suite. In *http://www.spec.org*.

[40] Sunspider Javascript Benchmark. [n. d.]. Sunspider. In *http://www.webkit.org/perf/sunspider/sunspider.html*.

[41] Gary S. Tyson and Todd M. Austin. 1999. Memory Renaming: Fast, Early and Accurate Processing of Memory Communication. *Int. J. Parallel Program.* 27, 5 (Oct. 1999), 357–380. https://doi.org/10.1023/A:1018734923512

[42] V. Zyuban and P. Kogge. 1998. The energy complexity of register files. In *Proceedings. 1998 International Symposium on Low Power Electronics and Design (IEEE Cat. No.98TH8379)*. 305–310. https://doi.org/10.1145/280756.280943