Efficient Load Value Prediction using Multiple Predictors and Filters

Rami Sheikh, Derek Hower
Qualcomm Technologies, Inc.
{ralsheik, dhower}@qti.qualcomm.com

***Abstract*—Value prediction [1], [2] has the potential to break through the performance limitations imposed by true data dependencies. Aggressive value predictors can deliver significant performance improvements, but usually require large hardware budgets. While predicting values of all instruction types is possible, prior work has shown that predicting just load values is most effective with a modest hardware budget (*e.g.*, 8KB of prediction state [3], [4]). However, with hardware budget constraints and high prediction accuracy requirements (99%), prior work has struggled to increase the fraction of predicted loads (coverage) beyond the low 30s.**

In this paper, we analyzed four state-of-the-art load value predictors, and found that they complement each other. Based on that finding, we evaluated a new composite predictor that combines all four component predictors. Our results show that the composite predictor, combined with several optimizations we proposed, improve the benefit of load value prediction by 54%-74% depending on the total predictor budget. Moreover, our composite predictor delivers more than twice the coverage of first championship value prediction winner predictor (EVES [4]), and substantially increases the delivered speedup by more than 50%.

I. Introduction

As microarchitectures continue to extract more Instruction Level Parallelism (ILP) from increasing out-of-order scheduling windows (*e.g.*, 97 instructions in Intel Skylake [5]), performance is increasingly limited by true data dependencies in a program. Value prediction is a technique to break those true dependencies by allowing consumer instructions to speculatively execute ahead of their producer. Value prediction works because instructions exhibit *value locality*, meaning that the same static instruction often produces a predictable value [1]. In the case of load instructions, it is also possible to predict a load memory address, followed by a data cache access, to generate a speculative value that does not necessarily exhibit value locality (e.g., DLVP [3]). While value predictors can generate speculative results for all instruction types, recent work has shown that load-only predictors are most efficient with a modest hardware budget (*e.g.*, 8KB) [3], [4].

In this study, we investigated techniques to increase the effectiveness of load value prediction. First, we thoroughly analyzed and compared four state-of-the-art load value predictors[1], shown in Table I, to determine how they complement one another. We found that no individual predictor is strictly better than another since they all target loads with different characteristics. Further, we found that a composite predictor that uses all four variants simultaneously outperforms any single predictor in isolation for the same hardware state budget.

Table I: Four component load value predictors.

	Predicts	
	Load values	**Load addresses**
Context agnostic	Last Value Prediction (LVP) [1]	Stride Address Prediction (SAP) [6]
Context aware	Context Value Prediction (CVP) [7], [8]	Context Address Prediction (CAP) [3]

Second, we investigated techniques to increase the effectiveness of a composite load value predictor. We improved a composite predictor by:

- Adding an Accuracy Monitor (AM) that throttles component predictors when their accuracy drops below a threshold,
- Using heterogeneous component predictor table sizes to find the best use of limited resources,
- Using a smart training algorithm that avoids redundantly training multiple predictors that are equally effective, and
- Dynamically fusing predictor tables to reallocate resources from under-performing predictors to better-performing predictors.

After combining all the techniques, we found that a composite predictor can best the performance of a single component predictor of the same size by 54%-74%, depending on the total predictor budget. We found that many of the optimizations are most effective at small to modest predictor sizes, thus allowing designers to trade off design complexity and area with similar performance. Moreover, we show that our composite predictor significantly outperforms the winner of first championship value prediction (EVES [4]) in terms of coverage (more than doubles) and speedup (increases by more than 50%).

In the rest of this paper, we present the results of a comprehensive design space exploration of composite load value prediction. We present the details of our methodology and evaluation framework in Section II. Then, we explore the theory and performance of the four state-of-the-art

[1]We analyzed several other predictors, like last address and stride value predictors. These predictors showed limited or no benefit in the presence of the four selected predictors.

Benchmark Suite	Applications
EEMBC (ARMv7)	a2time, aifirf, basefp, bezier, canrdr, cjpeg, coremark, dither, djpeg, fbital, huffde, iirflt, matrix, mp3player, mp4dec, mp4enc, mpeg2dec, mpeg2enc, nat, pktcheck, pntrch, rotate, routelookup, rspeed, typescript
Miscellaneous (ARMv7)	browsermark, ibench, linpack, mplayer
Octane/Javascript (ARMv7)	avmshell, codeload, dromaeo, earleyboyer, filecycler, gbemu, mandreel, pdfjs, regex, scimark, splay, sunspider, v8, v8shell, zlib
SPEC2K (ARMv8)	apsi, bzip2k, crafty, eon, equake, 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 II: Applications used in our evaluation.

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

Table III: Baseline core configuration.

component predictors in Sections III and IV, including an analysis of how the predictors overlap. Section V contains the design and analysis of a composite predictor, including the four aforementioned optimizations. Finally, we conclude in Section VI.

II. Methodology and Evaluation Framework

A. 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 [9], SPEC2K6 [10], and EEMBC [11]. Moreover, we enriched our benchmark pool with other popular applications: Linpack [12], media player [13], browser benchmark [14], and various Javascript benchmarks [15], [16], [17], [18], [19], [20].

Table II 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.

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.

Unless otherwise noted, we present results as the arithmetic average across all workloads (geometric for IPC).

B. Evaluation Framework

All results were collected using 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 do not release it, and there is nothing publicly available that we can cite.

The parameters of our baseline core are configured as close as possible to those of Intel's Skylake core [5]. The baseline core uses state-of-art TAGE and ITTAGE branch predictors [21], [22], and a memory dependence predictor similar to Alpha 21264 [23]. We use a fetch-to-execute latency of 13 cycles. Table III shows the baseline core configuration.

III. Value Prediction

Value prediction enables speculation past true data dependencies in a program. It is effective because the values produced by dynamic instances of the same static instruction exhibit value locality. For example, some instructions generate a constant value for a computation while others might normally produce the same return value indicating successful completion. The result of any instruction type can be predicted, though in this paper we focus only on predicting load values since that is most effective with limited hardware resources [3], [4].

A. Value Prediction Approaches

There are two general approaches to load value prediction, which we discuss briefly below. The goal of both is to have a predicted value ready by the time any consumer of the load enters the instruction queue (aka the scheduler). If the predictors can get a correct value in time, then consumers can execute immediately, making it appear that the load has a zero-cycle load-to-use latency. Predictions are validated when the predicted load executes. If a prediction is found to be incorrect, recovery actions take place. In this work we assume a flush-based recovery microarchitecture, similar to the work of Perais and Seznec [7], [8]. Because the cost of a misprediction is usually high, it is important for value predictors to deliver very high accuracy (*e.g.*, 99% of predictions correct).

One approach to load value prediction is to directly predict the value that a static load will produce (*e.g.*, the load at PC *X* always returns zero). In Figure 1, we show the changes to support this style of value prediction in a baseline architecture. In step ❶, the value predictor is probed as a load is fetched, and if a high confidence prediction is found, the value is forwarded to the Value Prediction Engine (VPE) in step ❷ (we discuss the filter later when combining predictors).[2] VPE provides the mechanism needed to communicate the predicted values from the value-predicted producers to their consumers [3]. Consumers of the load can use the prediction by reading the stored value out of the VPE rather than waiting on a physical register to be ready. When the load executes, the correct value is read from the data cache and is validated

[2]We could wait and probe the value predictor at rename, but choose to place the predictor access in fetch for parity with the design of an address predictor. The placement in the front-end allows for more slack for prediction while increasing the prediction-to-update latency.

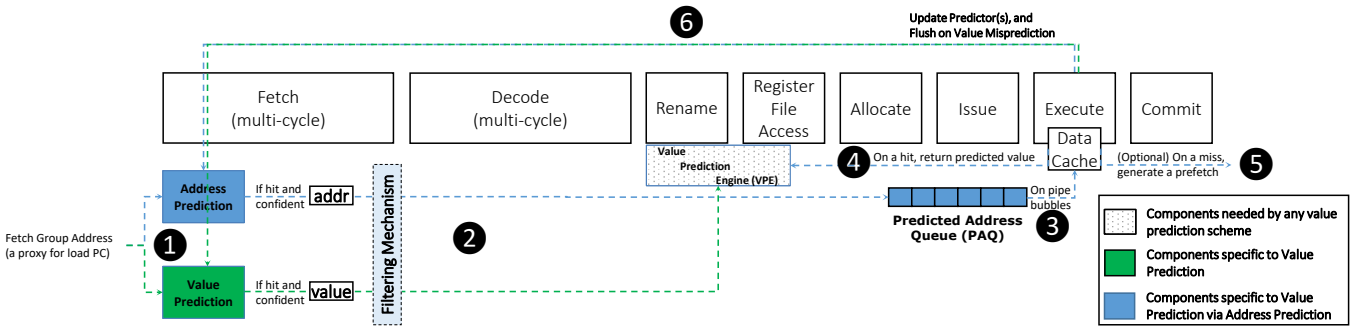


Figure 1: Pipeline with support for our proposed value prediction scheme.

against the speculative value. The predictor updates in step ❻ and, if a misprediction is detected, the affected instructions are flushed and fetch is redirected to the recovery address.

Another approach to load value prediction is to use address prediction, combined with a data cache read, to generate a speculative value. Load value prediction through address prediction is similar to data prefetching except that the address is predicted when a load is fetched so that its data can be ready in the pipeline by the time any consumer of the load enters the scheduler. Address predictors are probed when a load is fetched in step ❶, and if a high confidence prediction is found, the address is forwarded to the Predicted Address Queue (PAQ) in step ❷. The PAQ waits for bubbles in the load pipeline, and when it finds one, probes the data cache with a predicted address in step ❸. If the address hits in the data cache, then in step ❹, the value is forwarded to the VPE. As long as the data returns before a consumer of the load reaches rename, the load will appear to have a zero-cycle load-to-use latency. If the predicted address misses in the data cache, we can optionally generate a data prefetch request in step ❺ to accelerate the eventual execution of the predicted load (this feature is *disabled* in our work). If a predicted value is used, then the value (note, checking the address is insufficient as the value may have changed) of the load must be checked when the load executes, and if the speculative value was incorrect, a misprediction recovery is initiated in step ❻.

To learn more about the two load value prediction approaches and the recent advances towards practical implementations of value prediction, we encourage the readers to visit prior art papers [3], [7], [8].

Regarding memory consistency, ARM's relaxed consistency model allows for reordering most memory operations with one exception: dependent loads are not allowed to be reordered [24], [25]. Value prediction can violate this rule. To avoid violating ARM's memory consistency model, we employ a technique similar to the work of Martin *et al.* [26]. Also, address/value prediction is not used with memory ordering instructions, atomic and exclusive memory accesses.

B. Value and Address Prediction Strategies

We studied four different value predictors that we believe cover the current state of the art. We discuss how each predictor works, and then analyze the load types they target, where the predictors overlap, and how efficient each is in terms of hardware storage and power. We group predictors based on whether or not they consider program history (context) when making a prediction since doing so tends to increase accuracy but also decrease storage efficiency.

While the high-level concept of each predictor is based on published literature, the details (size, confidence, etc.) are based on tuning from our own evaluation (details in Section II). Because the cost of a value misprediction is so high, we tuned each predictor to achieve 99% accuracy (thereby sacrificing coverage). Our evaluation showed that lower accuracy tends to decrease performance gains. Readers may notice that our reported benefit for each design is sometimes significantly lower than previously published results; this is caused by different assumptions about the baseline ISA, microarchitecture, and storage constraints (Sheikh reports similar findings [3], [27]).

In all of the studied predictors, we use forward probabilistic counters to reduce the number of bits needed to track confidence [28]. Using FPC, a confidence counter is incremented with probability P, where P is found by indexing the FPC vector in Table IV with the current confidence value. By probabilistically incrementing, FPC lets us represent, for example, a $0..N-1$ counter using fewer than $log_2 N$ bits. To determine the FPC vector size and values, we first ran experiments using scalar confidence counters to determine the smallest confidence level that leads to 99% accuracy, and then construct an FPC vector that gives the same effective confidence using fewer bits.

1) Context-agnostic Predictors: **Last Value Prediction (LVP)** A last value predictor exploits the fact that consecutive dynamic instances of a static load will often produce the same value. This commonly occurs, for example, with PC-based loads that read large constants. The pattern can also occur when dynamic instances of a static load produce different addresses, such as when sequencing through an array

Table IV: Predictor parameters. We empirically determined confidence thresholds based on a 99% accuracy target. The confidence is listed both as the absolute threshold (*i.e.*, the counter value) and the effective level considering FPC (*i.e.*, the expected number of observations before achieving high confidence).

	bit fields / entry							Conf		FPC Vector	History Length	@ 1K entries		
	Total	Tag	Value	Addr	Conf	Stride	Size	Thold	Eff			Size (KiB)	Speedup	Speedup / KiB
LVP	81	14	64		3			7	64	$\frac{1}{2}, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{16}, \frac{1}{16}$		10.13	3.3%	0.3%
SAP	77	14		49	2	10	2	3	9	$1, \frac{1}{4}, \frac{1}{4}$		9.63	5.2%	0.5%
CVP	81	14	64		3			7	16	$\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{4}$	5, 13, 23	10.13	3.8%	0.4%
CAP	67	14		49	2		2	3	4	$1, 1, \frac{1}{2}$	16	8.38	6.5%	0.8%

just initialized with memset. Last value predictors can be viewed as members of the stride value predictors [29] family, where the stride is zero. In our evaluation, we observed very limited presence of stride loaded values (though did find strided values for other instruction types such as arithmetic instructions), therefore, we excluded stride value predictors from our pool of candidate predictors.

LVP uses a PC-indexed, tagged prediction table. Each entry contains a 14-bit tag, 64-bit value, and a 3-bit saturating confidence counter, for a total of 81 bits per entry. LVP is trained when a load executes by hashing the PC bits of a load to access an entry and then updating the entry's tag and value. If the new tag/value match the existing tag/value, then we probabilistically increase the confidence; otherwise, the confidence is reset to zero.

To make a prediction, the PC of a newly fetched load is hashed to access an entry, and if the tag matches and the confidence is above the threshold, then the stored value will be used as a prediction. We find, like other work before [7], [8], that LVP needs a high confidence to avoid reducing performance through mispredictions, and therefore use a confidence threshold of 7, which, with the FPC vector shown in Table IV, corresponds to an effective confidence of 64 consecutive observations of a value.

Stride Address Prediction (SAP) A stride address predictor identifies static loads that produce strided addresses (possibly with stride = 0), and then probes the data cache to retrieve a predicted value. The stride detection logic is similar to the logic in a stride-based data prefetcher.

In our implementation of SAP, we maintain a PC-indexed, tagged prediction table. Each entry contains a 14-bit tag, a 49-bit virtual address representing the last known load address for the PC, a 2-bit saturating confidence counter, a 10-bit stride, and a 2-bit load size indicator, for a total of 77 bits per entry.

To train SAP, when a load executes, it hashes the PC to identify a predictor table entry, writes the delta between the load address and the last known load address into the stride field, and updates the size field to the log base two of the load width. If the tag entry matches and the calculated stride equals the stored stride, then the confidence counter is incremented; otherwise, the confidence counter is reset to zero. Because short-lived strides cause many mispredictions, we found that a modest confidence of 9 consecutive observations was necessary to achieve 99% accuracy.

After confidence is high, SAP produces a predicted address by adding the last known load address to the stride and sends the address to the PAQ where it will wait for a pipeline bubble and probe the data cache. The returned value is used to speculate while the predicted value is verified. Similar to the enhancements described for the stride value predictor in EVES[4], SAP takes into account the number of inflight occurrences of the load instruction, when making a prediction.

2) Context-aware Predictors: **Context Value Prediction (CVP)** A context value predictor uses program history along with load PC to generate more accurate predictions. CVP is inspired by branch prediction, which has long observed that branch behavior is correlated with the path history leading to the branch. The seminal work on CVP found that the same holds true for all instruction values [7], [8], and subsequent work confirmed the same is true for load instructions in particular [3], [4].

Our implementation of CVP is similar to VTAGE value predictor [7] except that we excluded the untagged, last-value table. The predictor keeps three tables, all of which are indexed using a hash of the PC and a geometric sample of the branch path history. Each table entry stores a 14-bit tag, a 64-bit value, and 3-bit saturating confidence counter, for a total of 81 bits (same of LVP). When a load executes, all three tables are updated in a similar manner as LVP to train the predictor. When predicting a value, CVP uses a value from the table with the longest history whose entry has high confidence. Like LVP, CVP requires high confidence to build high accuracy, so we use forward probabilistic counters and set the confidence threshold to 4, corresponding to 16 consecutive observations.

Context Address Prediction (CAP) A context address predictor also uses program history along with load PC to generate more accurate predictions, but uses the data cache as a value store rather than directly generating values from

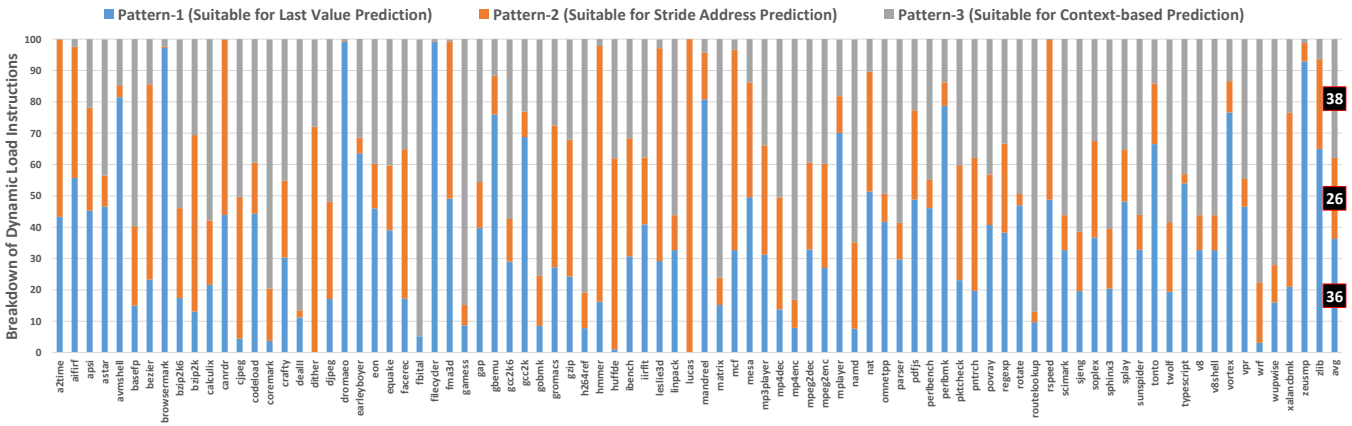


Figure 2: Load breakdown by pattern

the predictor. Prior work found that load addresses correlate more closely to global load path history (prior *N* load PCs) rather than branch path history.

We use the state-of-the-art DLVP predictor as a reference design [3]. The predictor consists of one tagged table indexed by a hash of PC and load path history. An entry contains a 14-bit tag, a 49-bit virtual address, a 2-bit confidence, and a 2-bit load size, for a total of 67 bits. When a load completes, it updates the table by setting the tag, value, and size. If the new tag, value, and size match the existing entry, the confidence is incremented; otherwise, the confidence is reset to zero. A prediction is made when a fetched load has a tag match and confidence is high. CAP has the lowest confidence threshold of all predictors, corresponding to four consecutive observations of a give path/load PC.

Please note that, for all of the studied predictors, the total storage can be considerably reduced by employing optimizations similar to the ones described for the enhanced VTAGE implementation in [4] (e.g., decoupling the value/address arrays and then sharing them among the predictors). We believe that employing such optimizations is outside the scope of this work, and that it will not impact the findings presented.

IV. Component Predictor Analysis

A. Load Classification

Each of the four studied load value predictors target different load characteristics. LVP is targeted at static loads that produce the same value. SAP targets static loads that produce a predictable (strided) address. CVP and CAP target dynamic loads that, when taken in context, produce predictable values or addresses, respectively.

We analyzed all benchmarks to determine the breakdown of loads covered by each of the four predictors. We placed dynamic instances of a load into one of three groups:

- **Pattern-1 (LVP proxy)**: The PC of the load highly correlates with the load value
- **Pattern-2 (SAP proxy)**: The PC of the load highly correlates with the load address
- **Pattern-3 (CVP/CAP proxy)**: All other loads

The classification was performed using infinite resources – *i.e.*, we perfectly remember load values/addresses. The patterns are ordered and exclusive, such that any load fitting Pattern-1 will not be considered for Pattern-2 or Pattern-3. We prioritized the patterns based on the preference of their proxy predictors: *value before address and context-unaware before context-aware.* We prefer load value predictors first because they do not require a data cache access, and are not susceptible to cache misses, cache bandwidth constraints, and the increased power of a cache access. We prefer context-unaware predictors because they are more storage-efficient; a single entry in a context-unaware predictor can cover more dynamic loads than a single entry in a context-aware predictor.

As Figure 2 shows, the breakdown of loads is almost evenly split between Pattern-1 (LVP), Pattern-2 (SAP), and Pattern-3 (CAP and SAP). This result motivates a composite predictor design. Even though loads from a lower-numbered pattern may also be covered by a higher-number pattern, using the proxy predictor from a higher-number pattern would result in a less efficient design. We explore predictor overlap more in Section V-A while studying composite prediction.

B. Speedup

We evaluated the speedup from each component predictor in isolation using the confidence values in Table IV, which were chosen empirically to hit a 99% accuracy target. Figure 3 shows the performance improvement of each predictor as we scale the storage budget (by scaling the number of table entries from 64 - 4K entries).[3] We observed that all four predictors hit a performance knee around 1K table entries (corresponds to 8-10KB storage), and therefore assume 1K entries as a starting baseline going forward. Even scaling out

[3]For CVP, total entries is the sum of the size of each of the three tables.

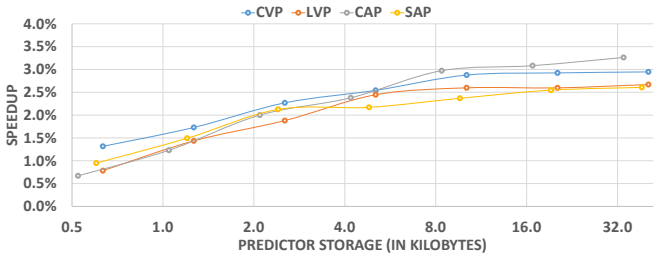


Figure 3: Speedup (geometric mean over all workloads) of each predictor as the number of entries scales up.

to 32K entries (not shown), which would rival the size of most L2 caches, predictors only see modest gain.

C. Predictor Overlap

Even though all four predictors use different strategies to predict load values, there will still be overlap in the predictions they provide. While the overlap may be an indication of inefficient resource usage in a composite predictor, it can at times be beneficial due to effects like training time or aliasing.[4] In this subsection, we explain this subtlety through a simple illustrative example.

```
1for (o=0; o < M; o++) {
2  memset(A, 0, N*sizeof(*A));
3
4  for (i=0; i < N; i++) {
5    auto load = A[i];
6    ...
7  }
8}
```

Listing 1: Simple loop

Consider the loop in Listing 1. The load on line 5 is highly predictable by all four value predictors, though each differ in how long it takes to generate a prediction and for how many iterations through the inner loop the prediction stays valid. SAP will quickly find a stride (equal to the size of an A element) and begin predicting after nine completed loads (i.e., when the effective confidence is reached). SAP has to retrain on each iteration of the outer loop since restarting breaks the stride. CAP will never predict when $i >= 16$ because at that point the load path history does not change but the address does. For iterations $i < 16$, CAP will establish entries in the prediction table and then build confidence in these entries over consecutive iterations of the outer loop. When $o > 4$, CAP will predict all iterations of $i < 16$. LVP will produce a correct prediction as soon as its high confidence threshold is met (*e.g.*, after encountering 64 instances of the load). LVP does not need to retrain on subsequent iterations of the outer loop, and can predict the load on the first iteration of the inner loop once the high confidence threshold is reached (e.g., when $o > 0$.) CVP will produce a correct prediction when enough iterations execute to fill the branch history register of the smallest CVP table (*e.g.*, 5 iterations) and after enough loads execute to build confidence (*e.g.*, 16 loads with same 5-bit branch history). E.g., CVP will predict iterations with $i > 22$ because when $i > 5$ the 5-bit history remains the same, and since the value is fixed, CVP will be confident after 22 iterations (6 iterations + 16 iterations). Also, for iterations of $o > 16$, CVP will predict all load values, including iterations of $i <= 22$.

[4]In their seminal work, Sheikh et al. proposed a hardware-only fault attack mitigation scheme that leverages the overlap between value predictors to reverse the trust model, trusting the predicted value over the faulted value [30], [31], [32].

Table V: Number of loads from inner loop of Listing 1 that must complete before making a prediction, assuming no predictor aliasing and for various iterations of the outer loop. A dash means that the predictor never makes a prediction, and a zero means that a prediction is made on the first iteration of the inner loop.

	$o = 0$	$o = 1$	$o = 6$	$o = 17$
SAP	9	9	9	9
CAP	-	-	0*	0*
LVP	64	0	0	0
CVP	22	22	22	0

* Only predicts first 16 iterations of inner loop

Note that due to pipelining and/or superscalar effects, the first iteration of the inner loop to see a benefit from value prediction may be larger than the value listed in Table V. For example, in SAP, by the time the ninth load completes to build confidence in the entry, some number of iterations greater than nine may have already been fetched and will therefore not be value predicted. In an aggressive machine, where value prediction shows the most benefit, this pipelining effect may be quite large. This highlights why having complementary predictors can often be beneficial. Consider a machine with SAP and CAP; CAP can predict the first 16 iterations while SAP trains (which could take close to 16 iterations).

V. Efficient Load Value Prediction

In this section, we present the results of a comprehensive design space analysis of a system using multiple load value predictors and filters. We begin by presenting the results of a simple composite predictor that simultaneously uses all four component predictors. Then we refine that design by adding a prediction filter, adjusting component table sizes, improving the training algorithm, and dynamically fusing component resources.

A. Composite Predictor

We first combined all four of the predictors from Section III to create a composite predictor. All four components train in parallel, and we use a prediction from any predictor that is highly confident (using thresholds from Table IV). Because

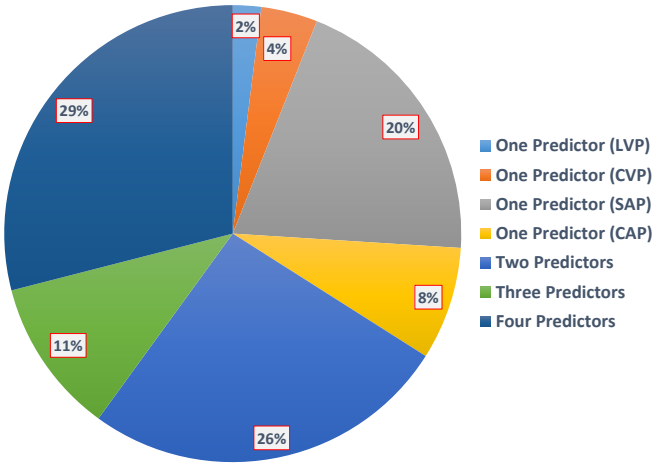


Figure 4: Percent of dynamic loads in the benchmark mix predicted by one, two, three, or four predictors when using 1K-entry tables for each of the predictor types.

all predictors are tuned for 99% accuracy, highly confident predictors rarely disagree (results, not shown, confirm that highly-confident predictors disagree less than 0.03% of the time). Therefore, choosing among highly-confident predictors has little impact on performance. However, there is a power implication; when multiple predictors are confident, we prefer the result of a value predictor first since it is most power efficient (e.g., there is no need to speculatively access the data cache), and then chose context-aware over context-agnostic (for accuracy reasons).

Figure 4 shows which components produce high-confidence predictions across all loads in the benchmark mix. We see that there is significant overlap – 66% of loads are predicted by more than one component. We also find that the address predictors (SAP and CAP) pick up most of the loads that can only be predicted by one predictor type, indicating that many more loads have predictable address patterns than predictable value patterns. As we previously established, however, such overlap is not always wasteful.

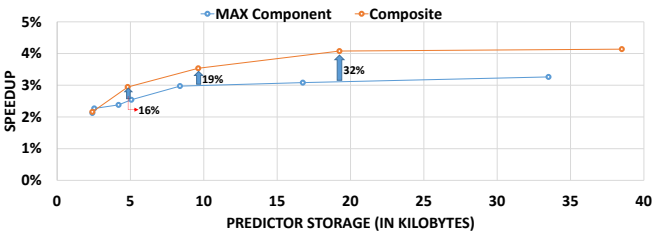


Figure 5: Speedup from a composite predictor with homogeneous table sizes vs. the best component predictor for a given storage size.

Figure 5 shows the speedup improvement of the composite predictor over the best component predictor as we scale the total number of predictor entries from 256-4K. Except for the smallest configuration, where each component predictor has 64 entries, the composite predictor significantly exceeds the speedup of a single component predictor. Thus, we conclude that a composite predictor is better able to use the available predictor state (though, of course, has more logic complexity than a component predictor).

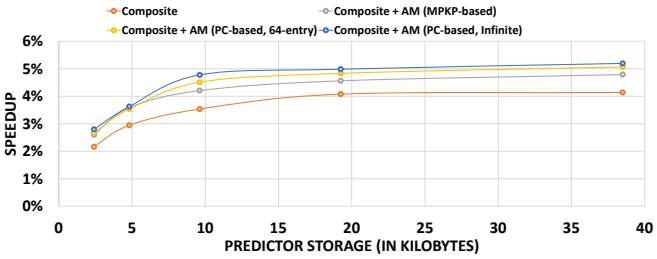


Figure 6: Speedup due to throttling from an accuracy monitor.

B. Accuracy Monitor

As previously discussed, the potential gain of a value predictor is often limited by the high cost of a misprediction. All the component predictors include a confidence threshold to throttle predictions from particular loads. In a composite predictor, we can also throttle an entire component predictor when it is producing a high misprediction rate overall. We studied two different throttling mechanisms, which we call Accuracy Monitors (AM).

We augment the design in Figure 1 by adding one AM per component predictor. At prediction time (Fetch) the AMs are looked up concurrently with the predictors. We squash a component predictor's confident prediction if the associated AM indicates the predictor is unreliable. The exact reliability metric depends on the AM variant, which we discuss next.

1) M-AM: The first AM variant, called M-AM, tracks the misprediction rate for each component predictor within an execution epoch. When the misprediction rate is high at the end of an epoch ($>$ 3 mispredictions-per-kilo-predictions, MPKP), the associated component predictor is silenced in the next epoch. Silenced predictors continue to train.

The M-AM uses two counters per component predictor: one to track mispredictions and the other to track total predictions. The counters are always reset at an epoch boundary, which in our evaluation is one million instructions.

2) PC-AM: While M-AM is a heavy hammer that silences a component predictor based on overall accuracy, our second accuracy monitor, PC-AM, attempts to enforce more targeted silencing by tracking accuracy per PC. PC-AM, uses a direct-mapped, PC-indexed and PC-tagged table. It is indexed by hashing the lower order bits of the PC (e.g., for a 64-entry AM, index is computed as $(PC >> 2) \oplus (PC >> 8)$).

A PC-AM entry consists of:

- **Tag**: A partial tag computed by folding the low order bits of the PC (e.g., for a 10-bit tag, tag is computed as $(PC >> 2) \oplus (PC >> 12)$)

Table VI: Selected configurations from heterogeneous composite predictor sizing exploration.

Total Entries	Speedup vs. No VP	LVP	SAP	CVP	CAP	Total Storage	Speedup/KB	Speedup vs. Homogeneous	comments
4096	5.07%	1024	1024	1024	1024	38.21KB	0.13%	0.00%	homogeneous was best
2048	4.84%	256	1024	512	256	19.31KB	0.25%	+19%	
1024	4.51%	256	256	256	256	9.56KB	0.47%	0.00%	homogeneous was best
512	3.92%	64	256	128	64	4.83KB	0.81%	+33%	
256	3.20%	32	32	128	64	3.67KB	0.87%	+48%	best speedup/KB

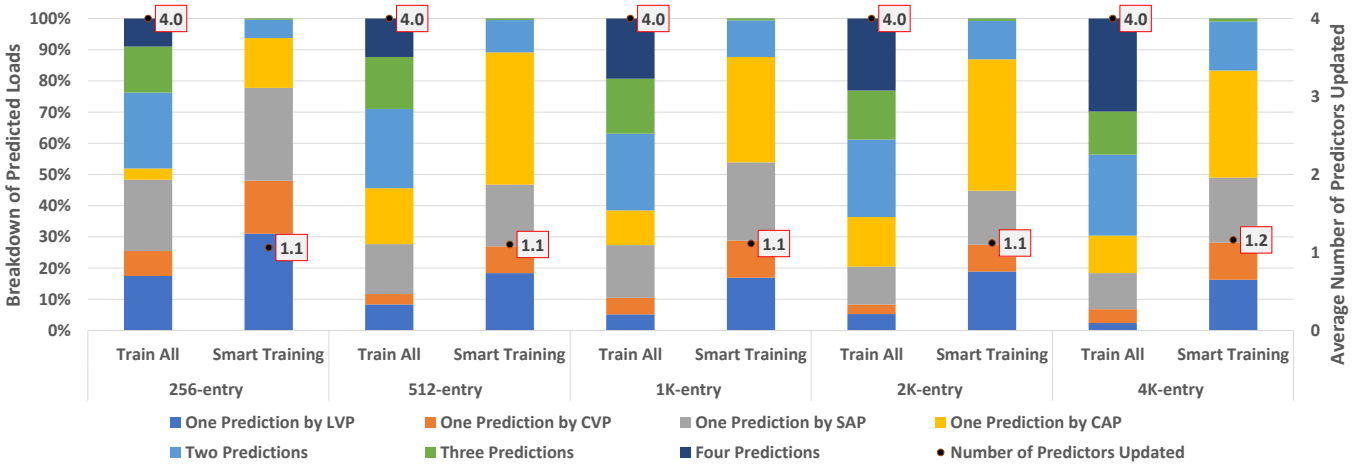


Figure 7: Breakdown of the number of predictions with and without smart training.

- **Counters**: Per predictor correct and incorrect counters that track the accuracy of the given predictor.

We use narrow counters because we only need a rough estimate of accuracy. There are eight counters total (two for each component). If the most significant bit of one of the eight counters gets set, all eight counters are shifted to the right. This action preserves the relative ratio of correct-to-incorrect while allowing for using only 8-bit counters. We silence a predictor using PC-AM whenever the accuracy $(correct/(correct+incorrect))$ for that PC is below 95%.

The PC-AM is managed slightly differently than M-AM. When a value predicted load mispredicts and triggers recovery actions (a pipeline flush), an entry is allocated in AM, possibly replacing the existing entry. A value predicted load (implies that at least one of the predictors was confident in its prediction) that has an AM entry will increment the corresponding counters for all confident predictors, even though only one of the predictions is actually used. This allows for monitoring the accuracy of the predictors that did not provide the final prediction. For each confident predictor, a correct prediction increments the correct counter, and an incorrect prediction increments the incorrect counter.

3) AM Effectiveness: Figure 6 shows the speedup gain from three accuracy monitors: M-AM, PC-AM with 64 entries, and PC-AM with infinite entries. All three variants improve the base composite predictor, and PC-AM generally outperforms M-AM. The finite PC-AM design performs nearly as well as the infinite PC-AM, leading us to conclude that a small PC-AM filter is sufficient. Unless otherwise is stated, we assume a 64-entry PC-AM in our evaluation.

C. Heterogeneous Predictor Tables

We studied whether or not a composite predictor could benefit from heterogeneous component predictor table sizes. To evaluate the potential, we swept the predictor table sizes independently from 0-1K entries, keeping the same training and selection policies as the baseline. Zero entries means that we left the component predictor out completely.

Table VI shows configurations that represent the best performance under different storage budgets. We found several interesting results. First, all winning configurations included all four predictors, indicating that the four complement each other. Second, we found that in two cases (4096 and 1024), the best configuration was actually the homogeneous table allocation. This makes sense in large configurations where none of the predictors are particularly resource starved. Third, we found that heterogeneous configurations were most effective for small predictors.

D. Smart Training

As Figure 4 shows, there is significant overlap in the predictions provided by each component in the composite predictor. To mitigate this overlap, we evaluated a training policy that steers loads to a subset, rather than all, of

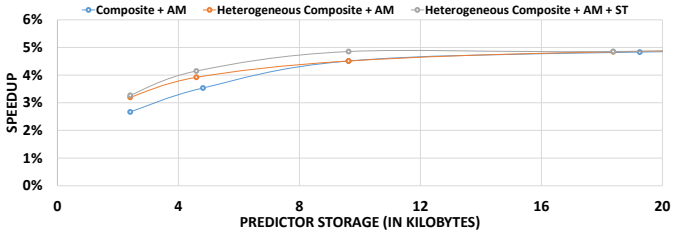


Figure 8: Speedup from smart training. ST: Smart Training.

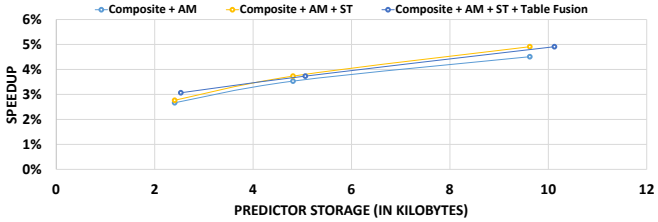


Figure 9: Speedups from table fusion. ST: Smart Training.

predictors at training time. This has the potential to improve performance by avoiding redundancy in the component predictor state.

The smart training algorithm works as follows. If no prediction was made, all predictors are trained to minimize the amount of time to get a confident prediction. However, if one or more predictions are made, we only train the predictors that (a) mispredicted, or (b) have the lowest cost according the heuristic listed below. By always training a component that produced an incorrect prediction, we encourage a quick eviction of the associated entry (a trained misprediction resets confidence). Among predictors that produce a correct prediction, we train them in the following order that prefers value over address and context-agnostic over context-aware: LVP, CVP, SAP, CAP. Additionally, whenever SAP produced a correct prediction but was not chosen for training, we invalidate the SAP entry. By skipping training, the SAP stride will be broken, effectively rendering the entry useless anyway. For example, if all four predictors produced correct predictions, we train LVP entry and invalidate SAP entry, meanwhile, CVP and CAP are not trained.

To validate that smart training worked as intended, we analyzed how many predictors were providing a prediction with and without smart training. Figure 7 shows a significant reduction in the number of times multiple predictions are made. For example, for a 1K-entry composite predictor, the percentage of time multiple predictions are made reduces from 62% to 12%. The Figure also shows the average number of predictors updated at training time; smart training, on average, updates close to one predictor.

Figure 8 shows the speedup from smart training. We found that smart training is most effective for small and moderate size predictors. This makes sense, since larger predictors are less sensitive to small changes in effective table size.

E. Table Fusion

Looking at the component predictors, we see that they have similar storage requirements. To see if it is possible to exploit that fact, we developed a table fusion mechanism that dynamically reallocates tables entries from predictors with low accuracy to predictors with higher accuracy. In order to do so, we assume a design in which all component predictors use the same table width – 81 bits in this case – and number of entries (no heterogeneous allocation).

The table fusion algorithm attempts to separate component predictors into two groups: donors and receivers. Donors are predictors that, over the recent past, have not been very productive (low number of used prediction). Conversely, receivers are predictors that have been useful in the recent past. After classifying the predictors, the fusion algorithm repurposes donor tables as extra storage for receiver predictors. We considered donating partial tables, but found that donating entire predictor tables resulted in the best performance.

The table fusion algorithm is epoch based (one million instructions in our design). During the execution of an epoch, we track the number of used predictions for each component predictor. At the end of an epoch, we compare the number of used predictions to a threshold (corresponding to 20 predictions per thousand instructions in our evaluation), and increment usefulness counter for any predictor exceeding the threshold. After N epochs ($N = 5$), we identify donor tables (used predictions lower than threshold in at least one epoch) and receiver tables (all other predictors). After M epochs ($M >> N, M = 25$), we revert the fusion and repeat.

If there is at least one donor table, fusion occurs. When there is one donor (and three receivers), the receiver with the highest number of used predictions gets the donor table. When there are two donors and two receivers, each receiver fuses with one of the donors. When there are three donors and one receiver, the receiver fuses with all three donors. Donors are flushed at fusion time, because they hold invalid information.

When fusion occurs, the donor tables are added as if they were additional cache ways of the now set-associative receiver table. This approach greatly simplifies indexing and data management, though it may incur extra power compared to an approach that tries to maintain a direct-mapped structure. When tables are unfused, the donor tables are flushed (again) while the receiver tables are maintained since they still contain valid data.

Figure 9 shows the performance gains from table fusion. Like the smart training optimization, table fusion is most helpful on small predictors. At 1K entries and above, table fusion results in no speedup.

F. Combined Benefit

Figure 10 shows the maximum benefit when combining all of the previously discussed composite predictor optimizations. At all sizes, the composite predictor provides a $>$ 50%

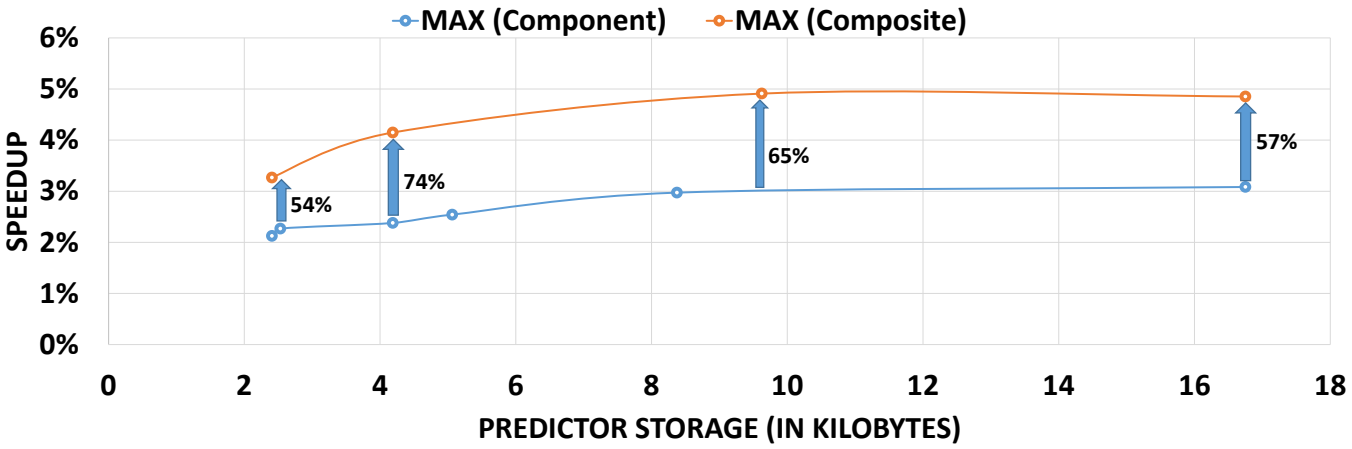
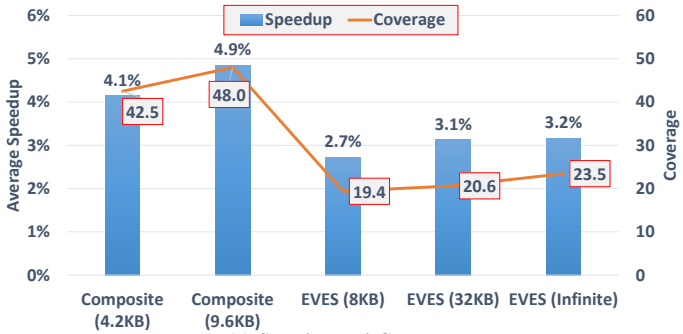
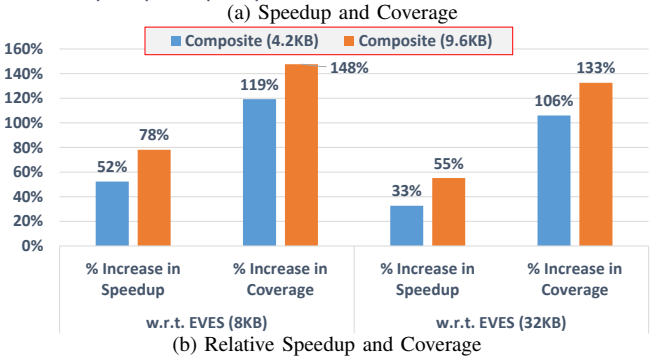


Figure 10: Best overall composite speedup vs. best overall component speedup.

performance boost over a single component predictor. Even though not shown, under comparable budgets, the composite predictor significantly outperforms any of the component predictors, in terms of speedup and coverage, on every individual workload. We would like to reiterate that due to several reasons (e.g., different assumptions about the baseline ISA, microarchitecture, and storage constraints), the speedups reported in this work can be significantly lower than previously published literature. Sheikh reports similar findings in [3], [27].



(a) Speedup and Coverage



(b) Relative Speedup and Coverage

Figure 11: Comparison between composite and EVES predictors.

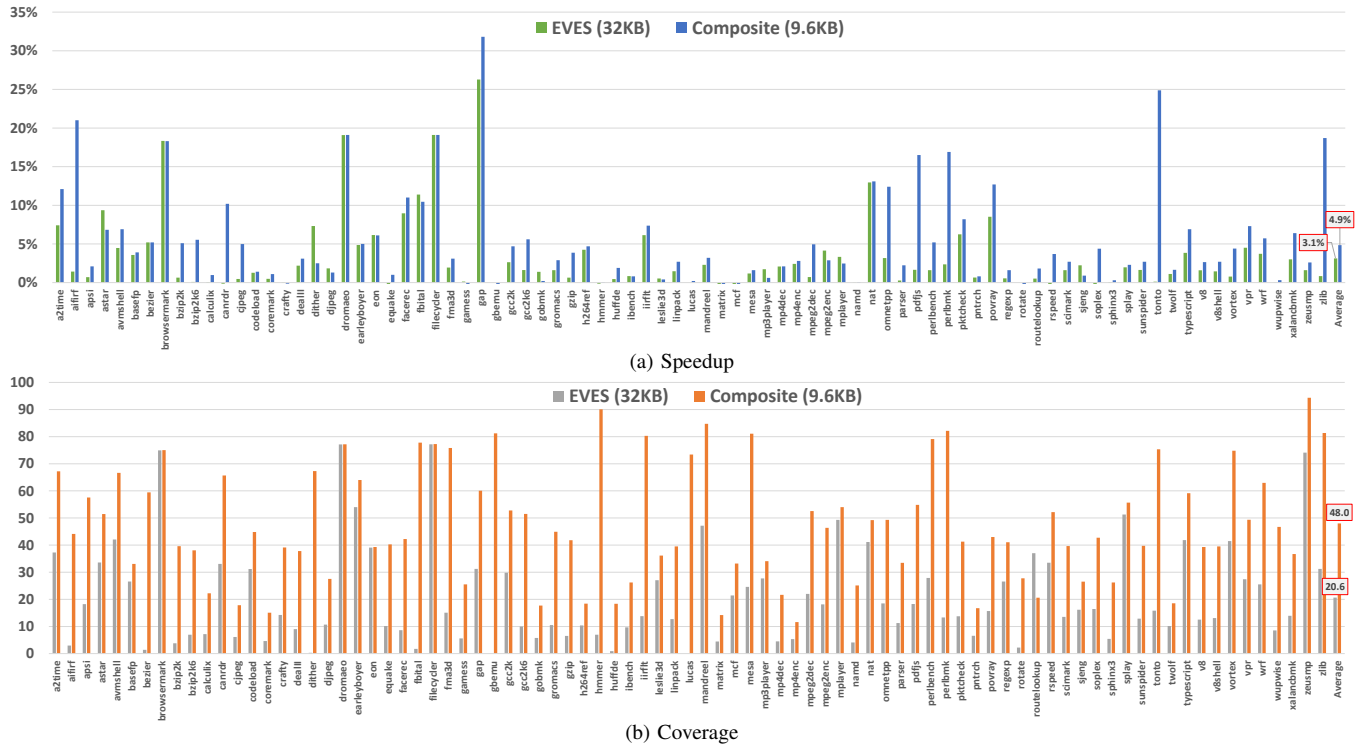
G. Comparison Against Championship Value Prediction

We integrated the winner of the first championship value prediction (CVP-1) [33], called EVES predictor [4], into our framework. EVES tunes VTAGE and augments it with a stride value predictor. Both predictors employ clever optimizations that improve the predictors' storage efficiency and prediction accuracy. In Figure 11 we compare the speedup and coverage of our proposed composite predictor and EVES predictor.

In Figure 12a, we show the per-workload speedup achieved by the best composite predictor (9.6KB budget) vs. EVES predictor (32KB budget). The composite predictor outperforms EVES in 67 out of 85 workloads, while the latter outperforms the composite predictor in 9 workloads only. On average, the composite predictor delivers a substantially higher speedup, 55% higher than EVES. In Figure 12b we show the per-workload coverage achieved by the two predictors. On average, the composite predictor delivers a substantially higher coverage, 133% higher than EVES.

VI. Conclusion

We presented a comprehensive analysis of four state-of-the-art load value predictors. Based on that analysis, we proposed and evaluated a composite predictor that uses all four component predictors together. We suggested four optimizations to make a composite predictor more efficient, namely an accuracy monitor that silences poorly-predicting components, a heterogeneous allocation of resources among components, a smart training algorithm, and a table fusion mechanism. We showed that the composite predictor outperforms the best component predictor by 54%-74% depending on predictor size. We showed that the composite predictor delivers more than twice the coverage of first championship value prediction winner predictor (EVES [4]), and substantially increases the delivered speedup by more than 50%.



(a) Speedup

(b) Coverage

Figure 12: Per workload comparison of composite (9.6KB) and EVES (32KB) predictors.

ACKNOWLEDGMENT

We thank the reviewers for their valuable feedback. 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] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value locality and load value prediction," *SIGPLAN Not.*, vol. 31, Sep. 1996.

[2] F. Gabbay, "Speculative execution based on value prediction," EE Department TR 1080, Technion - Israel Institue of Technology, Tech. Rep., 1996.

[3] R. Sheikh, H. W. Cain, and R. Damodaran, "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*, ser. MICRO-50 '17, 2017.

[4] A. Seznec, "Exploring value prediction with the eves predictor," in *First Championship Value Prediction, CVP 2018, Los Angeles, June 3, 2018.*

[5] J. Mandelblat, "Technology insight: Intel's next generation microarchitecture code name skylake." Presented at Intel Developer Forum, San Francisco, 2015.

[6] J. González and A. González, "Speculative execution via address prediction and data prefetching," in *Proceedings of the 11th International Conference on Supercomputing*, ser. ICS '97. ACM, 1997.

[7] A. Perais and A. Seznec, "Practical data value speculation for future high-end processors," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, Feb 2014.

[8] A. Perais and A. Seznec, "Bebop: A cost effective predictor infrastructure for superscalar value prediction," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015.

[9] Standard Performance Evaluation Corporation, "The SPEC CPU 2000 Benchmark Suite," in *http://www.spec.org*.

[10] Standard Performance Evaluation Corporation, "The SPEC CPU 2006 Benchmark Suite," in *http://www.spec.org*.

[11] J. A. Poovey, T. M. Conte, M. Levy, and S. Gal-On, "A benchmark characterization of the eembc benchmark suite," *IEEE Micro*, vol. 29, Sept 2009.

[12] Linpack Benchmark, "Linpack," in *http://www.netlib.org/benchmark/hpl*.

[13] Media Player Benchmark, "MPlayer," in *http://mplayerhq.hu/design7/dload.html*.

[14] Browser benchmark, "Browsermark," in *http://web.basemark.com*.

[15] Sunspider Javascript Benchmark, "Sunspider," in *http://www.webkit.org/perf/sunspider/sunspider.html*.

[16] Google V8 Benchmarks, "V8," in *http://code.google.com/apis/v8/benchmarks.html*.

[17] Google Octane Benchmark, "Octane," in *https://developers.google.com/octane*.

[18] Dromaeo: Javascript Performance Testing, "Dromaeo," in *http://dromaeo.com*.

[19] iBench Benchmark, "iBench," in *http://ibench.sourceforge.net*.

[20] SciMark: Java benchmark for scientific and numerical computing, "SciMark," in *http://math.nist.gov/scimark2/*.

[21] A. Seznec, "A new case for the tage branch predictor," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. ACM, 2011.

[22] A. Seznec, "A 64-kbytes ittage indirect branch predictor," in *Third Championship Branch Prediction*, ser. JWAC-2, 2011.

[23] R. E. Kessler, "The alpha 21264 microprocessor," *IEEE Micro*, vol. 19, Mar 1999.

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

[25] N. Chong and S. Ishtiaq, "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)*, ser. MSPC '08. ACM, 2008.

[26] M. M. K. Martin, D. J. Sorin, H. W. Cain, M. D. Hill, and M. H. Lipasti, "Correctly implementing value prediction in microprocessors that support multithreading or multiprocessing," in *Microarchitecture, 2001. MICRO-34. Proceedings. 34th ACM/IEEE International Symposium on*, Dec 2001.

[27] R. Sheikh, "Panel discussion: Speculation: Past, present and future." First Championship Value Prediction., 2018.

[28] N. Riley and C. Zilles, "Probabilistic counter updates for predictor hysteresis and stratification," in *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, Feb 2006.

[29] R. J. Eickemeyer and S. Vassiliadis, "A load-instruction unit for pipelined processors," *IBM Journal of Research and Development*, vol. 37, July 1993.

[30] R. Sheikh, R. Cammarota, and W. Ruan, "Value prediction for security (vpsec): Countering fault attacks in modern microprocessors," in *2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, April 2018.

[31] R. Cammarota and R. Sheikh, "Vpsec: Countering fault attacks in general purpose microprocessors with value prediction," in *Proceedings of the 15th ACM International Conference on Computing Frontiers*, ser. CF '18. ACM, 2018.

[32] R. Sheikh and R. Cammarota, "Improving performance and mitigating fault attacks using value prediction," *Cryptography*, vol. 2, 2018.

[33] "First championship value prediction." in *https://www.microarch.org/cvp1*, 2018.