Minimizing Replay under Way-Prediction*

Ricardo Alves
Uppsala University
ricardo.alves@it.uu.se

Stefanos Kaxiras
Uppsala University
stefanos.kaxiras@it.uu.se

David Black-Schaffer
Uppsala University
david.black-schaffer@it.uu.se

***Abstract*— Way-predictors are effective at reducing dynamic cache energy by reducing the number of ways accessed, but introduce additional latency for incorrect way-predictions. While previous work has studied the impact of the increased latency for incorrect way-predictions, we show that the *latency variability* has a far greater effect as it forces replay of in-flight instructions on an incorrect way-prediction. To address the problem, we propose a solution that learns the confidence of the way-prediction and dynamically disables it when it is likely to mispredict. We further improve this approach by biasing the confidence to reduce latency variability further at the cost of reduced way-predictions. Our results show that instruction replay in a way-predictor reduces IPC by 6.9% due to 10% of the instructions being replayed. Our confidence-based way-predictor degrades IPC by only 2.9% by replaying just 3.4% of the instructions, reducing way-predictor cache energy overhead (compared to serial access cache) from 8.5% to 1.9%.**

1. Introduction

Processors require low-latency first-level caches, which often results in L1 cache designs wherein all ways are probed in parallel to avoid the latency of serializing tag and data accesses. However, this approach wastes a significant amount of energy, as all but one of the probed ways is wasted. A common solution to this problem is the use of a way-predictor [2], [3], [4], [5], which attempts to predict which way contains the data, and thereby avoid probing unnecessary ways. On a way-misprediction, however, the access latency increases, as the cache must be accessed a second time for the correct way.

While way-predictors improve energy and latency, their impact on instruction scheduling has been largely ignored. This impact arises from the need to speculatively schedule instructions in deep pipelines based on estimates of when their source operands will be available [6], [7], [8]. Since way-predictors are typically quite accurate, instruction scheduling is based on the latency of a correct way-prediction. However, when the way-predictor mispredicts, all dependent in-flight instructions must be re-scheduled, or *replayed*, such that they execute when the results of the mispredicted load are available. (See Figure 2.) The number of instructions that must be replayed increases with both the issue-to-execute delay (depth of the pipeline) and the pipeline width (number of execution units). Instruction replay costs energy (re-scheduling and re-issuing instructions) and hurts performance (lost execution slots).

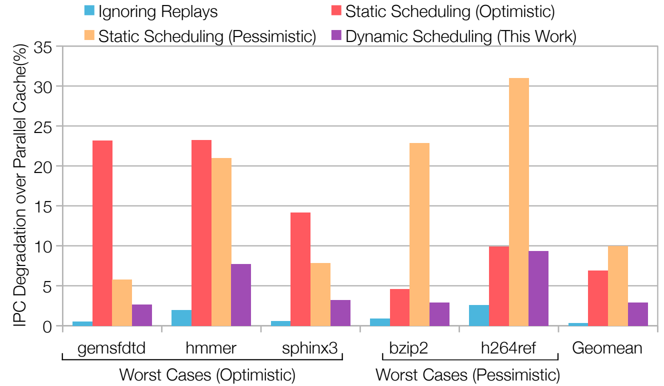

Fig. 1: IPC impact (lower is better) of way-prediction ignoring instruction replays, optimistically scheduling for correct way-prediction, pessimistically scheduling to minimize replays, and our dynamic approach. The chosen benchmarks show that neither static option works well across all benchmarks.

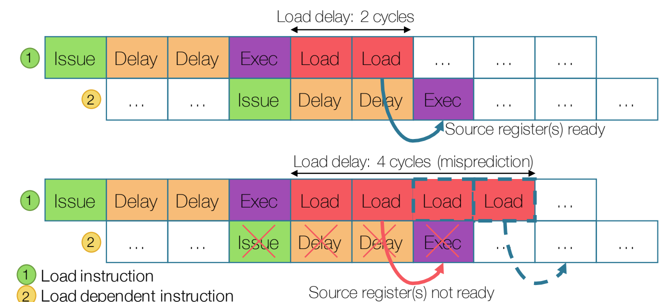

Fig. 2: Speculatively scheduling of a load-dependent instruction on a correct way-prediction (top) and on a way-misprediction, which increase load latency (bottom).

When instruction replay is included, the impact of way-mispredictions is much larger than previously reported. Figure 1 shows the performance degradation (IPC reduction) normalized to a parallel access cache for way-prediction when ignoring replays (blue bar) and when including replays (red bar). Across SPEC2006 (Geomean), ignoring instruction replays reports an average IPC loss of only 0.5% for way-

***Extension of Conference Paper** This work is an extended version of the paper presented at the 36th International Conference on Computer Design[1]. We expand on the evaluation by including CPU pipelines with varying issue-to-execute latencies, showing that instruction replay due to way-mispredictions is worse on deeper pipelines, but it can still be significant on shallower ones. This extension also shows that the original proposed solution is beneficial on both shallow and deep pipelines. Moreover, we make the observation that instruction replay is more detrimental to performance than higher load-to-use latency and propose an improved solution that bias the predictor to avoid instruction replay. This new solution, while increasing the prediction error, counter intuitively improves both IPC and L1 dynamic energy compared to the original solution.

predictors, but if we include replays, the performance penalty increases to 6.9%. Figure 1 further shows the effect of scheduling assuming way-mispredictions (orange bar, Pessimistic) vs. scheduling assuming correct way-predictions (red bar, Optimistic). Neither statically scheduling for correct nor incorrect way-predictions works well across all benchmarks, as we need to trade-off the increased latency of scheduling for mispredictions with reduced replays.

We propose a mechanism to measure the confidence of the way-prediction and use it to dynamically disable prediction and schedule dependent instructions accordingly (purple bar, Figure 1). This avoids the latency cost of scheduling for mispredictions when the way-predictor is likely to be accurate and avoids the cost of instruction replays when it is not. As a result, we can reduce instruction replay without hurting latency, while also reducing additional cache data array probes, and thereby saving energy.

For a CPU with an issue-to-execute delay of 4 cycles, our results show that a standard way-predictor causes an average of 10% of issued instructions to be replayed (up to 42.3%), resulting in a 9.6% loss of performance (up to 23.2%), which is far greater than the 0.5% loss when ignoring replays. Our solution only replays 3.4% of the instructions (up to 13.2%) which degrades performance by only 2.9% on average (up to 11.3%). Moreover, despite decreasing prediction accuracy, our solution avoids enough double-probes on mispredictions that it reduces the number of extra cache data array probes, which further improves cache energy efficiency. On the worst benchmark, a standard way-predictor uses 33.8% more energy than a serial cache, while our design uses only 4.8% more, as it disables the way-prediction when it will be ineffective.

The contributions of this paper are:

- We present the first evaluation of the performance impact of way-prediction considering instruction replay on a multi-cycle issue pipeline.
- We identify scheduling variability of way-predictors as a significant performance and efficiency problem that has not been addressed by previous work.
- We demonstrate that even in short issue-to-execute delay pipelines, instruction replay due to way-mispredictions is more detrimental to performance than the load latency increase.
- We propose a method to learn the accuracy of the way-predictor and use it to dynamically disable prediction and adjust instruction scheduling accordingly.
- We improve energy and performance compared to a standard way-predictor by reducing both cache probes and replayed instructions.

2. Background

A. Way-Predictors

The most power-efficient cache design is to read the tags and data serially, which allows the cache to probe only the needed data way. This minimizes the total number of bits read, thereby reducing energy, but comes at the cost of increased latency compared to parallel access of the tags and data [9].

Way-predictors [3], [4] address this problem by using a prediction table in parallel with the tags to predict the correct way and speculatively probe only that way. A simple and effective strategy is to predict that the most-recently-used (MRU) way of a set will be the next one used [10]. A way-predictor misprediction increases latency and energy, as a second access to the data array is required. However, after the first access, the correct way (if any) is known from the tag array lookup. With sufficiently high correct prediction rates, way-predictors can achieve the latency of a parallel cache and the energy efficiency of a serial cache.

B. Speculative Scheduling

Most processors today have significant issue-to-execute delays due to pipelining. As a result, instruction scheduling happens many cycles before execution. For instructions with fixed latencies, the scheduler issues dependent instructions the correct number of cycles after their producers [11], [12]. However, in the case of load instructions, execution latency is not deterministic, and deciding when to issue dependent instructions is a challenge: issuing too late hurts performance, while issuing too early may force instruction replays if the data is not ready at the expected time. The two main sources of this load-latency variability studied in literature are cache misses [13], [14], [15] and bank conflicts [15], [16].

Caches Misses. Load latency depends on where the data is found in the memory hierarchy. Most schedulers assume an L1 hit and schedule dependent instructions accordingly. However, for workloads with low hit-ratios, this results in many dependent instructions being replayed.

Hit predictors [13], [14], [15] can delay scheduling of instructions when an L1 miss is predicted. However, these solutions are not directly suitable for way-predictors as way-predictor success rates may not be correlated with cache hits. For example, *libquantum* is the SPEC2006 benchmark with both the best way-predictor success ratio and the worst L1 hit ratio. This problem is somewhat simpler for way-predictors as the latency difference between a correct way-prediction and a way-misprediction is static, unlike cache misses, which may return data from anywhere in the memory hierarchy. This means that the scheduler could deterministically account for the way-misprediction latency by scheduling for the worst case. Such a pessimistic approach would avoid replays, but increase latency.

Bank Conflicts. Cache data arrays are divided in banks to reduce latency and energy. This allows the cache to satisfy simultaneous data requests to different banks. The frequency of bank conflicts depends on the number of banks, the data mapping, and the access pattern. Bank conflicts can be addressed by conflict predictors [15], [16] and scheduling to avoid simultaneous execution of non-critical loads [16].

Neither of these solutions are applicable to way-misprediction since delaying the issuing of a way-mispredicting load will still cause it to mispredict, just later on in time, and therefore hurt performance further.

Frequency	2.5GHz
IssueWidth/LoadUnits/LQ/SQ/IQ/ROB	4/2/64/40/48/128
Caches Size Latency Associativity	L1I/L1D/L2/L3 32kB/32kB/256kB/4MB 1c/2c/12c/20c 8w/8w/8w/16w
DRAM	DDR3/1600MHz/64bits

TABLE 1: Simulation processor configuration. The way-predictor is applied to the L1 data cache.

C. Instruction Replay

Speculative scheduling of instructions is necessary when an instruction depends on an earlier instruction with a variable latency. If the source is not available at the scheduled time, the pipeline needs to cancel and replay the dependent instructions. This can be done by either replaying the dependent in-flight instructions (selective replay), or all in-flight instructions (dependent and independent instructions alike)[1].

Efficiently implementing selective replay is a challenge [6] since the scheduler needs to keep track of all the in-flight speculative instruction chains. Several strategies have been proposed to address this, including *token-based selective replay* [6], wherein only a small number of in-flight instructions, those marked as likely to be misscheduled, are tracked and replayed if needed. All other misscheduled instructions need to be squashed and re-fetched. A second solution is to place speculatively scheduled instructions in a *recovery buffer* [17] such that the speculatively scheduled instructions are easily found in the buffer on a misschedule. This design can also be extended to replay only a particular misscheduled instruction chain, reducing replays even further.

When it comes to industrial replay mechanisms, only the Alpha 21264 [18] and the Pentium 4 [19] have any public documentation. The Alpha 21264 replays all in-flight instructions when a misspeculation is detected, while Intel provides no details. Perais et al. [16] identified replay queues [20] and replay loops (similar to the Cyclone scheduler [11]) as two possible implementations based on Intel patents.

3. Simulation Environment

We use the gem5 [21] simulator in full-system mode with 10 uniformly-distributed checkpoints for each SPEC2006 benchmark. Checkpoints are warmed for 100M instructions, followed by 10M instructions of detailed simulation.

CPU Model. We simulate a “medium” complexity out-of-order CPU (128 ROB, 4 issue wide, see Table 1). This represents a balance between a large out-of-order CPU (more latency tolerant with a larger issue width, reducing the performance benefit of a way-predictor but increasing replay cost) and a small out-of-order CPU (where way-prediction would have a larger impact on performance, but there would be a lower cost of replay).

Cache Energy Model. We use a modified version of Cacti [22] that does not assume a full cache-line readout on each access to avoid overestimating the impact of way-prediction [23]. Our modeled cache is banked with cache-lines striped across multiple sub-arrays in groups of 64 bits to minimize dynamic energy on a parallel access. The modeled parallel access 8-way L1 uses 3.7x more energy than a serial access[2]. Additional structures such as the way-prediction table and the confident measuring unit (discussed in Section 5) are also modeled, but do not significantly impact energy.

Replay Mechanism. We added instruction replay to the gem5 simulator following the work of Perais et al. [16], which is based on the Alpha 21264, i.e. all in-flight instructions are replayed. It is not clear if replayed instructions in the Alpha 21264 retain their IQ position or if a recovery buffer is used. As we did not see any performance degradation due to increased IQ pressure, we simply allow speculative scheduled instructions to retain their IQ entry for re-scheduling on a replay.

Other Sources of Replay. To isolate and study the impact of replays caused by way-mispredictions, we assume perfect L1-hit prediction and perfect L1 bank conflict resolution. This means we do not experience replays due to either L1 misses or bank conflicts. These assumptions are reasonable given that both of these issues have been studied and solved in prior work: Our experiments with a history-based hit-predictor [15] shows an average accuracy of 94%, and address-based solutions [13], [14] have even higher accuracy; Bank conflict replays can be completely avoided by shifting the issuing of loads with no performance degradation [16]. We do model replays and instruction squashes due to TLB misses and branch mispredictions, but they have little impact due to their rarity.

Way-Predictor. We model an MRU way-predictor using a prediction table with one entry (3 bits) per cache set (full coverage). This approach is based on previous work that has shown that such small prediction tables can be paired with fast address generation units (AGU) to allow access either outside the critical path or without increasing it [24]. A successful way-prediction therefore achieves the same load-to-use latency as a parallel cache, while a way-misprediction requires a second data array probe, which doubles the access time.

4. Impact of Way-mispredictions on Instruction Replay

Way-prediction has been considered an effective way to reduce the dynamic energy of first-level caches since its high prediction accuracy directly reduces dynamic energy (fewer ways probed) while incorrect predictions have minimal impact on performance. Figure 3 compares the success ratio of an L1 data cache way-predictor with its performance impact (normalized to a parallel access cache) ignoring replays.

[1] A third alternative would be to squash and re-fetch misspeculated instructions but that strategy is costly in energy and latency.

[2] The difference between parallel and serial accesses to the 8-way cache is not the expected 8x due to the more optimal cache-line striping [23] and the larger relative overhead of the tags when only reading the minimum amount of data bits instead of a full cache-line. Note that the difference is also different than the one in [23] due to the striping of cache-lines in groups of 64 bits instead of 32.

The average IPC degradation is 0.5%, peaking at 2.6% for *h264ref*. *H264ref* is a load latency-sensitive benchmark, but the way-predictor accuracy is high enough (90.6%) so as to keep the IPC impact low. On the other hand, for latency-insensitive benchmarks with low way-prediction accuracy, such as *gemsfdtd* (51.7%), the IPC degradation is only 0.5%.

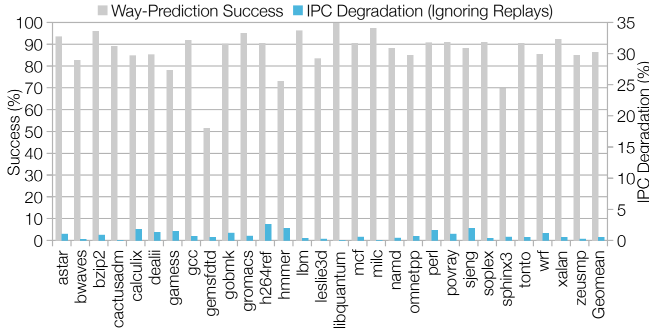

Fig. 3: Way-prediction success ratio (higher is better) and its effects on performance ignoring replays (IPC degradation, lower is better) compared to a baseline parallel cache.

These results make way-predictors look promising, as latency-sensitive benchmarks tend to have a high enough way-prediction success ratio to not suffer from mispredictions, while applications with many mispredictions are not particularly latency-sensitive. However, these results ignore the impact of instruction replay (e.g., they essentially assume a pipeline with a 0-cycle issue-to-execute delay), which is not realistic for out-of-order processors.

When the penalty of instruction replay is included, way-mispredictions not only increase the latency of loads, but also affect the issue stage of the pipeline by forcing new instructions to be delayed due to replayed instructions being re-scheduled. This effect is shown in Figure 4, where we see how the previous results (Ignoring-Replays) compare to a realistic implementation with a 4 cycle issue-to-execute delay, that optimistically schedules assuming a correct way-prediction (Replay-Optimistic).

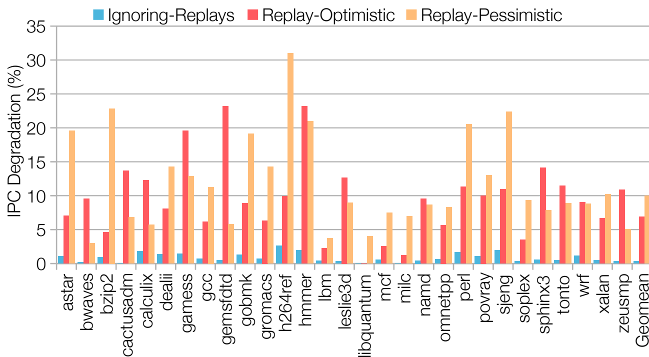

Fig. 4: Perforamnce impact of including replay costs in way-prediction performance analysis. Performance degradation (over a baseline parallel access cache) for way-prediction with No Replay and for an issue-to-execute-delay of 4 under Optimistic and Pessimistic scheduling (lower is better).

The difference when taking into account replays is substantial. We see that the average IPC degradation for the way-predictor is now 6.9%, which is significantly worse than the 0.5% when ignoring replays. The difference is particular evident in benchmarks with low way-prediction accuracy, such as *gemsfdtd*, *sphinx3* and *hmmer* (51.7%, 69.7% and 73.1%, respectively) where IPC degrades by 23.2%, 14.1% and 23.1%, respectively. When we ignore replays, these benchmarks are not affected by mispredictions, as they are not sensitive to load latency. But when the cost of replay is included, their high misprediction rates lead to significant performance losses from instruction replay.

One can eliminate the need for replays by *pessimistically* scheduling dependent instructions assuming way-mispredictions (Figure 4, Replay-Pessimistic). This will eliminate the overhead for benchmarks with high way-mispredictions ratios that are not latency-insensitive. However, this hurts benchmarks that are sensitive to load latency (*bzip2* and *h264ref*). For these benchmarks, the way-predictor has a high success ratio (96% and 90.6%), and the benchmarks benefit from the earlier scheduling of load-dependent instructions. Overall, a pessimist scheduling strategy produces even worse results and defeats the purpose of way-prediction. From this point on we assume an optimistic scheduling strategy for way-predictor designs.

We have shown that *instruction replay* due to way-mispredictions has a far greater impact than the previously reported by studies that only considered the latency increase of re-probing the data arrays on way-mispredictions. This effect is more pronounced for applications with low way-prediction accuracy, as they suffer from more replays, but even applications with high-prediction accuracy see significantly more performance loss due to replays than re-probing. While replays can be eliminated by statically scheduling instructions for way-mispredictions, this approach hurts performance significantly.

5. Selective Way-Prediction

To avoid excessive replays without increasing latency, we propose *Selective Way-Prediction*, which learns when the way-predictor is likely to mispredict and schedules accordingly. By doing so we can minimize replays without hurting latency.

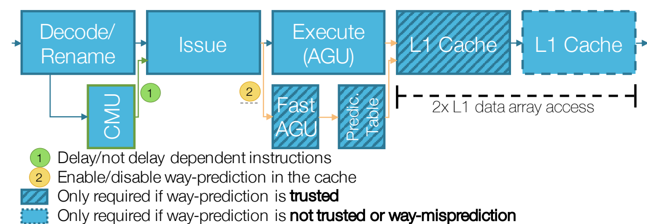

Fig. 5: An early way-predictor confidence measurement from the confidence measuring unit (CMU) is used to control scheduling of dependent instructions and to enable or disable way-prediction.

To predict when the way-predictor is likely to mispredict, we add a confidence measuring unit (CMU in Figure 5) that follows the same simple strategy proposed for branch

predictors [25]: The CMU is a table of saturating counters, indexed by the least significant bits of the load PC, that are incremented/decremented on a correct/incorrect way-prediction. If the count is equal or greater than a threshold, we deem the way-predictor accurate for that load, and thus enable way-prediction and optimistically schedule dependent instructions. Otherwise, the way-prediction is disabled and load-dependent instructions are scheduled accordingly.

A. *Scheduler and Cache Modifications*

Depending on the confidence for a particular load, the scheduler makes different scheduling decisions for the load's dependent instructions. This information allows the scheduler to take advantage of the low load-to-use latency of high-confidence (correct) way-predictions, while avoiding paying the replay cost of overly-eager scheduling for unreliable way-predictions. In addition, if the scheduler knows to avoid immediately scheduling dependent instructions, it is likely to find other instructions to schedule instead, thereby limiting the performance impact further.

The confidence is also used to dynamically enable/disable way-prediction. For low-confidence predictions we disable the first (speculative) data access and wait for the tag results. This avoids the extra data array probe on a misprediction, which provides the energy-efficiency of a serial cache for low-confidence loads. Alternatively, a performance-oriented optimization could execute a parallel load on low-confidence predictions to ensure low access latency at the cost of increased energy. Simulations of this approach showed little IPC benefit despite a 24% energy increase, even with a perfect confidence prediction.

B. *Confidence History Table*

Our Confidence History Table is indexed by the instruction PC, is not tagged, and has 256 entries, each holding 2-bit counters. The confidence threshold is set 2, such that 2 or more correct predictions in a row are deemed trustworthy. We explored the impact of other history table sizes and counter depths, and found little benefit to more than 256 entries (as the L1 has 256 sets) or more than 2 bits of counter depth (higher resolutions did not increase accuracy).

Since a modern CPU is capable of fetching, decoding and issuing several instructions per cycle, the history table needs to allow multiple accesses per cycle. The most obvious solution would be to multi-port the table. Since the table has 256 entries with 2 bits per entry, the total size of the structure is of 512 bits which is a reasonable size for multi-porting. However, since the structure is directly mapped and load instructions issued in the same cycle are likely to have small PC differences, it should be possible to bank the table and use an index function that will largely avoid bank conflicts (e.g., banking on the least significant PC bits). A banked structure will reduce the dynamic energy compared to a multi-ported design. For this work, we assumed the use of a multi-ported structure to store the history, which enables arbitrary concurrent accesses.

Since the PC is available early in the pipeline, and the table is indexed by the PC, we place the confidence history table in the decode/rename stage. This means that the table can be accessed in parallel with the decoding of the instruction, allowing the instruction to be sent to the issue stage along with its way-predictor confidence. As a result, the scheduler and the cache both have early access to this information.

Placing the history table in the decode stage does have the undesirable side-effect that all instructions (loads and otherwise) access the table as they have not yet been decoded. Since only the opcode is needed to determine if the history table needs to be accessed, is reasonable to assume that we can include a partial decode and disable history table access for non-loads. Even if this is not the case, the extra accesses to the table will affect the prediction, and will have a small energy impact due to the very small size of the table.

6. Evaluation

We evaluate the accuracy of Selective Way-Prediction and its impact on instruction replay, performance, and cache probes (energy). The results are presented for four configurations:

- **WayPred:** a standard way-predictor where the scheduler optimistically assumes a correct way-prediction for dependent instructions.
- **Selective WayPred:** a confidence history table is used to disable way-prediction and delay scheduling of dependent instructions for low-confidence way-predictions.
- **Biased Selective WayPred:** the confidence history table is biased to minimize errors due to *incorrectly enabling* way-prediction (Sections 6-C and 6-D).
- **Baseline (for performance results):** a standard parallel access cache. This provides the highest performance as all hits are returned in the shortest time.
- **Baseline (for energy results):** a standard serial access cache. This provides the lowest energy as there are never any unnecessary data array probes. (This is used for results presented in Section 6-F.)

A. *Confidence Measure*

If the confidence measure was perfect, selective way-prediction would only disable way-prediction for incorrect way-predictions, and not affect way-prediction accuracy. However that is not the case and Figure 6 shows that selective way-prediction reduces the average correct way-prediction ratio from 86.6% to 83.7% across all benchmarks, with the worst impact being a decrease of 7.8 percentage points for *sphinx3*. To understand the impact on performance and energy we must first look at the type of errors introduced.

The selective way-predictor strategy has two types of errors:

- *Incorrectly enabling* way-prediction when it will mispredict the way. This causes instruction replays and costs energy for a second data array probe.
- *Incorrectly disabling* way-prediction when it would have correctly predict the way. This has the performance cost of scheduling dependent instructions later, but does not cost additional energy on data array probes nor does it cause instruction replays.

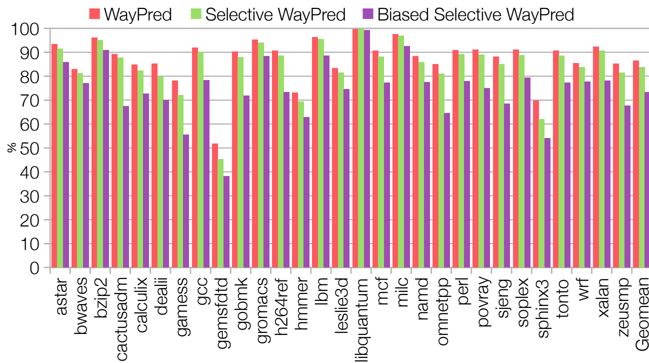

Fig. 6: Prediction success ratio of a standard Way-Predictor and the proposed Selective Way-Predictor and Biased Selective Way-Predictor. (higher is better).

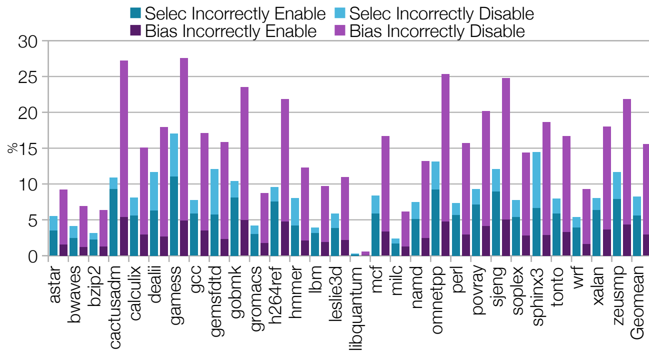

Fig. 7: Way-prediction errors caused by Selective Way-Prediction (Selec) and Biased Selective Way-Prediction (Bias).

The breakdown of these errors is shown in Figure 7. The light blue bars shows the number of *incorrectly disabled* way-predictions, i.e., the accuracy difference shown in Figure 6. These errors affect performance for latency-sensitive applications. The dark blue bars show the *incorrectly enabled* way-predictions, which result in both extra energy, from an additional data array probe, and instruction replay. In the majority of the benchmarks the CMU accurately detects where way-prediction will fail or succeed, which allows us to correctly enable/disable way-prediction for 91.8% of the loads.

B. Instruction Replay and Performance

Although selective way-prediction hurts way-prediction accuracy, it substantially improves instruction scheduling accuracy by detecting most mispredictions and minimizing instruction replays. Selective way-prediction can only cause misscheduling on *incorrectly enabling* errors, while a standard way-predictor can cause misscheduling on all mispredictions. Figure 8 shows the percentage of replayed instructions for a standard way-predictor compared to our selective way-predictor. We see that many of the benchmarks with the lowest way-prediction accuracies (*gemsfdtd*, *hmmer*, *sphinx3*) generate very large numbers of instruction replays (42.3%, 38.4%, 23.6%). Note that the lowest way-prediction accuracy alone does not guarantee the most replays, since the number of replayed instructions is a function of both the number of in-flight instructions and the number of way-mispredictions. Since selective way-prediction only causes replays when it *incorrectly enables* way-prediction, the percentage of replayed instructions is reduced from 10% for a standard way-predictor to 5.6%, on average, and the benchmark with the largest replay ratio goes from *gemsfdfd* with a 42.3% ratio, to *cactusadm* with a 16.3% ratio.

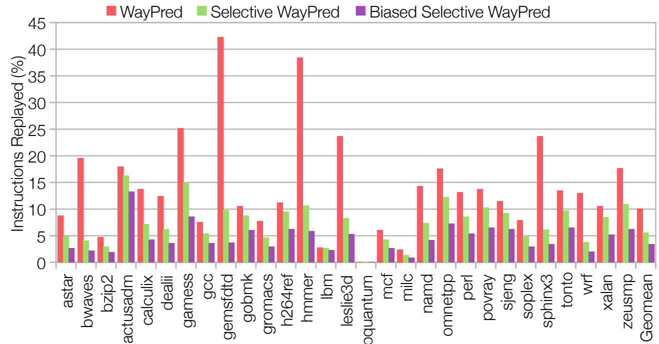

Fig. 8: Ratio of instructions replayed to instructions issued with an issue-to-execute delay of 4 (lower is better).

The performance impact of reducing the number of replayed instructions is shown in Figure 9. For an issue-to-execute delay of 4 (purple line), selective way-prediction improves performance across the benchmark suit, but has a more pronounced effect on benchmarks with a steep reduction in instruction replays, such as *gemsfdtd*, *hmmer*, and *sphinx3*. For these applications, IPC improves by 22.1%, 16.8% and 10.7%, respectively, compared to a standard way-predictor. For applications where the confidence measurement unit is unable to significantly reduce the number of replays, such as *cactusadm* and *lbm*, the performance improvement is minimal (1.2% and 0.4%). It is important to note that benchmarks such as *libquantum* and *milc* have little room for improvement as their way-prediction success rates are very high (99.7% and 97.5%, Figure 3). Across all benchmarks, selective way-prediction degrades performance by 4.4% compared to a parallel cache, versus the 6.9% degradation of a standard way-predictor.

C. Analyzing Performance: Latency vs Replay

Selective way-prediction causes a decrease in way-prediction accuracy (potentially hurting performance), but also reduces the number of misscheduled instructions and replays (potentially improving performance and energy). To understand the contribution of these two aspects to the overall IPC, we conducted simulations with and without a replay penalty (blue and purple lines in Figure 9, respectively).

In the configurations without replay (blue line), there is no penalty for way-mispredictions aside from the increased load latency. As expected, the standard way-predictor cache performs best as it has the highest way-prediction accuracy due to always predicting the way and we ignore replay effects. That is, it does not suffer from *incorrectly disable* errors, Figure 7.

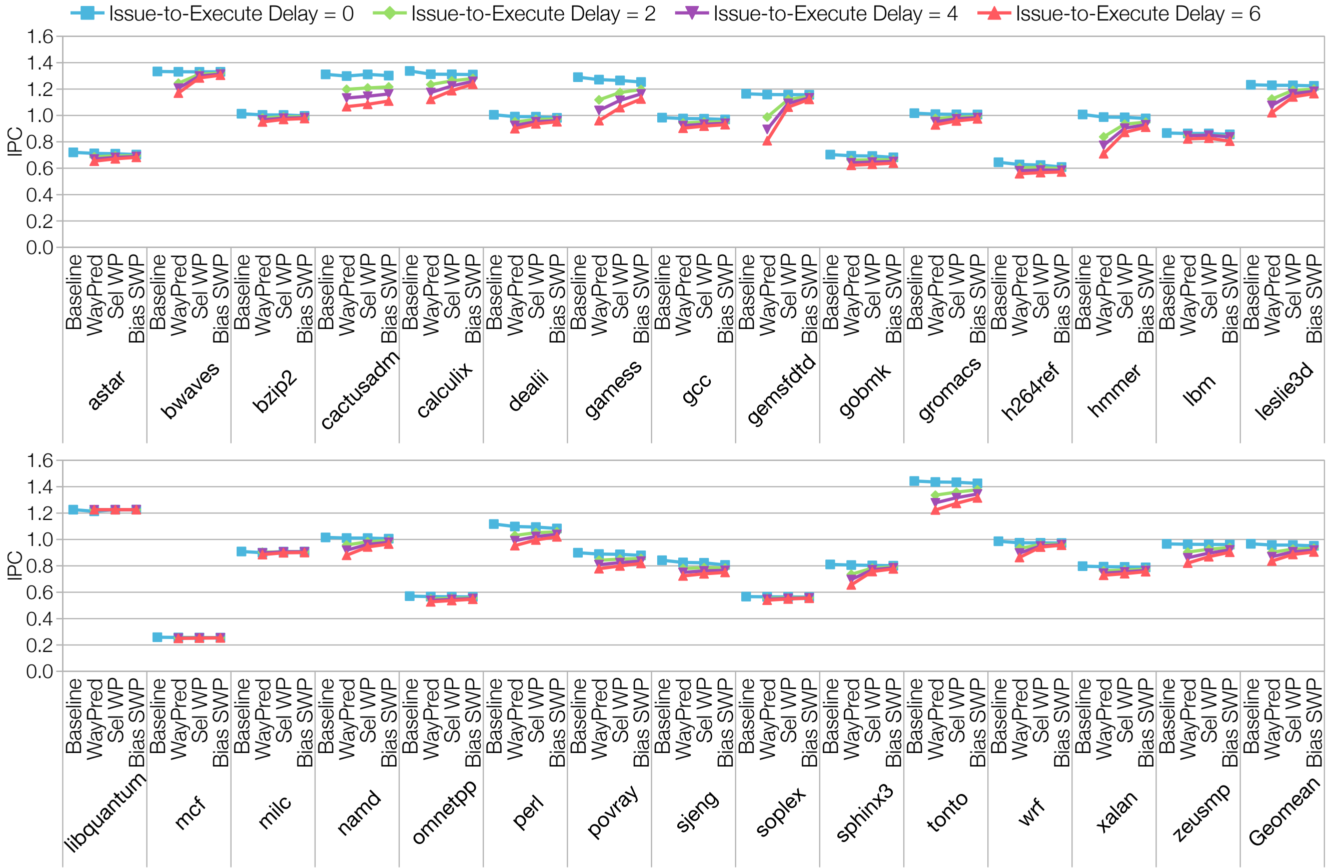

Fig. 9: Performance of the explored way-prediction strategies with varying issue-to-execute delay. (higher is better).

When replay is introduced (purple line, for an issue-to-execute delay of 4), the trend is reversed, with the configuration with more instruction replays (standard way-predictor) showing the larger performance loss. (The effect of varying the issue-to-execute delay is discussed in Section 6-E.) These findings show that the misscheduling and replays introduced by the standard way-predictor are more detrimental to performance than the increase in average load latency (due to *incorrectly disabling* errors) introduced by the selective way-predictor.

These results suggest that it would be beneficial to *bias* selective way-prediction to be more aggressive in avoiding *incorrectly enabling* errors. Such a bias would further reduce replays at the cost of an increase in latency due to additional missed opportunities to hit in the predicted way (additional *incorrectly disabling* errors). To explore this, we must change the confidence measure to be less eager to trust the prediction (a higher threshold or smaller increases in the confidence counter on correct predictions) and/or increase the penalty for mispredictions (a larger decrease in the confidence counter on incorrect predictions).

Many combinations of these modifications can be used to bias the confidence. We explored a range of options for counter increments, decrements, and thresholds, and found the best trade-off between decreasing *incorrectly enabling* and increasing *incorrectly disabling* errors is to decrease the confidence counter by two on a misprediction, increase by one on a correct prediction, and only trust the prediction when the two-bit counter is saturated. We use this configuration for our biased selective way-predictor.

D. Biased Solution: Instruction Replay and Performance

Figure 7 shows the error for the biased selective way-predictor (purple bars). The biased strategy is successful in reducing instances of *incorrectly enabling* the way-predictor across the benchmarks, with an average reduction from 5.6% to 2.9%. This reduction allows the biased selective way-prediction to further reduce the percentage of replayed instructions to 3.4% (Figure 8), an improvement over the 5.6% achieved by the non-biased approach. Benchmarks that had the highest ratio of instruction replay previously, *gamess*, *omnetpp* and *zeusmp* (15%, 12.2% and 11%) are further reduced to 8.6%, 7.3% and 6.2%, respectively. *Cactusadm* stands out as the benchmark with the worst ratio in both configurations, but its replays are now reduced from 16.3% to 13.2% with the bias (both significantly better than 18% of a standard way-predictor).

However, the bias significantly increases the *incorrectly disabling* errors from 2.6% to 12.6%, increasing the overall error. Despite this increase, the biased selective way-predictor is able to improve performance on all benchmarks (purple line, Figure 9) with the exception of *lbm*, where IPC decreases by 1.2% compared to a standard way-predictor. Benchmarks such as *gemsfdtd*, *hmmer* and *sphinx3* see

IPC improvements of 26.8%, 20.1% and 12.8% over a standard way-predictor (further improving on selective way-prediction) since they are not latency sensitive and thus benefit more from the reduction in *incorrectly disable* errors than *incorrectly enable* errors. On the other hand, latency sensitive benchmarks such as *bzip2* and *h264ref* see only modest improvements of 1.7% and 0.6% for the opposite reason.

Overall, the biased selective way-predictor degrades IPC by 2.9% compared to the baseline, improving on both the standard way-predictor and selective way-predictor, which degrade IPC by 6.9% and 4.4%, respectively.

E. Varying Issue-to-execute Delay

To provide a broader understanding of the interaction between way-predictor scheduling variability and replay, we vary the pipeline issue-to-execute delay from 0 to 6 cycles (Figure 9). This models an increased pipeline depth (latency between the issue and execute stages), which increases the number of potential in-flight instructions that need to be replayed. As a result the penalty from way-misprediction replays increases as well. This is most pronounced for benchmarks with high way-misprediction rates, the wost being *gemsfdtd*, *hmmer* and *sphinx3*. The effect is smaller, but still significant, even at relatively short issue-to-execute delays of 2 cycles (green line). For such short pipelines, we still see the standard way-predictor hurting the IPC of *gemsfdtd* and *hmmer* by 15.2% and 16.8% respectively. Overall, the standard way-predictor decreases IPC by 4.4%, 6.9% and 9.3% for pipelines with an issue-to-execute delay of 2, 4 and 6, respectively.

Due to the increase in in-flight instructions, selective way-prediction is most beneficial for deeper pipelines (red line, Figure 9). With shallower pipelines (green line), selective way-prediction still manages to reduce the negative impact of benchmarks with high misprediction ratios such *gemsfdtd*, *hmmer* and *sphinx3* from 15.2%, 16.8% and 9.3% to 3.7%, 7.3% and 3.4%, respectively. Biased selective way-prediction improves the results even further by reducing the negative IPC impact to 1.7%, 5.7% and 2.4% on those same benchmarks. Overall, for a short issue-to-execute delay of 2 cycles, selective way-predictor reduces IPC by 2.9% slightly worse than a biased selective way-predictor (2.1%). These results show that selective and biased selective way-predictors are beneficial even on short pipelines.

F. Dynamic Energy Reduction

With selective way-prediction the data array is only probed twice when the confidence measure incorrectly enables the way-prediction instead of on all mispredictions. Figure 10 compares the total dynamic cache energy (including energy from cache misses) of the way prediction strategies to the baseline serial cache. The baseline serial cache will always be the most energy efficient design as no extra probes are ever required.

The benefit of selective way-prediction is most noticeable on benchmarks with the highest way-misprediction ratios, where selective way-prediction is able to detect and avoid most of the way-mispredictions. The benchmarks *gemsfdtd*, *hmmer* and *sphinx3* have dynamic L1 energies for a selective way prediction cache that are 4%, 3.9% and 3.1% higher than a serial L1, which is a significant reduction from the 33.8%, 25% and 14.2% increases of the standard way-predictor. On average, selective way-prediction reduces the energy overhead from 8.5% to 3.7% and the worst benchmark switches from being *gemsfdtd* at 33.8%, to *gamess* at 9.5%.

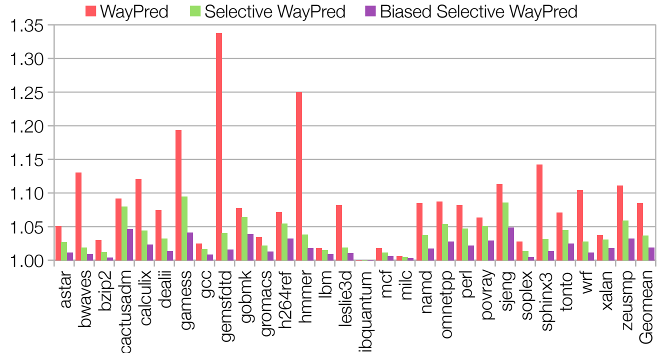

Fig. 10: Increases in dynamic energy compared to a serial access cache (lower is better). This shows the ability of both the selective and biased selective way-predictors to reduce cache dynamic energy. These results include the energy of the way-prediction table and confidence measuring unit. The results include the energy of the way-prediction table and confidence measuring unit.

Adding a bias to the selective way-prediction allows us to reduce the *incorrectly enable* errors from 5.6% to 3%, further reducing the additional dynamic energy of way-prediction from 3.7% to only 1.9%. The reduction in additional data array probes not only further reduces the energy overhead of *gemsfdtd*, *hmmer* and *sphinx3* benchmarks (to 1.6%, 1.9% and 1.4%), but also improves the overhead of *gamess*, the non-biased worst benchmark (from 9.5% to 4.1%). This is due to a reduction of *incorrectly enable* errors from 11.1% to 4.9%. The worst benchmark is now *sjeng* with an energy overhead of 4.8% over a serial cache, or 7x lower than the benchmark with the worst overhead in the standard way-predictor.

7. Related work

In addition to the standard way-prediction approaches discussed earlier [2], [3], [4], [5], way-estimation tries to determine where the data is through fast partial tag comparisons [26], [27] or bloom filters [28] to avoid probing those ways. Such techniques try to identify the minimum number of ways that require probing to guarantee a hit, which is not necessarily one. This guarantee comes at the cost of additional energy from probing multiple ways. Cache decay[29] can also be considered a way-estimation technique since it disables the probing of ways dynamically, but may lead to an increase in cache misses.

We have also discussed two classes of strategies to handle variability in instruction replay for cache bank conflicts [15], [16] and L1 misses [13], [14], [15]. Bank conflict strategies focus on delaying problematic loads to avoid causing the

conflict, and L1 miss strategies focus on delaying issuing dependent instructions until the missing load is resolved. Unfortunately, neither are directly applicable to way-misprediction. Delaying the issuing of way-mispredicting loads, unlike bank conflicting loads, will still cause the load to mispredict, increasing the way-misprediction penalty even further. On the other hand, delaying the issuing of load-dependent instructions when a way-misprediction is expected is similar to the strategy for avoiding replays under cache misses. However, way-mispredictions have two properties that make them distinct from cache-misses: (1) the misprediction latency penalty is small enough that the dependent instruction still should be issued speculatively to minimise the negative impact in IPC, instead of waiting until the load is resolved as on cache-misses, and, (2) the way-misprediction penalty is deterministic, unlike cache-misses, which makes speculative scheduling of load-dependent instructions viable and trivial on both correct and way-mispredictions.

8. Conclusions

We have shown that the previously unstudied effects of instruction replay are the dominate performance concern for way-predictors in out-of-order processors, far surpassing the previously studied impact of increased latency from mispredictions. We have shown that this effect is largest for deep pipelines with longer issue-to-execute delays, but that it is still quite significant for shorter pipelines.

To address this, we proposed the use of confidence measuring to disable way-prediction and appropriately schedule dependent instructions when a load is likely to mispredict. This minimizes instruction replays and additional data array probes caused by standard way-predictors, improving performance and energy efficiency. We have shown that even a simple confidence measure technique is accurate enough to address the majority of the way-mispredictions without unduly disabling the way-predictor when it is accurate. We found that the increase in access latency due to way-mispredictions has a smaller effect on overall performance than replays, which allowed us to further improved the results by *biasing* the confidence measure to avoid the predictions that lead to replays, decreasing prediction accuracy, but further improving performance and energy.

Overall *Biased Selective Way-Prediction* reduces the performance penalty of a standard way-predictor from 6.9% to 2.9% by reducing the percentage of instruction replays from 10% to 2.9%. As a result, the additional dynamic energy of a way-predictor over a serial cache is reduced from 8.5% to 1.9%.

References

[1] R. Alves, S. Kaxiras, and D. Black-Schaffer, "Dynamically disabling way-prediction to reduce instruction replay," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pp. 140–143, IEEE, 2018.

[2] B. Calder, D. Grunwald, and J. Emer, "Predictive sequential associative cache," in *High-Performance Computer Architecture, 1996. Proceedings., Second International Symposium on*, pp. 244–253, IEEE, 1996.

[3] R. E. Kessler, R. Jooss, A. Lebeck, and M. D. Hill, *Inexpensive implementations of set-associativity*, vol. 17. ACM, 1989.

[4] K. Inoue, T. Ishihara, and K. Murakami, "Way-predicting set-associative cache for high performance and low energy consumption," in *Proceedings of the 1999 international symposium on Low power electronics and design*, pp. 273–275, ACM, 1999.

[5] M. D. Powell, A. Agarwal, T. Vijaykumar, B. Falsafi, and K. Roy, "Reducing set-associative cache energy via way-prediction and selective direct-mapping," in *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pp. 54–65, IEEE Computer Society, 2001.

[6] I. Kim and M. H. Lipasti, "Understanding scheduling replay schemes," in *Software, IEE Proceedings-*, pp. 198–209, IEEE, 2004.

[7] S. Palacharla, N. P. Jouppi, and J. E. Smith, *Complexity-effective superscalar processors*, vol. 25. ACM, 1997.

[8] J. Stark, M. D. Brown, and Y. N. Patt, "On pipelining dynamic instruction scheduling logic," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pp. 57–66, ACM, 2000.

[9] J. H. Edmondson, P. I. Rubinfeld, P. J. Bannon, B. J. Benschneider, D. Bernstein, R. W. Castelino, E. M. Cooper, D. E. Dever, D. R. Donchin, T. C. Fischer, *et al.*, "Internal organization of the alpha 21164, a 300-mhz 64-bit quad-issue cmos risc microprocessor," *Digital Technical Journal*, vol. 7, no. 1, p. 0, 1995.

[10] K. So and R. N. Rechtschaffen, "Cache operations by mru change.," *IEEE Trans. Computers*, vol. 37, no. 6, pp. 700–709, 1988.

[11] D. Ernst, A. Hamel, and T. Austin, "Cyclone: A broadcast-free dynamic instruction scheduler with selective replay," *ACM SIGARCH Computer Architecture News*, vol. 31, no. 2, pp. 253–263, 2003.

[12] P. Michaud and A. Seznec, "Data-flow prescheduling for large instruction windows in out-of-order processors," in *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pp. 27–36, IEEE, 2001.

[13] Y. Liu, A. Shayesteh, G. Memik, and G. Reinman, "Scaling the issue window with look-ahead latency prediction," in *Proceedings of the 18th annual international conference on Supercomputing*, pp. 217–226, ACM, 2004.

[14] G. Memik, G. Reinman, and W. H. Mangione-Smith, "Precise instruction scheduling," *Journal of Instruction-Level Parallelism*, vol. 7, pp. 1–29, 2005.

[15] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan, "Speculation techniques for improving load related instruction scheduling," in *ACM SIGARCH Computer Architecture News*, vol. 27, pp. 42–53, IEEE Computer Society, 1999.

[16] A. Perais, A. Seznec, P. Michaud, A. Sembrant, and E. Hagersten, "Cost-effective speculative scheduling in high performance processors," in *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, pp. 247–259, IEEE, 2015.

[17] E. Morancho, J. M. Llabería, and À. Olivé, "Recovery mechanism for latency misprediction," in *Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on*, pp. 118–128, IEEE, 2001.

[18] R. E. Kessler, E. J. McLellan, and D. A. Webb, "The alpha 21264 microprocessor architecture," in *Computer Design: VLSI in Computers and Processors, 1998. ICCD'98. Proceedings. International Conference on*, pp. 90–95, IEEE, 1998.

[19] G. Hinton, D. Sager, M. Upton, D. Boggs, *et al.*, "The microarchitecture of the pentium® 4 processor," in *Intel Technology Journal*, Citeseer, 2001.

[20] A. A. Merchant, D. D. Boggs, and D. J. Sager, "Processor with a replay system that includes a replay queue for improved throughput," Apr. 3 2007. US Patent 7,200,737.

[21] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[22] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP laboratories*, pp. 22–31, 2009.

[23] R. Alves, N. Nikoleris, S. Kaxiras, and D. Black-Schaffer, "Addressing energy challenges in filter caches," in *Computer Architecture and High Performance Computing (SBAC-PAD), 2017 29th International Symposium on*, pp. 49–56, IEEE, 2017.

[24] D. Nicolaescu, B. Salamat, A. Veidenbaum, and M. Valero, "Fast speculative address generation and way caching for reducing l1 data cache energy," in *Computer Design, 2006. ICCD 2006. International Conference on*, pp. 101–107, IEEE, 2007.

[25] J. E. Smith, "A study of branch prediction strategies," in *Proceedings*

of the 8th annual symposium on Computer Architecture, pp. 135–148, IEEE Computer Society Press, 1981.

[26] C. Zhang, F. Vahid, J. Yang, and W. Najjar, “A way-halting cache for low-energy high-performance systems,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 2, no. 1, pp. 34–54, 2005.

[27] D. Moreau, A. Bardizbanyan, M. Själander, D. Whalley, and P. Larsson-Edefors, “Practical way halting by speculatively accessing halt tags,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, pp. 1375–1380, IEEE, 2016.

[28] M. Ghosh, E. Ozer, S. Ford, S. Biles, and H.-H. S. Lee, “Way guard: a segmented counting bloom filter approach to reducing energy for set-associative caches,” in *Proceedings of the 2009 ACM/IEEE international symposium on Low power electronics and design*, pp. 165–170, ACM, 2009.

[29] G. Keramidas, P. Xekalakis, and S. Kaxiras, “Applying decay to reduce dynamic power in set-associative caches,” in *International Conference on High-Performance Embedded Architectures and Compilers*, pp. 38–53, Springer, 2007.