RESEARCH-ARTICLE

Towards the adoption of Local Branch Predictors in Modern Out-of-Order Superscalar Processors

NIRANJAN KUMAR SOUNDARARAJAN, Intel Corporation, Santa Clara, CA, United States

SAURABH GUPTA, Intel Corporation, Santa Clara, CA, United States

RAGAVENDRA NATARAJAN, Intel Corporation, Santa Clara, CA, United States

JARED WARNER STARK, Intel Corporation, Santa Clara, CA, United States

RAHUL PAL, Intel Corporation, Santa Clara, CA, United States

FRANCK SALA, Intel Corporation, Santa Clara, CA, United States

View all

Towards the adoption of Local Branch Predictors in Modern Out-of-Order Superscalar Processors

Niranjan Soundararajan[1] Saurabh Gupta[1] Ragavendra Natarajan[1] Jared Stark[2] Rahul Pal[2]
Franck Sala[2] Lihu Rappoport[2] Adi Yoaz[2] Sreenivas Subramoney[1]
niranjan.k.soundararajan@intel.com
[1]Processor Architecture Research Lab, Intel Labs, Intel Corporation [2] Intel Corporation

ABSTRACT

Branch prediction accuracy plays a dominant role in the performance provided by modern Out-of-Order(OOO) superscalar processors. While global history-based branch predictors are more popular, local history-based predictors offer an additional dimension towards enhancing the overall branch prediction accuracy. Integrating the local predictors in modern cores, though, comes with non-trivial challenges associated with managing the local predictor's state and repairing this state on any branch misprediction is essential for the local predictor to operate effectively. Using a highly accurate, industry standard simulator modeling a Skylake-like OOO core and workloads spanning diverse categories including Server, High Performance Computing (HPC) and personal computing suites, besides SPEC, we methodically highlight the issues that need to be tackled, why local predictor repair is non-trivial and the performance opportunity that is lost when the local predictor repair is not handled efficiently. We discuss the issues with prior techniques and quantify their limitations when using them in current OOO cores. Further, we propose three practical, implementable and efficient repair techniques with minimal storage requirements that provide significant performance gains for local predictors. Unlike prior repair techniques that can only attain 50% of the oracular gains, our realistic repair techniques retain about 80% of the oracular gains resulting in significantly better application performance.

CCS CONCEPTS

• **Computer systems organization** → **Superscalar architectures**; **Pipeline computing**.

KEYWORDS

Local predictors, Branch Prediction, Performance, Superscalar cores

ACM Reference Format:
Niranjan Soundararajan[1] Saurabh Gupta[1] Ragavendra Natarajan[1] Jared Stark[2] Rahul Pal[2] Franck Sala[2] Lihu Rappoport[2] Adi Yoaz[2] Sreenivas Subramoney[1] . 2019. Towards the adoption of Local Branch Predictors in Modern Out-of-Order Superscalar Processors. In *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3352460.3358315

1 INTRODUCTION

Even as domain-specific accelerators become popular, single thread (ST) performance for general purpose compute remains extremely relevant for real-world applications [27]. As we continue to optimize memory system design, there is still a non-trivial latency involved in fetching data from deeper levels of the memory hierarchy. To counter this, superscalar processors adopt deeper and wider pipelines looking for more independent instructions to extract the instruction level parallelism (ILP) [15, 28]. However, for the work done to be useful and have tangible performance improvements, a very efficient conditional branch predictor, capable of handling deep levels of speculation while maintaining a high accuracy, becomes highly imperative. Architects, therefore, are continuing to investigate newer designs to improve the branch prediction accuracy [22, 24, 31].

Branch predictors tend to use information built from the knowledge on how prior branches behaved. There have been several global history-based and local-history-based branch predictors proposed in literature [26, 29, 32, 37]. The global-history based predictors use information from all prior branches while local history-based predictors use information that is specific to individual branches when providing a prediction. TAGE is one of the most popular and accurate branch predictor designs [32]. Most of the recent Championship Branch Predictor (CBP) winners incorporate TAGE, a testament to its efficiency. TAGE relies on a single large global history which is used to access entries from its tables to make a prediction. Since TAGE relies on a single global history that gets updated for every branch instruction, it can be inefficient for branches whose outcomes do not exhibit much correlation with the global history [30]. Two-level local branch predictors [20, 21, 37], that utilize per instruction patterns to predict the branch direction, offer an alternate design choice to the global branch predictors. In these predictors, a Branch History Table (BHT) maintains the per-branch direction histories (local state) of multiple branches and this direction history is used to access a pattern table (PT) entry to make the direction prediction. Some proposals, including the recent Championship Branch Predictor 2016 winner(CBPw), have combined the global and local history predictors to lower MPKI (Mispredictions Per Kilo Instructions) [30, 31, 33].

Prior proposals on local branch predictors suffer from two major drawbacks. Firstly, they establish the efficiency of their predictor using only MPKI as the metric. Secondly, they perform their evaluations only on functional models that neither model modern speculative OOO processing nor intensively study the impact of the branch mispredictions on Instructions Per Cycle (IPC). This is especially important for local predictor designs since they carry significant per-PC state (in the BHT) that needs to be repaired and most (if not all) of their performance gains depends on repairing this per-PC state accurately and efficiently in time and space. By not repairing their per-PC state, we show that local predictors lose the opportunity to reduce MPKI by as much as 40-50%. Worse, for some workloads, we find that the local predictor actually does worse than the baseline system when its state is not repaired. Repair is the key issue preventing the adoption of local predictors in modern cores. To the best of our knowledge, only one prior work [34] has looked at the challenges in implementing a local predictor in an OOO processor. Given the benefits that local predictors can provide, repair definitely warrants further investigation for current day superscalar OOO cores.

Loop predictors [16, 30] are an important sub-class of local predictors. Unlike generic local predictors, loop predictors, as the name indicates, target loops by capturing their dominant exit iteration count and predicting this exit iteration in future occurrences of the loop. For loops, the predominant direction is a taken (T) and we exit the loop on a not taken (N). Loop predictors are more effective than generic local predictors for these branches since the number of bits required to capture the state is $log_2(iteration)$ which reduce the storage requirements significantly. In recent proposals like CBPw, the loop predictors have been extended to cover forward conditional branches (if-then-else) as well. It does it by reversing the branch state captured. For the forward conditional branches, the counter captures the predominant direction as the series of continuous NT which get terminated when the branch is T.

In our work, we use the loop predictor from CBPw-8KB category (referred as CBPw-Loop) as the primary vehicle to establish the performance potential that is available with local predictors. We also highlight the issues with adopting local predictors, in general, in OOO cores and how to tackle these issues. All the performance issues discussed and techniques proposed are extensible to any generic local predictor. The difference between the loop predictor and the generic local predictor is only in the state saved and restored. For the generic local predictors, the state is a sequence of bit-patterns while for the loop predictor the state is a counter. Given that we are not optimizing the contents of the local predictor state, our techniques can be directly extended to any local predictor design.

As a first step, we show the effectiveness of CBPw-Loop predictor using 200+ representative workloads drawn from diverse applications categories. We show that CBPw-Loop provides 31% reduction in MPKI over TAGE. We integrate CBPw-Loop in an OOO Skylake-based core [6] using a highly accurate industry-standard simulation setup. CBPw-Loop, with perfect and instantaneous repair of the local predictor state, gives 3.8% IPC gain over baseline TAGE across these 200+ workloads.

Studying local predictors using perfect and instantaneous repair of the per-PC speculative state provides an incomplete picture. It only quantifies the performance opportunity that is available since the perfect and instantaneous repair is only possible with unbounded resources to checkpoint state and the ability to repair the structures in zero cycles. As we will show, the primary challenge to tackle is the non-trivial state management associated with repair post mis-speculation events and evaluating the storage-performance-time tradeoffs therein. These perfect repair studies only establish the best-case performance potential but still do not pave the way for implementation in a real processor. Works like [34] offer a limited view of the issues involved, since the size of the repair structures they study is quite large. At realistic structure sizes, and increasing number of inflight instructions, repair issues significantly diminish the gains obtained from the local predictor and require efficient handling to translate the MPKI reductions to IPC gains.

In this work, we identify and address the issues that a realistic design of a local predictor presents and propose simple techniques that help harvest most of the gains that perfectly repairing the local predictor state can provide. The main contributions of this work:

(1) We establish that integrating local predictors in modern superscalar OOO pipelines involves non-trivial state management during recovery from mispredictions. Across the 200+ workloads, we see that in some cases we need to repair >60 PCs in the BHT which adds substantial complexity.
(2) We delve into the hardware issues that need to be tackled with handling repair in OOO cores, highlighting the elements that have not been discussed in earlier works and shedding light on why these issues are important.
(3) Using the loop predictor from CBP 2016 winner-8KB category (CBPw-Loop) as a demonstration vehicle, on 200+ workloads picked from Server, Office tools, personal computing and SPEC suites, we show that perfect and instantaneous repair of the BHT state can provide 3.8% IPC gain resulting from a 31% MPKI reduction. We show that prior techniques can provide only about 16.5% MPKI reduction resulting in about 2% IPC gain. The is about 52% of the best case gains.
(4) We propose three intelligent realistic repair techniques with limited storage requirements that efficiently extract most of the best case gains. The proposed realistic repair schemes establish the most efficient priority order in which to repair the PCs to retain a large part of the gains that is obtainable from perfectly repairing the local predictor state. Our repair techniques provide up to 27% MPKI reduction, resulting in 3% IPC gains, which is 79% of the best case gains. Compared to the prior techniques, our schemes significantly improve both the MPKI and IPC gains seen with the local predictor. The mechanisms we propose are easily extendable to work with any generic local predictor designs.

Section 2 provides a background and motivation for why efficient local predictor design is important in OOO cores. It also discusses local predictor repair and prior works in repairing local predictors. We propose our repair techniques in section 3 highlighting what each of these techniques enable. Post that, we discuss the different workloads we use in our study in section 4 and the evaluation setup in section 5. Section 6 shows the performance impact seen with prior techniques and our new techniques and we conclude our work in section 7.

2 BACKGROUND AND MOTIVATION

In this section, we summarize the differences in the operation and state management between global and local branch predictors. We also discuss the prior works on local predictor repair and highlight the complexities in handling local predictor repair. Finally, we motivate why handling this complexity is important.

2.1 Summary of branch predictor designs

It is widely understood that branch prediction accuracy and OOO superscalar performance go hand-in-hand [36]. This is due to the frequent occurrence of control flow instructions in program code coupled with the requirement that branch predictions be provided very early in the pipeline. As works like [27] point out, single thread IPC is critical for many modern workloads and therefore improving branch prediction accuracy remains relevant.

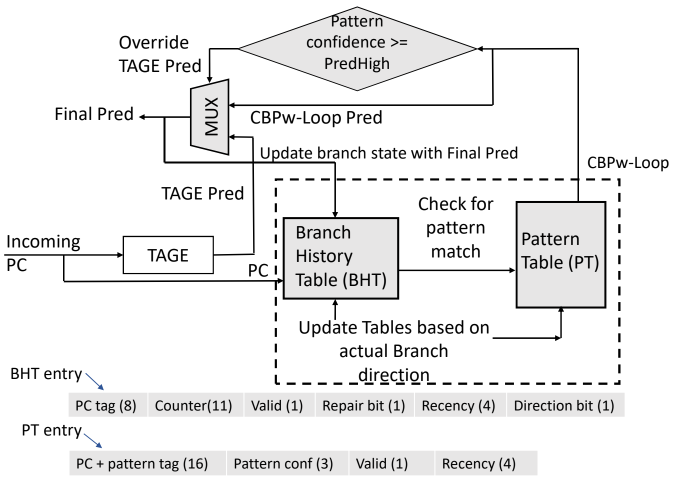


Figure 1: CBPw-Loop designed with a BHT to track the current state of the PCs and PT that provides a prediction. CBPw-Loop is an adjunct predictor to TAGE.

Conditional branch predictors have evolved over time. Some of the predictors use the global history of all prior branches and others use only the specific branch's prior history to predict the direction. Hybrid predictors use both histories to provide even better branch predictions. The TAGE predictor, proposed in [32], is composed of a tagless bimodal component and a set of partially tagged global-history based tables. The bimodal [35] component is simple and handles most of the predictions. The global tables are accessed based on an index computed using the branch PC and the global path history (PHIST) and direction history (GHIST). At prediction time, the bimodal component provides a prediction which can get overridden by the global table with the longest matching history, provided the tag matches. Allocations are made when there is a misprediction of the branch direction. These allocations can happen across multiple global tables depending on entry availability and the table that provided the prediction.

2.2 Two-level Local Predictors

Local predictors offer an effective design alternative to global history-based predictors. Program code involving branches that are less dependent on global history like constant iteration loops, loops whose exit iteration counts have low entropy, if-then-else branches with repeating patterns lend themselves naturally to local predictors. Two-level local predictors, as proposed in [37], work by tracking the recent history of branch directions at a per instruction granularity in a Branch History Table (BHT). This history is then used to access a pattern table (PT) which provides a prediction for the given history. The BHT tracks the recent history (or "state") per instruction. Accuracy of this history is important for a local predictor to give the right prediction. At any point, the BHT tracks the history of several instructions. Post the execution of the branch, the eventual branch direction is updated in the BHT history and any confidence value and/or bias counters tracked in the PT are updated.

2.3 CBPw-Loop

CBPw incorporates TAGE together with a loop predictor and multiple predictor components within the statistical corrector (SC). The loop predictor in CBPw targets both backward (loops) and forward (if-then-else) branches. It tracks specific patterns which are sequences of TTT...N or NNN...T. The periodicity (or iteration count) of the branches are captured using a counter and if it builds enough confidence, the loop predictor gives a prediction.

We use this loop predictor to discuss the repair issues that need to be tackled and also to evaluate the repair techniques. While the SC also incorporates a generic local predictor, we found our workloads to be more sensitive to the loop predictor. As mentioned earlier, the repair issues discussed here apply for any local predictor and all our techniques proposed to alleviate these issues are also extensible to any generic local predictor.

Figure 1 shows how the CBPw-Loop is setup and how it works with TAGE. We redesigned the CBPw-Loop and adopted a conventional two-level design taking the OOO system design into account. The loop predictor in CBPw has one single table tracking both the final iteration count and the current iteration count. Since these two tables get updated at different times, which we discuss in section 2.4, we split this predictor table into the BHT tracking the current iteration count and PT which gives the final iteration count to lower the port requirements on the predictor table. Now only the BHT gets updated post giving a prediction. PT is updated only after the branch completes execution. Also, the local predictor repair is related to only recovering the current iteration count (current BHT state), this design simplifies the repair logic to pick only the BHT state.

2.3.1 *State management in local predictors.*

Superscalar processors have several instructions in flight within the pipeline. Most of these instructions are on the right path of execution but due to events like branch mispredictions and exceptions, some of these instructions need to be flushed from the pipeline. Subsequent to these flush events, pipeline state needs to be recovered to the right state for the local predictor to provide the correct overrides. Part of recovering the pipeline state involves repairing the branch state. For global predictors like TAGE this involves recovering the state of their GHIST and PHIST registers. Each instruction carries its

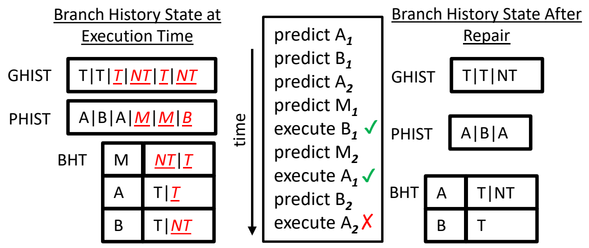


Figure 2: Capturing the changes to Global and local history based predictors pre and post a branch misprediction. A, B and M are different PCs and 1,2 refer to different instances of the same PC.

pre-updated GHIST/PHIST state and uses it to repair the registers on a misprediction. This greatly simplifies repair for the global predictors since it involves only the GHIST and PHIST registers and therefore can be done in a deterministic number of cycles. On the other hand, for local predictors, recovery involves repairing the BHT state across several entries which belong to multiple instructions. Figure 2 highlights this using an example. As seen, irrespective of the number of instructions in the pipeline, for global predictors the steps involved in branch state recovery is constant. For the local predictor, its BHT state recovery has to repair all three instructions (A, B and M). This number increases as we encounter more PCs in the program flow. Also, increasing pipeline depth and width increases the number of inflight instructions which also adds to the state that is tracked in the BHT.

This non-trivial recovery of BHT state is the single biggest hindrance to widespread adoption of local predictors in modern OOO processors, as observed by earlier works as well [33]. As we show below, restoring the BHT state is critical as it determines how effective a local predictor can be.

2.4 Pipeline events associated with integrating local predictors in OOO core

Figure 3A highlights the main events that occur when CBPw-Loop (or any other local predictor) is integrated in an OOO core. The predictor typically gets accessed from different pipeline stages either to provide a prediction or to update its state. These events include,

(1) The incoming PC accesses the BHT to get the current pattern/counter (state).
(2) If the PC exists, the current state is checkpointed in the checkpoint structure (CS). Every PC carries an CS id corresponding to the entry in which the PC got checkpointed.
(3) The PC and the current state are used to access the PT for a prediction.
(4) If there is a hit in the PT, the final prediction is chosen from TAGE's prediction and the one from the local predictor.
(5) The final prediction is used to update the BHT
(6) Once the instruction completes execution, if the prediction turns out correct, no update for the BHT is required since it has already been updated to the correct state. Only the TAGE and PT confidence counters are updated. On a misprediction, besides updating the TAGE and PT counters, the CS is accessed and BHT repair is triggered.
(7) BHT is recovered to its state prior to the mispredicting branch and its state is updated based on what the branch execution provides.

2.5 Issues with repairing the BHT state in OOO cores

Given that the OOO core pipeline is active when BHT repair is going on, repair techniques have to take the pipeline dynamics into account. Figure 3B captures the different issues that need to be tackled while building a realistic repair infrastructure for a local predictor. These include,

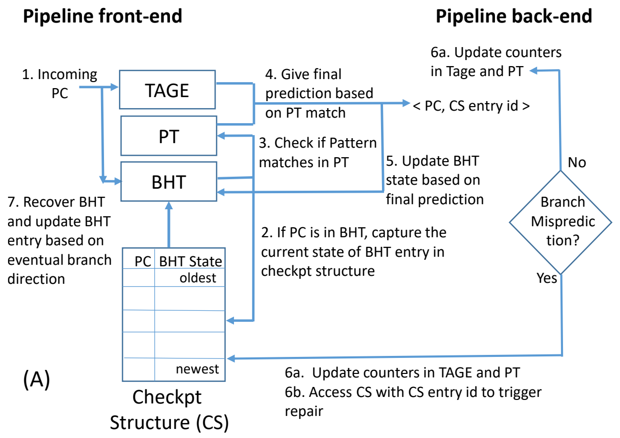
a) **BHT is unavailable during repair:** Due to port limitations and design complexity, it may take several cycles to repair the BHT to its correct state. During this time the BHT cannot provide predictions for an incoming PC since the pattern/counter has not been repaired.
b) **BHT state cannot be checkpointed correctly during repair:** During repair, if a new instance of a PC present in the BHT enters the pipeline, the BHT cannot get updated or checkpointed since its current state is not correct. Alternately, we could increase the misprediction penalty by stalling the pipeline, recover the BHT and then proceed. But this results in a high performance loss, as expected.
c) **Handling multiple repairs:** Even when the BHT is recovering, execution of other branches in the pipeline can result in more mispredictions and hence trigger newer repair events. Due to OOO nature of the pipeline, the mispredicting branch could be older in program order and therefore the repairs need to be handled correctly. The newer branch also could trigger a BHT repair which needs to be merged with the ongoing BHT repair.
d) **Size of the checkpointing structure:** The checkpointing structure will be a limited size structure and depending on the workload characteristics can run out of entries. This will lead to the BHT entries not getting checkpointed which may hurt performance due to incorrect repair of the BHT. This can happen often as the front-end runs much ahead of the back-end and as we increase the pipeline depth in the front-end, the amount of state to hold increases and along with it the associated complexity of state management.

The repair schemes we develop take these factors into account and provide practical and implementable techniques to achieve BHT repair to make local predictors practical and effective in OOO cores.

2.6 Prior works on local predictor repair

Skadron et al. [34] elaborately study the speculative repair issues in local predictors and propose techniques that preserve and recover the BHT state. This includes the History File and Future File based techniques discussed below.

In History file based repair, the BHT is speculatively updated after giving a prediction. Before updating its state, the BHT records its pre-update state in a outstanding branch queue (OBQ) which serves as a History File. Each entry in the OBQ holds the PC and pre-update BHT history of the PC. On a misprediction, entries from the OBQ are walked "backwards" from the youngest entry up to the


Pipeline front-end
Pipeline back-end
1. Incoming PC
TAGE
PT
BHT
4. Give final prediction based on PT match
< PC, CS entry id >
6a. Update counters in Tage and PT
3. Check if Pattern matches in PT
5. Update BHT state based on final prediction
No
Branch Misprediction?
Yes
7. Recover BHT and update BHT entry based on eventual branch direction
2. If PC is in BHT, capture the current state of BHT entry in checkpt structure
PC BHT State
oldest
newest
(A)
Checkpt Structure (CS)
6a. Update counters in TAGE and PT
6b. Access CS with CS entry id to trigger repair

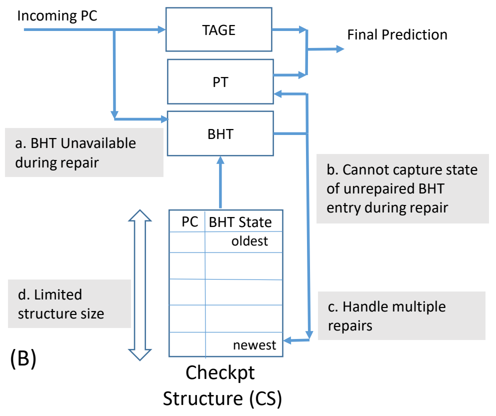

Incoming PC
TAGE
PT
BHT
Final Prediction
a. BHT Unavailable during repair
b. Cannot capture state of unrepaired BHT entry during repair
PC BHT State
oldest
newest
d. Limited structure size
c. Handle multiple repairs
(B)
Checkpt Structure (CS)

Figure 3: (A) Interaction between the BHT, PT and checkpointing structure which helps to save and restore BHT state. (B) Different issues that need to be tackled when implementing a realistic local predictor repair scheme.

entry corresponding to the mispredicting instruction and the BHT is repaired to its state prior to the misprediction. Going forward, we refer to this technique as *backward walk-based HF repair*.

In a future file (FF) based repair, the OBQ records the latest pattern speculatively in its entries while BHT is only updated at instruction retirement. Newer instructions entering pipeline access the OBQ for the current pattern. While this technique simplifies repair, it only involves reverting the OBQ's tail pointer to the mispredicting instruction, it adds significant complexity to the common-case branch prediction path. Since the updated state of a PC can be in any OBQ entry, it requires an associative search of all the OBQ entries to identify the correct entry. Beyond 8-way or 16-way associativity, the associative access of the FF becomes a performance bottleneck and a power hog. We therefore do not discuss this design any further.

Similar to register file and register alias table (RAT) checkpointing [19], we can checkpoint/take snapshots of the BHT to enable the repair [34]. At each update to BHT, the state of all the BHT entries get captured in a snapshot and these snapshots can be maintained in a snapshot queue (SQ). While simple, snapshots cost significantly in terms of storage. Also they increase the number of ports in the BHT which has storage and complexity implications.

While these techniques address the need to save and restore BHT state, they do not tackle 1) the impact of realistic-sized OBQ and related storage-performance tradeoffs, 2) the impact of limited set of read and write ports to update state BHT, 3) the challenge of handling multiple repair events in deeply pipelined OOO cores, 4) unavailability of BHT to give predictions when repair spans over multiple cycles. We intensely grapple with these challenges and propose multiple solutions to efficiently integrate a local branch predictor into modern OOO processor pipelines.

2.7 Need for local predictor repair

To understand why tackling these issues is important, in Figure 4 we show the MPKI opportunity a highly accurate local branch predictor with no misprediction provides. The data is across the 200+ general-purpose workloads (refer section 4) we study in this work. The MPKI reduction is about 44% across the workloads. Not all of these

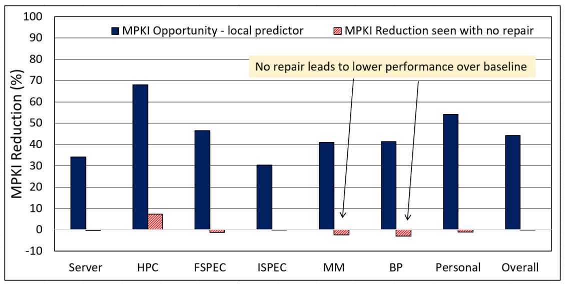


Figure 4: Highlighting the MPKI opportunity that local predictors have across workload categories and the limited returns we get from not repairing the local predictor.

gains are attainable due to cold branch misses and data entropy but the data shows the large opportunity that local predictions can provide in current OOO cores. The figure also shows the fraction of these misses avoided using a local predictor which does not repair its BHT state. We see that almost all of the potential MPKI reduction opportunity is lost thus emphasizing the importance of repairing the BHT state for efficient local predictor design. Multimedia (MM) and business productivity (BP) workloads actually lose performance with respect to the baseline when the BHT state is not repaired. This data shows why efficient repair techniques are critical for local predictors to realize their performance potential.

3 REALISTIC REPAIR TECHNIQUES FOR LOCAL PREDICTORS

In this section, we present our new repair techniques and how they efficiently handle repairing the BHT state with limited additional hardware and complexity requirements.

3.1 Forward walk-based HF repair

We observed that with backward walking the OBQ, from the youngest entry to the entry corresponding to the mispredicting instruction, to repair the BHT, the same PC in BHT may get updated multiple times. Given that BHT and OBQ come with limited read and write

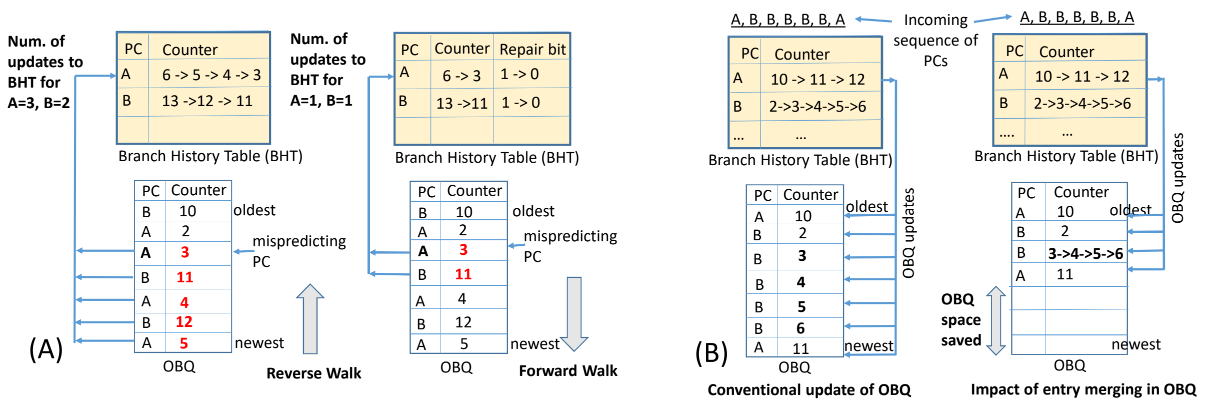


Figure 5: (a) Benefits of forward walk over backward walk. BHT entry shows the transition of the pattern counter as the repair walk happens, and (b) Impact of merging multiple OBQ entries.

ports, updating the same entry multiple times can delay another PC from repairing and therefore have a significant performance impact. Figure 5a shows this in detail where PC-A and PC-B get updated multiple times as OBQ entries are walked backwards. To address this, we propose a simple but very effective tweak in which instead of walking backwards, we start the repair at the mispredicting PC and walk *forward*, towards younger entries in OBQ. This provides twin benefits, 1) Each instruction is only updated once in the BHT. By capturing a 1 bit flag per BHT entry that flips when an instruction is repaired the first time (per repair), we avoid overwrites. We call this a *repair bit*, refer Figure 1. The repair bit gets set across all entries when repair starts and is set to 0 when the first update is applied to an entry. 2) In deeply pipelined cores, instructions from the correct path are temporally close to the mispredicting instruction (post a reconvergence point). Therefore, repairing these instructions early is critical for the local predictor to have more opportunities to override the baseline prediction. Repairing the instructions in this order allows the local predictor to start giving predictions for these PCs since they are already in their right state. This is not the case with backward walk since it starts at the youngest entry and therefore cannot guarantee that a PC's state is repaired to the correct state till it completes repair.

To limit the OBQ from running out of entries, we propose an optimization. When multiple instances of the same PC get checkpointed consecutively in the OBQ, the intermediate instances of the PC aren't provided a separate entry. They are all (logically) merged to a single entry and hence carry the same OBQ entry id. Only the first and the last instance remain in the OBQ as separate entries. As Figure 5b shows, this saves OBQ entries which in turn gives a performance boost.

To enable this optimization, each instruction carries its pre-update BHT state (11-bit counter) through the pipeline. Now, during repair, if it is an instruction before this PC sequence that mispredicts we then use the first instance of the PC in the OBQ to repair the PC's state. If the mispredicting PC is after the last instance in the sequence, then there is no recovery needed. If one of the intermediate instances of the PC gets mispredicted, we use the state carried with the instruction to recover the BHT state. For the other instructions, they get repaired from the OBQ depending on their entry ids. Assigning the same OBQ entry id is to maintain the causal order of this PC with respect to other PCs. We find that this optimization allows the mispredicting PC to recover quickly and also reduces the number of entries we need to access on a repair.

Despite this optimization, the OBQ can still run out of entries. During these periods, we continue to update the BHT state speculatively but the PCs that enter the pipeline are not assigned an OBQ entry id. In case one of these instructions mispredict, the OBQ state is not recovered. Note that patterns in the BHT can recover to the right state when the corresponding PC flips direction and the counter is reinitialized.

Lastly we handle multiple mispredictions based on the program order of the PCs. Younger branch PCs already in the pipeline get flushed and do not trigger a repair. For PCs older than the current PC that triggered the misprediction, we restart repair. The repair bits, for all BHT entries, gets set again to allow entries that have been repaired to be updated once again.

3.2 Multi-stage prediction

Since BHT repair using forward walk can still take multiple cycles, providing predictions for newer branches entering the pipeline and repair may need to happen simultaneously. This adds to the BHT port requirements and in turn increases the design complexity. Here we investigate the impact of *deferring* the CBPw-Loop from the branch prediction stage to a later pipeline stage. The deferring allows BHT to fully repair its state before needing to provide its next prediction. While this simplifies the design requirements, we now have a two-stage branch prediction where TAGE provides an early prediction and later CBPw-Loop overrides it. Prior works like [23] have shown overriding to be effective when predictors need more cycles to provide better predictions. In our studies, we defer the CBP-Loop predictions to just before instructions enter the allocation queue [2]. Whenever CBPw-Loop's prediction overrides the baseline prediction, we trigger an *early misprediction* and re-steer the pipeline based on the direction CBPw-Loop provides. This change requires CBPw-Loop's prediction to be even more accurate as a misprediction from it now suffers the penalty of re-steering the pipeline (due to the early misprediction) together with the actual misprediction penalty post execution.

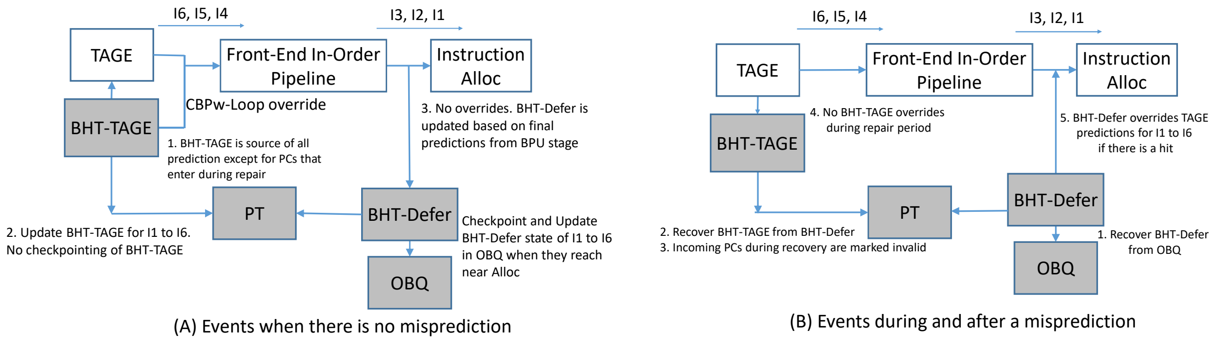

I6, I5, I4
I3, I2, I1
TAGE
Front-End In-Order Pipeline
Instruction Alloc
BHT-TAGE
CBPw-Loop override
1. BHT-TAGE is source of all prediction except for PCs that enter during repair
3. No overrides. BHT-Defer is updated based on final predictions from BPU stage
PT
BHT-Defer
2. Update BHT-TAGE for I1 to I6. No checkpointing of BHT-TAGE
Checkpoint and Update BHT-Defer state of I1 to I6 in OBQ when they reach near Alloc
OBQ
(A) Events when there is no misprediction
4. No BHT-TAGE overrides during repair period
5. BHT-Defer overrides TAGE predictions for I1 to I6 if there is a hit
2. Recover BHT-TAGE from BHT-Defer
3. Incoming PCs during recovery are marked invalid
1. Recover BHT-Defer from OBQ
(B) Events during and after a misprediction

Figure 6: Multi-stage CBPw-Loop prediction with Split BHT

3.2.1 *Multi-stage prediction with split BHT.*

To minimize the impact of early mispredictions, we split the BHT into BHT-TAGE and BHT-Defer. Each of the tables have half the BHT entries as what we had otherwise. Figure 6 shows the overall design and the different events when there is no misprediction and events post a misprediction and how repair is handled. The BHT-TAGE, present in the branch prediction stage (along with TAGE), can now provide the overrides immediately without a penalty. None of the BHT-TAGE entries get checkpointed in the OBQ but the patterns get speculatively updated based on the incoming instructions. BHT-Defer, on the other hand, is at the alloc stage and its entries get checkpointed in the OBQ. Both these tables operate independently depending on the instructions they see at their pipeline stage. The PT is accessed by both the BHTs. In case sharing the PT is difficult between these two stages, we also investigate the effect of splitting the PT.

On a repair, the BHT-Defer recovers its state from the OBQ. Post its recovery, the BHT-TAGE gets repaired from the BHT-Defer entries. We use the repair bits in the BHT-Defer entries to identify the PCs that got repaired and update only those PCs in the BHT-TAGE. During this *repair period* - start of repair in BHT-Defer to when BHT-TAGE repair is completed - PCs that entered the pipeline do not get any predictions from BHT-TAGE. They are handled by BHT-Defer (since it is later in the pipeline). This 2-stage repair prevents CBPw-Loop from losing coverage when BHT-TAGE is getting repaired. Valid bits for the instructions that enter the pipeline during the repair period are reset in BHT-TAGE to avoid incorrect overrides. The valid bits get set once the PCs flip their directions and their counters are reset. BHT-TAGE does not provide any predictions during the repair period and therefore this design does not need any additional ports to support repair. The same ports used for providing predictions are sufficient to support repair. Since BHT-Defer is much later in the pipeline, new instructions entering the pipeline take time to reach it. This provides sufficient time for BHT-Defer to repair its state in most cases. If an instruction reaches BHT-Defer in the middle of its repair, it does not provide any predictions for this instruction and its state is marked invalid. In our studies, this was a very rare event.

3.3 Limited-PC repair

OBQ-based repair techniques need additional structures to repair the BHT state. Further the duration of the repair is not deterministic as it can be done in one cycle or span several cycles. These factors add to the design complexity. One of the learnings we had when developing the forward walk technique was that **not all PCs are equally important to be repaired**. The primary reasons include,

a) The PC does not benefit from the local predictor.
b) The current pattern associated with a PC, even if wrong, if it does not hit in the PT it does not trigger a wrong prediction.
c) The pattern exists in the PT but the confidence value being low cannot trigger an override of TAGE's prediction. Saving and restoring these PCs does not help performance.
d) A branch's pattern even if it is not repaired can return to the right state. So the wrong state is temporary. For the loop predictor, this happens when the branch PC changes direction. For a generic local predictor, this occurs when the incorrect state is shifted out of its history. During this period if the PC does not hit in the PT, there are no additional mispredictions that get triggered.

Based on these observations, we decided to look at a non-OBQ technique in which we limit the repairs in the BHT. On every misprediction, we identify "M" PCs to repair. The pre-update BHT state of these PCs (24 bits per PC - 5-bit set, 8-bit tag and 11-bit pattern) is carried with each instruction through the pipeline, similar to them carrying the GHIST and PHIST for TAGE's history recovery. On a misprediction, these M PCs are repaired from the state carried by the instruction. While limiting the coverage, this simplifies design since we will know the exact number of cycles required to complete repair.

To select the set of M PCs to repair, we developed the following heuristic,

- Pick from the recent set of PCs that did a correct override of TAGE's predictions. If there are more than M PCs, we use LRU to replace the instructions in this set.
- If there are lesser than M PCs, the remaining PCs are picked based on recency of updates in the BHT
- We always repair the state of the instruction that mispredicted. This mean each instruction picks M-1 other instructions based on the above two criteria

This heuristic works because it covers utility and recency - utility in terms of picking PCs that gave right predictions and recency in terms of picking PCs that occurred in the immediate past (as seen by the BHT). Allowing an instruction to always pick itself is driven by recency.

For the instructions that do not get repaired (non-repaired PCs), we tried two policies: Mark the non-repaired PCs invalid in BHT so that they do not give any more predictions or alternately leave them as is. Leaving them as is simpler to implement and we found that this scheme also results in better gains. This is because the BHT tracks several PCs and a portion of these PCs are outside the scope of the mispredicting PC. These PCs are in their right state in the BHT and do not need to be repaired. By marking them invalid, we lose the opportunity to override the baseline prediction for these PCs.

In the following sections, we show the different workloads we use in this study, provide details on the system on which we evaluate and using these we establish the performance gains from these different repair schemes in section 6.

4 WORKLOADS

Our workload list includes more than 200 workloads spread over multiple application suites. The workloads were characterized using a Simpoint-like methodology [18] to identify the representative phases. Table 1 shows the distribution of the workloads and the broad categories they fall into. We picked ones that had reasonably high MPKI while also containing branch PCs exhibiting local patterns. We picked these workloads to show that when the workload is sensitive to a local predictor which brings in MPKI gains, our repair techniques can help retain those gains.

Category	**Description**	**Count**
Server	Data analytics on Hadoop, compression on cloud, streaming using Spark [11], BigBench [17], Transaction process using Cassandra [4], SPECjbb [14], Web search, Particle rendering	29
HPC	HPlin-pack [7], SPECmpi [9], Molecular dynamics, Signal processing application, FFT processing	8
ISPEC	ISPEC06 [8] & ISPEC17 [13]	34
FSPEC	FSPEC06 [8] & FSPEC17 [13]	64
Multimedia(MM)	Photo-editing, Animation, Video conversion, Mediaplayer	15
Business Productivity(BP)	SYSmark [10], PDF editing, Email, Presentations, Spreadsheet & Documents	16
Personal	Email, Voice-to-text tools, Image converters, Games, Mobilexprt [3], Geekbench [25], Tabletmark [5], EEMBC [12]	36

Table 1: Evaluated benchmark categories

5 EVALUATION SETUP

Our performance evaluation is done using our in-house cycle-accurate simulator modeling a Skylake-based x86 core clocked at 3.2 GHz. The microarchitectural parameters are similar to the latest Intel Skylake processor [6] and we list some of the relevant parameters in Table 2. Our core model incorporates a uop cache of appropriate size [1] to quickly feed the back-end. We integrate TAGE, from the CBPw-8KB category [31], and CBPw-Loop in this setup to enable the study. Since the core is 4-wide, BHT and PT require 4 read and write ports. This is similar to what TAGE requires to update its entries. BHT will share the 4 write ports between updating the counters post giving a prediction and those coming post the branch execution. This is possible since the updates coming from execution is needed by BHT only after a branch misprediction, refer section 2.4.

We first establish the potential MPKI and IPC gains that perfectly repairing CBPw-Loop can provide. This data is key to show the gains that CBPw-Loop can provide and establish an upper bound performance for the repair techniques. We then discuss the repair mechanisms in detail and establish the efficiency of each scheme by normalizing their performance with what we see with perfect repair. We show that even with limited storage and port requirements, most of the perfect repair gains can be attained which was not possible using prior techniques.

Core	4-wide OOO, 224-entry ROB, 64-entry allocation queue, 72-entry load buffer, 56-entry store buffer
Baseline Branch Predictor	TAGE - 7.1 KB
Branch Target Buffer	2K entries
CBPw-Loop256	256 entries, 8-way BHT, PT - 1.5 KB
CBPw-Loop128	128 entries, 8-way BHT, PT - 0.75 KB
CBPw-Loop64	64 entries, 8-way BHT, PT - 0.38 KB
L1 cache	Private, 32KB, 64B line, 8 way, 5 cycles latency, prefetchers enabled
L2 cache	Private, 256KB, 64B line, 8 way, 15 cycle latency, prefetchers enabled
LLC	Shared, Inclusive, 8MB, 64B line, 16 way, 40 cycle latency, prefetchers enabled
Main Memory	Dual channel DDR4-2133MHz

Table 2: Simulator Parameters

OBQ design: The OBQ operates as a circular buffer with newer entries added to the tail. Entries are evicted when the corresponding instruction retires from the pipeline. In our studies, each entry is 76 bits, with a 64-bit PC, 11-bit pattern and the valid bit from the BHT entry. We can avoid storing the PC by recording the set, way and tag information from the BHT entry which reduces the entry size to 26 bits but for simplicity we went with the PC. Entries in the OBQ are evicted only when the corresponding instruction retires.

Also, to enable repair, every incoming PC, even if it does not hit in the BHT, is assigned an OBQ entry id. For PCs that hit in the BHT, the entry id is set to the tail pointer while for those that do not hit in the BHT, it is the entry before the tail pointer.

6 RESULTS

In this section, we first show the performance opportunity that CBPw-Loop provides when we can perfectly and instantaneously repair its state. Using this as the target, we discuss the performance that different repair schemes can provide including prior techniques that have been proposed. We summarize the different schemes in section 6.6 and conclude the section with sensitivity studies on a larger TAGE design.

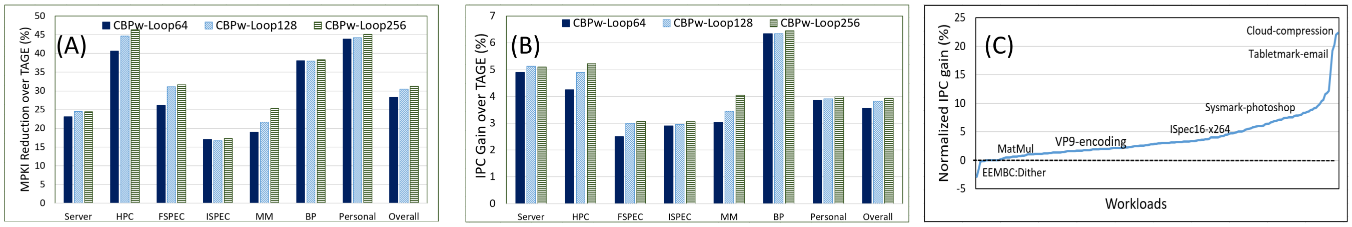


Figure 7: (a) MPKI reduction CBPw-Loop provides on top of TAGE across different workload categories, and (b) IPC gain across the different categories. (c) IPC S-curve for CBPw-Loop128 highlighting some of the workloads

6.1 Performance opportunity with perfect repair of BHT state

In this section, we show the best case gains that *perfectly repairing* CBPw-Loop can provide with unbounded resources and also being able to instantaneously repair (not taking any cycles) its state on a misprediction.

We breakdown this section by looking at the MPKI reduction that CBPw-Loop provides and then discuss the performance impact that this reduction translates into. Table 2 shows the different CBPw-Loop configurations we study below.

Figure 7a captures the MPKI reduction that CBPw-Loop provides over TAGE. CBPw-Loop64, with 64 entries in the BHT and PT, provides about 28.3% MPKI reduction which increases to 30.5% with CBPw-Loop128. CBPw-Loop256 does slightly better and provides about 31.2% MPKI reduction. Most categories of workloads see reductions of at least 15% (on top of TAGE) with the HPC, BP and Personal computing categories seeing the highest reduction. Server workloads typically see a lot of distinct branch PCs whose state need to be tracked. CBPw-Loop is able to effectively capture the right set of PCs to provide good overrides. Further, since CBPw-Loop supports both loops and if-then-else branches, it works well for Server and ISPEC workloads which have a reasonably good combination of both branch types. Figure 7b captures the IPC gains for the same configurations. Overall we see 3.6% IPC gain from CBPw-Loop64 which increases to 3.8% for CBPw-Loop128 and 3.95% for CBPw-Loop256 on top of TAGE. Most categories, except FSPEC, see around 3% gain. Figure 7c shows the performance S-curve capturing different workloads and their IPC gain. Workloads like data compression on cloud (Cloud-compression) and Tabletmark-email are very sensitive to the local predictor and see > 15% IPC gain. EEMBC-Dither sees IPC loss primarily because the BHT and PT tables see thrashing from the number of PCs tracked in the BHT and PT tables. On the CBPw-Loop256 config, this workload sees 0.7% IPC gain.

Figure 8 shows the average and maximum number of repairs required in the BHT per misprediction, to enable the perfect and instantaneous repair, across the different workloads. We see that, in some workloads, even the average the number of repairs required is as high as 16, resulting in those many writes in the BHT to repair it to the correct state. As the figure shows, in the worst case, this can be as high as 61 writes. This data helps establish why local predictor repair is non-trivial and unless we have efficient techniques the overhead to support repair can limit the adoption of local predictors.

Taking the performance and storage requirements into consideration, going forward, we use CBPw-Loop128 as the default configuration. Since perfect repair captures the best case performance, all the performance graphs below are normalized to the 3.8% IPC gain we see with CBPw-Loop128.

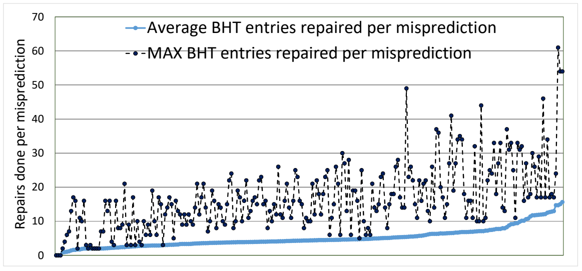


Figure 8: The average and maximum number of repairs required to repair the BHT per misprediction across the different workloads

6.2 Performance impact of prior techniques

In this section we discuss the performance gains provided by the prior techniques, discussed in section 2.6, and their limitation in a realistic setup.

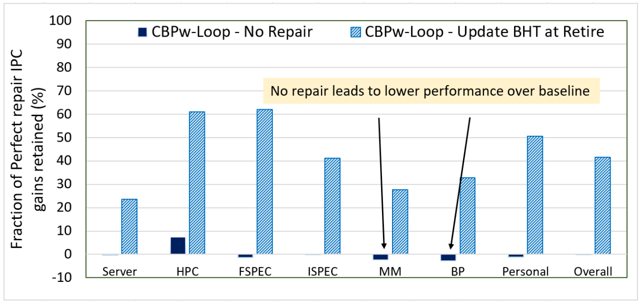


Figure 9: IPC impact when BHT is updated at retire and when not doing any repair

We first characterize the impact of updating BHT at retire stage when the instructions retire. Typically, the BHT is updated after giving a prediction but this is a speculative update since its based on the branch's direction prediction. By updating BHT at retire we avoid this speculative update and only update based on the actual direction. This avoids the need to track and repair the speculative state but the update to the BHT state is delayed till the branch retires. Figure 9 shows that the performance impact across the different workload categories. As seen, not repairing the BHT state results in the local predictor providing no performance on top of the baseline system. Updating BHT at retire gives only about 41% of the perfect repair gains. The primary reason for the lower performance is the delayed update to BHT state which is insufficient to extract higher performance gains. As pipeline depth increases in future cores, and the number of instructions in the pipeline increases, the effect of the delayed update will only exacerbate. We also show the impact from not repairing the local predictor to reiterate the claim we made in section 2.7 that efficient repair techniques are critical to realize the performance gains from local predictors.

Figure 10 shows the performance characteristics of backward walk-based HF and snapshot-based repair. When the number of entries in the repair structures (OBQ or SQ) and the ports to read and write to them are high (64-64-64), both techniques are quite efficient and retain large portion of the gains that perfect repair provides. Even with a smaller OBQ or SQ (16 entries), more than 65% of the performance is retained by backward walk. But once we make the number of ports reasonable, the performance quickly drops. With SQ, even the 32-8-8 configuration retains much less than 50% of perfect repair gains while backward walk retains 50% of the perfect repair gains.

Going back to Figure 8, we see that the average number of repairs required across the workloads is about 5 on a misprediction (can be as high as 15 for specific workloads). Backward-walk gets limited by the need to walk through all entries before it can start giving predictions again. Even for the average case of 5 repairs, with 4 additional ports for repair, every misprediction takes 2 cycles to repair BHT before its ready for providing the next prediction. Workloads like tabletmark-email and Sysmark-photoshop which gained significantly with perfect repair (refer Figure7c) suffer because these workloads have much higher repairs required (>7 PCs repaired) than average. This leads to the repair spilling over multiple cycles and in turn the local predictor being unavailable to provide predictions for incoming instructions and thereby lowering the performance potential.

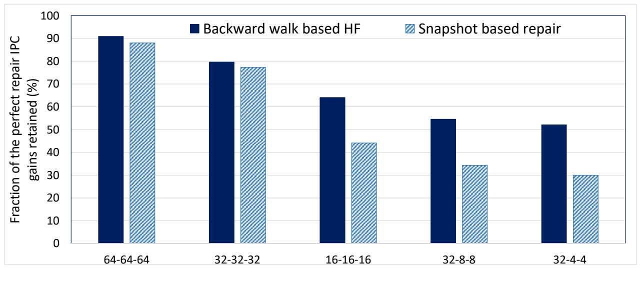


Figure 10: Performance of backward walk based HF and Snapshot scheme across different configurations. Config M-N-P stands for M OBQ/snapshot entries, N OBQ/snapshot read ports, P BHT write ports

6.3 Forward walk-based HF repair

Figure 11 shows the performance gains that the forward walk mechanism provides. Expectedly, a smaller OBQ (going from 64 OBQ entries to 32 OBQ entries) lowers the performance since some PCs do not get checkpointed. Also, the configurations with fewer read/write ports give correspondingly lower performance compared to those with more ports due to the additional time needed to complete repair. Interestingly, the FWD-32-4-2 configuration retains 76% of the gains that oracular repair provides. Further, the last bar in the figure shows the impact on FWD-32-4-2 configuration when merging of OBQ entries is enabled. This gives an additional 3.5%, taking the overall gains from forward walk to 79.5%. This design requires about 4 ports in the OBQ to read and write to its entries and 2 write ports in BHT to repair its entries. With forward walk, Sysmark-photoshop and Tabletmark-email (refer Figure7c) are now able to retain most of the perfect repair IPC gains showing the effectiveness of this technique.

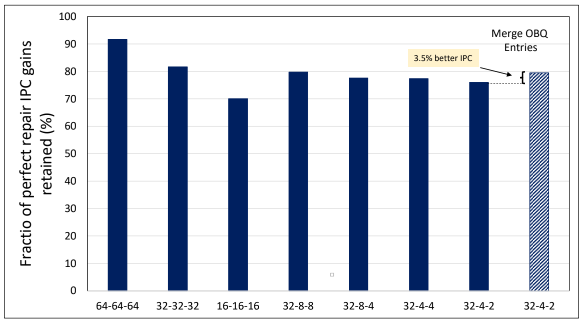


Figure 11: Performance impact of the forward walk mechanism. The config $M - N - X$ expands to M OBQ entries, N OBQ read ports, X BHT write ports.

6.4 Multi-stage prediction with split BHT

Figure 12 shows the performance gains with this technique. The dark bars are the performance gains from forward walk while the other two bars are with the multi-stage design. The striped bars show the performance when the PT is shared and the chequered bars are if we split the PT also between the two stages. Depending on whether the PT can be accessed by BHT-TAGE from the instruction alloc stage, the choice can be made to pick the corresponding design. The gains are lower than forward walk due to the additional delay required for re-steering from the alloc queue and also the the size of the tables (in each stage) being only 64 entries while in forward walk it was 128 entries. As mentioned earlier, ther is no need for extra ports to support repair unlike forward walk.

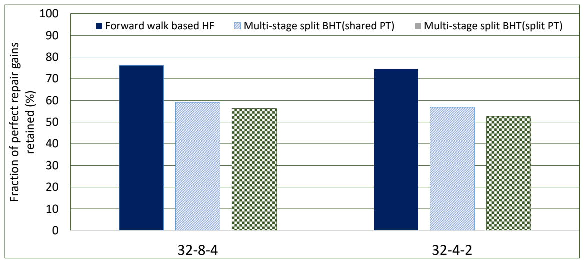


Figure 12: Performance impact of Multi-stage prediction with split BHT with shared PT and split PT.

6.5 Limited-PC repair

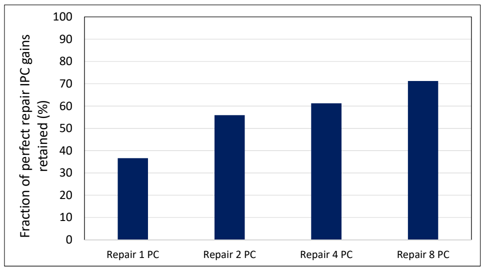


Figure 13: Performance impact of repairing only a limited set of PCs.

Here we study the performance impact of repairing only a limited set of PCs. We capture the performance seen as we scale the number of PCs in Figure 13. Even the 2 PC repair does better than backward walk with limited ports. While surprising, this is the result of being able to identify the right set of PCs to repair. As we saw in section 6.2, no repair leads to a potential loss in performance. Also, in backward walk, when we wait for repair to complete, we lose significant opportunity to give prediction for PCs that are outside the scope of the ongoing repair.

Limited-PC repair provides a simple scalable solution but requires capturing as many bits as determined by the value of "M" (48 bits for 2 PC repair). Alternately, we extended the snapshot queue (SQ) design to checkpoint only these limited PCs. The instruction only carries the entry id in the SQ. This also brings down the storage requirement of the SQ considerably. Checkpointing and repairing 8 PCs with a 32 entry SQ, we retain 57% of the perfect repair gains. This only requires 0.33KB for the SQ which is comparable to the storage required by OBQ.

6.6 Summary of the different techniques

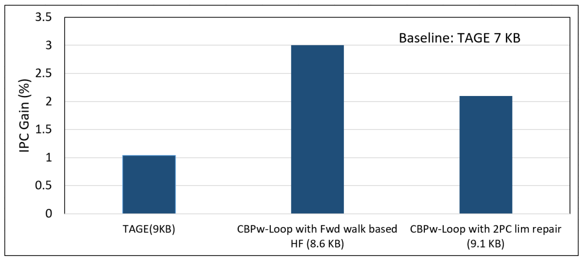
Table 3 summarizes the prior work and the different techniques we looked at till now. Overall, the forward-walk based HF repair which has significantly better performance gains than prior techniques and adds only a moderate hardware cost (0.77 KB in total - 0.03KB for repair bits in BHT, 0.3 KB for OBQ, 0.44KB for adding 16 bits to ROB entry (224 entries in total): 5 bit OBQ entry id and 11 bit counter tracking the current state of an instruction). The multi-stage design with split BHT avoids the additional port requirements for repair and the limited-PC repair schemes allow repair to complete in deterministic time. Each of these schemes have their own advantages and help make the local predictor more effective.

6.7 Sensitivity studies

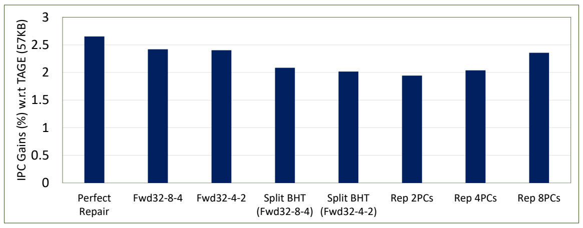
We breakdown this data in this section into two parts,

1. **Iso-storage comparisons:** We compare CBPw-Loop with realistic repair with a TAGE predictor scaled to equal storage. Figure 14A captures the IPC gains where we see that TAGE(9KB) only gains by about 1% which is much lower than combining TAGE with CBPw-Loop integrated with a repair technique like forward walk. In fact, combining TAGE with CBPw-Loop and forward walk gives 3x more performance gain than enhancing TAGE with the same storage.

2. **Larger baseline predictor:** We integrated CBPw-Loop with the much larger TAGE configuration from CBPw-64KB [31] predictor. TAGE is about 57KB in this configuration. As Figure 14B shows, CBPw-Loop continues to be effective improving IPC by 2.7% (with perfect repair). Each of the repair techniques continue to give good IPC gains, for the same additional hardware as shown in Table 3, which shows the efficiency of these techniques.



(A) Comparing performance of TAGE(9KB) with iso-storage configurations of TAGE-CBPw-Loop with Repair



(B) IPC Gain seen when integrating CBPw-Loop with TAGE(57KB), part of CBPw(64KB category)

Figure 14: Sensitivity studies with CBPw-Loop

7 CONCLUSION

Local predictors bring in significant benefits in modern OOO cores. We showed that to reap these benefits, local predictors require careful state management. We proposed different repair techniques including the forward walk, multi-stage prediction with split BHT and limited-PC repair which perform significantly better than prior techniques not only in terms of performance benefits but also required only limited hardware. Our techniques retain 79% of the perfect repair gains while prior techniques only retained 52%, clearly establishing the effectiveness of our proposals. The repair techniques are extensible to other local predictors and will therefore help in wider adoption of local predictors in OOO cores.

8 ACKNOWLEDGMENTS

The authors would like to the thank our fellow colleagues at PARL for their help in improving the quality of the submission. The authors would also like to thank the anonymous reviewers for their thoughtful feedback which helped improve the quality of our paper.

REFERENCES

[1] 2015. Anandtech. https://www.anandtech.com/show/9582/intel-skylake-mobile-desktop-launch-architecture-analysis/.
[2] 2015. Intel Skylake Processor Architecture Overview. https://pcper.com/2015/08/intel-skylake-processor-architecture-overview-scaling-from-tablets-to-servers/.

Configuration	MPKI Redn	IPC gain	Percentage of perfect repair gains retained	Storage (KB) (TAGE + local Predictor + local Repair)	TAGE/BHT Ports for prediction (R\W) 4-wide core	Ports for repair (R\W)	Takeaways
Baseline TAGE	0%	0%	0%	7.1	0	0	
No Repair	0%	0%	0%	7.9	4/4	0	Makes local predictors ineffective
Snapshot	9.1%	1.14%	30%	18.2	4/4	4/4	Storage concerns and more time to repair
Update BHT at Retire	9.6%	1.56%	41%	7.9	4/4	0	Gains will lower as pipeline depth increases in future cores
Backward-walk	16.5%	1.98%	52%	8.3	4/4	4/4	More cycles to repair due to unnecessary BHT updates
2PC lim repair	21%	2.13%	56%	9.1	4/4	0/2	Only 2 PCs repaired. 48 bits carried with instruction for BHT repair
Split BHT Repair	21.5%	2.17%	57%	8.3	4/4	4/0	BHT split into BHT-TAGE and BHT-Defer. Uses Fwd walk for repair. No additional write ports to BHT-Defer needed
4 PC lim repair	22%	2.32%	61%	10.5	4/4	0/4	Only 4 PCs repaired. 96 bits carried with instruction for repair
Forward-walk	26%	2.92%	77%	8.6	4/4	4/2	Relevant PCs updated first and only one update per PC per BHT repair
Forward-walk (with coalescing)	27%	3.0%	79%	8.6	4/4	4/2	Extends forward walk by limiting OBQ entry pressure. 11 bits carried with instruction
Perfect Repair	31%	3.8%	100%	NA	4/4	NA	Best case for this local predictor

Table 3: Summary of the different techniques. Repair Ports are split as read ports (for the checkpointing structure - OBQ/SQ) and write ports for the BHT. Schemes are ordered in increasing order of IPC gains.

[3] 2015. MobileXPRT. http://https://principledtechnologies.com/benchmarkxprt/mobilexprt/.

[4] 2016. Apache Cassandra. http://cassandra.apache.org/.

[5] 2017. TabletMark. https://bapco.com/products/tabletmark/.

[6] 2018. 6th Generation Intel Processor Family. https://www.intel.com/content/www/us/en/processors/core/desktop-6th-gen-core-family-spec-update.html.

[7] 2018. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. http://www.netlib.org/benchmark/hpl/.

[8] 2018. SPEC CPU 2006. https://www.spec.org/cpu2006/.

[9] 2018. SPECMPI2007. http://spec.org/mpi2007/.

[10] 2018. SYSmark 2014. https://bapco.com/products/sysmark-2014/.

[11] 2019. Apache Spark. http://spark.apache.org/.

[12] 2019. EEMBC. https://www.eembc.org/.

[13] 2019. SPEC CPU2017. https://www.spec.org/cpu2017/.

[14] 2019. SPECjbb2015. http://spec.org/jbb2015/.

[15] Mike Clark. 2016. A new× 86 core architecture for the next generation of computing. In *Hot Chips 28 Symposium (HCS), 2016 IEEE*. IEEE, 1–19.

[16] Hongliang Gao and Huiyang Zhou. 2005. Adaptive information processing: An effective way to improve perceptron branch predictors. *Journal of Instruction-Level Parallelism* 7 (2005), 1–10.

[17] Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen. 2013. BigBench: towards an industry standard benchmark for big data analytics. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*.

[18] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. 2005. Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism* 7, 4 (2005), 1–28.

[19] W. W. Hwu and Y. N. Patt. 1987. Checkpoint Repair for Out-of-order Execution Machines. In *Proceedings of the 14th Annual International Symposium on Computer Architecture (ISCA '87)*.

[20] Yasuo Ishii. 2007. Fused two-level branch prediction with ahead calculation. *Journal of Instruction-Level Parallelism* 9 (2007), 1–19.

[21] Yasuo Ishii, Keisuke Kuroyanagi, Takeo Sawada, Mary Inaba, and Kei Hiraki. 2010. Revisiting Local History to Improve the Fused Two-Level Branch Predictor. *3rd Championship Branch Prediction* (2010).

[22] D Jiménez. 2016. Multiperspective perceptron predictor. *Championship Branch Prediction (CBP-5)* (2016).

[23] Daniel A Jiménez, Stephen W Keckler, and Calvin Lin. 2000. The impact of delay on the design of branch predictors. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*.

[24] Daniel A Jiménez and Calvin Lin. 2001. Dynamic branch prediction with perceptrons. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*. IEEE, 197–206.

[25] Primate Labs. 2019. Geekbench. https://geekbench.com.

[26] Scott McFarling. 1993. Combining Branch Predictors. *Techincal Report DEC* (1993).

[27] D. Richins, T. Ahmed, R. Clapp, and V. Janapa Reddi. 2018. Amdahl's Law in Big Data Analytics: Alive and Kicking in TPCx-BB (BigBench). *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2018).

[28] Satish Kumar Sadasivam, Brian W Thompto, Ron Kalla, and William J Starke. 2017. IBM Power9 processor architecture. *IEEE Micro* 37, 2 (2017), 40–51.

[29] A. Seznec. 2005. Analysis of the O-GEometric history length branch predictor. In *32nd International Symposium on Computer Architecture (ISCA'05)*.

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

[31] André Seznec. 2016. Tage-sc-l branch predictors again. *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)* (2016).

[32] André Seznec and Pierre Michaud. 2006. A Case for (partially) Tagged Geometric history length branch prediction. In *Journal of Instruction Level Parallelism*. http://jilp.org/vol8

[33] André Seznec, Joshua San Miguel, and Jorge Albericio. 2015. The Inner Most Loop Iteration Counter: A New Dimension in Branch History. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 347–357. https://doi.org/10.1145/2830772.2830831

[34] Kevin Skadron, Margaret Martonosi, and D Clark. 2000. Speculative updates of local and global branch history: A quantitative analysis. *Journal of Instruction-Level Parallelism* 2 (2000).

[35] James E. Smith. 1981. A Study of Branch Prediction Strategies. In *Proceedings of the 8th Annual Symposium on Computer Architecture (ISCA '81)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 135–148. http://dl.acm.org/citation.cfm?id=800052.801871

[36] Eric Sprangle and Doug Carmean. 2002. Increasing processor performance by implementing deeper pipelines. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*.

[37] Tse-Yu Yeh and Yale N. Patt. 1991. Two-level Adaptive Training Branch Prediction. In *Proceedings of the 24th Annual International Symposium on Microarchitecture (MICRO 24)*.