Opportunistic Early Pipeline Re-steering for Data-dependent Branches

Saurabh Gupta
Processor Architecture Research Lab, Intel Labs, Intel Corp.
saurabh2.gupta@intel.com

Niranjan Soundararajan
Processor Architecture Research Lab, Intel Labs, Intel Corp.
niranjan.k.soundararajan@intel.com

Ragavendra Natarajan
Intel Corporation
ragavendra.natarajan@intel.com

Sreenivas Subramoney
Processor Architecture Research Lab, Intel Labs, Intel Corp.
sreenivas.subramoney@intel.com

ABSTRACT

As Out-of-Order (OOO) cores scale to very large instruction windows to extract higher instruction level parallelism (ILP), the cost of mis-speculation increases tremendously. Branch predictors sitting at the head of the pipeline are a significant contributor to the speculatively executed code. It is critical to limit the execution time down the wrong path for the sake of performance and energy efficiency. Architects continue to increase the branch predictor sizes and improve the prediction algorithms to mitigate the increasing cost of the mis-speculations. Unfortunately, there still exists branches whose outcomes are predominantly data dependent, and that significantly contribute to the overall mispredictions. This work is the first to characterize data dependent branches at a scale spanning 100+ workloads drawn from several application categories and establish a clear motivation to address them. We find that branches which have only one load instruction feeding them and only simple operations to compute the branch direction from the load value are responsible for a significant fraction of the overall mispredictions. We call such branches Direct Data Dependent (3D) branches. We develop a family of synergistic techniques to avoid mispredictions from such 3D-branches. We describe 3D-Branch Overrider, our novel branch overriding technique that operates in the front-end of the pipeline to minimize the branch misprediction penalties. On a modern Icelake-like OOO core equipped with the state-of-the-art branch predictor, we find that our technique provides a 12.7% reduction in branch mispredictions resulting in 3.1% Instructions Per Cycle (IPC) gain while needing just 5.7KB in additional storage. As future cores become wider and deeper, we show that the gains from 3D-Branch Overrider nicely increases, even if the size of the underlying branch predictor is aggressively scaled.

CCS CONCEPTS

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

PACT '20, October 3–7, 2020, Virtual Event, GA, USA

ACM ISBN 978-1-4503-8075-1/20/10…$15.00
https://doi.org/10.1145/3410463.3414628

KEYWORDS

Branch Prediction, Data-dependent branches, Instructions per Cycle.

ACM Reference Format:
Saurabh Gupta, Niranjan Soundararajan, Ragavendra Natarajan, and Sreenivas Subramoney. 2020. Opportunistic Early Pipeline Re-steering for Data-dependent Branches. In *Proceedings of the 2020 International Conference on Parallel Architectures and Compilation Techniques (PACT '20), October 3–7, 2020, Virtual Event, GA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3410463.3414628

1 INTRODUCTION

Given the continued importance of single thread performance in general-purpose cores [30], architects are looking to extract more instruction level parallelism (ILP) by building deeper and wider cores. This in turn increases the number of in-flight instructions within the pipeline. The conditional branch predictor unit (BPU) sits early in the pipeline to predict the program control flow path. As the pipeline depth increases, the delay between branch prediction and branch execution increases, leading to increasing performance penalty per misprediction across processor generations. Overall, branch prediction accuracy significantly impacts the performance and energy efficiency of general-purpose OOO cores [37].

Figure 1 shows a typical OOO pipeline illustrating different stages and highlighting different events. As a branch instruction enters the pipeline, the branch predictor predicts the direction it would take very early in the pipeline while the branch gets executed and its true direction is resolved much later in the pipeline. Most of the branch mispredictions in modern workloads are caused by conditional branches (92% in the workloads listed in Section 6). State-of-the-art conditional branch predictors operate by tracking the branch direction and target address history of all prior branches to capture the path taken to reach the current point in the execution. This information, referred to as global branch history, is then used to predict the incoming branch's direction. Branch predictor proposals in literature track specific branches in the global history [27] or only use the specific branch's prior history (referred to as local history) to provide a prediction. TAgged GEometric history length predictor (TAGE) [35] is one of the most successful branch predictor proposals. The most recent CBP winner (CBP-5) [15], integrated TAGE with a statistical corrector (SC) component and a loop predictor (referred to as TAGE-SC-L) and involved predictors based on both global history and branch specific local history. This combination of global and local history is typically sufficient to

predict branch directions with a high degree of accuracy as shown by CBP results [14, 15].

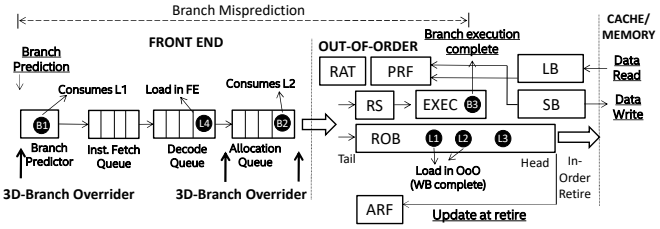


Figure 1: Typical OOO pipeline highlighting different the parts of the pipeline and the individual structures. Front-end of pipeline includes the branch predictor, fetch, decode and allocation queues. OOO includes Register Alias Table (RAT), Physical Register File (PRF), Reservation Station (RS), Execution units (EXEC), and Re-Order Buffer(ROB), Load Buffer (LB) and Store Buffer (SB). Architecture Register File (ARF) is updated on instruction retirement. *We point out where different events of interest happen.* {L1, L2, L3, L4} are load instructions and {B1, B2, B3} are branches.

But there are conditional branch instructions across several applications that are poorly correlated to the prior branch history and hence cause frequent mispredictions. These conditional branches, broadly, fall into the category of data-dependent branches where the data values themselves exhibit high entropy. Scaling the predictor sizes for these branches provides only a marginal gain. As newer branch predictor designs like Perceptron and TAGE integrated with statistical corrector emerge [24, 33], *mispredictions from data-dependent branches remain the single most important category of mispredictions to tackle.* Prior works like EXACT [9], SLB [18] have proposed different techniques to lower the mispredictions from such data-dependent branches. These include mechanisms to track the store addresses and their values using large hardware structures or add ISA extensions to identify the data-dependent branches that are then optimized. As such, both designs suffer from high complexity that has impeded their adoption into commercial state-of-the-art processors.

Data dependent branches have been relatively less explored and optimized in literature, especially in comparison to branches that exhibit correlation to global or local branch history. Specifically, our work analyzes data dependent branches from the programmatic context of the load instructions that feed these branches. *We study 100+ workloads spanning several application categories including cloud, enterprise and personal computing to show that data dependent branches are quite common* and demonstrate effective solutions to mitigate misprediction penalties arising from them will benefit workloads across multiple domains.

We first characterize the dependence chains of mispredicting branches based on the number and type of instructions leading to the load instructions that feed data to these branches (referred to as "feeder" loads). Interestingly, in 52% of the cases, the branches are fed by only one load PC. We also observed that in most of these cases there is very little intermediate computation between the load value and the downstream data dependent branch instruction. We refer to such branches that have only one feeder load instruction and only simple operations to compute the branch direction as **Direct Data Dependent (3D) branches** and target these branches in our study. Figure 2 shows different branch instances and their dependence chains to illustrate the cases that qualify as 3D-branches.

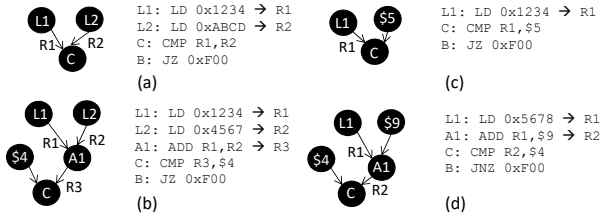


Figure 2: Examples highlighting the conditions that qualify a branch as Direct Data Dependent (3D). Cases (a) and (b) are not 3D-branches while (c) and (d) are 3D-branches.

Given that 3D-branches are a significant contributor to the total mispredictions, this paper proposes **3D-Branch Overrider**, a technique that overrides the default branch prediction of data-dependent branches when the feeder load's value is available, and re-steers the processor's pipeline as early as possible. As shown in Figure 1, 3D-Branch Overrider operates in the front-end of the pipeline to opportunistically limit the number of wrong-path instructions entering the pipeline after the branch. 3D-Branch Overrider is flexible in that the re-steering event can be triggered at the branch predictor or near the allocation queue depending on when the load value becomes available. In Figure 1, branch B1 which is dependent on load L1 can use the load's value for prediction override when entering the pipeline while branch B2 which is dependent on load L2 can use its value only near the allocation queue.

While 3D-branches present a significant opportunity, it is often the case that the feeder loads occur close to the branches in the program and therefore might not have their values available in time to compute the branch direction and re-steer the pipeline sufficiently early. Taking timely load value availability into consideration, our overall opportunity shrinks to 7% on average across the 100+ workloads.

Based on our initial estimate that 3D-branches constitute 52% of the mispredictions, there is still a significant opportunity that remains untapped. Towards this, we propose three enhancements to the 3D-Branch Overrider, i) Accelerate the execution of the feeder load instruction, ii) Feed the value from a prior store that wrote to same address as the feeder load using a memory dependence predictor [40], iii) Predict the load address and prefetch the load value. While each of them improve the gains from 3D-Branch Overrider, predicting the load address and prefetching the load value in a timely manner provides significant additional gains. In this technique, we uniquely re-purpose a load address predictor to provide an address to prefetch the load value in time to compute the branch direction and override the baseline prediction.

Our contributions include,

1) Across **100+** workloads drawn from several real-world application categories and important benchmarks, we characterize and study their branch mispredictions. We identify that a considerable portion of branch mispredictions (52%) are fed by only one

load PC and only have simple operations to compute the branch direction. We call these branches **Direct Data Dependent (3D)** branches. We show later that the high prevalence of 3D-branches in branch mispedictions is a key insight as it would enable architects to tackle a major component of the total mispredictions with limited hardware costs.

2) We propose 3D-Branch Overrider, a simple and efficient branch overriding technique for mispredictions from 3D-branches. 3D-Branch Overrider computes the branch value based on the feeding loads to override the baseline branch prediction and re-steers the pipeline early to limit the impact of branch mispredictions. Extending our design with multiple optimizations, including a load address predictor to prefetch the load values in a timely manner, **3D-Branch Overrider provides a 12.7% Mispredictions Per Kilo Instructions (MPKI) reduction resulting in 3.1% gain in IPC**.

3) Looking into the future, with aggressive OOO processors, where the impact of branch misprediction penalties will be even higher, we show that the gains from 3D-Branch Overrider only increase. Scaling our processor cores to approximately be 1.5x and 2x our baseline core, 3D-Branch Overrider provides an even higher 3.7% and 4.3% gains in IPC respectively. **We show that merely scaling the sizes of conventional branch predictors is insufficient to address mispredictions from data dependent branches**. Integrating the CBP-unlimited predictor configuration from CBP-5 [32], 3D-Branch Overrider still remains largely effective and achieves 9.2% MPKI reduction and 1.3% in IPC gains establishing the unique effectiveness of our proposed branch direction overriding technique.

2 BACKGROUND

This section provides a brief overview of the state-of-the-art branch predictor design to expand on its limitations in addressing data dependent branches and motivate our work. We then look at how some of the existing techniques handle data-dependent branches and their limitations.

2.1 64KB TAGE-SC-L Branch Predictor

Conditional branch predictors have evolved over time from simple tag-less bimodal predictors to long branch history based tagged prediction by partial matching (PPM) predictors [26, 35]. The TAGE branch predictor, proposed in [35], has proven itself to be very efficient and is part of Championship Branch Prediction(CBP) winning predictor [33] in both 2014(CBP-4) and 2016(CBP-5). The TAGE predictor is composed of a tag-less bimodal component and a set of partially tagged global-history based tables. In CBP-5, TAGE is augmented with a statistical corrector [33] that uses local history, loop predictor and different sets of global history-based predictors to provide a more accurate branch prediction. This predictor combination, referred as TAGE-SC-L, won CBP-5 and is the current state-of-the-art branch predictor. In this work, we incorporate this large 64KB predictor as the baseline branch predictor. While CBP-5 also had an 8KB predictor, we use the larger predictor to help emphasize that data-dependent branches have less sensitivity to the predictor size and prior branch history size.

We now look into some of the prior works that have addressed data dependent branches and their limitations.

2.2 Existing Techniques to Handle Data Dependent Branches

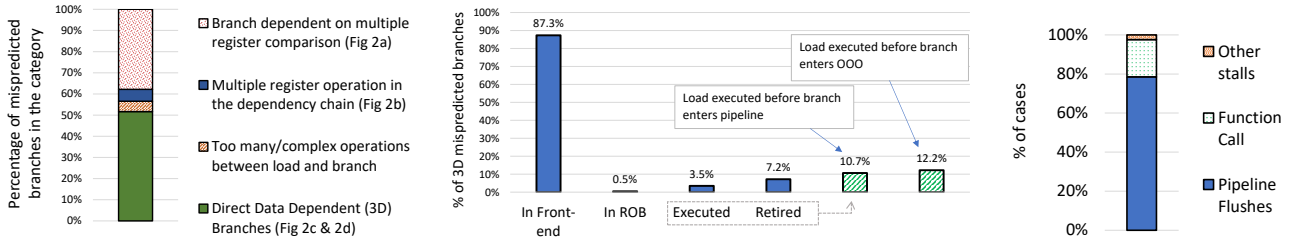
EXACT [9] is a *branch prediction technique* proposed to lower mispredictions from data dependent branches. By distinguishing branch instances based on their feeder load's addresses it can better classify different branch instances in turn leading to better predictions. Since the load addresses are not always available at branch prediction time, EXACT uses a prior instance of the branch, that had retired much earlier, as a proxy identifier for the current branch to access another table for the prediction. Using this proxy prior instance can be inefficient since contexts can vary across instances, changing the load value which in turn can change the branch direction. EXACT also looks at forwarding value from a prior store to the consumer load after establishing the store-load-branch dependency. Though, the set of distinct load-branch and store-branch pairs that need to get tracked quickly increase the storage requirement over 10KB. As mentioned in [9], EXACT predictor does not win over an similarly sized TAGE predictor.

Learning from EXACT's limitations, our proposal targets very specific cases of data dependent branches which depend on only one load instruction and have only limited computations to determine the branch direction. We also propose overriding near the allocation queue for increased coverage. Further, by computing the actual branch direction from the load value allows us to have higher accuracy.

Store-Load-Branch (SLB) predictor [18] tracks the stores that feed the value to the data-dependent branches, and uses profiling and compilers to mark data-dependent branches with ISA extensions. Since SLB forwards values from the store to a dependent branch, there is load address prediction involved to identify the load that will consume the data. In addition to ISA changes, SLB requires load/store hint instructions and additional compare instructions to be added to the program. SLB requires about 8KB to provide its predictions for the conditional branches. 3D-Branch Overrider, on the other hand, is a completely hardware-driven solution with a lower storage requirement and can transparently be integrated in any modern processor's pipeline.

3 CHARACTERIZING DIRECT DATA DEPENDENT (3D) BRANCHES

All branches have one or more load instructions in their dependence chain that provide the value for computing the branch direction. For most branches, the data value governing the branch outcome has a low entropy or is well-correlated with the prior branch direction and path history. On the other hand, data dependent branches showing a high branch misprediction rate have a higher entropy in their data value and using just the direction and path histories is insufficient to provide highly accurate predictions for these branches. In this work, we target a specific class of data dependent branches. These branches have only one feeding load in their dependency chain and also have only a limited set of operations leading to the branch instruction. So, once the load value is available, the branch direction can be computed independent of other instructions using minimal arithmetic logic. We call these branches as the Direct Data Dependent (3D) branches. As we saw in Figure 2, case (c) and (d)



(a) Characterizing the branch mispredictions based on their dependency chain

(b) Solid blue bars show the status of feeder load when the branch is entering the pipeline. Striped green bars show the fraction of loads that completed execution before branch enters the pipeline versus OOO.

(c) Reasons for increase in pipeline slack(cycles) between feeder load and 3D-branch.

Figure 3: Characterizing the branch mispredictions, status of feeder load in the pipeline, and reasons for slack between feeder load and 3D-branch

represent 3D-branches. We call load L1 in these examples as "the feeder load".

Below, we show two examples of 3D-branches from the SPEC2006 benchmark *gobmk*. In the Example 1, instruction A1, which is a load, reads the data from memory and later instructions A3/A4 test and jump based on this data. In Example 2, data is loaded by instruction B1, and B2/B3 and B4/B5 compare and jump based on B1's data. Example 1 and 2 also highlight cases when the load and branch are closer in program layout but calls to intermediate functions, or runtime events like branch mispredictions can increase the time between when the load and the branch enter the pipeline.

```
        Example 1:                      Example 2:
A1: movl 0x1c(%esp),%esi        B1: movzbl 0x79a(%ebx),%edx
…                               …
A2: Call F (function)           B2: cmpl $0x3,%edx
…                               B3: je B6 – wrongly predicted
…                                                   taken
A3: testl %esi,%esi             B4: testl %edx,%edx
A4: jne A6                      B5: jne B7
```

In the following sections, we establish the prevalence of 3D-branch mispredictions, understand the opportunity better by tracking the load value availability, characterize the number and type of operations seen between the feeder load and the 3D-branch.

3.1 Mispredictions from 3D-branches

As a first step to emphasize the need to tackle 3D-branch mispredictions, we characterize the data flow feeding into all branch instructions that the baseline predictor mispredicts. For the 100+ workloads that we study in this work (refer section 6), Figure 3(a) depicts the different categories in which the mispredictions fall. While a considerable fraction of the mispredictions involve multiple loads or multiple register operations or complex arithmetic operations, *majority of the mispredictions in these workloads are from 3D-branches*. Focusing on mispredictions from 3D-branches not only simplifies the hardware requirements but also this covers a very significant fraction of the mispredictions and therefore motivates further analysis.

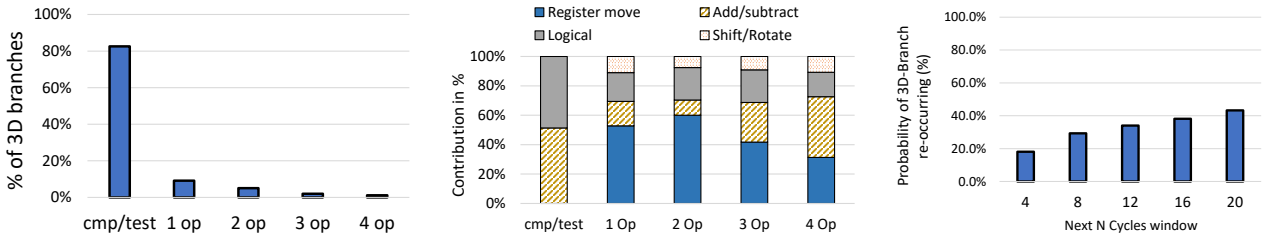
3.2 Data Availability from the Feeder Loads

While the 3D-branches constitute a significant fraction of the overall mispredictions, the opportunity to do an effective override is dependent on the load value being available in a timely manner. The blue solid bars in Figure 3(b) categorizes the status of the feeding loads when the corresponding 3D-branches enter the pipeline. In most cases, the branch and the load are close to each other in the pipeline and the load is still in the allocation queue by the time branch enters the branch prediction stage. In about 10.7% of the cases, we see that the feeder loads have either retired or has completed and the value is available by the time the branch enters the pipeline and therefore the branch direction can be computed accurately.

While loads and their dependent branches are often near each other in the code layout, pipeline events like pipeline flushes due to intervening branch mispredictions or mis-speculations from incorrect store-load forwarding and other memory subsystem optimizations, stalls from structures being full or even program constructs like intermediate function calls can increase the time between when the load goes through different stages and when the branch does. In Figure 3(c), we characterize these events and see that *most often pipeline flushes and function calls provide an opportunity to compute the branch direction before the branch enters the pipeline*. We observe that an additional opportunity to override a branch prediction and have a front-end re-steer early is available in 1.5% of the mispredicting branches. In these cases, the load value becomes available before the branch has entered the OOO part of the pipeline. Across both these scenarios, we find that 12.2% of the 3D-branch mispredictions have their feeder load value available for an early re-steering before the branch enters the OOO pipeline. Referring back to Figure 3(a), where we saw that 3D-branch mispredictions constitute 52% of the total mispredictions, *our overall opportunity becomes 6.7% of the total mispredictions.*

3.3 Dependence Chains of 3D-branches

Computing the 3D-branch direction involves some operations on the load value. Given that replicating the full execution unit in the front-end of the pipeline comes at a very high hardware cost, we categorize the number of operations seen between the feeding load to the mispredicting 3D-branches in Figure 4(a) and the type of those operations in Figure 4(b). We see that in most of the cases, there is only one intermediate computation (compare or a test) between the load and the 3D-branch. Based on the number and type of the computations, once the load instruction completes execution, branch direction can be computed within few cycles.



(a) Distribution of number of operations between the load and 3D-branch instruction (cmp refers to compare).

(b) Distribution of type of operations seen between the load and 3D-branch instruction.

(c) Probability of a 3D-branch re-occurring with the next N cycles of another 3D-branch.

Figure 4: The characteristics of instructions seen between feeder load and 3D-branch, and burstiness of 3D-branches.

As shown in Figure 4(b), the operations seen include compare operations, register move operations and simple ALU operations like increment, decrement, additions, subtractions and shift operations involving the load and an immediate values.

While supporting the intermediate computation necessitates the need for exclusive hardware, we believe that to reduce the mispredictions from data dependent branches the area is better spent incorporating the computing hardware than continuing to increase the branch predictor sizes. In section 7, we show that even with an extremely large baseline predictor, 3D-Branch Overrider continues to remain effective without losing its impact emphasizing the need for specialized techniques to tackle data dependent branches.

3.4 Burstiness of 3D-branches

If the 3D-branches are very bursty in occurrence, the computation hardware has to account for this behavior. In Figure 4(c), we study their burstiness by plotting the probability of a 3D-branch entering the pipeline within N cycles of another 3B-branch having entered the pipeline. We see that there is only about a 20% chance of a 3D-branch occurring within the next 4 cycles and it does not increase significantly if we increase the monitoring window to 20 cycles.

Based on insights from our detailed characterization, we next describe our proposed 3D-Branch Overrider that opportunistically re-steers pipeline early to reduce the misprediction penalty.

4 3D-BRANCH OVERRIDER

It is to be noted that the 3D-Branch Overrider is designed as an override mechanism to the branch predictor's prediction and is not an enhancement to the branch predictor itself. It sits in the front-end of the OOO pipeline and monitors the branch instructions to identify those that often mispredict and satisfy the 3D-branch qualification criteria as described in Section 3. For those branches, depending on the load data's availability, the overrider computes the branch direction and triggers an early re-steering of the pipeline if baseline branch prediction of the branch direction was wrong. In this section, we describe the different components of the 3D-Branch Overrider, and how the load value is obtained and used to enable the branch prediction override.

4.1 3D-Branch Tracker Table (BT)

To limit the hardware requirements, we target only the frequently mispredicting branches from the baseline TAGE-SC-L predictor that satisfy the 3D criteria as discussed in section 4.2. These branches are then tracked in the 3D-Branch tracker table (referred as BT from here on). Each entry in the table, as in Figure 5(a), records the branch PC, the load PC, an index into a load tracker (LT) to identify the latest instance of the load PC, the computed branch direction, operations associated with getting the branch direction computed and other meta-data. We discuss the need for the LT index in section 4.3. Based on our sensitivity studies, tracking about 128 3D-branches, in a 8 way set-associative table, with age and usefulness-based replacement scheme is sufficient. The table is indexed by lower bits of the branch PC.

Once a branch instruction in the tracker enters the pipeline, the BT is searched for the branch PC and if there is a hit, the computed branch direction, if ready, is made available. If this computed branch direction is different from the baseline branch prediction, we trigger a pipeline re-steer immediately to start fetching from the correct path. Sometimes, due to delays in the load execution or additional cycles required to perform the intermediate computation, the opportunity to override and re-steer may be missed at the branch prediction stage. In that case, the override is attempted again at the allocation queue.

To enable an override at the allocation queue, once the feeder load completes and the branch direction is computed, LT index in the BT is matched against the LT index of the returning load data value. If they don't match, then we have missed the opportunity to override at branch prediction time, and we should attempt a re-steer at allocation stage and the LT index along with the computed branch direction is sent to the 3D Alloc re-steer table. This table is a direct-mapped structure indexed by the LT index and stores a 1-bit computed branch direction along with a valid bit. When an LT index is assigned to an incoming load instruction, the corresponding valid bit in the 3D Alloc re-steer table is reset. Only after the computed branch direction reaches the 3D Alloc re-steer table is the valid bit set. Using LT index ensures that correct overriding prediction is used for the 3D-branch. As the 3D-branches enter or leave the allocation queue, they use their LT index, carried by the 3D-branch instruction, to index and read the computed branch direction. The pipeline is re-steered if the computed and predicted directions mismatch.

As described in section 4.3 in more detail, each instruction carries the LT index in the front-end of the pipeline. Once it moves to the back-end, the load buffer (LB) is extended to store the LT index. As soon as the load completes execution, irrespective of whether the corresponding 3D-branch has entered the pipeline or

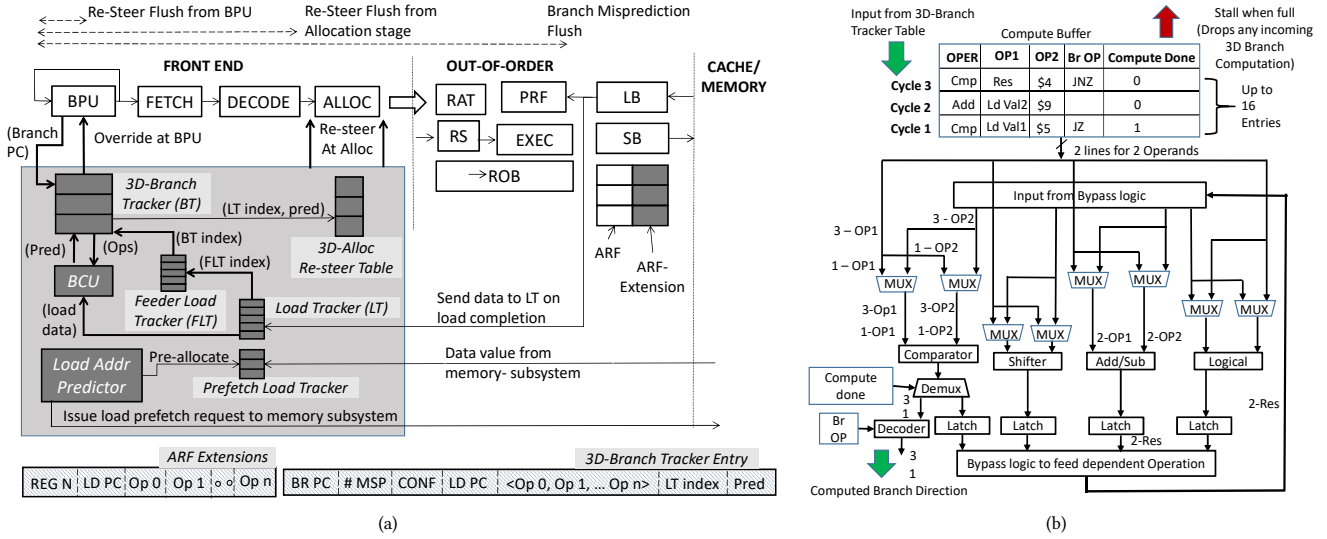


Figure 5: (a) Modern OOO core with the 3D-Branch Overrider components highlighted, and (b) Microarchitecture of the Branch Computation Unit (BCU).

not, we trigger the Branch Computation Unit (BCU) to compute the branch direction. This proactive computation combined with the opportunity we discussed in Figure 3(c), gives us a window where we seek to opportunistically override either from the branch prediction stage or at the allocation queue.

The learnt dependence chain information is kept with the entry in the form of a <Load PC, Op0, Op1...,Opn, Br Op> tuple. Each Op records the compute operation along with any immediate operand and the Br Op captures the final branch operation (JZ, JNZ) that is required to be done. This compact representation is possible because we target 3D-branches which, as discussed earlier, are dependent on only one load and cannot have multiple register operations in their dependence chain. In this manner, the data-flow is implicit in the order of the operations. The first operation (Op0) is dependent on the load value, Op1 is dependent on Op0 and so on. One optimization we do, when recording the operations in the BT, is to remove the register moves and directly pass the value as operand to the next instruction.

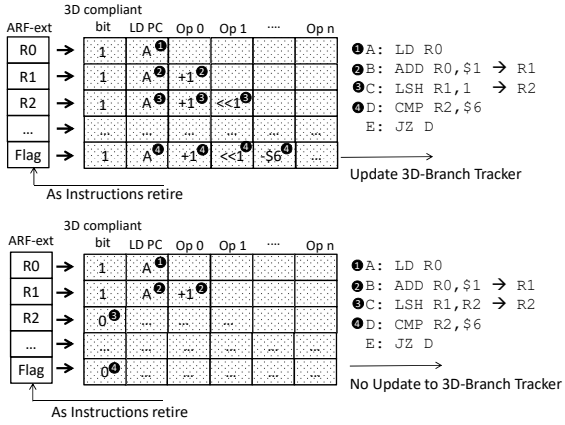


Figure 6: Tracking the loads and the associated operations that directly feed the 3D-branch.

4.2 ARF Extensions to Support 3D-branch Detection

A very important component of 3D-Branch Overrider is the detection of the branches that have only one feeding load and only simple computations in the data path leading to the branch. To enable this, we employ a scheme similar to EXACT predictor ID generation [9] where the ARF is extended to track the load addresses. Instead of tracking the load address, we track the load PCs and any operations that alter the load value before reaching the branch instruction. This information is propagated across registers till a branch reads from it. Figure 6 shows an example of the load PC and the operations that are propagated through the ARF.

There is one bit added into the ARF extension entries to identify if the register is 3D compliant. This bit is reset for those registers which are the destination of compute or comparison operation involving multiple registers. This bit is also reset if the intermediate computation is complex - multiplication, division etc. Also, we reset this bit when the number of prior operations reaches a maximum threshold before reaching the branch instruction (Op-n in Figure 5(a)). If the same register is the destination of two different load PCs, the newer load PC overwrites the ARF extension entry. Only when the 3D complaint bit is set, are the load PC and operations propagated to the next register in the data-path. If the 3D compliant of a register is reset, its dependent instructions will also not be 3D compliant. The 3D complaint bit is again set for a register when it is the destination of a load instruction. When a branch instruction retires, we can use the tracked 3D compliant bit to identify a 3D-branch. For x64 ISA-compliant cores [6], since the conditional branch instructions only read the flag register, we track the 3D compliant bit in the flag register as well. If the compare or

test instruction before the branch had its 3D compliant bit set, it is passed to the flag register. Now, if the 3D compliant bit is set in the flag register, at branch retire time, we add the branch PC, load PC, and the set of operations to the BT.

4.3 Feeder Load Tracker and the Load Tracker

When a load instruction, tracked in the BT, enters the pipeline we need to identify if it is a PC of interest. To determine this, we have two design options. We can either a) store the load PCs in the BT as content-addressable memory (CAM) and search across all entries, or b) have another table, Feeder Load Tracker (FLT), that is indexed by the load PC and track BT entries which are dependent on a particular load PC. We go with the latter option to limit the scalability and power issues associated with searching across 128 entries. The FLT table is configured similar to BT. Also, based on sensitivity studies, we track only two BT entries per FLT entry which was sufficient to retain most of our gains. When a new 3D-branch is looking to make an allocation in the BT, we check the feeder load's entry in FLT to see if it is already tracking multiple entries in the BT. If it is, we use the confidence bits of these BT entries to decide if the new branch makes an allocation or not.

To track the different feeder load instructions going in the pipeline, we use Load Tracker (LT) table. The LT is a 128-entry set-associative table in which we store an index into the FLT. When a load that hits in FLT table enters the pipeline, an LT entry is assigned to the load instruction and this LT index is carried in the pipeline for usage after the load executes. Once the load PC is assigned an LT entry, it also updates its LT index in the BT which is used by the branch to obtain the load value when it enters the pipeline. Since multiple dynamic instances of the same load instruction can be simultaneously in the pipeline, adding it to BT is important to correctly associate the incoming 3D-branch to the youngest instance of the feeder load. Once a load with LT index completes execution, we use FLT index stored in the LT to identify the BT entries of interest. The operations are then triggered to compute the branch direction based on availability of the Branch Computing Unit (BCU), as described below.

4.4 Branch Computation Unit

The critical component required to compute the branch direction is the Branch Computation Unit (BCU). Figure 5(b) illustrates how the computation of the branch direction is done when feeder loads of two 3D-branches complete. The operations required to compute the branch direction are read from the BT and sent to the BCU. BCU expands the format stored in the BT to a simpler format for internal consumption. BCU incorporates a 16 entry Compute Buffer (CB) to record the incoming operations and their operands. The CB is a simple in-order queue of operations and records the specific operation, their operands, any branch operation to be done at the completion of the current instruction, and a Compute Done flag to indicate the last instruction required to compute the branch direction. In Figure 4(b), we plot the distribution of the number and type of computations seen in the dependence chains of 3D-branches. As seen, in a significant number of the cases, the value is used directly by the compare instruction before the branch. Further, from Figure 4(c)we concluded that 3D-branches are not very bursty and do not occur very close to one another very often. Taking both these factors into account, we size the CB at 16 entries. While all the 16 entries are seldom full, since the load value (on completion) only gets stored in the CB entries, we want to make sure the computation requests are rarely dropped.

To keep the scheduling logic simple, only one operation gets triggered in a cycle. As shown in Figure 5(b), even independent operations are serialized and buffered in the CB and scheduled in-order. This is shown in Figure 5(b), where we see that the three operations required by the two 3D-branches complete in 3 cycles. While this simple design increases the latency to compute a branch direction, we find in practice that most branches only require a simple compare/test and hence there is marginal benefit from a more aggressive BCU design.

4.5 Storage Overhead

We summarize the overall storage required for the different components here. In our design, ARF-based learning happens at retire stage and the learned dependence chains (feeder load PC, intermediate operations and 3D-Branch PC) are sent to the BT. We assume that opcode and other instruction metadata is available with the ROB entry similar to [9, 28].

1) **3D-Branch Tracker Table (BT):** Each entry has 1 valid bit, 8 bits for branch PC tag, 4 bits for tracking the per PC mispredictions, 3 bits of confidence, 7 bits for LT index, and 1 bit for computed prediction from the load data. Additionally, for the dependence chain we require 16 bits for the Load PC and 1 byte for the compute operation, 2 bytes for each immediate operand and 1 byte for the final branch instruction. Therefore, storing four operations along with the load PC and branch instruction requires 15 bytes. This makes the total entry size to be of 18 bytes. Total storage required for the 128 entry BT is 2.25KB.

2) **3D Alloc Re-steer table:** It only needs to maintain the branch direction and valid bits. For a 128 entry table, this requires about 0.03KB.

3) **ARF Extensions:** For tracking and identifying 3D-branches using an extended ARF, we require 14 bytes per entry. Since x64 ISA [2, 38], only has 16 general purpose architectural registers and one flag register that need to be tracked, the total overhead of this extension is 0.23KB.

4) **FLT and LT:** Each FLT entry has 8 bit load PC tag and 14 bits for the two BT entries. For 128 entries, this requires 0.34KB. LT table of 128 entries with 7 bits to track the FLT entry and 4 bits of additional information for valid and replacement bits and requires 0.17KB.

5) **CB in BCU:** Each entry in the CB, within BCU, has 1 byte each for operation and the branch opcode, 8 bytes for OP1 (either the load value or bit to indicate the result of prior computation) and 2 bytes OP2, 1 bit for the compute done flag. For the 16 entries, CB requires about 0.18KB.

5) **Pipeline extensions:** Extending the fetch and allocation queues and the LB and to track the LT entry requires an additional 0.4KB.

Overall, 3D-Branch Overrider requires 3.2 KB.

5 OPTIMIZING THE 3D-BRANCH OVERRIDER

Given that 3D-branches constitute a significant portion of the branch mispredictions and the feeder loads are quite often close to the dependent branch, the load value is not available in time to override the predicted direction. In this section, we look at three optimizations to the baseline 3D-Branch Overrider to enhance our coverage to achieve further reductions in branch mispredictions.

5.1 Extending 3D-Branch Overrider with Memory Dependence Prediction

In many cases when the load is too close to the branch, refer Figure 3(b)), a possible optimization is to find the store prior to the load that wrote to the same address and use the store data value for override. EXACT's active update unit [9] and SLB [18] use this observation, but the hardware requirements were prohibitive for EXACT and SLB requires ISA extensions.

In our studies as well, we find that pairing instructions based on addresses or values rather than PC-based pairing (branch-load pairing or load-store pairing) can be expensive as the number of distinct addresses or values seen is typically significantly higher than PC values. On average, we find that that a load-branch PC pair goes through over 160 data values and over 2800 addresses while a load-store PC pair sees up to 380 data values and 260 different addresses. Therefore, to limit the hardware requirements, we enhance 3D-Branch Overrider with a Memory Dependence Predictor (MDP) [40] to capture the strongly correlated store PC-load PC pairs that have producer-consumer relationship. If the load has not yet completed but a strongly correlated store PC for the load PC has been identified, the data value from producer store instruction can be directly utilized for branch direction computation. To limit the wrong re-steers through this technique, besides updating the store-load pairs in MDP, we conservatively also lower the confidence counters in the 3D-branch table.

5.2 Accelerating the Feeder Load in the Pipeline

A fraction of the loads whose value is not available in time before the branch is allocated in the OOO, complete their execution within a few cycles of this event. To benefit from this observation, we could either delay the branch or prioritize the load to complete its execution early. Delaying the branch stalls all instructions after it and is therefore not ideal. Hence we devised a Load Prioritization scheme to prioritize the execution of the instructions in the backslice of the load (instructions that the feeder load is dependent on). This allows the address computation to be completed early, consequently, allowing the loads value to be available earlier as well. We leverage from prior proposals such as [10, 12] to build a simple and efficient module to capture the dependence chain of instructions which can then be prioritized for execution. We capture this dependence chain in a Load Backslice Table (LBT), which is configured similarly to the FLT. To limit the storage requirements, we track only 8 lower bits per PC and also limit the number of PCs tracked to 8 PCs per feeder load instruction. So once the feeder load enters the pipeline, we read the backslice information from the LBT and prioritize those instructions for execution.

5.3 Extending 3D-Branch Overrider with a Load Address Predictor

One potential solution to obtain the load data value early is to predict the load address and prefetch the value from memory. We observe that for some feeder loads the memory addresses from which they read the value are highly predictable and the data can be prefetched in time if the value is in the L1 data cache. Since this additional load access will consume memory pipeline bandwidth, we have to be very stringent and only use the prefetcher when the 3D-overrider is highly confident in avoiding a misprediction.

In our studies, we morph a highly accurate value predictor called EVES [34] into a load address predictor (LAP). It is indexed using the load PC and uses a last-value component, and a global branch history based component to predict the load address. We disable the stride component to simplify LAP. As observed by [34], this predictor achieves high accuracy predictions with limited storage. While EVES was picked to extend the 3D-Branch Overrider, our proposal will work with any other load address predictor such as [29] as well. The load address predictor generates a load prefetch when a feeder load PC (with an highly predictable address) enters the pipeline. It is tracked in a structure similar to LT, called Prefetch Load Tracker (shown in light grey in Figure 5(a)). Once the prefetched data is available, branch direction can be computed and pipeline re-steer is initiated if required.

In the pipeline, there can be in-flight store instructions that will write to the same address as the feeder load's data access. In these cases, the data obtained by the load prefetch may not have been accurate. We lower the confidence field in the BT on those feeder loads once such wrong re-steers are observed. To limit the size of the load address predictor and also to reduce the overhead on the memory subsystem, we filter the load PCs based on the timeliness of the prefetched load value being available in time to perform the branch override. We increase the confidence associated with this 3D-branch when there is a correct override from the prefetched load value. In this manner, only the 3D-branches that have high confidence continue to issue load prefetches.

5.4 Storage overhead

We capture the storage requirements for each of the optimization techniques. All of this storage is on top of the 3.2 KB the baseline 3D-Branch Overrider requires.

1) **Memory Dependence Prediction:** The store/load cache module (that enables the MDP) is a 8-way, 1024 entry structure and has a 8 bit tag per entry and therefore requires 1KB. Value File (VF) is sized at 128 entries to hold the store values requiring another 1 KB. So, this technique requires 2 KB.

2) **Load Prioritization:** Enabling the 128 entry LBT tracking 8 PCs and 8 bit tag per PC requires 1KB for storing the backslice of the feeder loads.

3) **Load Address Prediction:** As mentioned earlier, we repurpose the EVES predictor [34] to predict the load addresses. We integrated the 384-entry data table to store the load addresses along with a smaller 256-entry predictor array. We optimized the table

storing load addresses based on the observation that several them had the same 32 most significant bits (MSB). Using a small 8 entry table, we capture the MSB bits and only capture the 3 bit entry id in the data table. This results in a 2.41KB load address predictor and 32 entry Prefetch Load Tracker requires 0.05KB. Together, this technique requires 2.46KB.

6 WORKLOADS AND EVALUATION SETUP

The 3D-Branch Overrider is evaluated across 104 workloads spanning multiple application suites. Simpoint-like methodology [21] is used to identify the representative phases while running reference set of inputs from benchmarks form a suite and for real world workloads the inputs were drawn from actual usage. Table 1 shows the distribution of the workloads and the broad categories they belong to. We study workloads that are likely to be bottlenecked by branch mispredictions (had a high MPKI (>2)), and exhibit a modest fraction of these mispredictions arising from 3D-branches.

Table 1: Evaluated Benchmark Categories

Category	Description	Count
Cloud	Data analytics on Hadoop, Data streaming using Spark [5], BigBench [20], Transactional processing using Cassandra [13]	13
Enterprise	SPECjbb [8], Web search, Particle rendering	7
ISPEC	ISPEC06 [3] & ISPEC17 [7]	28
Multimedia(MM)	Video Games, Photo-editing, Animation, Video conversion, Mediaplayer	12
SYSmark	SYSmark [4]	8
Personal Computing	Email, Voice-to-text tools, Image converters, HTML backend workload, Geekbench [25]	11
Web Browsing	Firefox, JavaScript workloads	25

Our performance evaluation is done using our in-house cycle-accurate simulator modeling a Icelake-like x64 core [6] clocked at 3.2 GHz. We list some of the relevant parameters in Table 2. While 3D-Branch Overrider is not a predictor but an accurate branch overriding technique, MPKI reduction here refers to the number of expensive branch-misprediction related pipeline flushes triggered from the execution stage that are saved. These are replaced instead by low-latency front-end re-steer events thus benefiting both performance and efficiency.

Table 2: Simulator Parameters

Core	5-wide OOO core, 352-entry ROB, 128-entry Load Buffer, 72-entry Store Buffer, 128 entry Allocation Queue [1, 6]
Baseline Branch Predictor	TAGE-SC-L - 64 KB [33], and ITTAGE [31] 15 cycles misprediction penalty
Branch Target Buffer	8K entries
L1 cache	Private, 48KB, 64B line, 8 way; Prefetcher: Instruction pointer-based stride prefetcher [23]
L2 cache	Private, 512KB, 64B line, 8 way, Prefetcher: Streamer [23]
LLC	Shared, Inclusive, 8MB, 64B line, 16 way
Main Memory	Dual channel DDR4-2133MHz

7 RESULTS

In this section, we discuss the MPKI and performance impact from integrating 3D-Branch Overrider in a Icelake-like OOO core. We discuss the workloads that gain significantly from 3D-Branch Override and also categorize the early re-steers depending on where in the pipeline they are triggered. We also quantify the performance benefits from applying the different optimization techniques besides quantifying their storage overheads. We also present sensitivity studies that show that as the structure sizes are varied, 3D-Branch Overrider can bring in additional gains. Further, we show that the impact of 3D-Branch Overrider only increases in future OOO cores and, lastly, we integrate the CBP-unlimited predictor to highlight that *3D-Branch Overrider continues to remain effective even with an impractically-sized baseline predictor.*

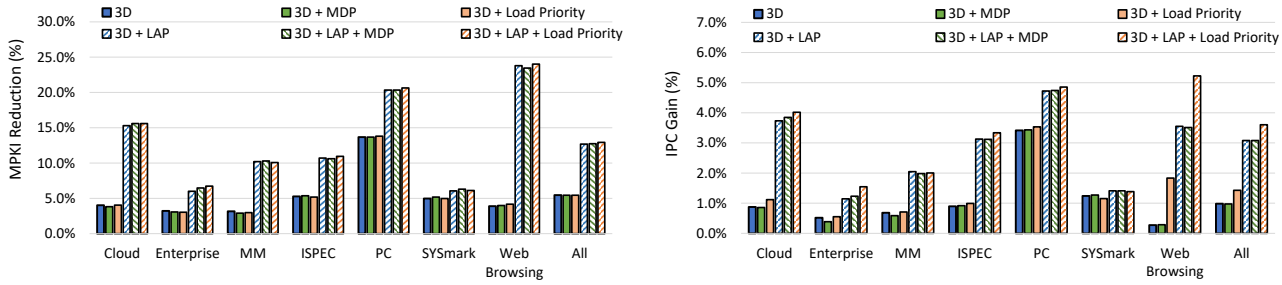
7.1 Performance Benefits from 3D-Branch Overrider

In Figure 7(a) and 7(b), we show the MPKI reduction and performance gains from the baseline 3D-Branch Overrider and the different optimizations on top of our baseline system. Overall, across the 104 workloads, 3D-Branch Overrider provides about 5.4% MPKI reduction resulting in 0.98% IPC gains. Some of the workloads like gzip gain as much as 6% in IPC demonstrating that 3D-Branch Overrider is efficient at limiting the branch misprediction penalty when the opportunity exists. Specific categories like Personal Computing show good performance sensitivity as geekbench, HTML backend, and compression workloads are significantly benefited by misprediction savings from 3D-branch override from executed loads. The gobmk workload that was discussed in Section-III experiences a 4.2% reduction MPKI resulting in a 1.44% gain in IPC. In Figure 8(a), the stacked-bar on the left, categorizes the misprediction reduction based on the location in the processor's pipeline the re-steering was triggered from. We find that most of the re-steers are from the branch prediction stage while having the flexibility to re-steer at the allocation queue does provide additional gains.

Combining the MDP with 3D-Branch Overrider to feed the load value from a prior store does not provide significant gains. Specific workloads such as 445.gobmk(SPEC06) and 631.sjeng(SPEC17) showed sensitivity which results in 2.1% and 3.5% IPC gain respectively. We find MDP to be accurate and capture most of the opportunity when the store PC is relatively closer to the load PC. Since MDP is quite sensitive to the ROB, LB and SB sizes, we scaled these structures to 2x the default configuration and there we see that 3D-Branch Overrider with MDP provides 1.1%IPC gain across the 100+ workloads.

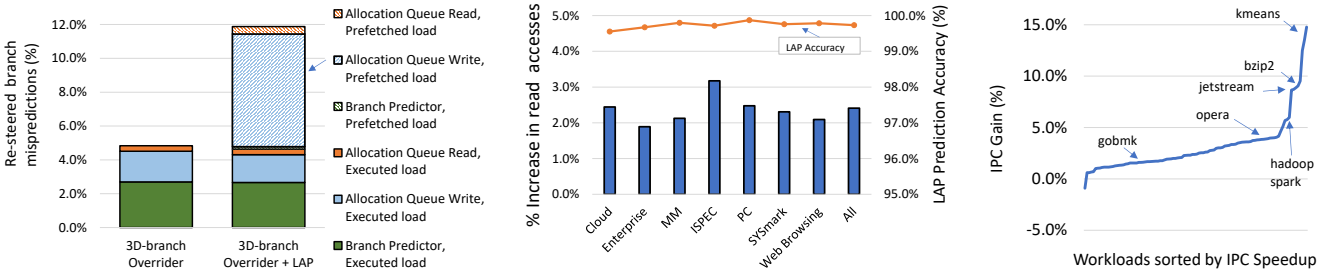
On prioritizing the backslice of the feeder load, the IPC gain increases to 1.4% while the MPKI reduction change is not significant. Specific categories like Cloud and Web Browsing are quite sensitive to this technique. After careful analysis, we realized that almost all the gains are purely from accelerating the feeder load backslice but only a small portion translated into an override from the 3D-Branch Overrider. While this is an interesting observation, we do not claim novelty here.

Enhancing the 3D-Branch Overrider with the LAP and prefetching the loads provides very significant MPKI reduction and IPC gains. Almost all categories reduce their MPKI by more than 5% from this enhancement with web-browsing seeing over 20% reduction in MPKI. Overall, integrating this technique results in the 3D-Branch Overrider providing 3.1% IPC gain. To shed further light into where the gains are coming from, in the stacked-bar on the right in Figure 8(a), we categorize the re-steers that are triggered with this technique. As seen, significant portion of the gains from 3D-Branch Overrider combined with the LAP are obtained when the re-steers are triggered as the branch is about to enter the allocation queue ("Allocation Queue Write, Prefetched load" in the right



(a) MPKI reduction per category (b) IPC gain per category

Figure 7: MPKI reduction and IPC gain (per category, overall) for 3D-Branch Overrider when applied with other optimizations.



(a) (b) (c)

Figure 8: (a) Number of re-steers with 3D-Branch Overrider (with and without LAP) normalized to number of total mispredictions from baseline predictor, (b) Increase in load traffic from LAP (primary axis), prediction accuracy of LAP (secondary axis), and (c) S-curve for 3D-Branch Overrider + LAP scheme.

stacked bar). The flexibility built into the 3D-Branch Overrider to inject overrides at multiple pipeline stages in the front-end helps significantly to enhance the overall gains.

We also evaluated LAP on top of the 3D-Branch Overrider, but without any LAP based re-steers (not shown in the figures). This helps us understand if the IPC gains of 3D-Branch Overrider with LAP are mainly due to data prefetching or indeed from LAP based 3D-Branch overrides. This scheme shows 0.93% IPC gain relative to the baseline which is even worse than 3D-Branch Overrider, and it is due to increased contention for L1 cache bandwidth. This shows that LAP based re-steers are key to getting higher performance.

In Figure 8(b), we graph the accuracy of the LAP. Across all categories, the LAP is very accurate (99.7%). This is important since the prefetching adds additional load accesses which can negatively impact the latency of baseline read requests. Overall, LAP shows a coverage of 14% for 3D-branches. In the same figure, we also see the read requests increase by about 2.3% with no category seeing more than a 3.2% increase in overall memory traffic. Despite the slight increase in the read requests, the MPKI reduction obtained and the cycles saved by re-steering the pipeline much earlier than actual misprediction event helps boost the IPC gains.

Figure 7(b) illustrates the benefits when combining the different optimization techniques. We find that combining the 3D-Branch Overrider with the prefetching for loads whose addresses are predictable while also prioritizing the backslice of the remaining feeder loads for execution (3D+LAP+Load prioritization) improves the overall IPC gains to 3.6%.

Figure 8(c) presents the IPC S-curve of the 3D+LAP configuration across individual workloads. We see that K-means clustering workload belonging to the Cloud category gains a significant 13% in IPC as our proposed techniques very effectively remove a large fraction of the misprediction related pipeline flushes in their execution (30% reduction in MPKI). 18 workloads see >4% IPC gains and as shown in the figure this includes workloads like 401.bzip2(SPEC06), hadoop-spark(Cloud), jetstream(web-browsing) spanning several categories *helping establish that 3D-Branch Overrider is impactful across several application categories.*

7.2 Results Summary

In Table 3, we summarize the IPC gains and the storage overhead for these configurations. Taking performance gains and storage overhead into consideration, combining the 3D-Branch Overrider with LAP predictor is an excellent design point with high MPKI reduction (12.7%) while also resulting in good IPC gains (3.1%). We use the 3D-Branch Overrider + LAP as the default configuration on subsequent studies.

Table 3: Results Summary

Configuration	MPKI Reduction	IPC Gain	Storage
3D	5.4%	0.98%	3.2 KB
3D + MDP	5.4%	0.98%	5.2 KB
3D + Load prioritization	5.4%	1.4%	4.2 KB
3D + LAP	12.7%	3.1%	5.66 KB
3D + LAP + MDP	12.8%	3.1%	7.66 KB
3D + LAP + Load prioritization	12.9%	3.6%	6.66 KB

7.3 Sensitivity Studies on 3D-Branch Override Parameters

Figure 9, we capture the IPC impact when scaling different components of the 3D-Branch Overrider, including the LAP predictor. While each of the optimizations provide modest gains, we see that scaling the LAP predictor to 256 entries (3.2KB) improves the IPC gain even more (from 3.1% to 3.5% across the 100+ workloads).

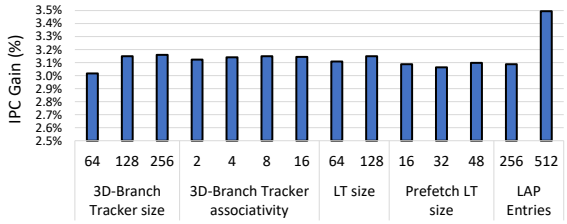


Figure 9: 3D-Branch Overrider sensitivity to configuration parameters such as 3D-branch tracker size, associativity, LT size, prefetch LT size, and LAP predictor size.

7.4 Impact of 3D-Branch Overrider in Future Cores

As discussed earlier, the importance of a technique like 3D-Branch Overrider to tackle data dependent branches only increases over generations. In Figure 10, we elaborate on this by measuring the IPC gain across different configurations scaled from our baseline Icelake-like core. The overall gains increase from 3.1% to 4.3% for the 2x configuration, thereby demonstrating our observation.

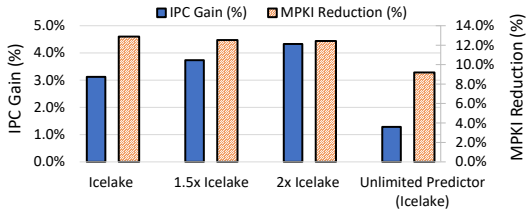


Figure 10: IPC gain and MPKI reduction from 3D-Branch Overrider for pipeline scaled to 1.5x and 2x of Icelake and also with the CBP-5 unlimited area branch predictor.

7.5 Impact of 3D-Branch Overrider with Very Large Baseline Branch Predictor

Finally, we examine whether the data dependent branch mispredictions can be addressed with just scaling the conventional predictor sizes. We use the CBP-5 winner [32] in unlimited storage category as the baseline predictor to showcase this. In Figure 10, even with large predictor as baseline we get 9.2% in MPKI reduction resulting in 1.3% IPC gains. For all the categories of workloads, a significant fraction of the MPKI reduction (60-70%) from 3D-Branch Overrider are still retained. We conclude that even in the presence of an unreasonably large branch predictor, that reduces the mispredictions by 40% compared to 64KB CBP-5 winner predictor, data-dependent branch mispredictions cannot be effectively mitigated without unique and targeted solutions such as 3D-Branch Overrider.

8 RELATED WORK

Prior works such as [11, 16, 19, 22, 39] on enhancing branch predictors based on the correlation with prior data values seen. But, when data has high entropy, it affects their accuracy. Among recent works, EXACT predictor [9] proposes associating data dependent branches with their corresponding producer load address and using this information to disambiguate and uniquely predict multiple instances of the same data dependent branch PC. Unlike the EXACT predictor, our work uses the load value to *compute* the branch direction and accomplishes this at much less storage.

We also discussed Store-load-branch (SLB) predictor [18] which requires compiler support to identify the store that produces the value eventually feeding into the data-dependent branches. Control Flow Decoupling (CFD) [36], is another approach to hoist the load out the loop to create the slack between the load and branch, which then feeds the dependent branches using a prediction queue. Our proposal differs from SLB and CFD in two important ways. Firstly, our work proposes a hardware-only technique to mitigate the misprediction penalty of data-dependent branches. Secondly, unlike SLB, which used the memory address prediction to forward the data from a prior store (producer) to the data-dependent branch (consumer), our work utilized it to prefetch the data from memory subsystem. *Since the producer stores happen much earlier than loads, the storage overhead to track these values is higher as compared to using the load to read the value in timely manner.*

Works such as [16, 22] use data dependence tracking for enhancing the branch predictor. The goal of these predictors are to track the dependence chain of branches and when some of the registers in the dependence chain are ready, a predictor table is looked up to obtain a prediction. Branch Outcome Anticipation [17] also looked at reducing the branch misprediction penalty rather than improving the branch accuracy. For unconfident predictions, [10] proposes to prioritize the issue of the instructions on the backslice of a branch to reduce the branch misprediction penalty. These works are complementary to our proposal and can be implemented on top of the 3D-Branch Overrider for additional gains. Even though the target of [10], [17] and our work is to reduce the branch misprediction penalty, our approach is distinctive in two ways: (1) they do not examine the data-dependent branches as the focus of design, and (2) if the load execution is delayed in the pipeline, 3D-Branch Overrider can still use the LAP scheme to prefetch the load value to perform the override.

9 CONCLUSION

Mispredictions from data-dependent branches remains a key challenge for conditional branch predictors. We identified that mispredictions from 3D-branches are quite common and proposed the 3D-Branch Overrider, to detect and address these mispredictions. Further, to enhance coverage, we integrated a load address predictor to fetch the load value in time to be used for branch direction computation. Together, these techniques resulted in lowering the mispredictions on 100+ workloads we studied by 12.7% leading to 3.1% IPC gain. We also established that in future OOO cores, the impact of techniques like 3D-Branch Overrider will only increase and scaling the baseline branch predictors is insufficient to limit the branch mispredictions from data dependent branches.

REFERENCES

[1] 2015. Intel Skylake Architecture Preview Quick Take From IDF 2015. https://hothardware.com/reviews/intel-skylake-architecture-preview-quick-take-from-idf-2015.

[2] 2018. 6th Generation Intel Processor Family. https://www.intel.com/content/www/us/en/processors/core/desktop-5th-gen-core-family-spec-update.html.

[3] 2018. SPEC CPU 2006. https://www.spec.org/cpu2006/.

[4] 2018. SYSmark 2014. https://bapco.com/products/sysmark-2014/.

[5] 2019. Apache Spark. http://spark.apache.org/.

[6] 2019. Examining Intel's Ice Lake Processors: Taking a Bite of the Sunny Cove Microarchitecture. https://www.anandtech.com/show/14514/examining-intels-ice-lake-microarchitecture-and-sunny-cove/3.

[7] 2019. SPEC CPU 2017. https://www.spec.org/cpu2017/.

[8] 2019. SPECjbb2015. http://spec.org/jbb2015/.

[9] Muawya Al-Otoom, Elliott Forbes, and Eric Rotenberg. 2010. EXACT: explicit dynamic-branch prediction with active updates. In *Proceedings of the 7th ACM international conference on Computing frontiers*. ACM, 165–176.

[10] Hideki Ando. 2018. Performance improvement by prioritizing the issue of the instructions in unconfident branch slices. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 82–94.

[11] Juan L Aragón, José González, José M García, and Antonio González. 2001. Selective branch prediction reversal by correlating with data values and control flow. In *Proceedings 2001 IEEE International Conference on Computer Design: VLSI in Computers and Processors. ICCD 2001.* IEEE, 228–233.

[12] Trevor E Carlson, Wim Heirman, Osman Allam, Stefanos Kaxiras, and Lieven Eeckhout. 2015. The load slice core microarchitecture. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 272–284.

[13] Apache Cassandra. 2014. Apache cassandra. *Website. Available online at http://planetcassandra. org/what-is-apache-cassandra* 13 (2014).

[14] Championship Branch Prediction (CBP-4) 2014. *4th JILP Workshop on Computer Architecture Competitions (JWAC-4)*. Retrieved August 7, 2020 from https://www.jilp.org/cbp2014/

[15] Championship Branch Prediction (CBP-5) 2016. *5th JILP Workshop on Computer Architecture Competitions (JWAC-5)*. Retrieved August 7, 2020 from https://www.jilp.org/cbp2016/

[16] Lei Chen, Steve Dropsho, and David H Albonesi. 2003. Dynamic data dependence tracking and its application to branch prediction. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.* IEEE, 65–76.

[17] A. Farcy, O. Temam, R. Espasa, and T. Juan. 1998. Dataflow analysis of branch mispredictions and its application to early resolution of branch outcomes. In *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture.* 59–68.

[18] M Umar Farooq, Lizy K John, et al. 2013. Store-load-branch (slb) predictor: A compiler assisted branch prediction for data dependent branches. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA).* IEEE, 59–70.

[19] Hongliang Gao, Yi Ma, Martin Dimitrov, and Huiyang Zhou. 2008. Address-branch correlation: A novel locality for long-latency hard-to-predict branches. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture.* IEEE, 74–85.

[20] 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.*

[21] 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.

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

[23] R Intel. 2014. Intel 64 and ia-32 architectures optimization reference manual. *Intel Corporation, Sept* (2014).

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

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

[26] Pierre Michaud. 2005. A PPM-like, tag-based branch predictor. *Journal of Instruction Level Parallelism* 7, 1 (2005), 1–10.

[27] Sparsh Mittal. 2019. A survey of techniques for dynamic branch prediction. *Concurrency and Computation: Practice and Experience* 31, 1 (2019), e4666.

[28] Anant Vithal Nori, Jayesh Gaur, Siddharth Rai, Sreenivas Subramoney, and Hong Wang. 2018. Criticality aware tiered cache hierarchy: a fundamental relook at multi-level cache hierarchies. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA).* IEEE, 96–109.

[29] Lois Orosa, Rodolfo Azevedo, and Onur Mutlu. 2018. Avpp: Address-first value-next predictor with value prefetching for improving the efficiency of load value prediction. *ACM Transactions on Architecture and Code Optimization (TACO)* 15, 4 (2018), 1–30.

[30] Daniel Richins, Tahrina Ahmed, Russell Clapp, and Vijay Janapa Reddi. 2018. Amdahl's law in big data analytics: Alive and kicking in tpcx-bb (bigbench). In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA).* IEEE, 630–642.

[31] André Seznec. 2011. A 64-Kbytes ITTAGE indirect branch predictor.

[32] André Seznec. 2016. Exploring branch predictability limits with the MTAGE+ SC predictor.

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

[34] André Seznec. 2018. Exploring value prediction with the EVES predictor. In *Championship Value Prediction, An All-Year Computer Architecture Competition.*

[35] 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

[36] Rami Sheikh, James Tuck, and Eric Rotenberg. 2012. Control-flow decoupling. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture.* IEEE, 329–340.

[37] Stephen J Tarsa and Chit-Kwan Lin. 2019. Branch Prediction Is Not A Solved Problem: Measurements, Opportunities, and Future Directions. *arXiv preprint arXiv:1906.08170* (2019).

[38] Michael E Thomadakis. 2011. The architecture of the Nehalem processor and Nehalem-EP SMP platforms. *Resource* 3, 2 (2011), 30–32.

[39] Renju Thomas, Manoj Franklin, Chris Wilkerson, and Jared Stark. 2003. Improving branch prediction by dynamic dataflow-based identification of correlated branches from a large global history. In *Proceedings of the 30th annual international symposium on Computer architecture.* 314–323.

[40] Gary S Tyson and Todd M Austin. 1997. Improving the accuracy and performance of memory communication through renaming. In *Proceedings of 30th Annual International Symposium on Microarchitecture.* IEEE, 218–227.