Auto-Predication of Critical Branches*

Adarsh Chauhan
Processor Architecture Research Lab
Intel Labs
Bengaluru, India
adarsh.chauhan@intel.com

Jayesh Gaur
Processor Architecture Research Lab
Intel Labs
Bengaluru, India
jayesh.gaur@intel.com

Zeev Sperber
Intel Corporation
Haifa, Israel
zeev.sperber@intel.com

Franck Sala
Intel Corporation
Haifa, Israel
franck.sala@intel.com

Lihu Rappoport
Intel Corporation
Haifa, Israel
lihu.rappoport@intel.com

Adi Yoaz
Intel Corporation
Haifa, Israel
adi.yoaz@intel.com

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

Abstract—Advancements in branch predictors have allowed modern processors to aggressively speculate and gain significant performance with every generation of increasing out-of-order depth and width. Unfortunately, there are branches that are still hard-to-predict (H2P) and mis-speculation on these branches is severely limiting the performance scalability of future processors. One potential solution to mitigate this problem is to predicate branches by substituting control dependencies with data dependencies. Predication is very costly for performance as it inhibits instruction level parallelism. To overcome this limitation, prior works selectively applied predication at run-time on H2P branches that have low confidence of branch prediction. However, these schemes do not fully comprehend the delicate trade-offs involved in suppressing speculation and can suffer from performance degradation on certain workloads. Additionally, they need significant changes not just to the hardware but also to the compiler and the instruction set architecture, rendering their implementation complex and challenging.

In this paper, by analyzing the fundamental trade-offs between branch prediction and predication, we propose Auto-Predication of Critical Branches (ACB) — an end-to-end hardware-based solution that intelligently disables speculation only on branches that are critical for performance. Unlike existing approaches, ACB uses a sophisticated performance monitoring mechanism to gauge the effectiveness of dynamic predication, and hence does not suffer from performance inversions. Our simulation results show that, with just 386 bytes of additional hardware and no software support, ACB delivers 8% performance gain over a baseline similar to the Skylake processor. We also show that ACB reduces pipeline flushes because of mis-speculations by 22%, thus effectively helping both power and performance.

Index Terms—Microarchitecture, Dynamic Predication, Control Flow Convergence, Run-time Throttling

I. INTRODUCTION

High accuracy of modern branch predictors [2]–[5] has allowed Out-of-Order (OOO) processors to speculate aggressively on branches and gain significant performance with every generation of increasing processor depth and width. Unfortunately, there still remains a class of branches that are Hard-to-Predict (H2P) for even the most sophisticated branch predictors [6]–[8]. These branches cost not only performance but also significant power overheads because of pipeline flush and re-execution upon wrong speculation.

Figure 1 shows the performance improvements from an oracle perfect branch predictor with increasing processor depth and width. For these results, the baseline is similar in parameters to the Skylake processor [1] and uses a branch predictor similar to TAGE [2], [3]. We show the performance impact of perfect branch prediction on a continuum of processors with varying OOO resources compared to Skylake. As is evident from Figure 1, the performance potential of perfect speculation increases with OOO processor scaling. For instance, a three times wider and deeper machine than the Skylake baseline is almost two times more speculation bound than Skylake. These results clearly motivate the need for mitigating branch mis-speculations, especially since future OOO processors are expected to scale deeper and wider [9]. As it gets harder to improve branch prediction, there is an

*Concepts, techniques and implementations presented in this paper are subject matter of pending patent applications, which have been filed by Intel Corporation.
urgent need to investigate solutions to address this problem.

One possible solution is to limit speculation when an H2P branch is encountered. A classic approach to achieve this is predication [10], which allows fetching both the taken and not-taken portions of a conditional branch, but the execution is conditional based on the final branch outcome. Because predication inherently limits instruction level parallelism, it can be detrimental to overall performance. To overcome this, several prior techniques have tried to predicate only those instances of H2P branches which have low confidence of prediction [7], [11]–[14]. Policies like Diverge Merge Processor (DMP) [7], [15] use careful compiler profiling to select target H2P branches, and then throttle their application using run-time monitoring of branch prediction confidence. These techniques showed great promise in mitigating the problems with H2P branches. Unfortunately for almost a decade, no advancement has been further made in these policies, and as we will show in this paper, on modern OOO processors with accurate branch predictors these policies end up creating severe run-time bottlenecks for some applications, thereby limiting their applicability. Moreover, these techniques need significant changes to the compiler and the instruction set architecture (ISA), which makes their adoption challenging.

In this work, we first perform a thorough study of the performance trade-offs created by limiting speculation using predication. Based on this analysis, we propose Auto-Predication of Critical Branches (ACB) that intelligently tries to disable speculation only on branches critical for performance. ACB needs no compiler or ISA support and has a micro-architecture which is implementable in modern OOO processors. Specifically, we make the following new contributions.

1) We present an analysis of the fundamental cost-benefit trade-offs that come to the fore when branch prediction is replaced by predication, especially how it impacts the program critical path. Guided by this understanding, we propose ACB, a light-weight mechanism that intelligently decides whether limiting speculation for a given critical branch is helpful or detrimental to performance. ACB is a holistic and complete solution that mitigates performance losses by wrong speculation, while ensuring that it does not create performance inversions.

2) We describe ACB’s implementation in a modern OOO processor with no ISA changes or compiler support. ACB learns its targeted critical branch PCs (program counters) using simple heuristics, and uses a novel hardware mechanism to accurately detect control flow convergence using generic patterns of convergence. This is unlike previous approaches [7], [12]–[14] that were dependent upon compiler analysis and profiling. With small changes to Fetch and OOO pipelines, ACB dynamically predicates critical branches, thereby reducing costly pipeline flushed and improving performance.

3) We also propose a unique throttling system (Dynamo) that monitors the run-time performance delivered by applying ACB on any targeted branch and promptly throttles ACB instances that are found to be degrading performance. This is in contrast to typical throttling mechanisms that rely on monitoring multiple local performance counters. Cost-benefit estimation is complex for predication based solutions as they influence performance, negatively or positively, in many different ways. By directly monitoring the dynamic performance, Dynamo makes holistic and informed decisions. With suitable adaptations, Dynamo’s generic approach can be applied to control any performance feature which similarly requires balancing of cost-benefit trade-off.

Our simulation results show that with just 386 bytes of overall additional storage, ACB delivers 8% performance improvement over a baseline processor similar to Intel Skylake [1]. Since ACB requires little additional hardware and saves 22% of the baseline mispredictions, it helps both power and performance. We also show that ACB overcomes some of the fundamental limitations of past compiler-based optimizations and scales seamlessly to future processors, that are expected to be even more bound by branch mis-speculations.

II. BACKGROUND AND MOTIVATION

Modern branch predictors use program history to predict future outcomes of a branch [2], [4], [5]. Decades of research have made them very accurate. However, there remains a class of branches that are still hard to predict. Many such branches are data dependent branches and are difficult to predict using just program history [6].

We characterized branch mispredictions on our selected workloads. We found that on average, in a given program phase, 64 branch PCs sufficiently contribute to more than 95% of all dynamic mispredictions. Analysing the type of H2P branches reveals that a majority of total mispredictions come from direct conditional branches, of which 72% comes from convergent conditional branches. We define convergent branches as those branches whose taken and not-taken paths can converge to some later point in the program (using the same convergence criterion as DMP [7]). Loops are naturally converging and contribute to another 13%. Remaining 13% conditional branches exhibit non-converging control flows. These observations lead us to conclude that the majority of branch mispeculations can be addressed by focusing on a small set of 64 convergent conditional H2P branches.

A. Program Criticality

The performance of any OOO processor is bound by the critical path of execution. Critical path can be conceptually understood as the sequence of (data/control) dependent instructions which determines the total execution cycles of a program. Fields et al. [16] presented a graph-based definition of the critical path where the critical path is the maximum weighted path in the data-dependency graph (DDG). Instructions, whose execution lies on this path, are critical for performance. Branch mis-speculation appears on the critical path as a control dependency between the mispredicting branch and the
correct target fetched after branch resolution. While most of the branch mispredictions usually lie on the critical path, not all instances are critical for performance. Some mispredictions lie in the shadow of other, more critical events (e.g. long latency loads that miss LLC) and may not be critical.

B. Predication

One possible solution to the branch misprediction problem is to prevent speculation when an H2P branch is encountered. Static predication provides code for both the taken and not-taken directions of conditional hammocks, but the run-time execution is conditionally data-dependent on the branch outcome. Most ISAs have some support for static predication [17], [18]. Even though predication reduces critical path length by preventing pipeline flushes upon mispredictions, it substitutes control dependencies with data dependencies in the execution of the program. This limits instruction level parallelism and can elongate the critical path. To mitigate this, past approaches have dynamically applied predication only on branch instances having low confidence from branch prediction [7], [13], [14].

Wish Branches [12] relies on the compiler to provide predicated code for every branch PC. For every dynamic branch instance, branch prediction confidence is used to select between fetching the predicated code or speculate normally. However, this approach increases the compiled code footprint. Dynamic Hammock Predication (DHP [11]) uses the compiler to identify simple, short hammocks which can be predicated dynamically (and profitably) and fetches both the directions of the hammock in hardware. Diverge Merge Processor (DMP) [7] improves upon both Wish Branches and DHP. DMP uses compiler analysis-and-profiling to identify frequently mispredicting branch candidates and modifies the compiled binary to supply the convergence information for frequently converging, complex control flow patterns. Using ISA support and changes to processor front-end, DMP fetches both taken and not-taken paths of the conditional branch. Register Alias Table (RAT) in the OOO is forked and both the paths are renamed separately. Select-micro-ops are injected to dynamically predicate the data outcome from both paths.

By predicting branch confidence separately at run-time, DMP tries to effectively predicate only those instances that are likely to mispredict and delivers significant performance. However, as we will analyze in the following section, predication-based strategies like DMP can create new critical paths of execution, which are difficult to comprehend just by monitoring branch confidence. Also, training data-sets used by the compiler (for developing static/profiling-based branch selection criteria) can be very different from actual testing data seen during execution. Since many H2P branches are data dependent, the efficacy of compiler analyses [15] is dependent on the quality of profiled input. As a result, application of DMP and similar schemes may result in performance inversions on certain workloads. Moreover, such schemes need simultaneous changes to the hardware, compiler as well as ISA, which makes their practical implementation challenging.

In Section V-C, we will quantitatively discuss the performance of DMP and contrast it with our proposal.

C. Effects of Predication on Critical Path

As mentioned above, there are costs of performing predication to realize the benefits of saving mispredictions by eliminating speculation on branches. An imbalance in this delicate trade-off for predication can cause performance inversions. Hence, it is important to understand and consider the factors influencing this balance. Additionally, to encourage adoption on modern processors, we need techniques that are easy to implement completely in hardware, without needing support from the compiler or ISA. In this section, we will hence use program criticality to first develop an understanding of how predication changes the critical path of execution. Through this analysis, we will motivate the need for our feature.

1) Limiting Allocation: Predication, by fetching both the taken and not-taken paths of a branch, alters the critical path of execution. Figure 2(a) shows an example DDG (using notations from [16]) with and without predication. Without predication on a branch, a branch misprediction introduces the misprediction latency on the critical path. However, with predication, the critical path involves the latency of fetching control dependent region on both the directions and allocating them into the OOO (whereas the baseline speculates and fetches on only one direction).

Consider the misprediction rate for a given H2P branch as mispred_rate, and the taken path has T and not-taken path has N instructions. Assume \( p \) to be the probability of the branch being taken. With predication, we need to fetch \((T+N)\) instructions for every predicated instance. alloc_width is the maximum number of instructions that can be allocated in the OOO per cycle and mispred_penalty is the penalty of misprediction, i.e. the total time taken to execute the mispredicting branch, signal the misprediction and the subsequent pipeline flush latency. For the baseline, misprediction increases the critical path of execution by \((\text{mispred_rate} \cdot \text{mispred_penalty})\) cycles. On the other hand, with predication, the critical path increases by \((\langle T+N \rangle - (p \cdot T + (1-p) \cdot N))/\text{alloc_width})\). Predication will be profitable if,

\[
\frac{(1-p) \cdot T + p \cdot N}{\text{alloc_width}} \leq (\text{mispred_rate} \cdot \text{mispred_penalty})
\]

Equation 1 clearly shows the trade-off between higher allocations and saving the pipeline flushes by mispredictions. Let’s assume that allocation width (alloc_width) is 4, pipeline flush latency (mispred_penalty) is 20 cycles and we have equal probability of predicting taken and not-taken. If misprediction rate is 10%, then predication will be beneficial only if the total instructions in the predicated branch body (taken and not-taken paths combined \((T+N)\)) are less than 16. On the other hand, if branch body size is larger, say 32 instructions, then predication should be applied only for branches having misprediction rate greater than 20%. Realistically, the actual penalty for a branch misprediction is higher than just the pipeline flush latency, since it includes the execution latency of
the branch-sources required for computing its outcome. Hence, equation 1 will have a higher value for mispred_penalty, and predication may be able to tolerate somewhat larger number of extra allocations. Therefore, we can conclude that both misprediction rate and branch body size need to be considered to qualify any branch for predication. For those micro-architectures that allocate in OOO in terms of micro-operations [19], this equation needs to be suitably adjusted.

2) Increasing Branch Mispredictions: Figure 2(b) shows a sample program where branch B1 frequently mispredicts. Since B1 is a small hammock, it should be very amenable to dynamic predication. However, there is another branch B2 that is perfectly correlated with B1, but is not amenable to predication. Interestingly, in the baseline, B2 usually does not see any misprediction since B1 is more likely to execute (and cause pipeline flushes) before B2 can be executed. Perfect correlation between them would mean that B2 will always be correctly predicted when it is re-fetched, since it knows the outcome of B1. This happens because the global branch predictor would repair the prediction of B1 when there is no predication (since global history is updated), and B2 will always learn the correlation with B1.

With predication, however, there is no update to global history from B1. Therefore, B2 will start mispredicting and the effective number of mis-speculations will not come down. In fact, because of predication on B1, B2 will now take a longer time to execute, thereby elongating the critical path. Hence, branches like B1 should not be predicated, unless B2 can also be predicated. This effect of increasing the baseline mispredictions is more pronounced in cases of dynamic predication on branches with complex control flow patterns and large control dependent regions. Since branch history update and resolution are separated in branch speculation, the branch history cannot be perfectly corrected to improve the prediction for branches following the predicated region.

3) Elongating Critical Paths: Figure 2(c) shows another example where the body of an H2P branch creates sources for a critical (long latency) load. Without predication, the load would still be launched, and may be correct if the branch prediction was correct. However, due to predication, this long latency load’s dispatch is dependent upon the execution of the predicated branch. As a result, the critical path of execution may get elongated. If this H2P branch is very frequent, predication can result in a long chain of dependent instructions. In all such scenarios, resorting to normal branch speculation, even if the accuracy of branch prediction is low, may be a more optimal solution than predication.

To summarize our learnings, we first need to detect our target branches and learn their convergence patterns. Secondly, the selection criteria for critical branches should take into account the size of the branch body and the misprediction rate. Thirdly, alterations to the critical path due to predication need to be detected and handled at run-time. Finally, predication needs to be dynamic and completely implementable in hardware. These problems motivate us towards our proposal which we will describe in detail in the following section.

### III. AUTO-PREDICATION OF CRITICAL BRANCHES (ACB)

The essential idea behind ACB is to eliminate speculation when the criteria discussed in Section II are satisfied. ACB first detects conditional critical branches and then uses a novel hardware mechanism to find out their point of reconvergence. Thereafter, a simple mechanism is used to fetch both taken and not-taken portions (up to the reconvergence point) of the conditional branch. After the ACB-branch executes in the OOO, the predicated-true path is executed, whereas small micro-architectural modifications in the pipeline make the predicated-false path transparent to program execution. Finally, a dynamic monitoring (Dynamo) scheme monitors the runtime performance and appropriately throttles ACB. We now describe the micro-architecture of ACB in more detail.

#### A. Learning Target Branches

As reasoned in Section II-A, not all mispredicting branch instances impact performance. However, branches that frequently mispredict, invariably end up having several dynamic instances that lie on the critical path. We found that the frequency of misprediction for a given branch PC is a good measure of its criticality. Our scheme hence uses a simple criticality filter ($\leq 16$ mispredictions in 200K retired instructions window) to filter out infrequently mispredicting branches. Once convergence is confirmed for a branch, we further ensure...
We also experimented with other criticality heuristics to improve the above qualification criteria. Offline analysis of data dependence graphs for different applications expectedly showed that some fraction of the branch misprediction instances are not on the critical path. However, segregating such instances on-the-fly, and with reasonable hardware, is very challenging. We considered the heuristic of counting a mis-speculation event as critical only if, at the time of misprediction, the branch is within a fourth of the ROB size from head of the ROB (i.e., oldest entry in the ROB). Those mispredictions which happen near the retirement are more critical for performance as they will cause a greater part of ROB to be flushed and consequently, more control-independent work to be wasted. This simple heuristic slightly improved the accuracy of the frequency based criticality filter. Such criticality heuristics can be improved by future research.

To track critical branches, ACB uses a direct-mapped Critical Table indexed by the PC of mispredicting conditional branches. Each table entry stores an 11 bit tag to prevent aliasing, a 2 bit utility counter for managing conflicts, and a 4 bit saturating critical counter. Every critical branch misprediction event (as defined by our heuristics) increments both critical counter and utility counter of its PC-entry. In case of conflict misses in the table, utility counter is decremented. An old entry will be replaced by a new contending entry only if utility counter is zero. As section II suggested, our experimental sweeps over this table size show that a small 64-entry table provides sufficient coverage useful for performance.

**B. Learning Convergent Branches**

The next step involves identifying convergent candidates among the identified critical branches. For this, ACB uses a single entry Learning Table (20 bytes) to detect convergence one-branch-at-a-time which is sufficient for its functionality.

**Types of Convergence:** Through analysis of various control flow patterns in different workloads, we identified three generic cases by which conditional direct branches can converge. Figure 3 illustrates the three types, that we refer to as Type-1, Type-2 and Type-3. Type-1 convergence is characterized by the reconvergence point being identical to the ACB-branch target. The simplest form of Type-1 branches are IF-guarded hammocks that do not have an ELSE counterpart. Type-2 convergence is characterized by the not-taken path having some Jumper branch, when taken, which when taken, has a branch-target that is ahead of the ACB-branch target. This naturally guarantees that the taken path which starts from the ACB-branch target will fall-through to meet the Jumper branch target, making it the reconvergence point in this case. Type-2 covers conditional branches having pair of IF-ELSE clauses. Finally, Type-3 convergence possesses a more complex control flow pattern (which can have either IF-only or IF-ELSE form). It is characterized by the taken path encountering a Jumper branch which takes the control flow to its target that is less than the ACB-branch target. This ensures that the not-taken path naturally falls through to meet the Jumper branch target.

We have generalized these three types so that other complex cases (see Figure 3) can also be contained within this set. However, the above description defines conditions that hold true for only forward-going branches (where the ACB-branch target PC is more than the branch PC). To cover the cases of backward-going branches, we adapted our algorithm by exploiting the commutative nature of convergence for back-branches. We use an important observation that by simply moving the original back-branch from the beginning of its Not-Taken block to the beginning of its Taken block, and modifying it accordingly to being a forward branch with target as its own original PC, the program remains logically unchanged. Thus, the reconvergence point detected in this modified scenario is going to be the same as original. Figure 4 illustrates this idea through an example.

Convergence detection mechanism is implemented during fetch since it needs to track only the PCs of instructions being fetched. When an entry in the critical table saturates its critical count, we copy the branch PC into the Learning Table which is occupied until we confirm convergence or divergence on both its directions. The mechanism first tries to learn if the ACB-branch is a Type-1 or Type-2 convergence. It begins by first...
inspecting the Not-Taken path. We track the first $N$ fetched PC’s following the ACB-branch. If we receive the target of the ACB-branch within this interval, we classify it as Type-1 and finish learning. Otherwise, if another taken branch is observed whose target is ahead of the ACB-branch’s target, then we record this branch’s target as the reconvergence point. We then validate the occurrence of the same reconvergence point on the next instance when the ACB-branch fetches the Taken direction, within the same $N$ instruction limit, before confirming it as Type-2. If neither Type is confirmed, we leave the ACB-branch as unclassified.

If still unclassified, we finally try to learn it as Type-3 by inspecting the Taken path. If, within $N$ instructions, we observe a taken branch whose target is before the ACB-branch, then we record this branch’s target as the reconvergence point. We then validate the occurrence of the same reconvergence point on the next instance when the ACB-branch fetches the Not-Taken direction. Upon success, we confirm it as Type-3.

At any stage, if we exhaust the $N$ instruction counting limit, we reset the Learning Table entry as a sign of non-convergence. Upon confirmation of any Type, we copy the branch PC to a new ACB Table entry, along with the learned convergence information. We then vacate the corresponding Critical Table entry and reset the Learning Table entry. Based on the analysis in Section II-C1 and experimental sweeps, we found $N = 40$ to be optimal to cover large-body convergences that can be supported while being profitable with the given misprediction rate thresholds.

**Criticality Confidence:** We use a 32-entry, 2-way ACB Table (indexed by branch PCs) having a 6-bit saturating probabilistic-counter. All the meta-data needed to fetch both the paths upon ACB application on a targeted branch PC is also stored in this table entry (detailed composition in Table I). Before ACB can dynamically predicate, we need to establish confidence in accordance with the trade-off described by Equation 1. During learning, we record the combined body size of both paths that need to be fetched (encoded in 2 bits) and proportionally set the required misprediction rate $m$ for this branch, using a static mapping of Body-Size-to-Misprediction-Rate (refer Table I). The confidence counter in the ACB table is incremented for every mis-predicting instance of this branch that triggers a pipeline flush. It is decremented probabilistically by $1/M$ (where $M = \frac{1}{m} - 1$) on every correct prediction. When this counter becomes higher than 32 (half of its saturated value), we start applying ACB.

**Convergence Confidence:** While critical counter is less than 32, we use a single-entry Tracking Table to monitor the occurrence of the learned reconvergence point PC on both taken and not-taken paths for every fetched branch instance. If the learned convergence does not happen, we reset its confidence counter. This way we exclude branches from getting activated which tend to diverge more often. Despite low-associativity of ACB Table, we did not observe any major contention/thrashing issues. In our sensitivity studies, increasing its size from 32 to 256 had negligible effect on performance (since Learning Table acts as a filter for allocation from Critical Table to ACB Table).

C. Run-Time Application

1) Fetching the Taken and Not-Taken Paths: After learning branches that are candidates for ACB, we need to fetch both directions for predicated branches at run-time. Upon fetching every dynamic branch instance whose PC has reached confidence in the ACB Table, we open an ACB Context that records the target of the branch (from the Branch Target Array), and the reconvergence point (from the ACB Table). If the branch is Type-1 or Type-2, we override the branch predictor decision to first fetch the Not-Taken direction. If it is Type-3, we fetch the Taken direction first. If the convergence was Type-1, then we will naturally reach the PC for the point of convergence. For convergences of Type-2 and Type-3, we wait for fetching the Jumper branch which is predicted taken and whose target is our expected reconvergence point. One should note that this Jumper is allowed to be a different branch than what was seen during training. Having found the Jumper which will take us to the point of reconvergence, we now override the target of this Jumper branch to be either ACB-branch target (when first fetched direction is Not-Taken) or next PC after the ACB-branch (when first fetched direction is Taken). This step is needed to fetch the other path. Once the convergence PC is reached, present ACB Context is closed and we wait for another ACB-branch instance. The ACB-branch, Jumper branch, Reconvergence point and ACB-body instructions are all attached with a 3-bit identifier for OOO to identify and associate every predicated region with the corresponding ACB-branch.

Occasionally, reconvergence point on either path may not be reached. In such cases the front-end only waits for a certain threshold (in terms of fetched instructions) beyond the allowed convergence distance after the ACB-branch; if convergence is not detected by then, we set the same 3-bit identifier to indicate divergence for this instance. When the OOO receives this signal, it forces a pipeline flush at the ACB-branch after it resolves itself. It continues fetching from the correct target normally thereafter. We also reset the confidence and the utility bits in the ACB Table to make it re-train. Since we train for convergence as well, divergence injected pipeline flushes are rare and do not hurt performance.

2) Effective Predication in the OOO: OOO uses the ACB identifiers set during fetch to handle the predicated region. ACB-branch is stalled at scheduling for dispatch until ei-
ther the reconvergence-point or the divergence-identifier is received. This stalling of ACB-branch is needed since a failure in convergence implies incorrect fetching by ACB. To recover, we force a pipeline flush on diverging ACB-branch instances once their direction is known upon execution.

All instructions in the body of the ACB-branch are forced to add the ACB-branch as a source, effectively stalling them from execution until the ACB-branch has executed. Instructions post the reconvergence point are free to execute. If they have true data dependencies with any portion predicated by the ACB-branch, they will be naturally stalled by the OOO. Once ACB-branch executes, instructions on the predicated-true path execute normally. However, since predicated-false path was also allocated and OOO may have already added dependencies for predicated-true path with predicated-false path, we need to ensure Register Transparency beyond predicated-false path.

To achieve this aim, every instruction in the body of ACB that is a producer of some logical register or flags, also tracks the physical register corresponding to its local destination. For example, an instruction of the type \texttt{mov RAX, RBX} will be tracking \texttt{RAX} (i.e. its destination) in the OOO. After ACB-branch resolution, if an ACB-body instruction is identified as belonging to the predicated-true path, we will execute it normally as a move from \texttt{RBX} to \texttt{RAX}. If it instead turns out as a predicated-false path instruction, then we will ignore the original operation and it will act as a special move from \texttt{RAX} to \texttt{RAX}: it copies the last correctly produced value of \texttt{RAX} to the register allocated to it for writing \texttt{RAX}. Since RAT provides us with the last writer to a given logical register during OOO allocation, we obtain the last written physical register ID from the RAT during register renaming. Hence, the predicated-false path is able to propagate the correct data for the live-outs it produces, making it effectively transparent. Any instruction on the predicated-false path, that does not produce register or flags (like stores or branches), instantly releases its resources.

Prior works [7], [11] have relied on select-micro-op based approaches to handle correctness of data dependencies after the predicated region. While using select-micro-ops also allows the execution of the predicated region before the reconvergence point (unlike ACB which stalls it until ACB-branch resolution), it requires complex RAT fork-and-merge on every predicated instance. This also causes frequent loss of performance-critical allocation bandwidth, which becomes more significant in future wider processors. ACB’s design choices included the relatively simpler logical-destination tracking approach. Using these less intrusive micro-architectural changes, we are able to achieve register transparency without resorting to complex RAT recovery mechanisms or re-execution as proposed in [7], [11].

3) Predicated-False Path Loads/Stores: All ACB body loads and stores are stalled in the OOO-IQ until ACB resolves its direction. Memory disambiguation logic [20] stalls on stores since their addresses are not computed yet. When the branch resolves, these are dispatched from IQ with predicated-true/false path information. Predicated-false path loads/stores are invalidated in Load-Store Queue (LSQ) and are excluded from matching addresses with younger loads. These invalidated loads/stores deallocate (upon retirement) without dispatching to caches/memory. Predicated-true path loads/stores participate in store-load forwarding within the LSQ and are dispatched normally.

4) Run-Time Throttling using Dynamo: Like other predication strategies, ACB can have undesirable and dynamically varying side-effects on performance as analyzed in Section II-C. Hence, ACB requires run-time monitoring and throttling to optimize for performance and prevent inversions. However, performance can be affected by various diverse phenomena which, by tracking limited local heuristics, cannot be accurately evaluated. In fact, this is a generic problem that affects many other features which involve balancing cost-benefit trade off to maximize performance.

We propose a novel dynamic monitoring (Dynamo) algorithm that monitors the run-time performance delivered by ACB. Dynamo is a first of its kind predictor that tracks actual performance and compares it with baseline performance. Figure 5 describes the various elements of Dynamo and their interactions. Dynamo assumes a 3-bit FSM-state for each entry in the ACB Table, with the possible states being \textit{NEUTRAL}, \textit{GOOD}, \textit{LIKELY-GOOD}, \textit{LIKELY-BAD} and \textit{BAD}. FSM-state transitions happen for all entries together at every \textit{W} retired instructions, which we call as one epoch. Entries reaching the final states (\textit{GOOD} or \textit{BAD}) do not undergo further transitions. Choosing a very small epoch-length will be highly susceptible to noisy IPC changes, whereas a very large observation window will not correctly evaluate the performance impact since major program phase change falling in this window might affect the overall IPC dominantly. Through experimental analysis, we found epoch-length of 8K to 32K instructions as optimal (16K chosen for best performance).

Dynamo computes the cycles taken to complete a given epoch using an 18 bit saturating counter. Allocation in the ACB Table initializes each entry with \textit{NEUTRAL} state. For the odd-numbered epoch, Dynamo disables ACB for all the branches except those in \textit{NEUTRAL} state. For the even-numbered epoch, Dynamo enables ACB for all the branches except those in \textit{BAD} state. At the end of every odd-even pair of epochs, Dynamo checks the difference in cycles between the two.
If the cycles have increased due to enabling ACB beyond a thresholded factor, then it means that doing ACB for this set of unconfirmed branches is likely bad and Dynamo transitions the state of all the involved ACB-branches towards BAD. On the other hand, if the cycles have improved due to ACB, then Dynamo moves the state of all the involved ACB-branches towards GOOD. We found this cycle-change factor to be optimum at 1/8. Intuitively, a high threshold will be insensitive to subtle performance degradation by ACB whereas a low threshold will be susceptible to minor IPC changes because of changing program execution behavior.

To identify the ACB-branches responsible for affecting IPC in a given epoch, Dynamo also counts the per-instance activity of each ACB-branch in a 4 bit saturating Involvement Counter, which is incremented on every predicated dynamic instance. State transitions of activated ACBs are allowed only if their involvement counter is saturated. This prevents Dynamo from associating unrelated IPC fluctuations (or natural program phase changes) to its judgment of any activated ACB. To make it even more robust, Dynamo does not directly transition any branch to the final (GOOD or BAD) states. Instead it relies on observing positive or negative impacts of the branch consecutively to obtain a final decision regarding GOOD or BAD. Branches in GOOD state will perform ACB while those in BAD state are disabled henceforth. If the cycle-change factor is within allowed thresholds, then we do not update states in either direction and continue with the next epoch-pair.

It must be noted that multiple ACBs may be learned and simultaneously start getting applied in a given epoch. Dynamo evaluates IPC changes with and without all the actively working ACBs together since they eventually will be working alongside each other. Also, since program phase changes can potentially change the criticality of some branches, we wanted to give a fair chance to the blocked candidates to re-learn through Dynamo. So, we reset Dynamo state information for all entries periodically (~10 million retired instructions).

D. Storage Requirement

Table I enlists all the tabular structures used by ACB. Aggregate storage required by ACB is just 386 bytes.

IV. SIMULATION METHODOLOGY

We simulate an Out-of-Order x86-ISA core on a cycle-accurate simulator that accurately models the wrong path on branch mispredictions. Simulated core runs at 3.2 GHz and micro-architecture parameters are similar to Intel Skylake [1] configuration. Detailed parameters enlisted in Table II.

We experimented with 70 diverse, single-threaded workloads from different categories (details in Table III). The performance is measured in instructions-per-cycle (IPC).

V. RESULTS

We first present the performance improvement by ACB on our workloads in Section V-A. We then evaluate the effectiveness of Dynamo as a throttling scheme in Section V-B. In Section V-C, we contrast ACB with state-of-the-art dynamic prediction approach. We evaluate ACB’s performance on future OOO processors in Section V-D. Finally, we perform a qualitative analysis of ACB’s effects on power in Section V-E.

A. Performance Summary of ACB

Figure 6 summarizes the performance benefits of applying ACB. ACB gives an overall performance gain of 8.0% (geometric-mean) while providing an effective reduction in branch mis-speculations by 22% on average. Figure 7 shows a line graph correlating the performance improvement with reduction in pipeline flushes for all our studied workloads. We see that mis-speculation reduction correlates positively with the observed performance gains. The largest positive outlier (lammps) provides more than 2X speedup. Due to Dynamo’s intervention, losses are contained within -5%. An interesting observation comes from the analysis of outliers like soplex (on the left-end of Figure 7), where despite significant reduction in total mis-speculations, the performance gains are unexpectedly low. Here, the counted branch mispredictions are not on the critical path of execution in the baseline itself. As seen in Section II-A, such mispredictions are not important for perfor-
Benchmarks          Category
perlbench, bzip2, gcc, mcf, gobmk, hmmer, sjeng, libquantum, b264ref, onnetpp, astar, xalanckm
bwaves, games, mile, zeusmp soplex, povray, cal culix, gemsfld, tonio, ibm, wif, sphinx3 gromacs, cactusADM, lesle3D, namd, deal
cale cbsnn, ibm, camk, pop2, imagick, nab, roms, perlbench, gcc, mcf, onnetpp, xalanckm, x264, deepsjeng, leela, exchange, tz
winzip, photoshop, sketchup, premiere  SY Smark [23]
tablemark [24], geekbench [25], compression, 3dmark [26], embc [27], chrome
lammps [28], parsec [29]

| TABLE III |
| ALL 70 WORKLOADS USED IN OUR EXPERIMENTS. |

Another side-effect of ACB is noticeable in the largest negative outlier (onnetpp), where the mis-speculations slightly increase after applying ACB. This relates to Section II-C2 as ACB overrides the branch predictor decision consistently (to fetch both paths), causing the branch history to get modified. This starts affecting the BPU’s predictability for some other branches due to correlation effects. These outliers represent those scenarios where the newly manifested mispredictions cannot be helped by ACB due to its selective coverage.

B. Analysis of Dynamo

Figure 8 compares ACB’s performance with and without Dynamo for all workloads. Dynamo brings up native ACB’s performance from 6.7% to 8.0%. Without Dynamo, the largest negative outliers (emmbc and SPEC-h264) suffer nearly 20% performance loss, strongly exhibiting the negative impacts of non-judicious predication. Dynamo helps throttle out harmful ACB-able PCs in such cases helping recover performance.

Prior to Dynamo, we also experimented with execution stalls (i.e. waiting for dispatch at issue queue) counting based simpler metric, since predication primarily creates additional data-dependencies. But in few cases, we observed that despite high stall counts, performing predication was favorable as saved pipeline flushes outweighed the additional stalls incurred. This was also vulnerable to bad tuning. Dynamo was designed to holistically evaluate this trade-off for ACB.

C. Comparison with Prior Compiler-based Solutions

In this section, we compare against Diverge-Merge Processor (DMP) [7], which relies on changes to the compiler, ISA and micro-architecture to perform selective predication on low confidence branch predictions. We modeled the enhanced DMP [15], which improved upon the DMP solution through profile-assisted compiler techniques.

Figure 8 compares the performance of ACB (both with and without Dynamo) and DMP. ACB and DMP both produce impressive positive outliers (category A). Workloads marked as B1 and B2 are the cases of DMP outperforming ACB. The category B1 benefits from DMP’s multiple reconvergence point support by compiler assisted convergence detection. ACB can be enhanced to support the same by actively learning and allocating multiple reconvergence points in ACB Table. For category B2, ACB’s approach of stalling both the paths reduces its performance gains compared to DMP which eagerly executes the predicated region before the branch resolves itself. DMP achieves this with the help of select-micro-ops.
based micro-architecture. We experimented with adding selecto
tive-op support to ACB which improves ACB’s performance
gains by only about 0.2%. Since Dynamo already throttles
negative outliers, this scheme only helps the positive gain-
ers slightly. This trade-off justifies ACB’s logical-destination
tracking approach to save hardware complexity (RAT and fetch
forking).

Fig. 9. DMP and Oracle DMP (DMP-PBH) for Categories D and E.

Workloads marked as C in Figure 8 suffer from negative
performance impact for both DMP and ACB without Dynamo.
This clearly highlights the utility of dynamic performance
monitoring. In these workloads, both ACB and DMP qualify
similar set of branches to be predicated. Despite ACB’s
stricter qualification constraints, these branches incur more
costs by creating data-dependency based stalls in the OOO (as
explained in Section II-C3). With Dynamo we are able to iden-
tify and block such delinquent candidates. While enhanced-
DMP also has a detailed cost-benefit analysis through static
compiler-profiling, the work itself acknowledges its limitation
in being able to account for only fetch related costs and
not execution related costs [15]. Compiler-based techniques
are also susceptible to inefficiencies arising from differences
between input sets used for profiling and actual execution.

Figure 9 focuses only on Category D and E workloads,
showing a correlation between mis-speculation ratio and per-
formance over baseline. There is a significant increase in
branch mispredictions by applying DMP. This may seem
surprising at first, but as we had reasoned in Section II,
predication (DMP or ACB) changes the branch history that
is being learned by the branch predictor. It is well known
that speculative update of the branch history is very important
for branch predictor accuracy [30]. In the baseline, branch
history is always speculatively updated, assuming the previous
branch predictions were correct. When a branch mispredicts,
a pipeline flush happens and subsequent branches that are
re-fetched use the updated branch history. Hence, branch
history is always up-to-date for all valid predictions (except
on the wrong path, that is eventually flushed out). However,
when a branch is dynamically predicated, subsequently fetched
branches do not have any knowledge of the predicated branch’s
direction, since no real prediction happened for it. They will
know the direction only upon branch resolution of DMP-
ed branch in OOO. By that time branch predictions have
already happened and front-end has moved ahead, making it
impossible to correct predictions for these branches.

In case of ACB, we remove all ACB predicated instances
from the branch history, so the branch predictor adapts itself to
predict without the knowledge of the ACB-ed branch. However
in DMP, based on the branch confidence some instances are
predicated and others are not. This effectively means that many
more possible branch histories are possible, including wrong
branch history. Recall that the TAGE branch predictor [2]
allocates a higher branch history prediction table on every
misprediction, and the presence of unstable branch histories
results in severe thrashing of the tables. This badly effects
the baseline accuracy of the predictor, not just for the target
branch but also other branches. Also as described in Section II,
many branches are perfectly correlated with older branches,
and if the older branch is removed from the branch history by
predication, they lose their accuracy.

DMP uses compile time profiling to pick the target H2P
branches. Unfortunately the application of DMP at runtime
changes the branch predictor behavior in some applications,
rendering the compile time profiling sub-optimal and causing
performance inversions in category D and E. To validate this
hypothesis, we ran category D and E workloads with an oracle
update of branch history and compared it to DMP and ACB
performance in Figure 9. As is evident from Figure 9, DMP-
PBH (oracle with perfect branch history), recovers most of
the losses for category D, and reduces mispredictions over
baseline. A similar observation was made by Klauser et al. [11]
for branch history update and dynamic predication.

Fig. 10. Allocation stalls comparison for Category E workloads.

Interestingly, Category E workloads are still not optimal
even with the perfect branch history. Figure 10 correlates their
performance with increase in allocation stalls of the OOO pro-
cessor. Even though these workloads reduce mispredictions, in
presence of perfect branch history, they suffer from allocation
stalls because of data dependencies with select-uops beyond
the reconvergence point. A throttling mechanism like Dynamo
is needed for such cases.
Comparison against DHP: Unlike DMP, DHP [11] performs predication only on simple and short hammocks, targeting minimal cost of fetching the additional path as compared to speculation. Limited by its simplicity of application, DHP cannot cover complex, non-traditional control flows which lead to convergence. On average, ACB delivers (8.0%) nearly double the performance of DHP (4.3%). Figure 11 illustrates this performance difference on a per-workload basis, clearly highlighting the impact of difference in targeted coverage.

Fig. 11. Comparison of ACB against DHP. DHP has lower coverage and hence many workloads do not show sensitivity to it.

D. Effect of Core Scaling

Simulations on a scaled-up version of the present configuration (8-wide with twice the execution/fetch resources) showed that performance of ACB improves to 8.6% owing to a wider and deeper processor amplifying the problem of inefficiency due to mispredictions. This also highlights ACB’s improved efficiency and robustness in tackling branch mis-speculations.

E. Qualitative Power Analysis

ACB reduces pipeline flushes by 22% leading to reduction in the number of speculative OOO allocations. While ACB also allocates additional instructions in OOO for the wrong fetched path, our analysis reveals that ACB effectively reduces the total number of OOO allocations by 5%, which naturally translates to reduction in energy consumption.

Since tabular structures used by ACB are small and are looked up only for branches, the front-end power increment is insignificant. Additionally, one must note that mispredictions cost power not just through pipeline flushes, but also through re-execution of already executed (and correct) control-independent instructions. Each eliminated misprediction contributes to energy savings by preventing this wastage of work.

VI. RELATED WORK

Software predication has been studied extensively in the past [10], [31], [32]. Popular ISAs support static predication [17], [18] but due to large overheads, the realistic benefits are diminished [7], [12]. Wish Branches [12] rely on the compiler to supply predicated code but applies predication dynamically only on less predictable instances. Dynamic Hammock Predication [11] targets only small, simple hammocks. Hyperblock predication [33] uses compiler profiling to predicate frequently occurring basic blocks. Generalized multi-path execution was proposed in [34]-[36]. Diverge-Merge Processor (DMP) [7], [15] uses branch prediction confidence to selectively predicate conditional branches, while using the compiler for convergence and branch selection information. DMP outperformed previous schemes and was the focus of our comparison. Joao et al. [37] extended dynamic predication to indirect branches. Stephenson et al. [38] proposed another compiler based approach to simplify prior hardware complexity needed for enforcing correct dependence flow in predication. Their targeted hammocks are restricted by having specific register writing patterns in the predicated region, which are provided by the compiler. As examined in Section II-C and comparatively analyzed in Section V-C, prior works do not fully comprehend the delicate performance trade-offs created by disabling speculation causing performance inversions in certain scenarios. Additionally, they need significant changes to hardware, compiler and ISA, making their implementation challenging. In contrast, ACB is a pure hardware solution.

Several mechanisms exploiting control independence [39], [40] also exist which perform selective flush on a branch mis-speculation wherein only the control dependent instructions are flushed and re-executed. In contrast to ACB, these techniques require complex hardware to remove, re-fetch and re-allocate the selectively flushed instruction, along with complicated methods to correct data dependencies post pipeline flush. Skipper [41] proposed out-of-order fetch-and-execute of instructions post-control flow convergence to exploit control independence but required large area (about 6KB) for supporting its learning and application. SYRANT [42] simplified this approach by targeting only converging conditional branches and smarter reservation of OOO resources. However, it is limited in application only to consistently behaving branches. Control Flow Decoupling (CFD) [8] is a branch pre-computation based solution which modifies the targeted branches by separating the control-dependent and control-independent branch body using the compiler. Hardware then does an early resolution of the control flow removing the need for branch prediction. Store-Load-Branch (SLB) Predictor [43] is an adjunct branch predictor which improves accuracy by targeting data-dependent branches whose associated loads are memory-dependent upon stores. It detects dependency between stores, loads and branches using compiler and modifies hardware to override branch prediction with available pre-computed outcomes. ACB is applicable on top of any baseline branch predictor, including SLB.

Rotenberg et al. [44] proposed a hardware to detect only forward convergence scenarios. Collins et al. [45] proposed detecting any type of reconvergence. Their mechanism identifies the common patterns of convergence and adds dedicated hardware to the backend to simultaneously learn the different reconvergence points of different branches, all at once, by broadcasting the PCs of instructions being retired. As a result it
requires significant area (nearly 4KB) and much more complex implementation. In contrast, ACB is extremely light-weight with the overall mechanism needing just 386 bytes, including the reconvergence detection hardware.

VII. SUMMARY

In this paper, we have presented ACB, a lightweight mechanism and completely implementable in hardware, that intelligently disables speculation by dynamic predication of only selective critical branches, thereby mitigating some of the costly pipeline flushes because of wrong speculation. ACB uses a combination of program criticality directed selection of hard-to-predict branches and a runtime monitoring of performance to overcome the undesirable side-effects of disabling speculation. Micro-architecture solutions invented for ACB, like convergence detection and dynamic performance monitor, can have far reaching effects on future micro-architecture research. Our results on a diverse set of workloads shows that ACB is a power-and-performance feature that delivers 8% average performance gain while reducing power consumption. ACB also scales seamlessly to future out-of-order processors and continues to deliver high performance at lower power.

REFERENCES


