Alternate Path Fetch

Aniket Deshmukh
University of Texas at Austin
United States
a.deshmuhk@utexas.edu

Lingzhe(Chester) Cai
University of Texas at Austin
United States
chestercai@utexas.edu

Yale N. Patt
University of Texas at Austin
United States
patt@ece.utexas.edu

***Abstract*—Modern out-of-order cores rely on a large instruction supply from the processor frontend to achieve high performance. This requires building wider pipelines with more accurate branch predictors. However, scaling the pipeline width is becoming more challenging due to limitations on the number of instructions that can be renamed and branches that can be predicted in a single cycle. Moreover, mispredictions reduce the useful fetch bandwidth that can be extracted from a wider frontend.**

Our work, Alternate Path Fetch (APF), effectively uses a wide frontend by dividing the pipeline into two parallel sections. One processes regular instructions, and the other uses a separate pipeline to Branch Predict, Fetch, Decode, and partially Rename instructions on the alternate path of hard-to-predict (H2P) branches. The pipelines operate simultaneously using a Parallel-Fetch scheme we developed. This allows APF to more efficiently utilize the bandwidth of a wider frontend without the overhead associated with building a monolithic, wider pipeline.

APF improves performance by reducing the pipeline re-fill delay on branch mispredictions. Unlike other solutions that fully rename and execute instructions on both sides of a branch, we show that stopping after partial Renaming on the alternate path provides better performance through improved coverage and avoids the complexity associated with further processing. APF provides a 5% geomean speedup over an aggressive 8-wide out-of-order core.

***Index Terms*—Branch Prediction, Predication**

I. Introduction

Modern out-of-order (OoO) cores achieve high levels of single-thread performance by speculating on large instruction windows and building deeper pipelines. The execution backend of these cores needs to be driven by a large instruction supply. This requires wider pipelines supported by accurate branch prediction. OoO pipelines have grown wider over the years: many modern designs have an 8-wide frontend [2], [3], [5]. However, further scaling is challenging. To build a wider pipeline, the width of all frontend stages (Branch Prediction, Fetch, Decode, and Rename) needs to be increased. The Fetch and Decode logic is mostly parallel and relatively easier to scale. Increasing Branch Prediction and Rename bandwidth, however, poses major limitations [29]. Moreover, branch mispredictions are still a problem, even with modern Branch Predictors such as TAGE-SC-L [37] and Hashed Perceptron [21], as frequent branch mispredictions from a few hard-to-predict (H2P) branches [46] reduce the number of useful instructions fetched by a wider frontend. We posit that the **additional fetch bandwidth obtained from a wider frontend is better utilized for fetching the alternate path of H2P branches**.

Building a wider Fetch stage requires reading more bytes from the I-Cache every cycle which can be easily done by banking the I-Cache such that consecutive cache lines are stored in different banks. The Decode width can be increased by adding more parallel decoders[1]. These changes require additional logic but are unlikely to impact the critical path and increase pipeline latency.

The Branch Predictor (BP) and Branch Target Buffer (BTB) limit frontend bandwidth as current branch prediction algorithms are sequential. To predict a branch, the Program Counter (PC) and branch history are used to access the BP and BTB tables, which in turn update these registers. Accesses for a subsequent prediction cannot begin until the previous prediction finishes, which takes at least one cycle. Most current predictor designs can predict up to one **taken** branch per cycle. In many applications, this severely limits the effect of a wider pipeline due to the density of taken branches. While there has been some prior work on predicting multiple taken branches per cycle [30], [38], [45], these are either expensive and hard to implement on current predictors or only work in specific scenarios.

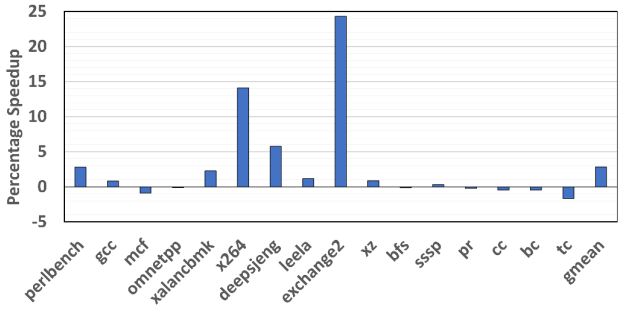

Fig. 1. Performance of a 16-wide OoO core with one extra cycle on Rename relative to an 8-wide baseline

In the Rename stage, each instruction within a fetch packet (instructions fetched in the same cycle) needs to check whether it depends on any older instructions within that fetch packet before accessing the Register Alias Table (RAT). Increasing

[1]In x86 machines, length decoding is sequential; going wider requires more logic than just duplicating existing decoders and increases the pipeline latency. However, using pre-decode bits or a uop cache allow x86 machines to provide higher decode bandwidth without additional latency.

979-8-3503-2658-1/24/$31.00 ©2024 IEEE
DOI 10.1109/ISCA59077.2024.00091

the Rename width directly impacts the critical path of the RAT read and dependency checking logic [29] and thus increases the number of pipeline stages in Rename [35]. Additional frontend latency reduces performance as it takes longer to refill a deeper pipeline after a branch misprediction. Even adding one additional cycle of Rename latency hurts performance for applications sensitive to branch mispredictions. Fig.1 shows the performance of a 16-wide OoO core[2] relative to an 8-wide baseline on SPEC2017 Integer and GAP benchmarks. Workloads with a high taken branch density show very little benefit. Benchmarks with high branch misses-per-kilo-instructions (MPKI) (Fig.2) drop in performance due to the increased misprediction latency.

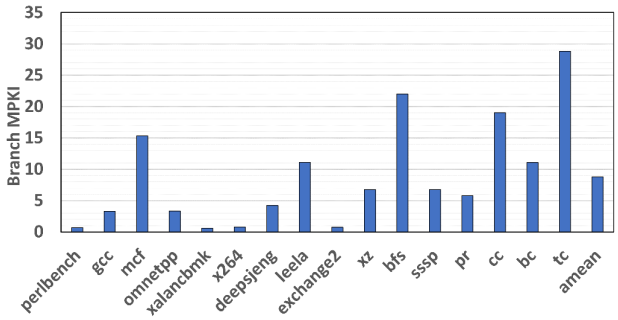

Fig. 2. Conditional Branch Mispredictions (8-wide core)

Our work, Alternate Path Fetch, aims to fetch and process instructions on both paths: the predicted path as specified by the branch predictor, and the alternate path which inverts its prediction, after H2P conditional branches. When an H2P branch resolves, if the prediction is incorrect, the previously fetched alternate path instructions can be preemptively inserted into the pipeline. This reduces the **pipeline re-fill delay**: the time it takes for correct path instructions to enter the backend after a branch misprediction is resolved, and in turn improves performance.

Alternate Path Fetch (APF) processes instructions on a separate pipeline until RAT access and is supported by a low-overhead parallel-fetch mechanism for the two paths. This breaks the 16-wide frontend into two separate pipelines and avoids the Renaming and predictor limitations associated with a normal 16-wide core. It does not add any additional Allocation width, functional units, or Retire width, but still provides significantly better performance. Our contributions are summarized as follows:

- We propose Alternate Path Fetch (APF), a technique that effectively utilizes the fetch bandwidth of a wider frontend.
- To support multiple taken predictions every cycle, we introduce a scheme that breaks the TAGE-SC-L predictor into several smaller predictors. Together with banking the I-Cache and BTB, this enables APF to predict branches and fetch instructions for both paths in parallel without adding additional ports to these structures.
- We analyze the trade-offs associated with the depth of the alternate path pipeline and show that processing alternate path instructions up until RAT access across multiple H2P branches both provides better performance and reduces hardware overhead and design complexity.
- When selecting an H2P branch for APF, we show that an oldest-first scheme with priority given to low TAGE confidence branches works well as it reduces wasted APF cycles and provides good misprediction coverage.

[2]The Allocation width, number of functional units, and Retire width are also scaled for the 16-wide OoO core to match the frontend.

II. Prior Work

The idea of fetching and executing instructions on both paths of a branch was first introduced by Eager Execution. Doing this for all branches has an exponential overhead, so later research looked at several variants to restrict the exponential branching [42] and limited Eager Execution to low confidence branches [26], [43], [44]. However, these approaches require a significant amount of hardware support to execute multiple alternate paths in parallel.

Predication techniques also fetch and execute both paths after a branch but only if the control flows after that branch converge within a few instructions. Compiler-only predication techniques [10], [16] take advantage of if-else statements with short bodies, commonly referred to as hammocks. They execute instructions on both paths and selectively commit the results on one path once the branch predicate is known. Wish Branches [23] dynamically pick between executing predicated code and performing branch prediction based on whether the prediction is accurate. Dynamic Predication techniques [17], [22], [25] find branches that can be predicated at runtime and do not require compiler support.

Predication-based techniques have a low misprediction coverage as they only target certain types of branches. The state-of-the-art, Auto-Predication of Critical Branches (ACB), relies on detecting specific code layouts to find the merge point for an H2P branch. ACB increases backend pressure as instructions from both paths compete for execution resources. In contrast, APF works for any mispredicted conditional branch and can target branches that are not amenable to predication. It does not have the complexity associated with renaming both paths and selectively discarding wrong path instructions. APF only executes predicted path instructions and does not increase backend pressure

The concept of not executing alternate path instructions was first introduced by Dual Path Instruction Processing (DPIP) [11]. DPIP performs Renaming and Allocation for alternate path instructions which has large hardware costs and limits which branches it can target. Moreover, it shares frontend cycles with the predicted path which is detrimental to performance with modern high-accuracy branch predictors as it takes useful fetch cycles away from the predicted path (which is often correct). In fact, most prior work on processing two paths reduce the effective fetch bandwidth available to the program as they cannot fetch alternate path instructions in parallel with the predicted path, which APF can do. Overall, APF does better both in terms of performance and hardware

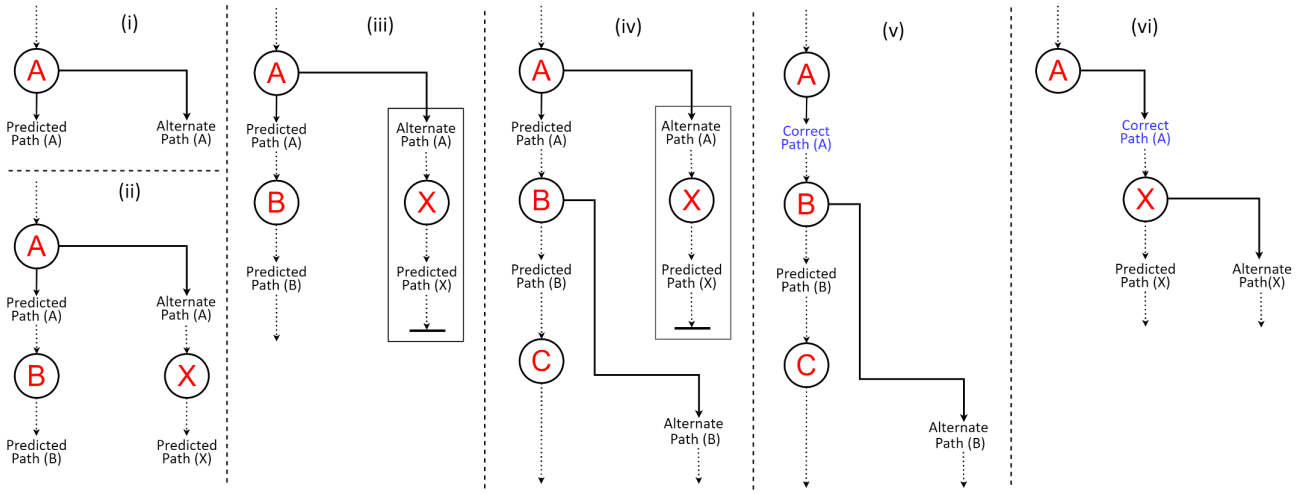

Fig. 3. Example showing how Alternate Path Fetch works. Each circular node represents an H2P branch. Branches are fetched in top-to-bottom order.

overhead. A detailed comparison against DPIP is provided in Section IV.

There is a clear tradeoff between coverage and per-branch saving associated with processing alternate path instructions. Eager execution and predication execute alternate path instructions and have high per-branch savings. DPIP stops short of execution. APF on the other hand, processes instruction only up until RAT access. Because it does not incur the overhead associated with Renaming, Allocating, or Executing alternate path instructions, **APF can target any conditional branch misprediction** and has comparatively higher coverage.

Misprediction Recovery Caches [15] record the correct path instructions seen after a mispredicted branch and replay them to reduce the latency of fetching and decoding the right path instructions. [19] similarly uses a Misprediction Recovery Buffer to replay prediction packets on recovery. Just-in-time hedge fetching [36] fetches the first few wrong path instructions into a buffer to avoid the I-cache access latency after a misprediction recovery. These solutions can only fetch a few instructions until the first branch as they cannot predict branches on the alternate path. **Our banked BP design allows us to predict alternate path branches in parallel with the predicted path.** Thus, we can process significantly more instructions on the alternate path allowing APF to save more cycles of re-fill penalty.

Elastic Fetch [31] dynamically converts a decoupled predictor to a coupled one, eliminating the cycle(s) required for prediction and accessing the prediction queue (can only save 1-2 cycles). Uop caches [40] reduce the baseline decode latency by storing decoded uops, which decreases the impact of mispredictions. However, APF can be employed on top of processors with these mechanisms and still provides substantial performance benefits (Section VI-G).

APF works in tandem with any solution that targets the full branch misprediction penalty for specific types of branches rather than competing with them. If a different solution reliably covers the branches that APF improves, APF can instead focus on other branch mispredictions and still provide additive benefits.

III. Alternate Path Fetch

Alternate Path Fetch uses a separate frontend, the APF pipeline, to process instructions on the alternate path for conditional H2P branches. We examined the branch resolution delays for various commercial products [1] [7] and selected an aggressive baseline configuration that has 15 pipeline stages in the frontend (Branch Prediction to Rename). The APF pipeline is 13 stages deep as we only process alternate path instructions up until RAT access. Alternate path instructions are thus fetched and processed for a maximum of 13 cycles. APF comprises the following steps:

- Find in-flight H2P branches.
- Fetch the alternate path for unresolved H2P branches.
- Save the fetched alternate path instructions to a buffer.
- Repeat for other unresolved H2P branches.
- If an H2P branch mispredicts, fill the main pipeline with the saved instructions for that branch and resume fetching from the last alternate path instruction.

This process is depicted with an example in Fig.3. The program in the example consists of several H2P branches depicted as nodes in a control flow graph and starts at the top-left (Fig.3-(i)). When H2P branch **A** is encountered, the APF pipeline begins fetching down the alternate path of branch **A** alongside its predicted path (using a shadow PC and branch history register). As instructions are fetched, subsequent H2P branches may be seen on either path. In Fig.3-(ii), H2P branch **B** is seen on the predicted path of **A**. Because the APF pipeline is already fetching the alternate path for branch **A**, it cannot begin APF for branch **B**. Instead, the PC and branch history at branch **B** are saved. OoO cores already keep track of the PC and history for all branches in the in-flight branch queue so only an additional bit needs to be added to this queue to indicate branch **B** was marked as H2P. Once branch **B** is predicted, the main pipeline continues down its predicted path.

In parallel, the APF pipeline continues fetching down the alternate path of branch **A** and eventually reaches the H2P branch **X** (Fig. 3-(ii)). Since the APF pipeline is busy, branch **X** is marked as H2P in a shadow in-flight branch queue (which saves the PC and history of branches on the alternate path). Branch **X** is then predicted and the APF pipeline continues fetching down the alternate path of **A**, which is now the predicted path of **X**. Note that both branches **B** and **X** can be predicted in the same cycle using the Parallel-Fetch mechanism described in Section V-B.

This process continues until the APF pipeline is full (Fig.3-(iii)). At this point, the alternate path instructions, the alternate path PC and branch history at that point, and the shadow in-flight branch queue are all copied over to an Alternate Path Buffer. This frees up the APF pipeline so it can fetch down the alternate path for the next H2P branch on the predicted path, which is branch **B** (Fig. 3-(iv)). The Alternate Path Buffers allow APF to fetch the alternate paths for multiple unresolved H2P branches simultaneously. **Since the APF pipeline does not access the RAT, only the PC and branch history (which are already saved in the in-flight branch queue) are needed to begin fetching down the alternate path of a subsequent H2P branch.**

Eventually, branch **A** is resolved. At this point, one of two things can happen. If branch **A** was correctly predicted, the alternate path of **A** is incorrect. This path is then flushed from the Alternate Path Buffers and we continue fetching down the predicted path and currently active alternate path (Fig. 3-(v)).

If branch **A** was mispredicted, the alternate path of **A** is in fact the correct path. In this case, all the saved alternate path instructions are restored to the main pipeline registers, and the PC and branch history are set to the saved values. This is done by reading instructions out the buffers cycle by cycle into the main pipeline stage register after the APF pipeline (Section V-G). In parallel, all other instructions (on the predicted and alternate paths) younger than branch **A** are flushed. The final state of the pipeline after the misprediction recovery is now reflected in Fig.3-(vi). After recovery, the state of the frontend is exactly as if the correct path instructions had been fetched and processed on the main pipeline for 13 cycles, except this happens in a single cycle. We now can immediately start fetching instructions on the predicted path of branch **X** as the PC and branch history have been updated to their respective values when they were processing the alternate path of **A**. The contents of the saved shadow in-flight branch queue are also restored to the main pipeline. Since the queue contains information about which branches were found to be H2P on the alternate path, we can also begin fetching down the alternate path for branch **X** in parallel. Note we do not consider branches on the alternate path for APF until the alternate path is found to be correct to avoid exponential branching.

A. APF on Large Footprint Applications

I-cache misses present another frontend bottleneck for applications with a large instruction footprint [12], [13], [41]. **We terminate Alternate Path Fetch for a branch on an I-Cache miss and do not send this miss out to memory.** As long as these misses do not occur alongside branch mispredictions, APF still improves performance on these applications.

Prior work such as Wrong-Path Instruction Prefetching [32] attempts to prefetch the first I-cache line on the alternate path of a branch. APF can be similarly modified to deal with I-cache misses, but has different tradeoffs. For example, APF would need to be active for longer and fetch alternate paths of easy-to-predict branches to achieve larger prefetching distances and get good coverage. Designing APF to work well for I-cache misses requires significant changes and we leave this to future work. Instead, we focus on dealing primarily with branch mispredictions.

B. Depth of the APF pipeline

The depth of the alternate path pipeline affects both how many H2P branches benefit from APF (misprediction coverage) and the pipeline re-fill penalty saved for each of these branches (per-branch savings). A deeper APF pipeline increases the number of re-fill cycles saved per branch as the alternate path instructions are processed for longer. However, spending more time fetching down the alternate path of one H2P branch reduces the number of APF cycles available to other H2P branches. This happens when H2P branches are clustered together. We observed that the improvement in per-branch benefit generally outweighs the decrease in coverage as the APF pipeline depth is increased (Section VI-D). This is because the first few alternate paths are more likely to be correct than later alternate paths. **However, increasing the pipeline depth beyond rename can significantly reduce coverage**, as explained in the next section.

IV. Comparison with DPIP

Dual Path Instruction Processing (DPIP) uses a custom low-confidence predictor that works in tandem with its branch predictor, gshare [28]. DPIP fetches alternate path instructions for low-confidence branches. It has separate PC and branch history registers for the alternate path, but time-shares the frontend structures of the core in a round-robin fashion, i.e., the fetch unit fetches instructions on the predicted path for one cycle and the alternate path for the next cycle. The alternate path instructions perform Branch Prediction, Fetch, Decode, and Rename (with a shadow copy of the RAT) and are Allocated to a duplicated ROB, Load Queue, and Store Queue. This approach limits performance and introduces significant hardware overhead.

A. Processing Alternate Path Instructions beyond Rename

Increasing the depth of the alternate path pipeline to include RAT access is much more expensive compared to prior stages (Fig.4). This is because alternate path instructions that finish Renaming can change the state of the RAT and Free List. The state of both these structures thus needs to be saved for each alternate path. Moreover, starting alternate path fetch at any H2P branch would require saving the state of the RAT and the Free List at all H2P branches on the predicted path and

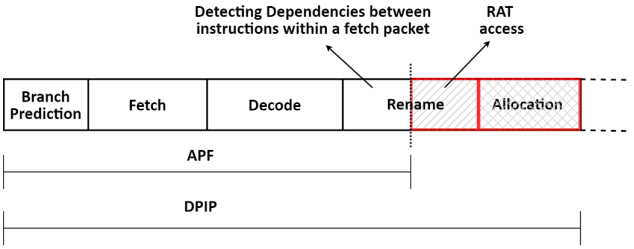

Fig. 4. Depth of the alternate path pipeline

each of the alternate paths. The number of the saved states grows exponentially as H2P branches are encountered. Real hardware can only target a limited number of H2P branches.

Dual Path Instruction Processing (DPIP) has a deeper pipeline for processing alternate path instructions that extends beyond Allocation. Since the alternate path instructions in DPIP perform RAT access, it does not target any intermediate branches encountered while fetching an alternate path to eliminate the need for saving the state of the RAT and Free list for branches on the predicted path. Referring to the example in Fig.3, DPIP fetches down the alternate path of **A** but ignores branch **B** on the predicted path. After fetching down the alternate path of **A**, the next branch it targets is **C**. DPIP processes instructions on the alternate path for 17 cycles and misses out on any intermediate mispredicted branches for this duration which severely limits its coverage. This applies to the alternate path as well: DPIP cannot fetch the alternate path for branch **X** (Fig.3-(vi)) as doing this would require recording the state of the RAT and Free list for branches on the alternate path.

DPIP also performs Allocation into the backend structures for alternate path instructions. To do this, it needs shadow Reservation Stations, a shadow Re-order Buffer, and shadow Load and Store Queues to hold alternate path instructions. Simple buffers cannot emulate the effect of allocation into these structures as copying instructions from a buffer into these structures cannot be done in a few cycles due to write port limitations. Instead, the shadow structures become the ”correct” ones if the corresponding H2P branch is found to be mispredicted. Duplicating the backend is expensive, even for one additional path, and prohibitively so for multiple paths. DPIP can only process one alternate path at a time and needs to wait for the initiating branch to be resolved before it can process another alternate path.

The coverage of DPIP is thus much lower due to the limitations imposed by processing instructions beyond Rename. Even though the per-branch benefit of DPIP is larger, the overall performance of APF is much better. This trade-off is quantitatively explored in Section VI (Fig.9 and Fig.10).

B. *Time-Sharing the Frontend vs. Parallel-Fetch*

An alternative to fetching the predicted and alternate paths in parallel is to time-share the frontend structures across different cycles. DPIP uses this scheme and distributes fetch cycles in a round-robin fashion. However, splitting fetch cycles between two paths, one of which is more likely, yields fewer useful fetch cycles on average. Giving fetch cycles to the alternate path of an H2P branch provides benefits only after several H2P branches are seen on the predicted path. For instance, it takes 7 branches with an accuracy of 80% for the probability of the predicted path being correct to equal the probability of the first alternate path being correct.

This effect is much more prominent with TAGE-SC-L since it has higher accuracy (especially on H2P branches) compared to gshare, which DPIP used. Time-sharing thus leads to lower than baseline performance in some cases (Section VI-E). The Parallel-Fetch scheme provides much better performance for both APF and DPIP compared to time-sharing. However, DPIP falls short of APF even with Parallel-Fetch.

V. Implementation Details

The implementation for Alternate Path Fetch consists of four major components: (a) the H2P Table which identifies hard-to-predict branches, (b) banked BP, BTB, and I-Cache which allow multiple parallel accesses, (c) the APF pipeline stages which process alternate path instructions, and (d) the Alternate Path Buffers which store the state of the APF pipeline. The overall block diagram is shown in Fig. 5.

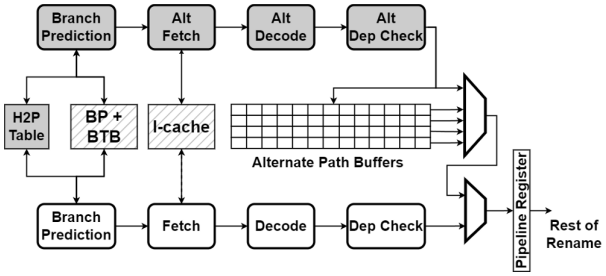

Fig. 5. Overview of Alternate Path Fetch

A. *Baseline OoO Core*

APF is implemented on top of a baseline OoO core with a Decoupled Branch Predictor [34] and a 16-entry Fetch Queue. The predictor can produce up to 1 taken prediction per cycle or up to 32B worth of instructions per cycle, similar to many industry products [2], [3], [5], [6], [19]. The Fetch stage reads predictions from the Fetch Queue and accesses the I-cache. The I-cache is banked to support cross-line accesses. As mentioned earlier, the APF pipeline has 13 stages: 3 for Prediction, 4 for Fetch, 4 for Decode, and 2 for Rename. Each stage has a width of 8 uops.

B. *Parallel-Fetch*

APF needs to fetch instructions from two different points in the instruction stream: the predicted path after a branch and its alternate path. To achieve this, both paths need to have a separate set of PC and Speculative Branch History registers. Each set can then be used to predict branches and fetch instructions in parallel if the BP, BTB, I-TLB, and I-Cache have two read ports. However, these structures contribute to most of the frontend area cost, and adding a read port exponentially increases this. Banking is an alternative that has

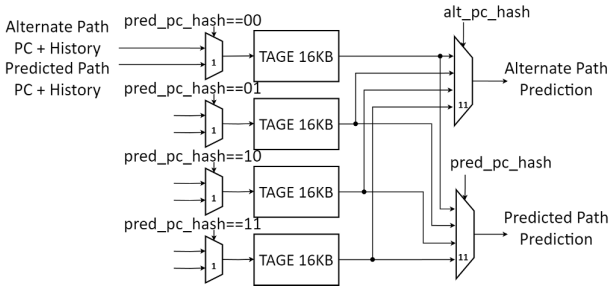

Fig. 6. Banking the Branch Predictor

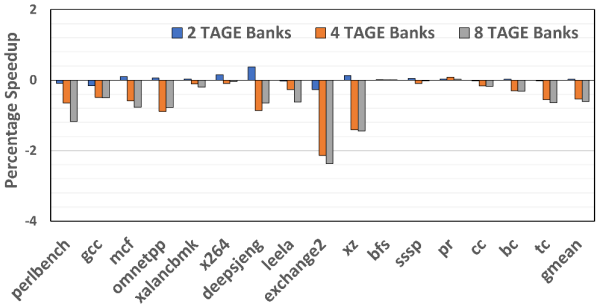

Fig. 7. Effect of TAGE banking on Baseline Performance

a significantly lower overhead and is employed in most caches. It works by dividing the cache into several memory banks, each of which has an access port. A cache line is mapped to a bank using certain bits of its address and performs read and write operations only on that bank. This allows two cache lines to be accessed in parallel if they map to different banks.

*1) **Banking the Branch Predictor**:* Banking the TAGE-SC-L predictor is challenging as it consists of multiple tables, each with different indexing. Instead of splitting up each table, we propose using four 16-KB mini-TAGE-SC-Ls as individual banks in place of the baseline 64-KB TAGE-SC-L. The PC determines which predictor bank a branch goes to for predictions and updates. To support two parallel predictions, the PC and history registers for both paths are sent to all the TAGE banks. If the bank bits for the two PCs are different, two separate predictions can be read out of the predictor banks in parallel (Fig.6). If there is a bank conflict, then priority is given to the predicted path. This allows the BP to achieve a throughput of two taken predictions per cycle when there are no conflicts.

Banking TAGE-SC-L into several smaller predictors can decrease its accuracy. This is because each static branch requires a variable number of predictor entries and branches may be distributed across the four banks such that one particular bank gets overloaded and faces capacity problems. This effect can be reduced if the allocations are equally distributed across the banks. Fig.7 shows the performance of various TAGE banking configurations relative to an un-banked TAGE on a baseline OoO core. Going from no banks to two banks marginally improves performance as forcing certain branches to go to a different bank can reduce aliasing. Moving to 4 or 8 banks hurts branch MPKI due to increased contention. The average MPKI increases by 0.1 in the 4-bank and 8-bank configurations. This reduces the performance of the baseline by 0.5% but is more than made up for the performance improvement of APF.

*2) **Choosing the Bank bits**:* The alternate path and predicted path instructions are often close to each other in instruction memory for conditional branches. Using the lower bits of the PC to determine the bank minimizes bank conflicts. Most stalls due to bank conflicts only last for a single cycle if the lower address bits are used as both paths normally cycle through the same sequence of banks. This is common when banking the I-Cache but less so for the BP. The hashing scheme we used for the BP is shown in Table I. We used the 4-TAGE configuration in all our experiments.

2 TAGE banks	bit0 = $PC[0] \oplus PC[4]$
4 TAGE banks	bit0 = $PC[0] \oplus PC[1] \oplus PC[5] \oplus PC[6]$ bit1 = $PC[2] \oplus PC[3] \oplus PC[4] \oplus PC[7]$
8 TAGE banks	bit0 = $PC[0] \oplus PC[1] \oplus PC[2]$ bit1 = $PC[3] \oplus PC[5] \oplus PC[6]$ bit2 = $PC[4] \oplus PC[7]$
I-Cache and BTB	{bit1, bit0} = $\{PC[7], PC[6]\}$

TABLE I
BANK COMPUTATION

*3) **Banking the I-Cache and BTB**:* The I-Cache is banked using bit 5 and bit 7 of the fetch address. Bit 5 is part of the line offset as we use 64B lines. This splits each I-Cache line into two 32B half-lines. Each group of half-lines goes to a different bank. These groups are then further subdivided into 2 banks based on bit 7, reaching a total of 4 banks. Both paths can read either one 32B chunk or two sequential 32B chunks split across two banks if the fetch packet requires two half-lines. The same bits are used to bank the BTB (which is organized as a region BTB with 64B regions).

The baseline performance is not affected by I-cache or BTB banking. The frontend reads at most a 64B chunk from the I-cache and BTB per cycle (enough for 8 instructions), which is distributed across two consecutive half-lines. Since consecutive half-lines are present on different banks, there is never a bank conflict for I-cache or BTB accesses in the baseline processor.

C. *H2P Table*

The H2P Table is responsible for marking static branches that are good candidates for Alternate Path Fetch. The table draws its insights from the Hard Branch Table proposed in [18], [33]. It is a 2-bank, 8-way set associative structure with 128 entries, indexed with the cache-aligned address of the branch PC. Each entry contains two 3-bit saturating counters that correspond to two H2P branches in that cache line. Each entry also contains two 6-bit fields to indicate the positions of the H2P branch in the cache line. Each bank of the H2P table is accessed once per cycle in parallel with the BTB (for the predicted path and the alternate path), and the results are compared against the BTB results to see if the counters correspond to any branches being predicted during this cycle.

This organization allows the counters for all the branches predicted in a cycle to be read out in a single access. A counter value of 0 indicates that entry is available for allocation and the two counters for each entry are initialized to 0.

An entry in the H2P Table is created for a branch when it mispredicts. Upon allocation, the counter for the branch is set to 1. If both counters have a non-zero value, the allocation request is dropped. If the same branch subsequently mispredicts again, its counter value is incremented. A branch is considered H2P for Alternate Path Fetch if it has an entry in the H2P Table and its counter value is greater than 2. All counters in the H2P Table are decremented by 1 every 20k instructions so that the counter values for branches that mispredict less often than once every 20k instructions tend towards 0. Counters for branches that are not H2P (have a counter value of 0) are prioritized for replacement. The periodic decrement allows the H2P table to mark branches above a certain MPKI threshold (0.2 MPKI here). We experimented with various decrement periods to find the best-performing parameters for APF.

D. Scheduling H2P Branches

*1) **Oldest-first ordering**:* Oldest-first ordering works well when picking unresolved H2P branches for APF because the alternate path for the oldest unresolved H2P branch is more likely to be correct than that of a younger H2P branch. For instance, if the prediction accuracy of an H2P branch is 80%, then its alternate path is correct 20% of the time. But the alternate path of the next H2P branch (assuming it has the same accuracy) is correct only 80%×20%=16% of the time. There are few instances where younger branches tend to have a much lower prediction accuracy than older ones, in which case this may not be true. However, dynamically determining these probabilities to compute the optimal ordering is challenging. Oldest-first is a simple algorithm that is easy to implement and is often the best ordering.

Note that we only perform APF for branches on the main pipeline. It does not make sense to perform APF for branches in the APF pipeline as the probability that this path is correct is even lower (20%×20%=4%). If a branch on the alternate path is H2P, we only mark it as such and consider it for APF if the alternate path is found to be the correct path.

The H2P Table tends to mark many unnecessary branches as H2P to achieve high misprediction coverage. Marking too many branches as H2P can hurt performance if many of them do not cause mispredictions. When this happens, the APF cycles are distributed amongst more H2P branches, decreasing the per-branch benefit without improving coverage as much. The H2P Table parameters need to be tuned to balance this tradeoff. We use two metrics described in past work [20] to measure the effectiveness of the H2P Table:

- Specificity: the percentage of all mispredicted branches classified as H2P.
- Predictive value of a negative test (PVN): the probability that a branch marked as H2P is mispredicted.

Specificity measures the misprediction coverage. A higher value of specificity means more mispredicted branches are marked as H2P. Higher coverage leads to better performance as more mispredictions can potentially be addressed by APF. PVN measures how efficient the H2P table is. A higher value of PVN indicates that fewer branches were marked as H2P even though they are correctly predicted. We use the inverted value of PVN (1-PVN) as this is directly correlated to the amount of wasted work that APF does: the higher this number, the more cycles APF spends on branches that are not mispredictions. Table II shows these metrics for the H2P Table. The H2P Table provides good coverage but ends up doing a lot of wasted work at the same time. While APF is designed for coverage and works well with just the H2P Table, we can get better performance by combining it with TAGE confidence.

	Coverage (Specificity)	Wastage (1-PVN)
H2P Table	95.43%	89.61%
TAGE confidence	56.33%	74.52%

TABLE II
MISPREDICTIONS DETECTED BY H2P TABLE AND TAGE CONFIDENCE

*2) **Using TAGE confidence**:* A useful optimization we discovered was to use TAGE confidence in addition to the H2P Table to decide which branches to pick for APF. TAGE provides three confidence levels based on whether the counter used for prediction is saturated. A prediction from an unsaturated counter is classified as a low-confidence prediction. The TAGE confidence mechanism does a worse job than the H2P Table at covering all mispredictions, as shown in Table II. However, it marks far fewer branches as low confidence leading to less wasted work. The counters used by TAGE confidence are stored in internal TAGE tables that are indexed using the PC and branch history. Thus, it operates on a per-path basis compared to the H2P Table which operates on just a per-PC basis. This allows TAGE confidence to filter out H2P branches that predict correctly only under specific history patterns. Moreover, TAGE counters are only 3 bits wide which allows them to saturate very quickly in one direction (taken or not taken) if a branch behaves consistently under a fixed control flow. TAGE confidence thus captures short-term changes in branch behavior more accurately. On the other hand, the H2P Table counters are decremented once every 20k instructions. This means that branches with a high long-term MPKI saturate the H2P Table counters and it captures static branches that may have bursts of correct predictions but an overall high MPKI. The H2P Table and TAGE confidence thus complement each other as seen from the coverage and wastage numbers in Table-II. The H2P Table can be tuned to provide high coverage and TAGE confidence can be used to preferentially select low confidence branches for APF. APF with only the H2P table provides a 3.3% performance improvement, but this number rises to 5% when TAGE confidence is used in conjunction.

E. APF Pipeline

The overall operation of the APF pipeline proceeds as follows. During Branch Prediction, the branch PC is sent to the H2P Table. The in-flight branch queue, which keeps track of the information about all branches currently in the pipeline is augmented with two additional bits: one to indicate whether

it is H2P (which is set by the H2P Table) and one to indicate whether the TAGE confidence for this branch was low. A 3-bit ID is also added which indicates whether APF has been performed for that branch and the location of the buffer in which the alternate path instructions for that branch are stored.

APF keeps track of the oldest unresolved branch marked by the H2P table and the oldest unresolved branch marked by the TAGE confidence in the in-flight branch queue. The one marked by TAGE confidence is picked first for APF. If no low-confidence branch is present, then the oldest H2P branch is picked.

Once a branch is picked, the APF pipeline is initialized by setting the alternate path PC to either the next PC or the branch target depending on the prediction. The speculative branch history register for the main pipeline is copied over with the inverted prediction pushed into the history. The APF pipeline can then begin fetching alternate path instructions at this PC. Note that the in-flight branch queue can be scanned while the APF pipeline is processing the current H2P branch so that APF for the next branch can begin immediately after the APF pipeline is free.

The APF pipeline contains a shadow PC register, a shadow branch history register, and a shadow in-flight branch queue (that can hold up to 20 branches). These track the control flow of the alternate path currently active on the APF pipeline. The branch confidence mechanism also sets the H2P and low confidence bits for branches on the alternate path in the shadow in-flight branch queue.

F. Alternate Path Buffers

Instructions enter the Alternate Path Buffers after the last stage of the APF pipeline. These buffers store all the information associated with an alternate path: instructions that have been processed by the APF pipeline, the PC and branch history at the end of the alternate path, and the contents of the shadow in-flight branch queue for that path. Each Alternate Path Buffer has a capacity of 104 uops (8 per cycle for an APF pipeline depth of 13 stages).

G. Branch Resolution and Misprediction

When an H2P branch is resolved and correctly predicted, we look at its in-flight branch queue entry to check whether an Alternate Path Buffer has been assigned to this branch. If so, it is cleared. If the alternate path of the branch is currently being processed, then the APF pipeline is cleared.

If the H2P branch is mispredicted, we initiate misprediction recovery. If no Alternate Path Buffer entry (or in-progress instructions in the APF pipeline) is present for this branch, misprediction recovery proceeds normally. If an entry is present (as indicated by the ID in the in-flight branch queue), we fast-forward to the end of alternate path instructions. This is done by overwriting the Branch History Register and the PC register of the main pipeline with history and PC saved in the Alternate Path Buffer entry. The contents of the saved shadow in-flight branch queue are also copied over. The alternate path instructions for the mispredicted branch may be present in the APF pipeline or the buffers. A 5-1 MUX decides which alternate path should be restored. The saved instructions are then moved into the main pipeline by feeding them cycle-by-cycle into the main pipeline register after the last APF stage (dependency check). A 2-1 MUX decides whether to load this pipeline register from the Alternate Path Buffers or the previous main pipeline stage. This is shown in Fig.5.

The rest of the misprediction recovery operations (flushing the backend, restoring the RAT, etc.) are the same. Since the fast-forwarding involves simple data movement, it can be done in parallel with the rest of the misprediction recovery operations. Note after being moved to the main pipeline, we may need to wait for the state of the RAT to be restored if the RAT has not been checkpointed for that branch. This rarely occurs since most processors checkpoint the state of the RAT for H2P branches, which are also the branches we target.

APF stops on indirect branches other than Returns as we do not bank the indirect branch predictor. APF handles Returns via a 4-entry shadow Return Address Stack (RAS) that stores the addresses of Calls made on the alternate path. The shadow RAS entries are also saved in the buffers and added back to the main RAS if the alternate path turns out to be correct.

H. Critical Path Analysis

Fig.5 (on page 5) shows the additional MUXes added by APF to the pipeline stages. The path from the APF pipeline (or the alternate path buffers) to the main pipeline only goes through two MUXes and does not impact the overall critical path. The other input of the 2-1 MUX comes from the dependency-checking logic in the main pipeline. This adds 1 to 1.5 gate delays to this path, assuming a skewed design is adopted for the 2-1 MUX. This stage is normally not the critical path and potentially has enough slack to accommodate the MUX. The logic within nearby pipeline stages can also be reshuffled to ensure enough slack is available without impacting the critical path.

In the unlikely scenario that enough slack cannot be found for the additional gate delays, we can shorten the APF pipeline so that the MUX is moved to a prior stage where slack is available for slightly reduced performance. Alternatively, the pipeline stage can be split into two so that the critical path is not affected. In either case, APF's performance improvement decreases to 4.0% at worst.

I. Discussion on Hardware Overhead

APF Pipeline Stages The APF pipeline consists of Fetch, Decode, and Pre-Rename stages. This overhead is lower than that of implementing a true 16-wide OoO core as that would require more logic for the Rename and Allocation stages as discussed in Section I. Moreover, replicating pipeline stages is much cheaper than replicating storage structures like caches as the area overhead for logic is considerably lower. According to McPAT [27], the APF pipeline stages take $\sim$2% of the core area, with decode contributing to ($\sim$1.6%). A 16-wide OoO core with additional allocation and execution bandwidth requires $\sim$20% more area. Implementing DPIP requires adding

shadow RAT, free list, ROB, LQ, and SQ in addition, leading to an ∼8% area overhead.

Other APF structures The Fetch Queue for the APF pipeline has 16 entries and takes up to 80B. The Alternate Path Buffers require 3.2KB of storage (∼800 bytes per buffer). Using four 16KB TAGE-SC-Ls storage-wise is the same as a larger TAGE-SC-L but adds some additional logic for folding the alternate path history register.

Energy Overhead The Fetch, Decode, and Dependency-Checking stages account for most of the additional power consumption of the APF pipeline (at most 10% of the core power according to McPAT). This excludes the BP, BTB, and I-cache as they are banked. The APF pipeline is active on average for 65% total execution time. APF also decreases the total static energy due to reduced execution time (∼5%).

VI. Evaluation

A. Methodology

To evaluate APF, we augment Scarab [8], an execution-driven cycle-accurate x86-64 simulator to implement APF and use Ramulator [24] to model main memory. Scarab supports wrong path execution which is essential for accurately modelling Alternate Path Fetch. The system details for the baseline OoO core and additional structures are listed in Table III. The baseline core parameters model an aggressive 8-wide OoO core with a deep backend, similar to many industry products [2], [3], [5]. We use a large TLB and perfect memory disambiguation as they do not affect the relative performance improvement of APF much in our benchmarks. The baseline does not have a banked BP. All results are relative to this baseline core configuration.

Core	3.2GHz, 8-wide issue, TAGE-SC-L Predictor [37] 512 Entry ROB, 256 Entry Reservation Station, 16-wide retire 15-cycle frontend latency, **Decoupled BP** 192 entry load queue, 128 entry store queue
Execution Ports	6 ALU (3 can handle BRs, 3 can handle FP), 3 Load, 3 Store
Predictors	TAGE-SC-L for condition branches History-based indirect branch predictor, RAS
Caches	32KB 8-way L1 I-cache (4-cycle access) 48KB 12-way D-cache (4-cycle access) 1MB 16-way LLC cache (18-cycle access), 64B lines
Memory	DDR4_2400R: 1 rank, 2 channels 4 bank groups and 4 banks per channel tRP-tCL-tRCD: 16-16-16
Alternate Path Fetch	H2P Table: 128-entry, 0.1KB, 1-cycle access Alternate Path Buffers (3.2KB), 4 Buffers
APF Pipeline 13-cycle latency	BP, BTB, and I-Cache banking, Fetch and Decode Stages Dependency check logic (Pre-Rename)

TABLE III
Frontend Parameters

B. Benchmarks

We use the SPEC CPU2017 Integer benchmarks [9] with the ref input sets and the GAP benchmarks suite [14] with inputs g=19 and n=300 in our evaluation. We use the SimPoints [39] methodology to generate up to 5 Simpoints per benchmark, with 200 million instructions per Simpoint, and use a warmup period of 200 million instructions.

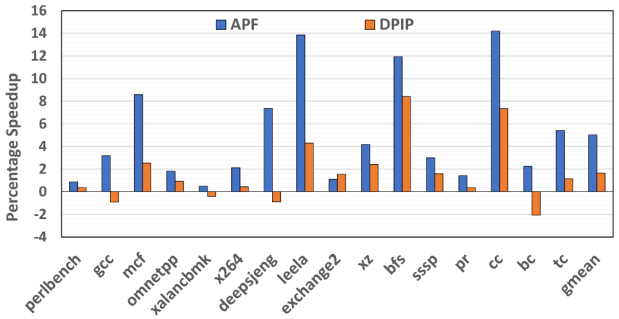

Fig. 8. Performance gains of Alternate Path Fetch

C. Performance Analysis

APF provides a geomean 5% speedup over an 8-wide baseline OoO core with no BP banking (Fig.8). Most benchmarks with a high branch MPKI (Fig.2) show substantial performance gains. However, the performance gains for *tc* and *bc* are relatively lower due to many conflicts on the banked BP (Table IV). *bc* is also less sensitive to branch mispredictions as they overlap with D-cache misses in this benchmark. *perlbench* and *xalanbmk* do poorly since they have fewer conditional branch mispredictions. *x264* does not have as many mispredictions but the coverage on this benchmark is good, which is reflected in its slightly better performance. On the other hand, even though *omnetpp* and *pr* have many mispredicted branches, these mispredictions are not always on the critical path of the program.

The DPIP configuration in Fig.8 uses time-sharing. The fetch cycles are shared with a 1:1 ratio between the predicted and alternate paths as this provided the best performance in our experiments. DPIP can keep track of the context (RAT and Free List state) of one extra branch on the predicted path and one on the alternate path but is limited to fetching from one alternate path at a time.

DPIP performs well on the benchmarks with a very high branch MPKI. However, due to its lower misprediction coverage, it is unable to extract much benefit from the rest of the benchmarks. *gcc*, *deepsjeng* and *bc* drop in performance because useful fetch cycles are taken away from the predicted path and spent on alternate path instructions that do not provide much value. Note that *exchange2* is the only benchmark that does a little better with DPIP as it suffers significantly due to TAGE banking (Fig.7) in APF.

D. Depth of the APF pipeline

As explained in Section III, the depth of the APF pipeline determines the coverage and per-branch benefits that APF can provide. We simulated 6 configurations to analyze the impact of alternate path pipeline depth on performance and misprediction coverage. The first four configurations all implement APF as described but with different pipeline depths: 3-stages (only Branch Prediction), 7-stages (till the end of Fetch), 11-stages (including Decode), and 13-stages (till Pre-Rename). The depth of the baseline frontend is not changed. The shorter

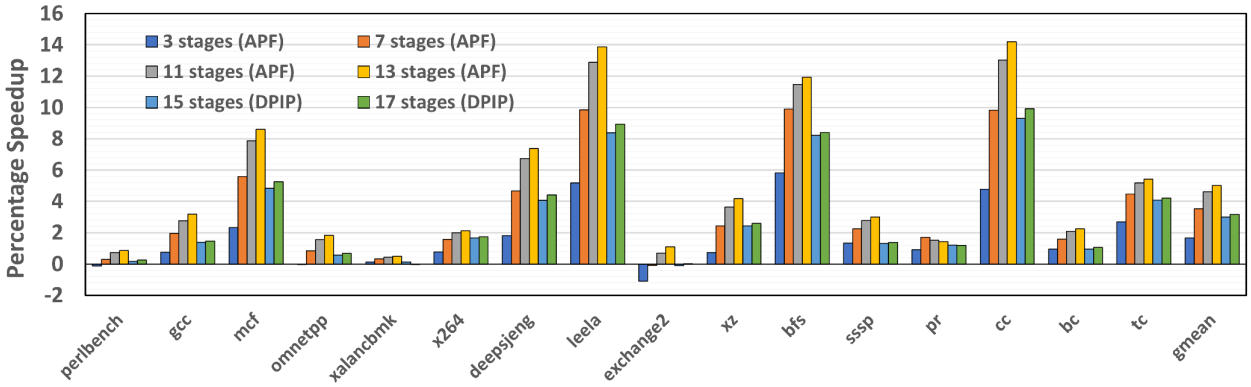

Fig. 9. Impact of changing the alternate path pipeline latency on performance

configurations have fewer APF pipeline stages and a lower area overhead. The last two configurations: 15-stages (till the end of Rename) and 17-stages (before Scheduling) model DPIP with Parallel-Fetch instead of time-sharing. The fetching mechanism for all configurations is the same, and H2P Table parameters were tuned to best suit each configuration.

Performance improves for deeper APF pipelines, going from 3 to 13 stages (Fig.9). Transitioning beyond 13 stages (past RAT access) reduces misprediction coverage significantly resulting in a steep drop in performance. After this, having a deeper pipeline with DPIP provides slightly better performance. These observations are all in line with the reasoning presented in Section IV. DPIP performs much better in these configurations compared to Fig.8 as it fetches more alternate path instructions with Parallel-Fetch and does not incur performance degradation due to time-sharing fetch cycles (3.2% vs 1.6%). An interesting point of comparison is that the performance of the best possible DPIP configuration is equivalent to that of the 7-stage APF pipeline. However, the area costs of the two are vastly different. The DPIP configuration needs to replicate all the frontend stages and backend structures to achieve this performance. The 7-stage APF configuration on the other hand only requires a duplicate Fetch Queue (for branch prediction) and Fetch unit.

Solutions like [4], [15], [19], [36] can only save up to 1-2 cycles of misprediction penalty (for most branches) and only provide a $\sim$1% performance improvement.

Fig.10 shows the misprediction coverage for the 6 configurations. Each category indicates what percentage of conditional branch mispredictions fall under that category. A small percentage of mispredictions in the first four configurations come from branches not marked as H2P. This is due to the warmup period for H2P Table counters and capacity constraints.

The rest of the categories indicate how many cycles of misprediction penalty were saved for a branch marked as H2P. 10%-30% of the branches show no improvements (0 cycles) as these branches were resolved while the APF pipeline was busy. This percentage increases with a deeper pipeline as the first few branches tend to starve out the rest.

The shallower APF pipelines have higher coverage, but each

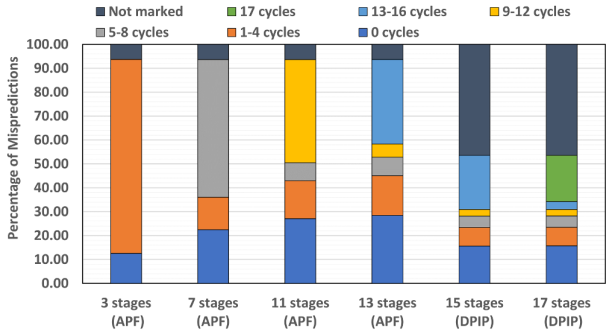

Fig. 10. Percentage of branch mispredictions categorized by how much pipeline re-fill penalty is saved for different configurations

branch that is improved only provides a small benefit. As the pipeline depth increases, the percentage of branches that show any benefit decreases from nearly 80% to around 65% for the APF configurations. However, a large chunk of the benefit comes from higher per-branch savings which results in better performance for the deeper APF pipelines.

At the transition between 13 and 15 stages, misprediction coverage drops drastically due to fewer branches being marked as H2P for DPIP. APF is thus at the sweep spot of this trade-off as it provides the best possible per-branch benefit without significantly losing out on coverage.

E. Fetch Schemes

There are three possible configurations for fetching instructions in APF: time-sharing, Parallel-Fetch via banking the frontend structures, or having two read ports to the frontend structures. Time-sharing is the cheapest alternative as it does not require the Fetch, Decode, and Pre-Rename stages to be duplicated, but also provides the least performance (Fig.11). For APF, the best ratio for sharing fetch cycles was 3:1, with 3 cycles for the predicted path. However *xalanbmk* and *bc* show a slight performance decrease even with the skewed distribution as the benefits of APF cannot make up for the lost cycles on the predicted path. Time-sharing provides more benefits with a decoupled BP as the prediction queues can partially absorb the impact of reduced prediction cycles. This

Benchmarks	perlbench	gcc	mcf	omnettpp	xalanbmk	x264	deepsjeng	leela
Bank conflicts	9.39%	10.45%	8.3%	6.11%	12.37%	12.23%	11.89%	11.13%
Benchmarks	exchange2	xz	bfs	sssp	pr	cc	bc	tc
Bank conflicts	12.38%	6.47%	22.69%	8.39%	10%	11.84%	4.34%	44.42%

TABLE IV
PERCENTAGE OF ALTERNATE PATH FETCH CYCLES SPENT IN BANK CONFLICTS

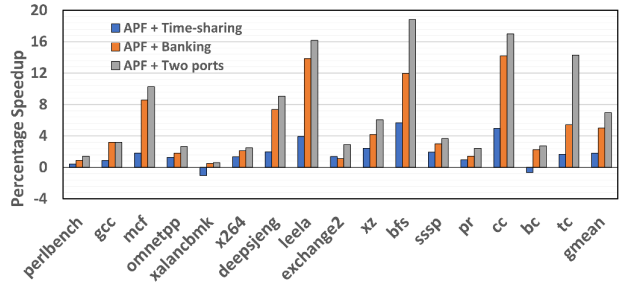

Fig. 11. APF with various Fetch Schemes

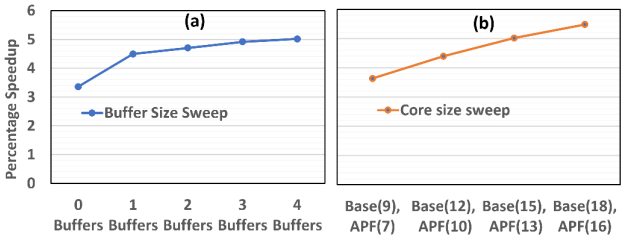

Fig. 12. (a) Sweeping the number of Alternate Path Buffers (b) Changing the baseline OoO pipeline depth. The number in the parenthesis indicates the depth of the baseline frontend and the APF pipeline

is why time-sharing shows some performance improvements across most of the benchmarks. The banked frontend provides better performance and does not require significantly more hardware. The only exception to this is *exchange2* which suffers mainly due to the lower baseline MPKI associated with banking the predictor (Fig.7). An extra port on the other hand improves performance but requires significantly more area and affects timing. Given these trade-offs, the Parallel-Fetch scheme via banking strikes the best balance between performance and area.

The hashing scheme we chose for banking utilizes the lower bits of the PC. This reduces bank conflicts significantly compared to using random bits to uniformly distribute entries across the banks (as explained in Section V-B). The percentage of bank conflicts (shown in Table-IV) is well below 25% for most benchmarks. *bfs* and *tc* are outliers as our hashing scheme does not work well for certain loop patterns. Thus, these benchmarks show the biggest disparity in performance between the banked and two-port configurations.

F. Alternate Path Buffers

The Alternate Path Buffers allow fetching down the alternate paths for multiple H2P branches simultaneously but provide diminishing returns as the number of buffers increases. Having even one buffer helps significantly (Fig.12-(a)) as an active Alternate Path Buffer entry is freed when the corresponding branch is resolved. Most benchmarks do not have a large number of outstanding H2P branches as they often resolve quickly, and therefore having a small number of buffers captures most of the benefits.

G. Effect of Frontend Latency on Performance

The latency of the baseline OoO core changes how much branch mispredictions hurt performance. We evaluated how APF performs on various baseline OoO core configurations, as summarized in Fig.12-(b). A deeper pipeline has a larger re-fill penalty and shows greater performance benefits with APF. A shorter frontend decreases the benefit of APF. For shorter pipelines, the reduced per-branch benefit is partly compensated for by an increase in coverage. Uop caches can reduce Decode latency by up to 3 cycles, but APF still provides a 4.4% performance improvement with them as seen in the Base(12) configuration in Fig.12-(b). Some processors [1], [7] have a 15-cycle frontend latency even with a Uop Cache hit. Similarly, Elastic Fetch decreases the prediction latency by 1-2 cycles on a misprediction. This can also be modeled with a shorter frontend and APF still provides 4.7% performance on top of this. Branch Prediction, Fetch, and Rename logic dominate frontend latency, and these are only likely to increase as frequency and pipeline width are scaled up.

VII. CONCLUSION

Branch mispredictions are still a cause for concern for single-thread performance. Both frequent mispredictions and branch prediction bandwidth limit the effectiveness of a wider frontend. Alternate Path Fetch provides a simple solution to this problem that works for any conditional direct branches: predict and fetch instructions on the alternate path for H2P branches using a separate pipeline that operates in parallel with the existing frontend. This makes effective use of a wider fetch by re-filling alternate path instructions into the main frontend if an H2P branch is found to be mispredicted. We introduce a simple banking scheme for TAGE-SC-L which allows two control flow paths to access the Branch Predictor in parallel without the overhead of an additional port. Alternate Path Fetch provides a 5% improvement over an 8-wide OoO core, compared to the 2.8% speedup that building a true 16-wide OoO core provides and has much lower area and energy overhead.

ACKNOWLEDGMENT

We thank the anonymous reviewers and the members of the HPS Research Group for their feedback and help in improving this paper. We also thank Apple, Intel, Arm, and NSF grant #2011145 for their financial support.

REFERENCES

[1] "AMD Zen2 measurements," https://www.7-cpu.com/cpu/Zen2.html.

[2] "AMD Zen4 microarchitecture," https://www.anandtech.com/show/17585/amd-zen-4-ryzen-9-7950x-and-ryzen-5-7600x-review-retaking-the-high-end/8.

[3] "Apple M1 microarchitecture," https://www.anandtech.com/show/16226/apple-silicon-m1-a14-deep-dive/2.

[4] "IBM System 370 Function Characteristics," https://bitsavers.org/pdf/ibm/370/funcChar/GA22-6935-0_370-165_funcChar_Jun70.pdf.

[5] "Intel Goldencove microarchitecture," https://www.anandtech.com/show/16881/a-deep-dive-into-intels-alder-lake-microarchitectures/3.

[6] "Intel Gracemont microarchitecture," https://www.anandtech.com/show/16881/a-deep-dive-into-intels-alder-lake-microarchitectures/4.

[7] "Intel Icelake measurements," https://www.7-cpu.com/cpu/Ice_Lake.html.

[8] "Scarab," https://github.com/hpsresearchgroup/scarab.

[9] "The standard performance evaluation corporation (spec)," 1997. [Online]. Available: https://www.spec.org/

[10] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '83, 1983, p. 177–189. [Online]. Available: https://doi.org/10.1145/567067.567085

[11] J. L. Aragón, J. González, A. González, and J. E. Smith, "Dual path instruction processing," in *Proceedings of the 16th International Conference on Supercomputing*, ser. ICS '02, 2002, p. 220–229. [Online]. Available: https://doi.org/10.1145/514191.514223

[12] T. Asheim, T. A. Khan, B. Kasicki, and R. Kumar, "Impact of microarchitectural state reuse on serverless functions," in *Proceedings of the Eighth International Workshop on Serverless Computing*, ser. WoSC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 7–12. [Online]. Available: https://doi.org/10.1145/3565382.3565879

[13] G. Ayers, N. P. Nagendra, D. I. August, H. K. Cho, S. Kanev, C. Kozyrakis, T. Krishnamurthy, H. Litz, T. Moseley, and P. Ranganathan, "Asmdb: understanding and mitigating front-end stalls in warehouse-scale computers," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 462–473. [Online]. Available: https://doi.org/10.1145/3307650.3322234

[14] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," 2017. [Online]. Available: https://arxiv.org/abs/1508.03619

[15] J. Bondi, A. Nanda, and S. Dutta, "Integrating a misprediction recovery cache (mrc) into a superscalar pipeline," in *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, 1996, pp. 14–23.

[16] P.-Y. Chang, E. Hao, Y. N. Patt, and P. P. Chang, "Using predicated execution to improve the performance of a dynamically scheduled machine with speculative execution," in *Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '95, 1995, p. 99–108.

[17] A. Chauhan, J. Gaur, Z. Sperber, F. Sala, L. Rappoport, A. Yoaz, and S. Subramoney, "Auto-predication of critical branches," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ser. ISCA '20, 2020, p. 92–104. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00019

[18] A. Deshmukh and Y. N. Patt, "Criticality driven fetch," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 380–391. [Online]. Available: https://doi.org/10.1145/3466752.3480115

[19] B. Grayson, J. Rupley, G. Z. Zuraski, E. Quinnell, D. A. Jiménez, T. Nakra, P. Kitchin, R. Hensley, E. Brekelbaum, V. Sinha, and A. Ghiya, "Evolution of the samsung exynos cpu microarchitecture," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 40–51.

[20] D. Grunwald, A. Klauser, S. Manne, and A. Pleszkun, "Confidence estimation for speculation control," in *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No.98CB36235)*, 1998, pp. 122–131.

[21] D. Jiménez, "Multiperspective perceptron predictor," in *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, 2016.

[22] H. Kim, J. A. Joao, O. Mutlu, and Y. N. Patt, "Diverge-merge processor (dmp): Dynamic predicated execution of complex control-flow graphs based on frequently executed paths," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, 2006, pp. 53–64.

[23] H. Kim, O. Mutlu, J. Stark, and Y. N. Patt, "Wish branches: Combining conditional branching and predication for adaptive predicated execution," in *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 38, 2005, p. 43–54. [Online]. Available: https://doi.org/10.1109/MICRO.2005.38

[24] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer Architecture Letters*, vol. 15, no. 1, 2016.

[25] A. Klauser, T. Austin, D. Grunwald, and B. Calder, "Dynamic hammock predication for non-predicated instruction set architectures," in *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '98. USA: IEEE Computer Society, 1998, p. 278.

[26] A. Klauser, A. Paithankar, and D. Grunwald, "Selective eager execution on the polypath architecture," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, ser. ISCA '98, 1998, p. 250–259. [Online]. Available: https://doi.org/10.1145/279358.279393

[27] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42, 2009, p. 469–480. [Online]. Available: https://doi.org/10.1145/1669112.1669172

[28] S. McFarling, "Combining branch predictors," Citeseer, Tech. Rep., 1993.

[29] S. Palacharla, N. Jouppi, and J. Smith, "Complexity-effective superscalar processors," in *Conference Proceedings. The 24th Annual International Symposium on Computer Architecture*, 1997, pp. 206–218.

[30] A. Perais and R. Sheikh, "Branch target buffer organizations," ser. MICRO '23, 2023, p. 240–253. [Online]. Available: https://doi.org/10.1145/3613424.3623774

[31] A. Perais, R. Sheikh, L. Yen, M. McIlvaine, and R. D. Clancy, "Elastic instruction fetching," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 478–490.

[32] J. Pierce and T. Mudge, "Wrong-path instruction prefetching," in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 29. USA: IEEE Computer Society, 1996, p. 165–175.

[33] S. Pruett and Y. Patt, "Branch runahead: An alternative to branch prediction for impossible to predict branches," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21, 2021, p. 804–815. [Online]. Available: https://doi.org/10.1145/3466752.3480053

[34] G. Reinman, B. Calder, and T. Austin, "Fetch directed instruction prefetching," in *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, 1999, pp. 16–27.

[35] E. Safi, A. Moshovos, and A. Veneris, "Two-stage, pipelined register renaming," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 10, pp. 1926–1931, 2011.

[36] D. A. Schroter and A. J. Van Norstrand, "Microprocessor instruction hedge-fetching in a multiprediction branch environment," Dec. 15 1998, uS Patent 5,850,542.

[37] A. Seznec, "A new case for the tage branch predictor," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44, 2011, p. 117–127. [Online]. Available: https://doi.org/10.1145/2155620.2155635

[38] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides, "Design tradeoffs for the alpha ev8 conditional branch predictor," ser. ISCA '02, 2002, p. 295–306.

[39] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," ser. ASPLOS X, 2002, p. 45–57. [Online]. Available: https://doi.org/10.1145/605397.605403

[40] B. Solomon, A. Mendelson, D. Orenstien, Y. Almog, and R. Ronen, "Micro-operation cache: a power aware frontend for variable instruction length isa," in *ISLPED'01: Proceedings of the 2001 International Symposium on Low Power Electronics and Design (IEEE Cat. No.01TH8581)*, 2001, pp. 4–9.

[41] N. K. Soundararajan, P. Braun, T. A. Khan, B. Kasikci, H. Litz, and S. Subramoney, "Pdede: Partitioned, deduplicated, delta branch

target buffer," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 779–791. [Online]. Available: https://doi.org/10.1145/3466752.3480046

[42] A. Uht, "Multipath execution," *High Performance Computer Architectures, D. Kaeli and P.-C. Yew, Eds. Boca Raton*, pp. 135–160, 2005.

[43] A. Uht, V. Sindagi, and K. Hall, "Disjoint eager execution: an optimal form of speculative execution," in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, 1995, pp. 313–325.

[44] S. Wallace, B. Calder, and D. M. Tullsen, "Threaded multiple path execution," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, ser. ISCA '98. USA: IEEE Computer Society, 1998, p. 238–249. [Online]. Available: https://doi.org/10.1145/279358.279392

[45] T.-Y. Yeh, D. T. Marr, and Y. N. Patt, "Increasing the instruction fetch rate via multiple branch prediction and a branch address cache," in *Proceedings of the 7th International Conference on Supercomputing*, ser. ICS '93, 1993, p. 67–76. [Online]. Available: https://doi.org/10.1145/165939.165956

[46] S. Zangeneh, S. Pruett, S. Lym, and Y. N. Patt, "Branchnet: A convolutional neural network to predict hard-to-predict branches," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 118–130.