Elastic Instruction Fetching

Arthur Perais† Rami Sheikh‡ Luke Yen† Michael McIlvaine† Robert D. Clancy†
†Qualcomm Datacenter Technologies, Inc. ‡Qualcomm Technologies, Inc.
{aperais,ralsheik,lyen,mmcilvai,rclancy}@qti.qualcomm.com

***Abstract*—Branch prediction (i.e., the generation of fetch addresses) and instruction cache accesses need not be tightly coupled. As the instruction fetch stage stalls because of an I-Cache miss or back-pressure, the branch predictor may run ahead and generate future fetch addresses that can be used for different optimizations, such as instruction prefetching but more importantly hiding taken branch fetch bubbles. This approach is used in many commercially available high-performance design.**

However, decoupling branch prediction from instruction retrieval has several drawbacks. First, it can increase the pipeline depth, leading to more expensive pipeline flushes. Second, it requires a large Branch Target Buffer (BTB) to store branch targets, allowing the branch predictor to follow taken branches without decoding instruction bytes. Missing the BTB will also cause additional bubbles. In some classes of workloads, those drawbacks may significantly offset the benefits of decoupling.

In this paper, we present ELastic Fetching (ELF), a hybrid mechanism that decouples branch prediction from instruction retrieval while minimizing additional bubbles on pipeline flushes and BTB misses. We present two different implementations that trade off complexity for additional performance. Both variants outperform a baseline decoupled fetcher design by up to 3.7% and 5.2%, respectively.

I. Introduction

Many high-performance general purpose processors decouple the branch prediction logic (a.k.a, the instruction sequencing logic) from the actual instruction retrieval from the instruction cache [1], [2], [3], [4]. Specifically, with the aid of a dedicated structure containing branch information such as a Branch Target Buffer (BTB), the branch prediction infrastructure generates fetch addresses and queues them in a decoupling queue. The fetcher can then consume those fetch addresses as fast as allowed by the instruction cache behavior. Decoupling has well-known benefits [5]. For instance, the decoupling queue can act as a very accurate instruction prefetcher: If an instruction cache miss takes place, the decoupling queue will get filled since branch prediction can run ahead. When this happens, prefetch addresses can be picked from the decoupling queue. Another, potentially more critical example, is that some implementations may suffer from a taken branch penalty, i.e., one or more bubbles inserted every time a branch is predicted taken in the fetch unit, even in the presence of a BTB.[1] Through decoupling, this bubble can be hidden as long as branch prediction is able to run ahead of the fetch unit. Nonetheless, decoupling branch prediction from instruction data retrieval has its own drawbacks. *First*, it requires the presence of a large BTB-like [6] structure to allow branch prediction to follow taken branches. Note that for decoupled fetching to increase performance through instruction prefetching, the BTB reach has to be greater than the I-Cache reach, which demands a larger BTB. *Second*, it mechanically increases the pipeline depth, such that a given fetch PC has to go through branch prediction first, then through instruction cache access. In a coupled design, both happen in parallel. This implies that the branch misprediction penalty is generally higher. Moreover, this increased pipeline depth also appears on the critical path on BTB misses, as the decode (or later) stage has to resteer the branch predictor once the relevant branch target is decoded/computed. In this paper, we propose ELastic Fetching (ELF), a mechanism that decouples branch prediction and instruction retrieval in steady state (referred to as *Decoupled mode*), but couples them after a pipeline flush to hide the additional latency introduced by decoupling (referred to as *Coupled mode*). We detail two possible implementations. The first, *Limited ELastic Fetching* (L-ELF), has low hardware overhead and yields modest IPC improvement – up to 3.7% – in relevant workloads. The second, *Unlimited ELastic Fetching* (U-ELF), may have higher implementation overheads depending on design choices left to the architect, but yields more substantial IPC improvement – up to 5.2% – in relevant workloads.

[1] In high frequency designs, a single cycle may not be sufficient to access a large BTB and branch predictor, process the BTB entry content, map the branch predictions to the BTB entry, and finally compute the next PC.

II. Related Work

The Branch Target Buffer (BTB) was initially disclosed by Losq [6], and was further described by Lee and Smith [7]. Interestingly, the BTB was initially envisioned as a way to reduce the cost of taken branches by pulling the prediction and target of a branch from the BTB in parallel with instruction fetch. If the branch was predicted taken, instructions at the target of the branch could be fetched in the following cycle, rather than having to wait for the instruction word to be decoded. However, the modern-day BTB typically stores – direct – branch targets and metadata while but conditional prediction is left to a dedicated predictor [8].

Reinman et al. [9], [5] first proposed decoupling fetch address generation from instruction retrieval by using the BTB and a decoupling queue to queue fetch addresses

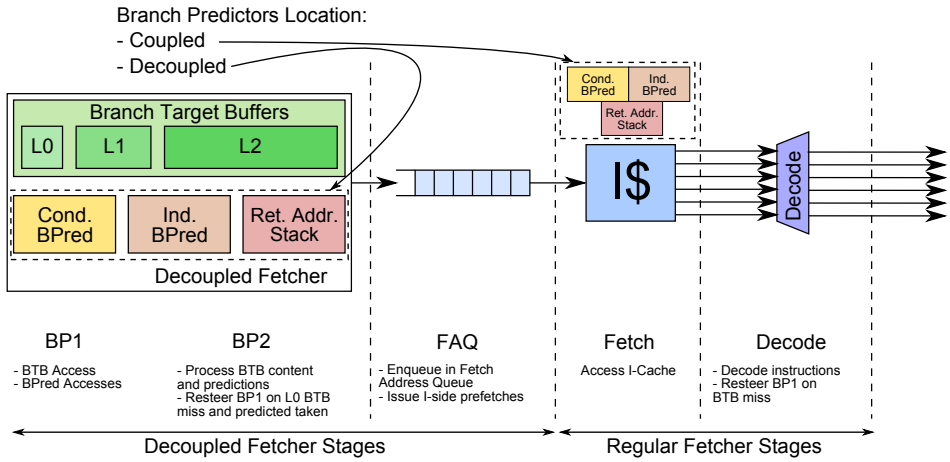

Figure 1: Decoupled fetcher pipeline consists of: decoupled fetch stages (BP1, BP2, and FAQ), and regular (non-decoupled) stages (Fetch, and Decode). One stage corresponds to one cycle.

during I-Cache misses or other long latency events. Through decoupling, very accurate instruction prefetching can be performed, and if the decoupling queue occupancy is high enough, taken branch bubbles can be squashed.

Predating Reinman et al., Stark et al. [10] similarly proposed to use the BTB to generate fetch addresses ahead of the instruction stream in case of an I-Cache miss. However, the proposed design aimed at *actually fetching* instructions past the I-Cache miss (i.e., out-of-order fetching) in a coupled fashion, rather than truly decoupling branch prediction from instruction fetch. As a result, this scheme was not able to squash taken branch bubbles, although it did perform instruction prefetching.

More recently, Kumar et al. described Boomerang [11], a technique that attempts to mitigate BTB misses by probing the I-Cache and decoding targets from the cache data directly, before they are needed. Indeed, many server workloads feature a gigantic instruction footprint that cannot fit entirely in the BTB, causing many branch prediction cycles to be wasted waiting for the decoded target. With Boomerang, the BTB miss is resolved as soon as possible, and its latency hidden if enough fetch addresses are en-route to the fetcher.

III. Decoupled Fetcher Overview

Decoupling branch prediction from instruction retrieval has one stringent requirement: the branch predictor has to be able to follow branches it predicts taken without accessing the instruction cache. As a result, a BTB is usually implemented side-by-side with the branch predictor. The BTB may be branch-target indexed and contains branch information as well as branch target(s) when relevant (i.e., for direct branches) for a given number of instructions. A single BTB entry tracks a finite number of branches, as well as a maximum number of instructions, or instruction bytes.

A. BTB Entry

Entry Content. In our framework, and as in AMD Zen, an entry in the BTB can track up to 16 sequential instructions, 2 of them being *"observed taken before"* branches [8]. That is, a conditional branch that was never observed taken will not occupy any of the two branch slots.

Entry Establishment. While it is possible to populate the BTB speculatively after instruction decode, we establish BTB entries non-speculatively as instructions retire. This limits complexity as entries being constructed never need to be – partially – flushed.

A new BTB entry being established ends when either: 1) An unconditional branch is encountered, 2) A third taken conditional is encountered, or 3) The entry spans 16 sequential instructions. In addition, an existing BTB entry may have to be amended if a *"never observed taken"* conditional branch becomes taken. This may cause an entry to be split into two to accommodate the fact that a single entry can track at most two *"observed taken before"* branches.

B. Frontend Operation

Figure 1 depicts the pipelined organization of the decoupled fetching (DCF) infrastructure we consider in this paper.

1) Coupled: Only the two rightmost stages in the Figure are used. During Fetch, the I-Cache and predictors are accessed using the fetch PC to retrieve N instruction words as well as N branch predictions. In this paper, we assume the presence of a smaller L0 I-Cache that is able to present data at the output in a single cycle. It is backed-up by a bigger L1 I-Cache that has multi-cycle latency, following Qualcomm's Centriq frontend design [12]. Therefore, assuming the presence of pre-decode bits (e.g., branch location) in the L0 I-Cache, predictions are attributed to branch instructions in parallel with Decode, causing a bubble to be inserted on a predicted taken branch. Various techniques exist to mitigate

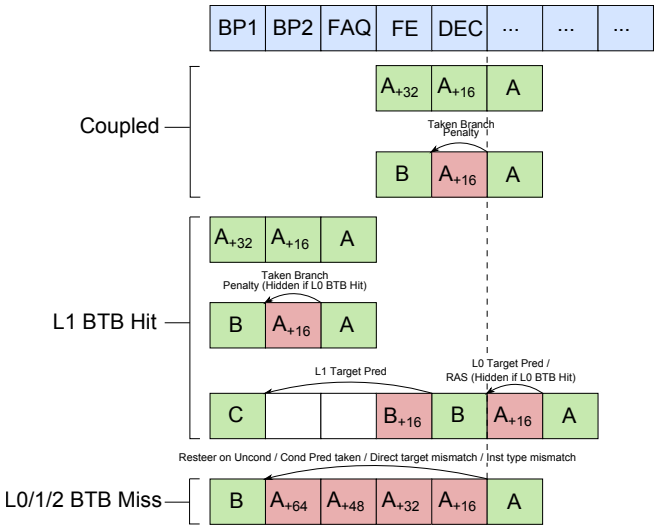

Figure 2: Timing depending on BTB content and instruction type. Green blocks are speculative addresses corresponding to branch and target predictions. Red blocks are discarded sequential addresses generated speculatively while waiting for a prediction or a correction.

this bubble such as using a BTB [7] or a Next Line Predictor [13].

2) Decoupled: Each cycle, in the "Branch Prediction 1" (BP1) stage, the L0 and L1 BTB are accessed with the current "BPred PC". In parallel, the branch predictor is accessed in order to retrieve N branch predictions, where N is the maximum number of conditional branches tracked per BTB entry (2 in this paper). Depending on the BTB entry branch information and the branch predictions, the next "BPred PC" is either the target of one of the N conditional branches, the target of an indirect branch if the BTB entry tracks one, the target of an unconditional direct branch if the BTB entry tracks one, or the fall-through of the BTB entry. Depending on the access time of the branch predictor, BTB and indirect target predictor, one or more bubbles may be inserted between the generation of two BPred PCs. In this paper, we assume one cycle for the processing of the content of the L1 BTB entry: "Branch Prediction 2" (BP2) stage. However, we are able to speculatively access the L0 and L1 BTB with the proxy fallthrough address (PC + 16 instructions) of the previous block each cycle. Consequently, a bubble is inserted if: 1) L1 BTB reports a taken branch in BP2 2) The L1 BTB entry does not track the maximum amount of sequential instructions (16 in this paper), causing the proxy fallthrough address to be incorrect despite the absence of a taken branch.[2]

[2]This may happen if an entry tracks e.g. 10 instructions, including two *observed taken before* branches. Assuming the last instruction is a tracked branch and its fallthrough is a third branch that has been taken before, the third branch cannot be given a slot in the BTB entry and will reside in a distinct entry. However, the speculative fallthrough access at PC + 16 insts. will be incorrect since the actual fallthrough is PC + 10 insts.

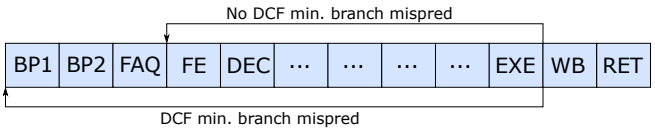

Figure 3: Minimum branch misprediction penalty in the presence of a decoupled fetcher infrastructure.

Moreover, if the L0 BTB hits, we are able to generate the next BPred address during the same cycle, using branch predictions from the bimodal component of TAGE [14]. As a result, even on a predicted taken branch, an L0 BTB hit prevents any bubble from being inserted in BP1. Note that if the tagged components of the TAGE predictor disagree with the bimodal, the prediction is overridden in BP2 and a bubble is inserted, as in the L1 BTB hit case.

In most ISAs, including ARMv8 that we use in this paper, indirect branches are unconditional. That is, the indirect branch predictor can be queried with the BPred PC for a single prediction since there will at most be one indirect branch per BTB entry: Such branch terminates the BTB entry. Therefore, indirect branches may not incur additional bubbles versus direct and conditional branches. In this paper, however, we implement a 2-level indirect target predictor that has variable latency: a hit in the L0 predictor or the Return Address Stack will cause a single bubble to be added (as with direct taken branches), but a miss in the L0 predictor will cause three bubbles to be added.

Figure 2 summarizes several examples of address generation timing depending on the BTB content, and also gives timing for the coupled organization as a reference.

Generated fetch addresses, along with the number of sequential instructions at each address, are enqueued in the "Fetch Address Queue" (FAQ) stage. Fetch addresses are consumed by the fetch unit as instruction words are retrieved from the L0 instruction cache in "Fetch" (FE). In the event that the BTB information is stale (e.g., cold miss, self-modifying code), the "Decode" (DEC) stage is in charge of re-steering the earlier stages (BP1/BP2/FAQ/FE). For instance, if it determines that the decoded target of a direct branch differs from the one stored in the BTB. Similarly, and as in a Coupled pipeline, the back-end will re-steer the whole front-end on branch and indirect target mispredictions as well as other pipeline flushing events.

C. BTB Misses

Until now, we placed ourselves in the context of BTB hits. However, since the BTB is a finite-size structure, it is possible that a "BPred PC" will miss the BTB. In that case, the decoupled fetcher (DCF) is essentially unable to determine what the next "BPred PC" is since it cannot tell whether there are branches and what their targets are. In that context, DCF can simply queue sequential fetch addresses (i.e., BPred PC + max instructions per BTB entry) in the decoupling

queue, although a misfetch is highly likely to occur since most BTB entries are terminated by taken branches.

As illustrated in Figure 2, it is usually possible to recover from a BTB miss earlier than from a branch misprediction. As soon as the decode stage detects a direct unconditional branch, it can resteer the frontend using the target contained in the instruction word. Similarly, this can take place for conditional branches if the branch predictor predicted taken for that branch, or for returns by explicitly stalling while the Return Address Stack (RAS) of the DCF is accessed (not shown in the figure). Indirect branches other than returns, however, must either wait for their target to be resolved at execution time or use the indirect target prediction to resteer the front-end. As a result, using a decoupled fetcher introduces another feedback loop that spans from the first branch prediction stage to the decode stage. This loop can be observed on the critical path every time the BTB misses, and its impact on performance will grow as the number of cycles between BP1 and Decode increases [15].

D. Other Pipeline Flushes

Pipeline flushes due to branch mispredictions or RAW hazards between a load and a store resteer the front-end. However, they also expose the additional latency introduced by the branch prediction stages, as illustrated in Figure 3. Overall, compared to a Coupled pipeline, DCF will incur additional penalty on all pipeline flushes as well as all BTB misses.

IV. ELastic Fetching (ELF)

The benefits of DCF can be belittled for workloads that often expose the additional pipeline depth introduced by DCF (due to frequent branch mispredictions, BTB misses ...etc.) [5]. This effect is likely to be exacerbated for workloads that do not benefit significantly from DCF in the first place, e.g., because they have a small instruction footprint that fits in the I-Cache and they are not sensitive to taken branch bubbles. As we will show in Section VI, in such cases, DCF is in fact detrimental to performance.

When considering only the instruction prefetch aspect of DCF, one might argue that implementing a dedicated instruction prefetcher can be superior to implementing DCF. However, DCF is orthogonal to other instruction prefetching schemes. Moreover, it also permits other optimizations such as hiding the taken branch penalty that may exist even in BTB-based designs as well as fetching across a taken branch in a single cycle. It is therefore broader in scope than instruction prefetching, which likely explains why it is widely used in commercial processors from AMD, Samsung, IBM and ARM [1], [2], [3], [4]. Therefore, hiding the additional latency (i.e., increase in pipeline depth), incurred by DCF on pipeline flushes, is of great value to modern frontend implementations.

A. Introducing ELF

To address the potential drawbacks of DCF, we introduce ELastic Fetching (ELF). The main idea behind ELF is that once a pipeline flush occurs (or once a BTB miss is resolved), the correct next PC is known. Consequently, we propose to probe the I-Cache immediately using this fetch PC while the decoupled fetching engine is restarting from BP1. This short-circuits the branch prediction stages and reduces the pipeline flush penalty. We note that this idea was first suggested by Reinman [9] and assumed to be a given, hence, no implementation was detailed and its impact on performance was not studied. This paper claims that this assumption may have been optimistic.

Allowing branch prediction and fetch to process the same PC concurrently is not a straightforward change as the processor now has two fetch address generation engines that are desynchronized. As a result, we have to provide a mechanism to resynchronize the fetcher with branch prediction when the latter catches up. Specifically, we introduce two modes for the fetcher.

Coupled Mode: In this mode, PC generation is done by the fetcher itself, as if the fetcher is not decoupled. This mode is the *transient* state, i.e., the processor should only spend a small fraction of cycles in this mode. Coupled mode is entered after a pipeline flush, or after a BTB miss is resolved in the Decode stage.

Decoupled Mode: In this mode, PC generation is performed by the decoupled fetcher with the help of the BTB. This is the *steady* state, and it is entered when the decoupled fetcher catches up to the fetcher that is currently running in coupled mode.

In the remainder of this paper, we use ”coupled” and ”decoupled” to either describe a dynamic resource (e.g., a coupled instruction is an instruction fetched in coupled mode), or a static resource (e.g., a coupled predictor is a structure that belongs to the fetcher, while a decoupled predictor belongs to the DCF).

B. Implementation #1: Limited ELF (L-ELF)

The fetcher will only fetch sequential instructions when in coupled mode. As soon as a control flow decision needs to be made, it will stall. Note that in this context, following an unconditional direct branch is **not** a control-flow decision, i.e., L-ELF can fetch past unconditional direct branches.

This implementation, which is similar in spirit to Reinman [9], is clearly the simplest one from the fetcher’s point of view, as it does not require branch prediction capability. Only a mechanism to resynchronize with the decoupled fetcher is necessary. However, if the number of instructions fetched in this mode is small, i.e., indirect/conditional branch density is high, then part of the additional latency exposed during pipeline flushes and BTB misses may not be hidden.

1) Resynchronizing DCF and the Fetcher: In this first implementation, switching from coupled mode to decoupled mode once the DCF has caught up is straightforward. Indeed, instructions fetched in coupled mode are guaranteed to be a subset of the FAQ blocks that the DCF will eventually generate. Consequently, the fetcher simply has to keep a count of instructions fetched in coupled mode, and compare this count to the count in FAQ blocks.

Every time an FAQ block becomes available for consumption, the fetcher will compare its current count of instructions fetched (in coupled mode) to the number of instructions that would already have been fetched (in decoupled mode) plus the size of the FAQ block. Three scenarios are possible:

1) The counts are equal: The fetcher has already fetched all the instructions from the FAQ blocks and said block can simply be popped. Decoupled mode can start from the next FAQ block. Alternatively, coupled mode can continue until one of 2.b) or 3) is met.
2) The coupled count is greater than the decoupled count. Either:
   a) DCF has not caught up, so no switching happens, or
   b) The fetcher overshot and fetched past a control-flow decision as a result of blindly fetching *fetch-width* instructions per cycle. This can easily be detected by embedding the cause of termination for each FAQ block. That is, if the cause is a taken branch (as opposed to sequencing to the next block), the coupled count cannot be greater than the decoupled count in L-ELF.[3] Upon detection, decoupled mode can start from the next FAQ block and Decode must be forced to throw away the instructions that overshot (which the fetcher would have done anyway in a non-decoupled design).
3) The coupled count is lesser than the decoupled count. Either DCF has caught up and/or the fetcher has encountered a control-flow decision and is stalled. The count in the newly generated FAQ block has to be adjusted and decoupled mode can start from the amended FAQ block.

When in coupled mode, each cycle the fetcher attempts to switch to decoupled mode if there is an FAQ block to be processed. That is, while the I-Cache access has started, the coupled count is speculatively updated assuming *fetch width* instructions will be fetched, and it is compared against the decoupled count adjusted with the count contained in the FAQ entry. This enables decoupled mode to start in the following cycle while still having retrieved instructions from the I-cache this cycle (unless the fetcher is stalled on a control-flow decision, in which case bubble(s) will be inserted).

Therefore, while very limited, L-ELF will be able to hide part or all the additional latency incurred by DCF as long as the fetcher is able to fetch a long enough sequence of sequential instructions without running into a conditional or indirect branch.

[3]The exception is self-modifying code where the coupled count could legitimately be higher as the BTB information could be stale. We assume that this case is already handled since any decoupled design must account for the possibility of self-modifying code.

C. Implementation #2: Unlimited ELF (U-ELF)

The fetcher implements branch prediction structures to be able to make control-flow decisions until the DCF catches up.

1) Branch Prediction Infrastructure: Intuitively, since most of the time is spent running in decoupled mode (steady state), the DCF features a complex branch predictor as well as a complex branch target predictor. Moreover, and for the same reason, in coupled mode, the number of static branches to track state for should be lower. Therefore, to limit overhead, it seems reasonable to implement limited-complexity predictors in the fetcher.

In addition, it is possible to only implement a specific predictor type and to stop coupled fetching if a branch that cannot be predicted is decoded. For instance, the fetcher may only feature a direction predictor and no indirect branch predictor (or only part of it such as the return address stack).

In this paper, we consider four predictor arrangements for the fetcher:

1) **RET-ELF**: 32-entry Return Address Stack (RAS), allowing to predict returns only.
2) **IND-ELF**: 64-entry direct-mapped Branch Target Cache that predicts all indirect branch types (except returns).
3) **COND-ELF**: 2K-entry bimodal predictor (3-bit counters), allowing to predict conditional branches.
4) **U-ELF**: All of the above.

We point out that these predictors are very simple, and that except for the RAS, they do not require checkpointing to restore state in case of a pipeline flush. The rationale for limiting ourselves is that ELF is a microarchitectural feature that targets specific classes of workloads, and as such, a **general purpose** design may not have significant area to spare to implement the coupled predictors. Nonetheless, there is no fundamental limitation in ELF that prevents the architect from implementing more complex prediction schemes.

Figure 4 gives an overview of a fetcher implementing full U-ELF. Note that branch tracking information from DCF is processed at Fetch since it is available from the FAQ, while branch tracking information from the coupled fetcher is processed after Decode since instructions must be decoded for branch information to become available. Next, we discuss the structures required to ensure correct operation.

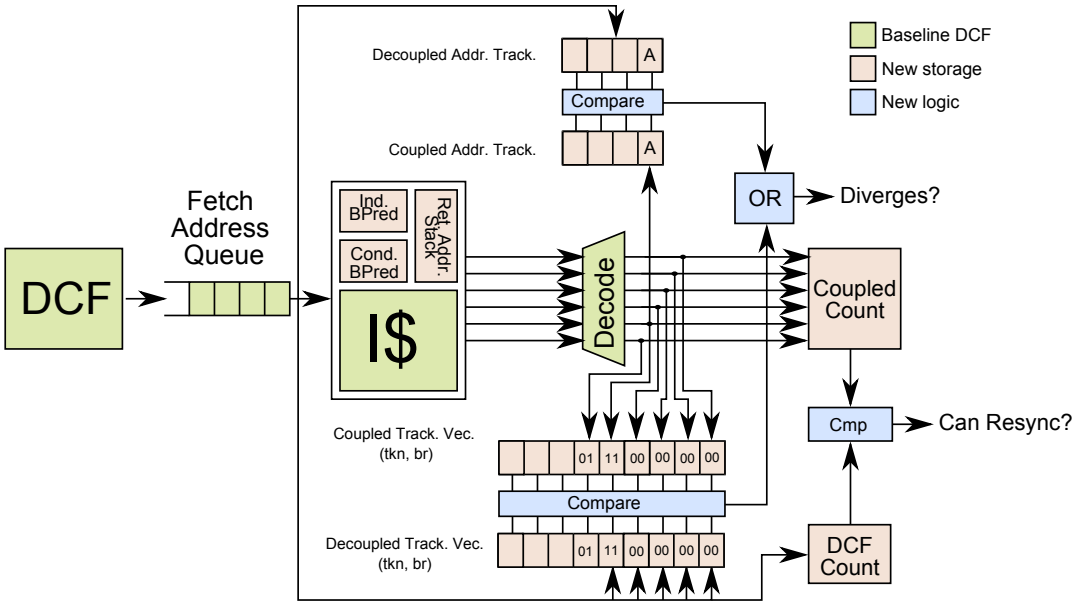

Figure 4: Pipeline block diagram for U-ELF. Decode Coupled Count is not shown in the Figure. Empty bitvectors/target queues entries have their *valid* bit set to 0 and do not participate in direction/target comparison.

2) Divergence: Using different control-flow speculation schemes in DCF and fetcher opens up the possibility of them going down different paths. As a result, the hardware must monitor both streams to be able to recover, preferably in a timely fashion.

Bitvectors: Two finite-size bitvectors are implemented in the fetcher, one tracks coupled stream and the other tracks decoupled stream. Each entry in these bitvectors consists of 3 bits: *taken*, *branch*, and *valid*. For *taken*, a 0 indicates a non-taken branch or non-branch instruction, while a 1 indicates a taken branch instruction. The *branch* bit is set for all branches. The *valid* bit, not shown in Figure 4, is used to prevent unallocated entries from triggering a mismatch signal. As a result, bitvectors (and target queues described below) are not managed as circular buffers. Rather, each entry has a sibling entry in the other bitvector/queue and comparison can be done directly, with the mismatch signal guarded by the *valid* bits. A circular buffer implementation is possible but complicates the comparison logic as it requires multiplexing.

The vectors are only populated in coupled mode. The first bitvector tracks decoded instructions and is therefore populated after Decode. The second tracks instructions that would have been fetched in decoupled mode, as recorded by the FAQ, and is therefore populated at Fetch, when a FAQ entry becomes available. The bitvectors are compared each cycle and a difference between two sibling entries with the *valid* bits set indicates divergence. At that point, we must either trust the DCF and flush any instruction fetched in coupled mode past the divergence point, or trust the fetcher and flush the DCF.

In general, since the DCF has the complex branch predictors, it would seem intuitive to trust the DCF over the fetcher. However, for the case of branch *direction* validation, there are exceptions where the fetch addresses generated by the DCF must be flushed while fetching continues in coupled mode:

1) On a BTB miss, any unconditional branch contained in the block tracked by the BTB entry will not have its target in the BTB. As a result, DCF will continue generating sequential fetch addresses. However, the fetcher will detect the unconditional branch in Decode, which will ultimately lead to detecting a divergence as DCF believes the stream is sequential, while the fetcher **knows** that there is a taken branch.
2) On a BTB hit ending by a taken branch (of any type), and in workloads featuring self-modifying code, it is possible that the decoded instruction corresponding to the taken branch is no longer a branch. This is detected using the *branch* bit which is 0 in the coupled bitvector and 1 in the decoupled bitvector.

Target Queues: Bitvectors as described above are not sufficient to detect divergence in case of an indirect branch being predicted differently by the fetcher and the DCF. Even worse, not detecting this case can lead to switching back to decoupled mode while the fetcher is on the correct path but DCF is not. Thus, the bitvectors are backed-up by two target queues and associated *valid* bits. As taken *direct and indirect* branches are encountered in DCF (Fetch timeframe) and the fetcher (Decode timeframe), the addresses are pushed in the target queues. As for the bitvectors, the addresses are compared each cycle and a difference between two entries with the *valid* bits set indicates divergence.

In the common case, we trust DCF when detecting divergence and we flush coupled instructions fetched past the divergence point. However, the following exception applies: On a taken direct branch (conditional or unconditional), the fetcher has the decoded target, which is the correct one. This target might differ from the one recorded by the BTB in the case of self-modifying code. If that is the case, then DCF is flushed and fetching continues in coupled mode. This entails that the Target Queues must also retain the type of the branch, to be able to choose the correct "winner" on a divergence. Specifically, on a divergence for an indirect branch, DCF wins, but on a divergence on a direct branch, the fetcher wins.

Interestingly, this exception is not a consequence of U-ELF, but of decoupled fetching itself. Indeed, since the BTB does not have to honor cache invalidation idioms used for self-modifying code (e.g., ARM64's IC_I*) because it does not contain instruction words, this case may arise during regular decoupled fetching. In other words, regular decoupled fetching already requires comparing the targets of direct branches. U-ELF simply adds another requirement: comparing the targets of indirect branches to detect divergence as well as buffering to hold decoded/predicted targets to allow the fetcher to run ahead.

3) Resynchronizing DCF and the Fetcher: As U-ELF is allowed to speculate past control-flow decisions, the precise coupled instruction count will only be known after instructions have been decoded and the branches identified. Therefore, a reasonable way to switch back to decoupled mode would be to start using FAQ information as soon as the DCF catches up with instruction decode. However, this implies adding bubbles corresponding to the fetch-to-decode latency when switching.

To avoid this penalty, we can use the two instruction counts used in L-ELF to switch back to decoupled mode as soon as possible, while precise divergence detection is done at Decode. To that extent, we add a *Decode Coupled Count* which, contrarily to the *Fetch Coupled Count* is non speculative and is therefore used for validation purposes as well as for ensuring that the current FAQ entry can be safely popped without the risk of losing information.

Note that after a switch to decoupled mode, we still need to track the few instructions that have been fetched (in coupled mode) but not yet decoded to prevent divergence. This means that we cannot reset the bitvectors and target queues upon switching to decoupled mode, but rather, we need to wait until all coupled instructions have passed through decode.

4) Resynchronization Example: Figure 5 describes a simple scenario where the fetcher is resynchronized with the DCF during cycle 1 even though some coupled instructions are still flowing from Fetch to Decode at the end of cycle 1.

Cycle 0: In the first cycle, the Decode stage, which already decoded 8 coupled instructions in the previous cycle, is receiving 8 instructions but discovers that the fourth one is a conditional branch that was predicted taken by the coupled predictor. As a result, the decode coupled count will increase to 12, and the Fetch stage will be resteered. Population of the coupled bitvector and target queue is not shown for clarity.

Concurrently, the Fetch stage is initiating a read of 8 instructions for the I-Cache, which speculatively increases the fetch coupled count by 8. Note that this access will be squashed due to Decode having detected a taken branch. In parallel, Fetch also processes the oldest FAQ entry, increasing the decoupled count by the number of instructions tracked in the FAQ. Similarly, populating the decoupled bitvector and target queue is not shown for clarity.

In that case, the updated decode coupled count is greater than or equal to the updated decoupled count, meaning that the information of the FAQ entry is not needed to drive the fetcher even if resynchronization happens now. Therefore, the FAQ entry can be popped safely.

Cycle 1: In the second cycle, Decode ignores the instructions flowing from Fetch as those instructions are the fallthrough of the branch that was predicted taken during the first cycle. Concurrently, the fetch coupled count is amended:

- It is *decreased* by 8 (the fetch width, FW in the Figure) since the cache access initiated during the previous cycle was squashed.
- It is *decreased* by 4 since during the previous cycle, the fourth instruction processed by Decode was a taken branch. That is, the cache access initiated two cycles ago only yielded 4 instructions when 8 were expected.
- It is *increased* by 8 since a new cache access for 8 instructions is initiated.

In parallel, a new FAQ entry has become available. Thus, the decoupled count is adjusted from 12 to 22, meaning that the number of instructions covered by the previous and current FAQ entries is now greater than the instructions fetched (or expected to be as we just initiated a cache access) in coupled mode. Therefore, we can switch to decoupled mode immediately. We also need to adjust the instruction count in the current FAQ entry as some of the instructions it references are currently being read from the I-Cache. We note that the adjustment is a fixed quantity: fetch width times Fetch to Decode latency.

Cycle 2: In the third cycle, Decode received the last 8 coupled instructions from the I-Cache and the decode coupled count is increased. In the Figure, the 8 instructions are sequential, therefore, the decode coupled count lines up with the fetch coupled count. This means that resynchronization is done and the bitvectors and target queues can be reset. Concurrently, Fetch initiates a cache access for only two instructions as the FAQ is now driving the fetcher.

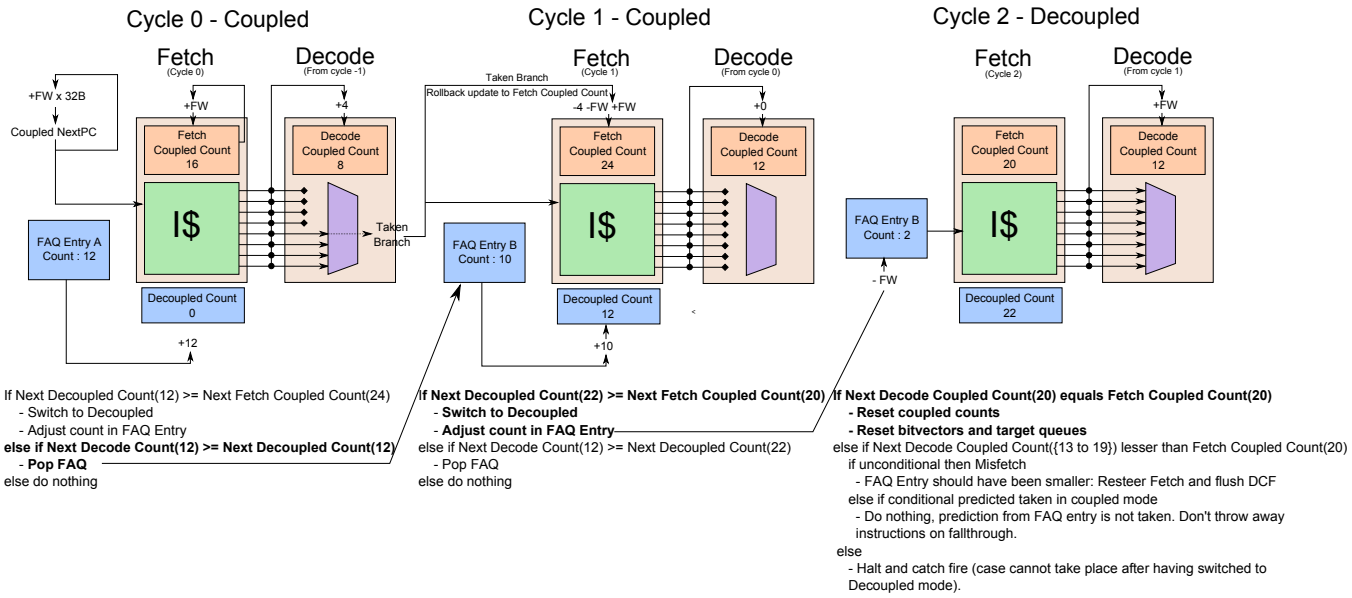

Figure 5: Resynchronizing in U-ELF. The algorithm in Cycle 0 and 1 is executed each cycle during operation in Coupled mode. The algorithm in Cycle 2 is executed during the next Fetch to Decode cycles after switching back to decoupled mode. For clarity, bitvectors and target queues are omitted in this figure.

Note that if Decode had found an unconditionally taken branch in any of the slots during cycle 2, then a misfetch would have occurred and the DCF would have been flushed. This misfetch would have been detected by observing that the fetcher is in decoupled mode but the decode and fetch coupled counts have not been reset, and they do not match although all coupled instructions have been processed by Decode. This is not possible unless the FAQ (and by extension BTB) information is stale. Conversely, in the case of one of the instructions being a conditional branch predicted taken by the coupled predictor and not taken by the decoupled predictor, the bitvectors would catch the mismatch.

D. Correctly Maintaining Branch Prediction State

Virtually all modern branch predictors use global or local state to provide predictions. To speculate efficiently, this state is updated speculatively using generated predictions, such that next predictions can be retrieved using a consistent yet speculative architectural state. Because the state is updated speculatively, it has to be rolled back on mis-speculation. This is usually achieved by checkpointing information prior to speculatively updating the predictor state. At a high level, each dynamic instruction is associated with a checkpoint ID such that the correct checkpoint is restored when a given instruction triggers a pipeline flush (e.g., AMD Zen [16]).

1) Decoupled Predictors State: With Elastic Fetching, it is possible that instructions fetched in coupled mode do not have a checkpoint ID associated with them as DCF may not have generated the corresponding checkpoint yet. Thus, if such an instruction triggers a pipeline flush, it would not know to what state to restore in the DCF predictors. Generally this may be addressed by waiting for the flush-triggering instruction to reach the head of the Reorder Buffer (ROB), guaranteeing that *all* the speculative state is younger and can be thrown away.

However, delaying the flush until the instruction reaches the head of the ROB can have a significant performance impact. Thus, another possibility is to associate each instruction fetched in coupled mode with a coupled checkpoint ID that will be bound as FAQ blocks are consumed. To do so, we can leverage the existing queue of checkpoints that branch instructions claim when they are fetched. Specifically, as branch instructions (more generally, all instructions that require branch prediction state to be checkpointed) are fetched in coupled mode, they obtain an entry in the checkpoint queue as per the regular operation of the fetcher. In decoupled mode, the entry payload would be populated using information coming from the FAQ entry being processed. However, in coupled mode, by construction, there might not be an FAQ entry from where to get the information to checkpoint *yet*. Therefore, we assume that the logic handling incoming FAQ entries *while in coupled mode* has the ability to populate the payload in the checkpoint queue at a later time as needed, even though the entry is allocated as usual. This is possible since the FAQ entry has knowledge about branch location within the block of sequential instructions and is carrying the information to be checkpointed (e.g., pointer to Global History Register bit that was updated). This allows instructions fetched in coupled mode to flush the pipeline as soon as their checkpoint queue entry gets populated, rather

Table I: Applications used in our evaluation.

Benchmark Suite	Applications
SPEC2K6 INT/FP	astar, bzip, calculix, dealII, gamess, gcc, gobmk, gromacs, h264ref, hmmer, leslie3d, namd, omnetpp, parser, perlbench, povray, sjeng, soplex, sphinx3, tonto, wrf, xalancbmk, zeusmp
SPEC2K17 INT/FP Speed	perlbench, gcc, mcf, omnetpp, xalancbmk, x264, deepsjeng, leela, exchange2, xz, bwaves, cactuBSSN, namd, parest, povray, lbm, wrf, blender, cam4, pop2n, imagick, nab, fotonik3d, roms
Server_1 (Large I-side footprint)	subtest_0, subtest_1, subtest_2
Server_2 (Large D-side footprint)	subtest_0, subtest_1, subtest_2

than having to wait until they reach the head of the ROB.

2) Coupled Predictors State: While we do not consider history-based predictors for the fetcher, the mere presence of a Return Address Stack in the fetcher raises the question of state checkpointing for the coupled predictor infrastructure. Fortunately, the coupled RAS, by definition, has to be updated even in decoupled mode. As a result, as long as both coupled and decoupled RAS are matching, the coupled top of stack pointer can be restored using the decoupled checkpoint information. This also happens when a divergence is detected.

For more complex branch and target predictors, however, checkpoints are typically stored in a queue that features a head pointer and a tail pointer to prevent overwriting checkpoints during speculation [16]. Therefore, to track checkpointed state for such coupled predictors, we can implement a dedicated queue. Entries are allocated in the queue by instructions fetched in coupled mode as well as instructions fetched in decoupled mode that need to update or restore the coupled predictor state (e.g., conditional branches to update the GHR). This allows an instruction fetched in decoupled mode to restore the correct state in the coupled predictors, even if younger instructions have been fetched in coupled mode while the decoupled instruction was still in the pipeline.

3) Coupled Predictors Updates: The goal of ELF is to spend most of the cycles in decoupled mode as it is the better performing mode. Moreover, it is likely that the coupled predictors will be much simpler and much smaller than the decoupled predictors. This raises the following question: Should the coupled predictors be updated on all branches, or only the branches fetched in coupled mode? Qualitatively, it makes little sense to allocate entries for branches that will never or very seldom be fetched in coupled mode, hence, this is the approach we used in this paper. However, as pointed out in the previous paragraph, the coupled RAS is updated regardless of the current fetching mode.

E. Impact of I-Cache Hierarchy on U-ELF

Typically, decoupled fetching allows for implementing a slower but pipelined instruction I-Cache. Indeed, since fetch addresses are provided by the FAQ directly, it is possible to fetch across taken branches in consecutive cycles, even if the actual cache access latency is several cycles. Conversely, with a coupled design, one has to wait for the instruction data to come out of the data array to determine if there is a taken branch at a specific location. In other words, every taken branch incurs *n* bubbles with *n* being the cache latency, unless a BTB is implemented in the fetcher specifically to cover the I-Cache latency [7]. Slower caches are compelling as they burn less power and have less stringent timing requirements.

In the context of ELF, and assuming the presence of such a multi-cycle instruction cache, the taken branch penalty can be observed in coupled mode when a taken branch is encountered. Specifically, as the I-Cache latency increases, the potential benefit of ELF will diminish since more and more bubbles will be inserted in coupled mode. To overcome this behavior, ELF requires that the regular fetcher feature a structure that allows to – at least partially – hide the main I-Cache latency on predicted taken branches. In our case, this structure is an actual small but fast L0 instruction cache (24KB, 3-way, 1-cycle latency). It is backed by the main L1 instruction cache (64KB, 8-way, 3-cycle latency). This maximizes the benefits of the coupled mode of ELF, and is a viable design point as commercially available chips use this design [12]. An alternative design could implement a coupled BTB [7] or Next Line Predictor [7], [13].

Lastly, although not explored in this work, using a fast L0 I-Cache renders several optimizations possible. First, as pointed out in Section IV-C1, the coupled predictors we considered are very simple. Combined with the fast L0I, this could permit coupled mode to hide taken branch bubbles by design as both L0I and predictors could be accessed in less than a full cycle. Similarly, it would also be possible to drive the L0I only in coupled mode, to benefit from the quick instruction turnaround, while driving the L1I only in decoupled mode to benefit from the power savings due to the slower access.

F. Variable-length ISA

Elastic Instruction Fetching was devised within our organization in the context of designing high-performance ARMv8 CPU cores. As a result, it is especially well-suited to fixed-length ISAs.

Adapting ELF to variable-length ISAs (e.g., x86) is beyond the scope of this paper. Indeed, ELF is tightly coupled with the fetcher machinery and ensuring correct and efficient operation will depend on low-level details of the microarchitecture that are generally not available.

Nonetheless, we believe that the main requirement is that the conditional coupled predictor be designed such that all branches fetched in a given cycle are predicted. This avoids penalties caused by stalling or by assuming the branch as not taken and mispredicting later. This may involve provisioning many predictions per entry.

We note that these challenges may be addressed gracefully through the presence of pre-decode bits[4] (e.g., to identify

[4]Those bits may be stored in the I-Cache or computed on the fly.

instruction boundaries, instruction types ..etc.) and a micro-op cache [1].

V. Evaluation Framework

A. Methodology

Our proposal aims at increasing sequential performance. Therefore, we focus on single-threaded workloads for our evaluation. Specifically, we consider benchmarks from the following benchmark suites: SPEC2K6 [17], SPEC2K17 [18]. We also use internally-developed, proprietary server-class workloads that are used to drive the design of future generation processors. We anonymized workloads' names, but we can describe them as follows: (*server_1*) is a transaction-based server workload with a significant instruction footprint, while the second suite (*server_2*) is a computation-intensive kernel that pressures branch prediction and data-side memory.

Table I lists the workloads used. All binaries were compiled using GCC 7.1 with -O3 except for *server_2* (-O2). We use 100-million instruction simpoints [19].

Table II: Baseline pipeline configuration.

Frequency	3.4GHz
Branch Target Buffer	16 insts. per entry, up to 2 taken branches and targets (if direct) **L0**: 24-entry, fully associative, 0 cycle (output address can drive input next cycle) **L1**: 256-entry, 4-way assoc., 1 cycle **L2**: 4K-entry, 8-way assoc., 3 cycle
Branch Prediction	**Cond. Branch Pred.**: state-of-art 32KB TAGE predictor (8 tagged tables, 1 cycle)[5][14] **Ind. Target Pred.**: 0.6KB 64-entry L0 indirect target cache (direct-mapped, 12 bit tags, 1 cycle) + 32KB ITTAGE predictor (4 tagged tables, 3 cycles) [20] 0.25KB 32-entry return address stack.
FAQ	32-entry FIFO.
Instruction Prefetch	On L0I idle cycles, prefetch using address from FAQ (older to younger). Up to 4 prefetch requests in flight.
Memory Hierarchy	L0I: 24KB, 3-way asso., 2-way set intlv., 1 cycle; 64B L1I: 64KB, 8-way asso., 3-cycle, 64B L1D: 32KB, 8-way asso., 3-cycle load-to-use, 64B L2: unified, 512KB, 8-way asso., 13 cycles, 128B L3: unified, 16MB, 16-way asso., 35 cycles, 128B Memory: 250 cycles, Advanced Stride-based prefetch
TLB	Perfect
BP1 to FE lat.	3 cycles (BP1, BP2, FAQ)
Fetch through Rename Width	8 instrs per cycle
Issue through Commit Width	9 instrs per cycle (4 ALU including 2MulDiv, 2 LD/ST, 2 SIMD, 1 StData)
ROB/IQ/ LSQ/PRF	256/128/128/256
BP1-EXE lat.	11 cycles
Memory Disambiguation	PC-based filter: violating load-store pair is recorded in the table. When load PC is renamed, load waits for older store if matching store PC was fetched.
Coupled Branch Prediction	2K-entry bimodal, 3-bit ctrs. (1 cycle) – 0.75KB 32-entry return address stack – 0.25KB 64-entry table, 12 bit tags, 1 cycle – 0.6KB
Divergence Tracking	64-entry 2-bit (*taken, branch*) bitvector + 16-entry target buffer for coupled information (**144B**) + 1 *allocated* bit per entry (**10B**). 64-entry 2-bit (*taken, branch*) bitvector + 16-entry target buffer for decoupled information (**144B**) + 1 *allocated* bit per entry (**10B**).

B. Evaluation Environment

The microarchitecture, presented in Sections III and IV, is faithfully modeled in our industry-class, in-house, cycle-level simulator. Since the model was correlated to RTL, the speedups we report, although low, are not within the noise of the simulator and are compelling enough to influence design decisions. This simulator runs ARMv8 ISA binaries in full-system mode and is used to drive the definition of future generation processors.

The parameters of our baseline core aims to model an aggressive out-of-order processor featuring a decoupled fetcher.

Table II shows the baseline core configuration. We point out that our DCF configuration features 3 levels of BTBs. Level 0 can be accessed and its content processed in a single cycle, such that if there is a taken branch, the target PC can be used next cycle (i.e., no taken branch bubble). However, this assumes that the direction/target prediction is also available in the same cycle. Consequently, on an L0 BTB hit, only predictions from the bimodal component of TAGE are used for conditional branches. We assume that if the tagged components of TAGE disagree with the bimodal, we are able to resteer BP1 in the next cycle, i.e., one bubble is introduced (as if it had been an L1 BTB hit). Similarly, on an L0 BTB hit, we assume that predictions from the L0 indirect predictor (direct-mapped partially tagged Branch Target Cache) and the RAS are generated fast enough to hide the bubble. That is, a miss in the L0 table will cause the full delay of the L1 (ITTAGE) access to be visible.

Table II also reports the size of the ELF structures we considered. The total storage cost of U-ELF is smaller than 2KB as the coupled predictors are very small compared to the decoupled ones, to minimize area overhead. Remaining additional logic includes the comparators used to compare bitvectors (128 bit comparisons) and target queues (16 virtual addresses comparisons).

VI. Results and Analysis

A. The Potentially Detrimental Impact of Decoupled Fetching

We begin our experiments with a comparison between the baseline detailed in Table II (*DCF*) and a similar pipeline without a decoupled fetcher(*NoDCF*). Note that the pipeline without DCF does not have any dedicated mechanism to hide the taken branch bubble, yet it still outperforms DCF in a few select cases. This highlights that DCF's ability to hide taken branch bubbles can be vastly outweighed by the additional latency on mispredictions and the BTB misses. Recall that in our framework, DCF may increase performance in two distinct ways:

1) via instruction prefetching using addresses queued in the FAQ.
2) by allowing the taken branch bubble to be hidden thanks to queuing effect in the FAQ as well as the presence of a small L0 BTB. This includes allowing

[5]Predictor access may span more than a single cycle as long as the next PC can be generated during BP2 using the predictions from the access that started in BP1.

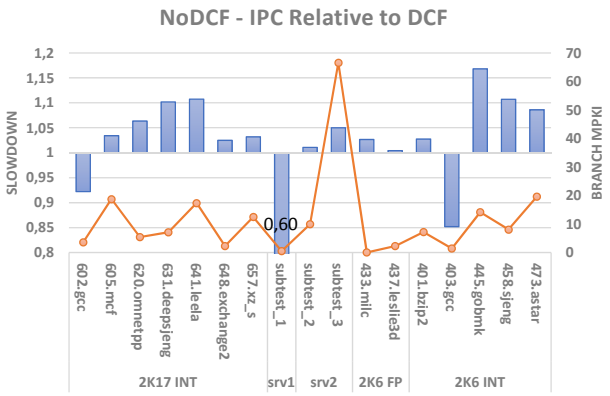

Figure 6: Performance of No Decoupled Fetcher (NoDCF) relative to baseline DCF for workloads that benefit from ELF.

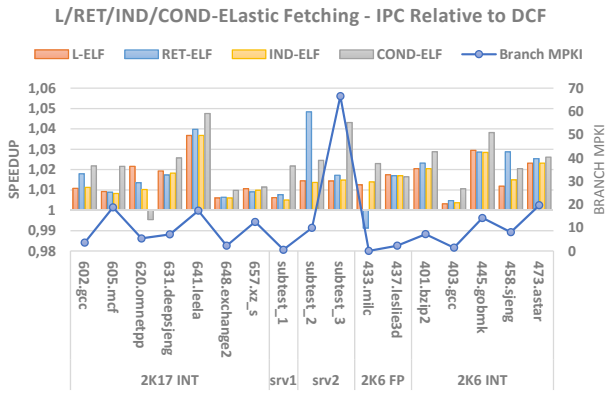

Figure 7: Performance improvement of L-ELF and different variants of U-ELF with respect to DCF.

the fetcher to fetch across a taken branch in a given cycle if the branch and the target map to the two different set interleaves of the L0I-Cache and if the FAQ has the block of the target available [21].

Figure 6 shows the performance and branch direction mispredictions per kilo instruction (MPKI) for benchmarks that benefit from ELastic Fetching. As previously stated, in some cases, performance is better without DCF. The reasons for this trend are that:

1) The pipeline flush penalty is increased by 3 cycles.
2) DCF adds a new feedback loop spanning from Decode to BP1, whose latency is observed on misfetches (BTB misses/BTB stale).
3) DCF introduces the concept of *non taken branch bubble*. That is, a BTB entry may end before 16 instructions because it ran out of space to store branch information. If this happens while missing in the L0BTB but hitting in the L1BTB, speculatively accessing the L1BTB with PC + *max_insts* at the beginning of the next cycle will be incorrect, and BP2 will have to resteer BP1, even in the absence of a taken branch.

In general, workloads for which performance degrades with DCF (primary y-axis greater than 1 in Figure 6) feature a reasonably high branch MPKI, which is one of the two features that ELF aims to mitigate. One especially interesting data point is *server_2 (subtest_3)*, which has the highest branch MPKI yet sees a limited slowdown with DCF. The reason is that this workload is in fact a graph processing workload with a huge memory footprint (several GBs). As a result, while DCF is detrimental due to the high number of branch mispredictions, the main bottleneck is in fact in the memory system.

The second main cause of degradation, BTB misses, is only significant in *server_1* (28.3%, 48.5% and 70.6% hit rate for L0/L1/L2BTB in *subtest_1*). This is expected as *server_1* was specifically considered in this study for its very large instruction footprint. Said footprint is also the reason why DCF is able to improve performance by 40% as a result of issuing I-side prefetches through the FAQ.

B. Different Flavors of ELF

In this section, we report results for L-ELF and and the 3 first U-ELF variants discussed in Section IV. Figure 7 shows the relative IPC of those variants normalized to the baseline.

Note that for IND-ELF, speculation past an indirect branch is only allowed if the branch hits the Branch Target Cache. If not, coupled mode stalls as in L-ELF. In that context, IND-ELF is not able to provide significant improvement over L-ELF, except in *458.sjeng* (1.5% vs 1.1% speedup).

In COND-ELF, to minimize adversarial effects due to wrong path instructions (e.g., useless cache accesses causing useful lines to be evicted), speculation past a conditional branch is only allowed if the 3-bit bimodal counter of the coupled predictor is saturated. We note that even with this optimization, performance degrades in *620.omnetpp* due to the bimodal prediction often being incorrect (+2 MPKI). This calls for a smarter filtering mechanism for allowing speculation past conditional branches, or simply for a better coupled predictor area/power resources permit it.

Finally, RET-ELF shows significant improvement in *Server_2 (Subtest_2)* as the RAS is very accurate and this particular workload relies on recursion. As a result, even though limited, being allowed to speculate past returns only is particularly interesting in very specific cases. In *433.milc*, however, we can observe a 1% slowdown, which is attributable to a single simpoint in which speculating across returns while in coupled mode creates a pathological case for the memory dependency predictor. This triples the number of pipeline flushes due to RAW hazard violations (27K to 95K). We observed a similar behavior for *server_1 (subtest_1)* in U-ELF.

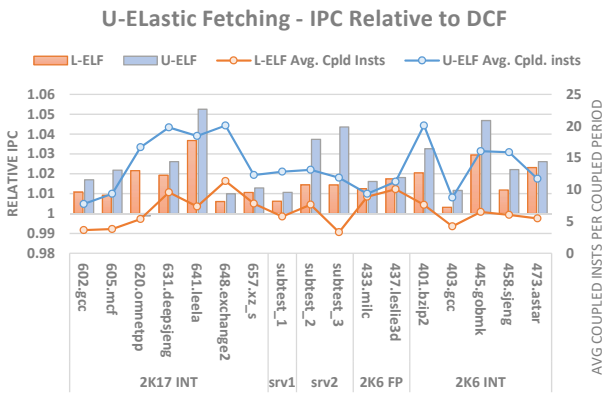

Figure 8: Performance improvement of L-ELF and U-ELF, as well as average number of instructions fetched during a run in coupled mode.

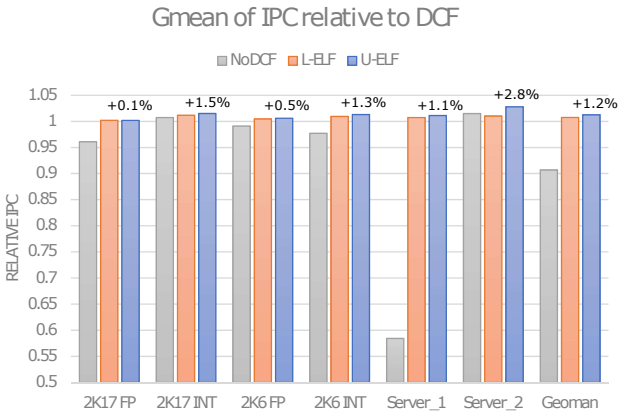

Figure 9: Speedup (geomean) of NoDCF, L-ELF, U-ELF relative to the baseline DCF configuration.

C. Unlimited ELF

Finally, we consider the final variant of ELF where all types of predictors are enabled (bimodal, RAS and indirect table), U-ELF. Figure 8 summarizes speedup over the baseline DCF configuration for L-ELF and U-ELF, and also reports the average number of instructions fetched from the time coupled mode starts until the fetcher switches back to decoupled mode.

In general, as more instructions are fetched in coupled mode, more latency is hidden and performance increases. In particular, performance improves by up to 3.6% and 5.2% in *641.leela* for L-ELF and U-ELF, respectively. Interestingly, the speedup of U-ELF in *server_1 (subtest_1)* remains modest (2.2%) even though this workload is often missing the 3 levels of BTB. The reason is that although ELF is theoretically able to hide the BTB miss penalty, a BTB miss often entails an instruction cache miss. In that event, the benefit of ELF comes from issuing the request to the L1I/L2 Cache earlier rather than from actually fetching instructions. Fully hiding the BTB miss penalty could be achieved through a mechanism such as Boomerang [11].

It seems like in order to extract more performance from ELF, better coupled predictors and/or better confidence mechanisms are needed. Indeed, if we consider *Server_2 (Subtest_2)*, we can observe that the speedup with U-ELF is 3.7%. However, with RET-ELF, speedup was 4.8%. The reason is that in this benchmark, while the coupled bimodal branch predictor is actually quite accurate (+0.1 MPKI), incorrect coupled predictions are sufficient to displace useful data from the L1D on the wrong path. Similarly, in *620.omnetpp*, L-ELF performs better than U-ELF by virtue of the bimodal predictor being inaccurate (+2 MPKI).

Nonetheless, with ELF, significant improvements can be obtained over a baseline with DCF. Moreover, despite the limited numbers of workloads strongly benefiting from ELastic Fetching, this new microarchitectural technique slightly improves performance on average for our benchmark suites, as shown in Figure 9.

VII. Conclusion

In this work, we proposed ELastic Fetching, a fetch mechanism that leverages a decoupled fetcher to implement well-known fetch optimizations (e.g., instruction prefetching), but that falls back to a more conservative coupled scheme when the pipeline is restarted, on a flush. This allows to hide part or all of the additional latency incurred by the DCF stages (3 stages, in our framework) as well as part of the penalty caused by BTB misses.

We detailed two different variants of ELF. First, Limited ELF, a low complexity design that only fetches sequential instructions in coupled mode, but yields moderate improvements, up to 3.6% (0.7% geomean).

We also presented results for different trade offs regarding what type of control decision L-ELF is able to speculate past, and found that in general, speculating past conditional branches (COND-ELF) yields the most benefits. However, given the predictor (3-bit bimodal) and the filtering mechanism (counter has to be saturated) we used, COND-ELF is also a riskier scheme as it may decrease performance in workloads sensitive to wrong-path effects such as spurious data cache accesses. As a result, future work may investigate the use of a better conditional predictor and/or filtering scheme to further improve COND-ELF and specifically ensure that performance does not decrease. The other variant, RET-ELF and IND-ELF, generally show less improvement

even though short recursive algorithms benefit significantly from RET-ELF.

Finally, we evaluated a more complex scheme, Unlimited ELF, that will allow coupled mode to fetch past any control-flow decisions, effectively combining COND-, IND- and RET-ELF. U-ELF yields substantial improvements in workloads that generally feature high branch MPKI, up to 5.2% (1.2% geomean). Therefore, as IPC is becoming more and more critical to new microarchitectures due to the end of Dennard Scaling and the slowdown of VLSI scaling, we consider ELF to be a worthy tool to add to the microarchitect's toolbox.

ACKNOWLEDGMENT

This research was supported by Qualcomm Technologies, Inc. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of Qualcomm Technologies, Inc.

REFERENCES

[1] M. Clark, "A new x86 core architecture for the next generation of computing." Presented at IEEE Hot Chips 2016, 2016.

[2] B. Burgess, "Samsung Exynos M1 Processor." Presented at IEEE Hot Chips 2015, 2016.

[3] C. Jacobi and A. Saporito, "The Next Generation IBM Z Systems Processor." Presented at IEEE Hot Chips 2017, 2017.

[4] Anandtech, "ARM's Cortex-A76 CPU Unveiled: Taking Aim at the Top for 7 nm." https://www.anandtech.com/show/12785/arm-cortex-a76-cpu-unveiled-7nm-powerhouse/2, 2017.

[5] G. Reinman, B. Calder, and T. Austin, "Optimizations enabled by a decoupled front-end architecture," *IEEE Transactions on Computers*, vol. 50, Apr 2001.

[6] J. Losq, "Generalized history table for branch prediction (in pipeline computers)," *IBM Tech. Disclosure Bull.;(United States)*, vol. 1, 1982.

[7] J. K. F. Lee and A. J. Smith, "Branch prediction strategies and branch target buffer design," *IEEE Computer*, vol. 17, Jan 1984.

[8] AMD, "Software Optimization Guide for AMD Family 17h Processors, 2.8.1.2." support.amd.com/TechDocs/55723_SOG_Fam_17h_Processors_3.00.pdf, 2017.

[9] G. Reinman, "Hardware optimizations enabled by a decoupled fetch architecture," Ph.D. dissertation, University of California, San Diego, 2001.

[10] J. Stark, P. Racunas, and Y. N. Patt, "Reducing the performance impact of instruction cache misses by writing instructions into the reservation stations out-of-order," in *Proceedings of the Annual International Symposium on Microarchitecture*, Dec 1997.

[11] R. Kumar, C. C. Huang, B. Grot, and V. Nagarajan, "Boomerang: A metadata-free architecture for control flow delivery," in *Proceedings of the International Symposium on High Performance Computer Architecture*, Feb 2017.

[12] T. Speier and B. Wolford, "Qualcomm Centriq 2400 Processor." Presented at IEEE Hot Chips 2017, 2017.

[13] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides, "Design tradeoffs for the alpha ev8 conditional branch predictor," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, 2002.

[14] 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. New York, NY, USA: ACM, 2011.

[15] E. Borch, S. Manne, J. Emer, and E. Tune, "Loose loops sink chips," in *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*. IEEE Computer Society, 2002.

[16] AMD, "Software Optimization Guide for AMD Family 17h Processors, 2.8.1.6." support.amd.com/TechDocs/55723_SOG_Fam_17h_Processors_3.00.pdf, 2017.

[17] Standard Performance Evaluation Corporation, "The SPEC CPU 2006 Benchmark Suite," in *http://www.spec.org*.

[18] Standard Performance Evaluation Corporation, "The SPEC CPU 2017 Benchmark Suite," in *http://www.spec.org*.

[19] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," in *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '03. New York, NY, USA: ACM, 2003. [Online]. Available: http://doi.acm.org/10.1145/781027.781076

[20] A. Seznec, "A 64-kbytes ittage indirect branch predictor," in *Third Championship Branch Prediction*, ser. JWAC-2, 2011.

[21] T. M. Conte, K. N. Menezes, P. M. Mills, and B. A. Patel, "Optimization of instruction fetch mechanisms for high issue rates," in *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, 1995.