Wrong-Path-Aware Entangling Instruction Prefetcher

Alberto Ros, *IEEE Senior* and Alexandra Jimborean

***Abstract*—Instruction prefetching is instrumental for guaranteeing a high flow of instructions through the processor front end for applications whose working set does not fit in the lower-level caches. Examples of such applications are server workloads, whose instruction footprints are constantly growing. There are two main techniques to mitigate this problem: fetch directed prefetching (or decoupled front end) and instruction cache (L1I) prefetching.**

This work extends the state-of-the-art Entangling prefetcher to avoid training during wrong-path execution. Our Entangling wrong-path-aware prefetcher is equipped with microarchitectural techniques that eliminate more than 99% of wrong-path pollution, thus reaching 98.9% of the performance of an ideal wrong-path-aware solution. Next, we propose two microarchitectural optimizations able to further increase performance benefits by 1.8%, on average. All this is achieved with just 304 bytes.

Finally, we study the interplay between the L1I prefetcher and a decoupled front end. Our analysis shows that due to pollution caused by wrong-path instructions, the degree of decoupling cannot be increased unlimitedly without negative effects on the energy-delay product (EDP). Furthermore, the closer to ideal is the L1I prefetcher, the less decoupling is required. For example, our Entangling prefetcher reaches an optimal EDP with a decoupling degree of 64 instructions.

***Index Terms*—Instruction prefetching, processor front-end, performance, energy efficiency.**

I. Introduction

A plethora of studies converge to the conclusion that modern workloads for servers, Cloud computing, etc significantly exceed the L1 instruction cache (L1I), leading to numerous processors stalls [11], [22] Instruction prefetching is the prime mechanism to address this problem, by caching in advance precisely the required instructions. Instruction prefetching can be initiated on L1I accesses [9], [14], [15], [20], [23], [31], [32], [37], [42], [45], or following the predicted execution path ahead of instruction fetch [19], [24], [25], [30], [33]–[35], namely decoupled front end [33] and also known as fetch-directed prefetching (FDP) [34]. FDP decouples the computation of the next instruction address from the instruction fetch through the Fetch Target Queue (FTQ). The FTQ records the generated instruction addresses on the predicted execution path, which are then used to issue prefetch requests. The larger the FTQ is, the more ahead can run the predicted execution path, so a larger decoupling degree is employed.

While the two mechanisms were sought to be complementary [21], latest developments and optimizations push both approaches to their limits, in an attempt to eliminate instruction misses and approach the ideal L1I behavior. Recent

A. Ros and A. Jimborean are with the Computer Engineering Department, University of Murcia, 30100 Murcia, Spain. E-mails: aros@ditec.um.es, alexandra.jimborean@um.es

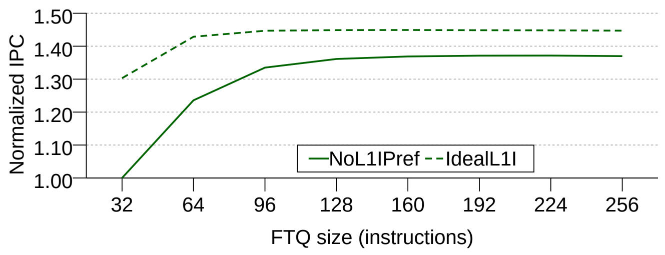

Fig. 1. Effect of FDP without LII prefetcher and with an ideal L1I prefetcher

work shows that FDP can achieve good performance provided that it has enough capacity to store the prefetched instructions and that it fetches instructions from the correct execution path [19], [21], [24]. This would translate to (1) a large FTQ to ensure enough capacity, (2) a perfect direction branch prediction and (3) a perfect Branch Target Buffer (BTB).

We complement the literature survey with a sensitivity study in which we analyze the interplay between L1I prefetching and FDP when varying the size of the FTQ. We integrate a two-level branch prediction scheme [4] with the best performing branch predictor [40] and indirect target predictor [39] at the second level, and model wrong-path (WP) execution (more details about our methodology can be found in Section V). The average IPC for the workloads employed in this work is illustrated in Figure 1 where we varied the FTQ size from 32 to 256 instructions (shown on the Ox) and evaluated a baseline system without a dedicated L1I prefetcher (*NoL1IPref*) and an ideal L1I prefetcher (*IdealL1I*). IPC is normalized to *NoL1IPref* with a 32-instruction FTQ. As expected, increasing the fetch decoupling improves performance when no L1I prefetcher is employed, saturating at about 160 instructions. An ideal L1I prefetcher however saturates at a lower FTQ size (around 96 instructions) and can push performance an additional 5.6%.

Increasing the FTQ size also comes with extra energy consumption since more instructions are fetched. Figure 2 shows the wrong-path effects when varying the FTQ size. As the FTQ grows, the percentage of wrong-path instructions inserted in the pipeline over committed instructions increases from 56% for a 32-entry FTQ to 155% for 256-entries. The majority of wrong-path instructions only reach the fetch stage, yet, the several folds increase from 32 to 256 entries leads increasing energy waste. We evaluated the full-system energy expenditure and observed that it increases by 6% when moving from 32 to 256 FTQ entries because of extra branch predictions, L1I accesses, and even main memory accesses (due

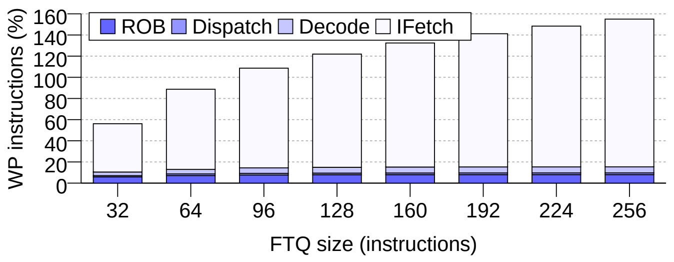

Fig. 2. Percentage of wrong-path instructions inserted in the pipeline over committed instructions and the stage that they reach (without L1I prefetcher)

mostly to accessing wrong-path instructions). Hence, simply increasing the FTQ size is not sufficient and complementing FDP with a precise, tailored and accurate L1I prefetching is essential to effectively eliminate L1I misses while keeping energy expenditure at bay.

This work extends our previous Entangling instruction prefetcher [36] and aims to bring Entangling closer to a ready-to-be-deployed solution. First, since Entangling was not designed nor evaluated in presence of wrong-path instructions, we propose simple extensions to avoid training the prefetcher for wrong-path instructions. Then, we propose a set of microarchitectural optimizations that push Entangling closer to an ideal L1I prefetcher. Finally, this paper sheds light on the L1I prefetcher's synergies with FDP.

This work makes the following contributions:

- We extend the Entangling instruction prefetcher, and show that it can easily deal with wrong-path execution with minor additions (accounting for a total of 280 bytes). As the original version of Entangling is wrong-path agnostic, that implementation suffers from wrong-path pollution, and in consequence cannot be deployed out-of-the-box in a real system. By adding such lightweight and simple support for wrong path, we are able to eliminate more than 99% (on average) of the wrong-path information learned by the original Entangling proposal, thus reaching 98.9% of the performance of an ideal wrong-path-aware solution.
- We present two novel, simple, yet effective optimizations that using only 24 extra bytes improve further the performance of the Entangling prefetcher by 1.8%, on average.
- In the view of integrating Entangling in a real system, we study the interplay of L1I prefetchers with FDP and show that they must work in tandem for optimal energy-delay product (EDP). Furthermore, the closer to ideal is the L1I prefetcher, the less decoupling is required. For example, our optimized Entangling prefetcher reaches an optimal EDP with a decoupling degree of maximum 64 instructions.

II. Background

This section describes the two instruction prefetching techniques more relevant to this work: Fetch Directed Prefetching (FDP) [34], driven by the branch prediction engine, and the Entangling Instruction Prefetcher [36], trained with L1I accesses.

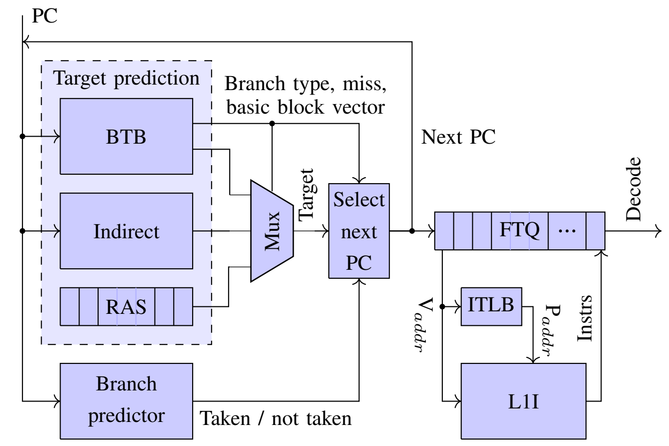

Fig. 3. Fetch directed instruction prefetching

A. Fetch Directed Prefetching

Typical L1I caches implemented in high-performance systems have a fetch latency of 4 cycles [6], [18]. Thus, waiting until an instruction is fetched to determine whether it is a branch and then predict if the branch is taken or not would dramatically serialize the processor's front end. To avoid this serialization, the next instruction to fetch should be known before the current instruction is fetched. This is accomplished by using the current instruction address, i.e., the program counter (PC), for computing, in parallel with the L1I accesses, the next addresses to be fetched. Generating instruction addresses ahead of fetch is commonly known as Fetch Directed Prefetching (FDP) [34].

Central to the FDP mechanism, depicted in Figure 3, is the FTQ. Each FTQ entry contains three fields: the status, the instruction address, and the corresponding instruction (if already fetched). The FTQ is fed by the branch prediction engine (shown on the left) which computes the address of the next instruction (next PC). Instruction fetch is shown below the FTQ. Next, we detail these two components.

1) The branch prediction engine: The mechanism for selecting the next PCs takes as input the outcome of the *target prediction* selected by a multiplexer (*Mux*) and the *branch* (direction) *prediction* which are accessed in parallel. The next PCs are then pushed to the FTQ and the last one of these next PCs is used as input for the next cycle prediction.

The branch target prediction consists of several predictors: the branch target buffer (BTB), the indirect branch target predictor (indirect), and the return address stack (RAS). The most critical predictor is the BTB since, apart from providing branch targets, it offers information about the next sequential instructions and, in case those instructions are branches, their type [44]. To this end, each BTB entry holds information for a maximum number of bytes/instructions and a maximum number of targets (e.g., up to two targets if the first branch in the sequence of instructions is conditional as in the AMD Zen cores [7]). The branch type then dictates which of the three target predictors should provide the target address through the target Mux: (1) the BTB is selected for direct branches, which do not change the target address, (2) the indirect target predictor for indirect branches, where the target address depends on a register value, and (3) the RAS for return branches, for which the return address register value can be easily identified using a stack of calls. The RAS prediction does not depend on the PC, so the top of the RAS is always given as input for the Mux.

Apart from the target, the branch direction (taken vs. not taken) is also necessary to compute the next PC. To this end, the branch predictor generates a prediction for the next conditional branches (e.g., two [7]). In case the target predictor fails to provide target addresses (e.g., on a BTB miss), the branches are considered as not taken until the target is computed. On a predicted taken branch, no more instruction addresses are generated in that cycle, and the target is used as input for the next cycle. On branches predicted as non-taken next instruction addresses are generated, until either a branch without target is found or the last instruction in the BTB entry is reached [7].

Re-steering fetch. Large BTBs and highly-accurate predictors incur several cycles to deliver their outcome. To avoid stalling fetch during this time, modern processors complement these large predictors with simpler predictors and smaller BTBs that can be accessed within one cycle, but deliver lower accuracy [6], [7]. In this two-level prediction setting, the highly-accurate predictors are trusted over the 1-cycle predictors, so in case of disagreement, the following instructions are squashed and fetch is redirected.

Mispredictions can also be detected at decode stage when the target provided by the branch instruction does not match the predicted target. As before, the subsequent instructions are squashed and the branch prediction engine is redirected.

Finally, mispredictions are also detected at the execution stage when the branch target and direction is computed. Although state-of-the-art predictors are very accurate and mispredictions are rare (≈ 3 branch MPKI [40]), these events have a considerable impact on the processor performance as they are detected only when the branch executes.

2) Instruction Fetch: As soon as an address (next PC) is inserted in the FTQ, the instruction fetch starts (provided that the L1I and ITLB have available ports). Since L1I caches are commonly virtually-indexed, physically-tagged (VIPT), the access to the ITLB and L1I is performed in parallel. Once the head of the FTQ has fetched the instruction, it can be sent to the decode stage.

If the ITLB and the L1I were ideal, then a FTQ size equal to the instruction's width $\times$ the number of cycles required to fetch an instruction could avoid bubbles, thus fetching at full speed. In the absence of ideal caches, larger FTQs compensate for ITLB and L1I misses by boosting the FDP's prefetching capabilities, but run the risk to fetch too much on the wrong-path. Therefore, the FTQ size has to be carefully adjusted for maximizing performance and efficiency.

B. The Entangling Instruction Prefetcher

L1I prefetchers and FDP have been proposed in a bid to more accurately prefetch instructions. There are a multitude of L1I prefetching approaches, as mentioned in the introduction. In this work we focus on the state-of-the-art Entangling Instruction Prefetcher [36], which won the 1st Instruction Prefetching Championship (IPC-1) [3]. The key concept in Entangling is *timeliness*, and it is achieved by computing the prefetch latency and by associating to each cache line missing in the L1I a corresponding cache line that should trigger the prefetch request, such that the target cache line is cached on time. The prefetch triggering cache line is called *source*, and the target of the prefetch cache line is called *destination*, while the pair built by the two is called *entangled pair*. Figure 4 depicts an overview of Entangling.

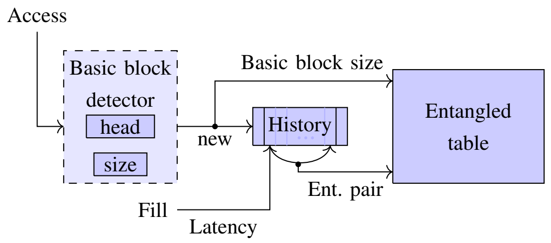

Fig. 4. Entangling prefetcher operation

1) Training: Learning is built on two pillars:

1. Basic block detection. Entangling detects *runtime basic blocks*, namely sequences of cache lines consecutively accessed. In other words, conditional branches that are not taken are embedded in the current basic block. Entangling stores the runtime basic block of maximum size.

2. Entangling pairs formation. Computing the latency of each prefetch instruction and pairing sources and destinations to build entangling pairs is the core of Entangling. To this end, the prefetcher records in the History table L1I accesses that are heads of basic blocks, i.e., the first cache line of a basic block (see basic block detector in Figure 4). Next, for each L1I cache miss, the *latency* of fetching the requested line is computed. Finally, Entangling tracks back in the History table and identifies a source instruction that executed at least *latency* cycles earlier than the requested instruction (i.e. the destination). The source and destination are entangled and recorded in the Entangled table.

This component ensures the timeliness of the approach, with a high accuracy (72%) and coverage (87%). Latency variations are also accounted for by means of confidence counters that are used to adjust the prefetching. To reduce the storage requirements, Entangling only entangles heads of basic blocks. Furthermore a compression scheme is designed that encodes each destination as a delta of the source instruction, to further reduce the storage demands. More precisely, the destination bits only contain the $signifB$ least significant bits of the destination, starting from the most significant bit that differs from the source. Considering that the distance between source and destination (in cache lines) is typically small, the destinations can be highly compressed. Based on the $signifB$ required to encode a destination, one can compute the compression "mode". The mode indicates the number of destinations that can be associated to a certain source ($1 \leq$mode≤ 6). All destinations must be encoded following the same mode.

2) Prefetching: On each cache access, the list of entangled destination for the current access is checked. If the currently accessed cache line corresponds to a source, two sets of prefetches are triggered. First, the basic block of the current cache line (i.e. the source) is fully prefetched. This leverages spatial locality and subsumes a next line prefetcher. Next, for each destination, the entire basic block of the destination instruction is prefetched, to fulfill timeliness.

In the presence of hard-to-predict branches, Entangling naturally prefetches instructions from both paths. In contrast, FDIP would fetch from one path only, which could be the wrong path with a not-too-low probability. When finally re-steering to the correct path, FDIP would suffer stalls when the correct-path instructions are not in the L1I.

III. Support for wrong path

This section proposes a set of microarchitectural solutions to deal with the effects of wrong-path accesses on the L1I prefetcher. As mentioned in the introduction, our target prefetcher is the Entangling instruction prefetcher [36], which was designed without considering the effects of wrong-path execution, other than a discussion on a potential solution.

Wrong-path execution affects the prefetcher behaviour with instructions that will be eventually squashed. First, the prefetcher is trained with those wrong-path accesses. For Entangling, wrong-path training refers to building basic blocks and entangled (correlated) pairs during the execution of the wrong path, and store them in the Entangled table. Second, the prefetcher triggers prefetch requests as a consequence of wrong-path accesses. For Entangling, it means to access the Entangled table looking for a source and prefetch based on information that was possibly created during correct-path execution, but that may not be useful as the access initiating the prefetch will be squashed. Prefetch requests triggered by wrong-path accesses may already be far in the memory hierarchy when the squash occurs and therefore canceling them is difficult.

A typical solution is to train the prefetcher only on correct-path (e.g., at commit [14]). While this is viable for many prefetchers, it is not a suitable solution for Entangling, which *Entangles* accesses on L1I fills based on the time at which previous accesses occurred. Moving this functionality to the commit stage would require to propagate large amounts of timestamps from the front-end to the back-end since entangling basic blocks based on commit timestamps would be inaccurate. A second solution is to update the prefetcher's structures upon a processor squash, thus canceling the training done on the wrong path, similar to how the branch history is recovered from a squash.

We first quantify the effects of wrong-path execution on the Entangling prefetcher, considering either training or triggering prefetch requests, and our findings are promising. Figure 5 shows this analysis as the FTQ size (i.e., FDP decoupling) increases. The plot shows in gray the Entangling version unaware of wrong-path accesses [36] (*Entangling*), an Entangling version that does not trigger prefetches on wrong-path accesses thanks to oracle knowledge (*Entangling-OraclePref*),

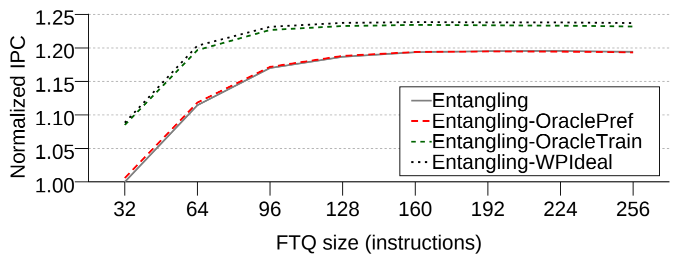

Fig. 5. Impact of wrong path on the Entangling prefetcher in FDP.

an Entangling version that never trains the prefetcher on wrong-path accesses thanks to oracle knowledge (*Entangling-OracleTrain*), and an Entagling version that fully ignores (neither train, nor trigger) wrong-path accesses (*Entangling-WPIdeal*). *Entangling-OracleTrain* differs from *Entangling-WPIdeal* in the fact that *Entangling-OracleTrain* can trigger prefetch requests due to wrong path accesses, if the information used for prefetching (i.e. the basic blocks and entangled pairs) was built on the correct path. This can happen for instance if the same path was executed previously as a correct path, thus populating the Entangling table. *Entangling-WPIdeal* does not trigger such prefetch requests.

The key observation is that most of the negative impact of the wrong-path execution stems from the prefetcher's training, since the curve representing Entangling trained only for accesses on the correct path (*Entangling-OracleTrain*) approaches an ideal version where the wrong path is fully neglected (*Entangling-WPIdeal*). In short, prefetches triggered by wrong-path accesses have a negligible negative impact.

Driven by these findings, we describe three simple, but effective microarchitectural techniques to annihilate on branch mispredictions wrong-path training, which would otherwise pollute the prefetching structures, evicting useful training, and ultimately polluting the cache. We call the resulting solution Entangling-WPA (wrong-path aware). Since in the Entangling prefetcher, prefetch requests are triggered by searching basic block sizes and entangling destinations in the Entangled table, our ultimate goal is that if this information (basic block sizes and entangling pairs) is created by any access in the wrong path, it will never be stored in the Entangled table. We achieve this goal by simply informing the prefetcher about squash events.

A. Updating runtime basic block information

Branch prediction, and handling the wrong-path thereof, has direct implications on the correct identification of runtime basic blocks, which are essential for the effectiveness of the Entangling prefetcher, as described in Section II-B. As the application starts its execution, neither the branch predictor nor the prefetcher are trained. Therefore, branches are initially predicted as not taken, until the branch predictor is warmed up and sufficiently trained. In parallel, the Entangling prefetcher already uses this inaccurate information for identifying the basic blocks. Without any branch being predicted as taken, the wrong-path leads to the building of very large (and incorrect) runtime basic blocks, which will be then entirely prefetched

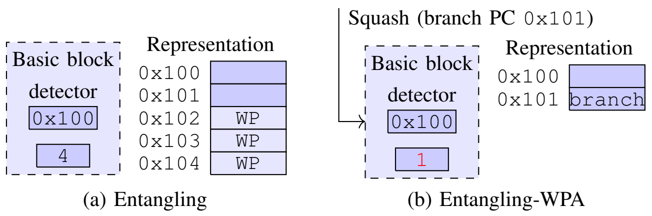

Fig. 6. Basic block size update

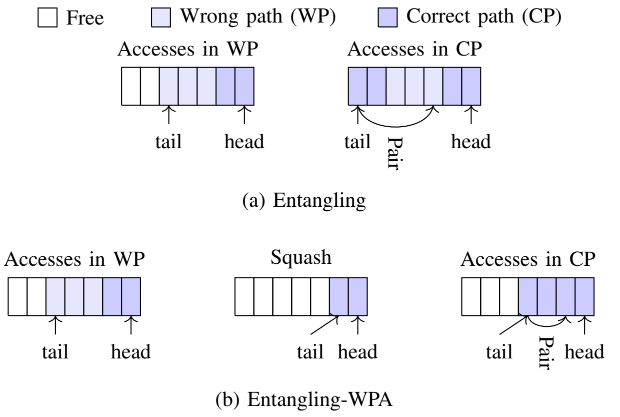

Fig. 7. Squash of the History

evicting useful data from the cache. A similar behavior is observed also when the application enters new phases, or simply paths that have not been explored before.

To circumvent this drawback, the first step in updating the prefetcher's bookkeeping information upon a branch misprediction is to adjust the size of the current basic block. This turns out to have also the highest impact in reducing wrong-path pollution and regaining performance, as shown in Section VI. Thus, upon a squash, Entangling is informed about the PC of the mispredicted branch. If the branch PC belongs to the current basic block being processed, i.e., its value is in between the head of the basic block and the head of the basic block plus the size of the basic block, then the basic block size is updated to the distance –in cache lines– from the head of the block to the branch. Figure 6 depicts this scenario where on the left appears the status of the basic block detector, and its representation in consecutive cache lines (indicating those only accessed in the wrong path), and on the right appears the status after the squash, updating the basic block size to 1.

This step ensures that the correct basic block information is propagated to the Entangled table and only the correct-path instructions are subsequently prefetched.

B. Squashing the history buffer

Another essential data structure of the Entangling prefetcher is the History, used for identifying the entangled pairs of cache lines. However, the History is also vulnerable to the wrong path. Wrong-path accesses recorded in it can incorrectly serve as entangled sources, as shown in Figure 7a (and thus Entangling may never issue the expected prefetch requests) or as entangled destinations (and thus prefetching unnecessary data). This section addresses the first problem by cleaning up the History, i.e. by flushing wrong-path entries upon a branch misprediction, as shown in Figure 7b. For this, we extend the History with an extra field per entry that holds the sequence number (or ID) of the first instruction accessing the recorded basic block. On a squash, the processor informs the prefetcher about the ID of the mispredicted branch. The prefetcher then flushes each entry in the History table whose ID is higher than the mispredicted branch. Additionally, each basic block in the History is updated following the mechanism described in the previous subsection.

C. Delaying insertion in the Entangled table

Despite flushing the History table and correcting the block size on a branch misprediction, the Entangled table is still polluted during the execution of the wrong path when (1) a new basic block is detected and (2) when a new entangled pair is built, since they are inserted in the Entangled table as soon as they are detected (Figure 4).

Since "cleaning up" the Entangled table is difficult, we opt for keeping the basic block information and the new entangled pairs in the smaller History table, which is squashed on each branch misprediction. As there is already information about the basic block sizes in the History, it is only necessary to add a new field to record the corresponding source of each destination. We opt for storing this information along with the destination because, on a History squash, pairs with wrong-path destination and correct-path source will be removed.

Basic block sizes and entangled sources reside thus in the History and are squashed each time wrong-path is detected. They enter the Entangled table when evicted from History due to its limited capacity. Eviction is performed in FIFO order, thus increasing the likelihood of only correct-path data to be inserted in the Entangled table. It could happen however that wrong-path basic blocks and entangled pairs reach the Entangled table if the execution continues in the wrong-path for a very long time. Yet, given the high accuracy of state-of-the-art branch predictors, this situation is unlikely and would cause negligible pollution.

D. Complexity and memory requirements

The extensions proposed for mitigating the pollution caused by wrong-path accesses require simple extra logic and only lightweight additions to the History table. Each entry is extended with a sequence number field (*ID*) and a source entangled field (*source*). For the processor configuration employed in our experiments (see Section V), 12 bits suffice to store the instruction ID field while accounting for rollbacks. The *source* field employs the full 58 bits of the cache line address. The format of an entry in the History and each field's size is depicted in Figure 8, with the newly added fields highlighted in blue. Hence, considering a 32-entry History, the total extra storage requirements amounts to just 280 bytes.

IV. Optimizing Entangling

By analyzing the behavior of the Entangling prefetcher under different situations, we identified that it performs sub-

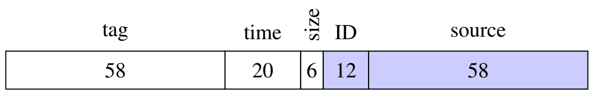

Fig. 8. History entry format for WPA

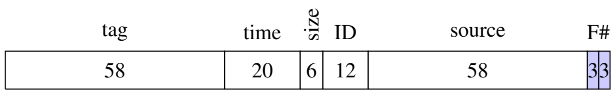

Fig. 9. History entry format for the optimizations

optimally when (1) selecting a timely source for each destination among several candidates and (2) filtering overlapping (redundant) prefetch requests. In this section we present two microarchitectural techniques to address these limitations and improve the Entangling's performance.

A. Selecting optimal source-entangled

Problem: For building each entangled pair, the Entangling prefetcher searches a suitable source instruction positioned *at least latency* cycles earlier than the destination instruction (where *latency* is the latency of the destination instruction). However, such a source is not unique. Since the Entangled table is a limited resource, the Entangling prefetcher already employed an optimization for maximizing its utilization [36]. Namely, the prefetcher performs two searches in the Entangled table, identifies the two youngest potential timely sources, and, when possible, selects the one that still has space in the list of destinations.

For example, consider two candidate source instructions, src_1 and src_2, where src_1 uses the compression mode 3 (i.e. can accept three destinations) and holds two destinations, while src_2 uses the compression mode 6 and holds five destinations. The new destination would have to be compressed in mode 4 with respect to both sources. But choosing one source or another yields a very different utilization of the Entangled table. If src_1 is chosen, we do not need to evict any destination. But if src_2 is chosen, we would need to evict two destinations to accommodate the new one, as the mode of src_2 would be set to 4.

Solution: Our goal is thus to increase the Entangled table's utilization. Nevertheless, selecting the optimal source is a costly operation because the Entangled table is the largest structure of the prefetcher and snooping it often entails significant costs.

To avoid snooping the Entangled table frequently, we propose to hold in the History table the compression mode of each inserted cache line along with the *number* of destinations already entangled. This information can be retrieved from the Entangled table as it is accessed on every L1I access. Going back to the previous example, the prefetcher can now easily select the optimal source, by computing the compression mode with respect to each potential source and choosing the one that maximizes utilization. Iterating the much smaller History table is faster than iterating the Entangled table and translates directly to performance improvements as more Entangled destinations can be held in the Entangled table.

Our algorithm for selecting the best source among the candidates works as follows. Each timely source candidate whose number of destinations is smaller than its compression mode (i.e., has space for accepting new destinations) computes the compression mode of the new destination if pairing with it. The optimal source is then chosen according to the following priorities: first, the sources that still have space and whose compression mode matches the one of the new destination; second, the ones with space to hold the new destination; third; the ones that are not present in the Entangled table (encoded in the History with compression mode 0), since we prefer to evict an old source (based on the replacement policy, thus making space for a new source) than a useful destination. In case of a tie, i.e. multiple sources fulfill the same criteria, the youngest source is preferred.

Once the source is selected, the new compression mode is computed (minimum of the current compression mode and destination compression mode) and updated, and the number of destinations of that entry is increased.

B. Keeping useful destination-entangled

Problem: As the Entangling prefetcher builds entangled pairs on each execution path, the same cache line (head of basic block) can be prefetched multiple times, by different sources. This can be beneficial for instance when the same basic block is reachable from two convergent paths. But in other situations, the prefetchers may be redundant (overlapping requests). For example, when the two sources are on the same execution path. This could happen for a destination with varying latencies, that gets entangled each time with a different source. While the confidence counters help to identify and remove the non-timely prefetches, if the redundant destinations are timely the counter is not decreased, thus occupying unnecessary entries.

Solution: We identify redundant prefetch requests upon a merge either in the Prefetch Queue (PQ) or in the Miss Status Holding Register (MSHR). In those cases, the confidence of the younger source is decreased. If this situation arises frequently, the counter will reach 0 and the redundant destination will be removed, leaving more space in the Entangled table for useful pairs.

C. Complexity and memory requirements

The presented optimizations require small additions to the History entries, on top of the solution derived from the previous section. In particular, each entry is now extended with two new fields: the format (F) of its destinations in the entangled table and the number (*#*) of such destinations. Since Entangling contemplates six different formats (from 1 to 6) each of them containing a maximum number of destinations as the value of the format, each field is comprised of 3 bits (Figure 9, again with additions in blue). For our 32-entry History, the added storage requirements is just 24 bytes.

TABLE I
BASELINE SYSTEM CONFIGURATION

Processor decoupled front-end: Two-level branch prediction: 1-cycle branch prediction (3K-targets 3-way L1 BTB, 3K-targets 3-way tagged indirect target array, 32-entry RAS, hashed perceptron predictor [43]); 2-cycle branch prediction (8K-targets 8-way L2 BTB, 64KB ITTAGE indirect target predictor [39], 64KB TAGE-SC-L branch predictor [40]); 32KB, 8-way uop cache; from 24-instruction to 256-instruction FTQ; 32-entry decode queue; 32-entry dispatch queue; 4-uop decode width; 6-uop dispatch width.
Processor back-end: 320-entry ROB; 136-entry LQ; 64-entry SQ; 8-uop execute width; 8-uop retire width.
Memory hierarchy: 32KB, 8-way, 4-hit-cycle VIPT L1I cache; no L1I prefetcher; 32KB, 8-way, 4-hit-cycle VIPT L1D cache; stride L1D prefetcher; 1MB, 8-way, 10-hit-cycle L2 cache; next-line L2 prefetcher; 32MB, 16-way, 20-hit-cycle LLC cache; no LLC prefetcher; 4 GB, one 8-byte channel, 3200MT/s DRAM.

V. METHODOLOGY

The evaluation has been conducted using the cycle-accurate ChampSim simulator [16]. ChampSim was the simulator employed for the 1st Instruction Prefetching Championship (IPC-1) and since then important improvements have been done towards a more accurate model, including a detailed front-end model. We have further modified ChampSim to model wrong-path execution and squashing of instructions (Section V-A) and a two-level branch and target prediction scheme[1] akin to the AMD Zen family [4], [6]. Table I shows the main configuration parameters of the baseline system evaluated in this work mimicking an AMD Zen 4 processor [6]. CACTI-P [28] and McPAT 1.3 [1], [27], using a 7nm process technology [5], are employed to model the energy expenditure of the whole system (core, cache hierarchy, and main memory).

We run the traces provided by Qualcomm Datacenter Technologies for the Championship Value Prediction (CVP) [2]. A subset of these traces was used for evaluating the prefetchers at IPC-1, but this study considers a larger number of those traces, whose conversion to ChampSim has been recently improved [13]. We use the 779 large server traces that show at least 1 L1I MPKI (misses per kilo-instruction) in the baseline system. Workloads are warmed up for 20M instructions, and then we gather statistics for their execution until completion.

A. Wrong-path model

ChampSim is a trace-based simulator and by default it does not model applications' wrong-path. However, not modeling wrong-path execution can lead to suboptimal decisions on the optimal FTQ size. Therefore, we modified ChampSim to model wrong-path execution and squashing of instructions.

Accurately modeling wrong path in a trace-based simulator is a difficult task [12]. However, this task simplifies when the focus is on instruction prefetching and when a decoupled front-end like FDP is employed. The reason is that FDP drives fetching based only on instruction addresses, not on the actual trace, and the main piece of information required to model the front-end is the instruction address.

[1]TAGE-SC-L branch predictor, ITTAGE indirect target predictor, and large BTBs cannot be accessed in a single cycle.

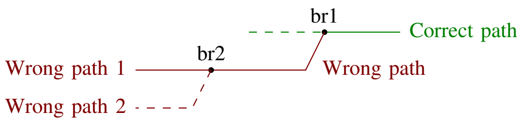

Fig. 10. Branch mispredictions on the wrong path

Therefore, we follow the FDP's branch prediction on the next addresses, then insert them in the FTQ and initiate the instruction fetch. The L1I will fetch the wrong-path addresses indicated by FDP, as in an execution-driven simulator, even when the prediction is towards the sequential path due to instructions never executed so far. In our model, wrong-path instructions advance to the next pipeline stages and are squashed when an older branch is considered mispredicted, either according to the L2 branch prediction engine, at decode stage due to a target misprediction, or later at execute stage. It is possible that a branch is detected as mispredicted more than once as the prediction is updated. We account for such cases, squashing and re-steering the fetch unit on each detected misprediction.

B. Limitations of the wrong-path model and solutions

Mispredicted wrong-path branches. Traces store information about taken/not taken branches on the correct path, but this information is not known for wrong-path instructions. Our approach is to follow the branch predictor decision and not squash branches on the wrong path (except when the L2 predictor contradicts the L1 predictor). This way, we can miss some branch mispredictions due to out-of-order execution.

Figure 10 shows the scenario where a mispredicted branch on the wrong path ($br2$) is corrected before a previous mispredicted branch on the correct path ($br1$). Continuous lines reflect the predicted path. In execution-based simulation, if $br2$ is resolved before $br1$, fetch is redirected from $Wrongpath1$ to $Wrongpath2$. In our model, $br2$ will not correct the path after its execution. Thus, we would continue prefetching instructions from $Wrongpath1$. In both cases, however, the cache would be polluted with incorrect instructions (either from $Wrongpath1$ or $Wrongpath2$), since execution follows the $Wrongpath$ with respect to $br1$.

Modeling instructions at the back-end. Modeling scheduling and execution at the back-end requires information such as the instruction type, the source and destination registers, and the memory locations. To obtain that information for wrong-path instructions, we keep a history of all the instructions loaded from the trace. When executing instructions on the wrong path, if their address matches the address of an instruction in the history, we retrieve its information. This way we can model load and store queues occupancy and register dependencies accurately. While type and registers are the same through different executions of an instruction, target addresses may change. We opted for retrieving the last address referenced by that instruction.

Since typically branches are predicted based on their previous behavior, it is very likely that a predicted path (either wrong or correct) has been visited before and hence recorded

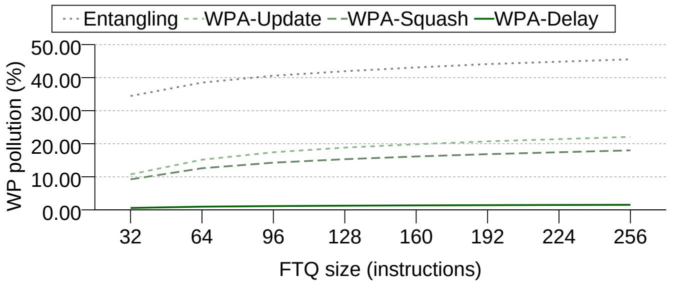

Fig. 11. Wrong-path induced pollution (lower is better) in FDP

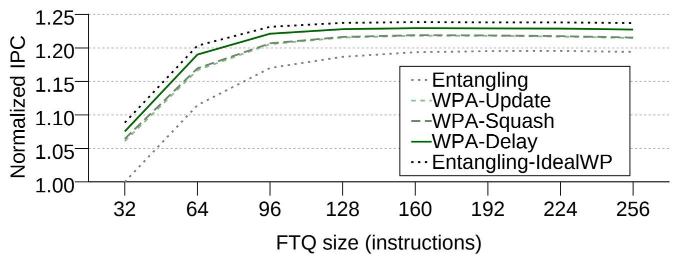

Fig. 12. Performance of Entangling-WPA in FDP

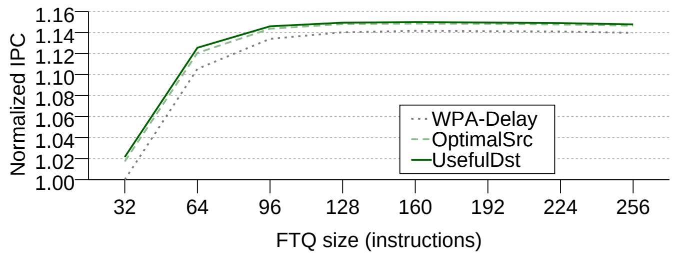

Fig. 13. Performance breakdown for optimized Entangling in FDP

in our history of instructions (exceptions are BTB misses for taken branches). Therefore, this simple solution provides useful information for most instructions on the wrong path. In particular, for the largest 256-instruction FTQ configuration, the percentage of wrong-path instructions dispatched to the back-end is 8.1%. Out of those, 78.0% can be retrieved from the history. Hence, 98.4% of the back-end instructions are known.

VI. Results

We start by measuring the wrong-path mitigation capabilities of Entangling-WPA. Then we explore the benefits of the proposed optimizations on top of Entangling-WPA. Finally, we compare our solution to other state-of-the-art proposals and show their impact on energy and (energy-delay) optimal FTQ size for each.

A. Wrong-path mitigation

Entangling-WPA employs three microarchitectural techniques to reduce the impact of wrong-path accesses on the L1I prefetcher. In order to quantify the effectiveness of each technique, we use a metric named *wrong-path pollution* defined as the percentage of insertions in the Entangling table that contain at least one cache line accessed only by wrong-path instructions.

The outcome of our analysis based on this metric is shown in Figure 11, in which we analyze the L1I prefetcher's performance in combination with an FDP of an increasing FTQ size (x axis). First, we show *Entangling* without wrong-path management. Then, we show the impact of updating the basic blocks (*WPA-Update*, Section III-A). Then, on top of *WPA-Update* is added the squashing of the History table technique (*WPA-Squash*, Section III-B). Finally, on top of *WPA-Squash* is added the proposal of delaying the insertion of new basic blocks and pairs in the Entangling table (*WPA-Delay*, Section III-C).

The first observation is that the pollution of the prefetcher grows with the FTQ size, since there are more wrong-path instructions being fetched. For the original Entangling prefetcher, the pollution is between 34%-46%, but it is considerably reduced when the basic block size is restored on a squash (*WPA-Update*), between 10%-22%. Squashing the History does not provide by itself significant reductions in pollution, but it is a necessary step towards the effectiveness of the third technique (*WPA-Delay*), which achieves less than 1.6% pollution when considering large FTQ sizes, and less than 1% for a 64-entry FTQ. The remaining pollution is due to pathological cases where a basic block starts on the correct path and ends on the wrong path. Addressing this pollution would complicate our design while offering modest performance benefits.

The reduction in wrong-path pollution translates to improvements in IPC, as shown in Figure 12. We show all the WPA proposals and the ideal WPA scenario described in Section III as an ideal version that fully ignores wrong path accesses (*Entangling-IdealWP*) – namely, wrong-path accesses are not used either for training the prefetcher nor for triggering prefetch requests.

The take away message of Figure 12 is that when all three techniques to mitigate the wrong-path pollution are applied (WPA-Delay), our Entangling-WPA approaches the performance of the ideal wrong-path fully aware version. In addition, Entangling-WPA shows a 2.8% performance improvements with respect to the previously published Entangling version without support for mitigating the wrong-path effects [36] for a 256-entry FTQ. Even higher improvements are reached for energy-aware FTQ sizes (6.8% for a 64-entry FTQ).

B. Optimizations

Figure 13 shows the improvements over the wrong-path aware Entangling (WPA) brought by our two microarchitectural optimizations: selecting optimal source-entangled (*OptimalSrc*) and, on top of it, filtering redundant prefetch requests and keeping only the useful entangled destinations (*UsefulDst*). Our analysis shows that when using an energy-aware 64-entry FTQ, OptimalSrc brings 1.4% performance improvements over Entangling-WPA, and together with the *UsefulDst* technique they obtain a speed up over Entangling-WPA of 1.8%.

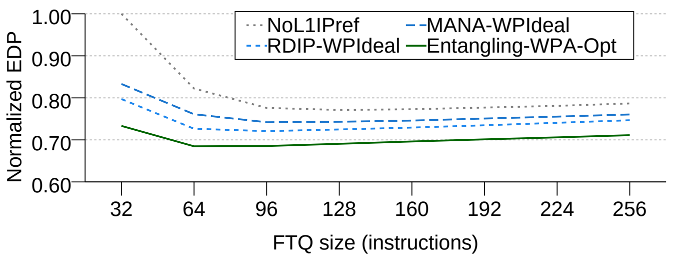

Fig. 14. EDP of the evaluated prefetchers

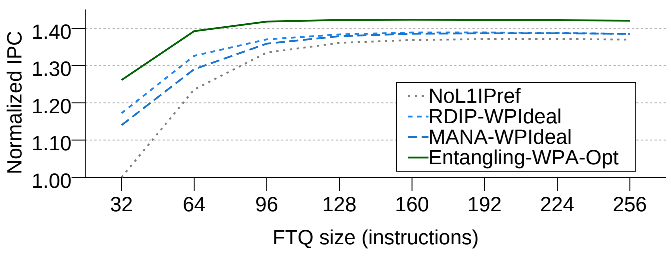

Fig. 15. Performance of the evaluated prefetchers

C. Comparison to state-of-the-art techniques

We compare Entangling equipped with the microarchitectural techniques proposed in this work to a baseline without L1I prefetching (NoL1IPref) and to two other state-of-the-art L1I prefetchers: *MANA* [8] and *RDIP* [23] (a more detailed analysis including comparison to other L1I prefetchers and larger L1I cache sizes can be found in our previous work [36]). *MANA* is a BTB-directed L1I prefetcher, providing a good performance-area trade-off. It is a successor of SN4L-Dir-BTB [9]. *RDIP* is a RAS-directed L1I prefetcher that associates prefetch operations with signatures that encode the RAS and its context. We modified the original MANA and RDIP proposals, that are wrong-path agnostic, to train them ideally only on the correct path. In what follows, we compare with these wrong-path ideal versions of the prefetchers (WPIdeal).

Energy delay product (EDP). We start this evaluation showing the EDP for all prefetchers since later the description focuses on the best FTQ size for each prefetcher in terms of EDP. Figure 14 shows the energy delay product normalized to NoL1IPref with a 32-entry FTQ. Energy is measured in nj and delay is measured in cycles per instruction. The lower is the EDP, the better. Increasing the FTQ size from 32 to 64 entries significantly improves EDP for all prefetchers because of the important boost in performance. Our Entangling version reaches optimal EDP with an FTQ of 64 instructions. The other competitors still benefit from larger FTQs. Indeed a 96-entry FTQ is optimal in terms of EDP for both of them. Finally, NoL1IPref obtain optimal EDP with 128 entries in the FTQ. Hence, different prefetchers achieve the best EDP at a different FTQ size. In what follows, for each prefetcher, we focus on the FTQ size for which it delivers the lowest EDP, as summarized in Table II.

Performance. Figure 15 presents the performance of the prefetchers, measured in instructions per cycle (IPC), the higher the better. Results are normalized to NoL1IPref with an FTQ of 32 entries. As the FTQ size increases prefetchers improve performance. RDIP-WPIdeal and MANA-WPIdeal reach a plateau with negligible improvements after 128-entries. Entangling reaches that plateau with a lower FTQ size (96 instructions). RDIP-WPIdeal and MANA-WPIdeal improve performance by 2.7% and 1.8%, respectively, over NoL1IPref, when all implement an FTQ size of 96, at which these prefecthers deliver the lowest EDP. Our optimized version of Entangling, Entangling-WPA-Opt, outperforms both MANA and RDIP and with a 64-entry FTQ obtains performance improvements of 12.7% over NoL1IPref with the same FTQ. As the FTQ size increases, the performance gap between the prefetchers decreases, i.e. for 256-entry FTQ, Entangling-WPA-Opt outperforms NoL1IPref by 3.7%. Yet, the performance benefits of NoL1IPref come with a higher energy-cost, as we detail next, while a 64-FTQ Entangling-WPA-Opt already obtains close to optimal (w.r.t FTQ size) performance.

Energy consumption. Figure 16 presents the dynamic energy expenditure of the system (core, cache hierarchy, and main memory), normalized to NoL1IPref with 32-entries FTQ (the lower, the better). Energy expenditure grows linearly with the FTQ size. Larger FTQs increase the number of accesses to BTBs and branch predictors, as well as L1I and, in case of cache misses, higher cache levels and main memory.

Generally, the energy trends are similar across all prefetchers. Interestingly, using a L1I prefetcher incur a lower energy expenditure than not using a dedicated L1I prefetcher. After analyzing this behavior, we found out that prefetch requests help positively updating the L1I replacement information. In essence, prefetch requests first search the L1I. Hence, although there is an increase in L1I accesses due to prefetching, they only access cache metadata (i.e., tags and replacement information), which incur low energy consumption. Prefetch requests then update the LRU policy on a hit, which yields less data accesses to the more costly L2 and LLC since the prefetch requests are accurate even when they hit L1I. We observed that most of the energy reduction comes from less accesses to the large L2 cache.

As a consequence, Entangling is consistently more energy efficient than NoL1IPref by 6-9%, and than MANA-WPIdeal and RDIP-WPIdeal, when considering the same FTQ size. For the FTQ sizes at which the prefetchers achieve the best EDP, the energy efficiency per prefetcher with respect to NoL1IPref of 128-entry FTQ is: 10.6% for Entangling-WPA-Opt with 64-entry FTQ, 6.5% for RDIP-WPIdeal with 96-entry FTQ, and 4.1% for MANA-WPIdeal with 96-entry FTQ. NoL1IPref with 128-entry FTQ increases energy expenditure by 3.1% over a 32-entry FTQ. As the FTQ size increases to 256-entries, all prefetchers become less energy efficient which around 6% more energy consumption than for a 32-entry FTQ.

Memory requirements. We evaluate MANA-WPIdeal with a 4K-entry MANA table, which reflects the low-cost configuration described by Ansari *et al.* [8]. It requires a total size of 17.25KB. For RDIP-WPIdeal, we evaluate a 4K-entry miss table with 3 trigger prefetchers for discontinuities and an 8-bit vector for consecutive cache lines. The total required storage is 63KB. For Entangling we model a 32-entry History and a 4K-entry Entangled table. This results in 41.85KB for our wrong-

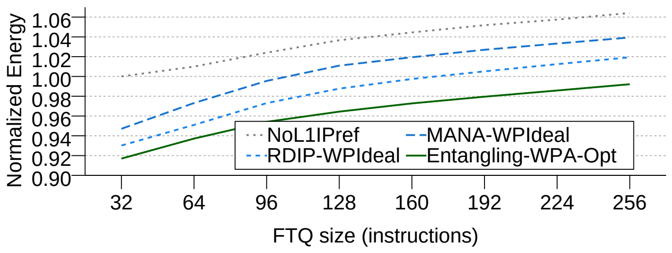

Fig. 16. Energy consumption of the evaluated prefetchers

TABLE II
MEMORY REQUIREMENTS AND OPTIMAL FTQ SIZE

Prefetcher	**Prefetcher memory (KB)**	**FTQ entries (optimal)**	**Total memory (KB)**
NoL1IPref	0	128	1.31
RDIP-WPIdeal	63.00	96	63.98
MANA-WPIdeal	17.25	96	18.23
Entangling-WPA	41.18	64	41.84
Entangling-WPA-Opt	41.21	64	41.87

path aware version and 41.87KB for the wrong-path aware version with the proposed microarchitectural optimizations. Table II summarizes the memory requirements of all prefetchers and shows the optimal FTQ size for each prefetcher considering the energy-delay-product metric averaged across all applications.

VII. Related Work

Wrong path modeling and effects. The discrepancy between modeling or disregarding wrong-path effects has been analysed in the past. For instance, Mutlu et al. [29] showed that the performance difference between modeling and disregarding wrong-path execution is up to 10%. Furthermore, the wrong-path negative effects are demonstrated to increase with larger pipelines. Modelling wrong path in trace-based simulators is a challenging task. Bhargava et al. [12] proposed to recreate a copy of the application code in other to model wrong path instructions in trace-based simulators. They find that often, more than 99% of the branch targets are found in their copy of the code. Although our wrong-path model is aimed at modeling accurately the front-end where we leverage the branch prediction information to discover new instructions to fetch (as a real processor would do), we also resort to a similar technique as the one proposed by Bhargava et al. to model instructions at the back end.

Other studies show that wrong-path memory accesses can be beneficial for performance in some cases [26], [31], as long as only superficial wrong-path prefetching is allowed, such as one, two or up to three wrong-path cache lines.

Wrong path effects in FDP. Fetch Directed Prefetching (FDP) [34] is inherent to wrong-path aware prefetches and, as prior work underlines [31] wrong-path “pollution” can even slightly boost performance. Elastic Instruction Fetching [30] proposes a solution to mitigate the negative impact of deep pipelines, at the cost of replicating hardware structures. Recently, Ishii et al. [19], brings to our attention the importance of evaluating a large enough FTQ, however their analysis does not consider wrong-path execution.

Mitigating wrong-path effects in FDP requires an ideal branch predictor. In practice, this translates to continuously growing BTBs. Yet, BTBs are already quite large structures, e.g., 561.5KB in the Samsung Exynos M6 [17], commonly organized in several BTB levels. Further augmenting the number of BTB levels or compressing them at the cost of indirection [38], [41], leads to front-end bubbles and increases the number of wrong-path instructions. In this work we use state-of-the-art branch and target prediction [39], [40], however close-to-perfect branch prediction is difficult to achieve due to BTB size limitations and hard-to-predict branches, and the front-end choice of predicting (and therefore prefetching) a single execution path. In contrast, the Entangling instruction prefetcher can explore both paths at the same time, thanks to entangling each source with several destinations when necessary.

An alternative to mitigate wrong-path effects is to recover fast from branch mispredictions [10]. This way, the wrong-path execution flow is redirected earlier, reducing the wrong-path impact on L1I accesses.

Wrong path management in L1I prefetching. Traditional L1I prefetchers, such as the *next-line* prefetcher, are commonly wrong-path agnostic. For prefetchers that do not require training, such as next-line prefetchers, this is not a problem as limited prefetching on the wrong path has been proved beneficial. More complex prefetchers are also active on the wrong-path. In particular, Confluence [21] proposes to unify the metadata of the BTBs used in FDP and of the L1I prefetchers, thus reducing the storage overhead and exposing the predicted path to the L1I prefetcher. It also evaluates FDP and an L1I prefetcher in tandem, but the prefetchers do not perform any particular action to mitigate wrong-path pollution. Another state-of-the-art prefetcher, SN4L-Dir-BTB [9], having as a main component a BTB-directed prefetcher, opts for not performing particular actions to recover from wrong-path execution.

An alternative is to train the prefetcher just on the correct-path, either by using the instruction commit sequence [14] or the non-speculative RAS state [23]. A drawback of recording instructions at commit-time is that the filtering of L1I accesses done by a decode cache in the front-end cannot be leveraged by the prefetcher. Additionally, timeliness can be jeopardized by recording the instructions much later than when they took place. As timeliness is essential for the Entagling prefetcher, we opted for a more precise handling of the wrong path.

VIII. Conclusion

In this work we show that a precise L1I prefetcher working in tandem with FDP can provide significant benefits. Furthermore, we show that depending on the precision of the prefetcher, different decoupling degrees yield lower energy-delay-product, and close to ideal L1I prefetchers work best with small Fetch Target Queues (FTQ). Modeling wrong path has a direct impact on performance and we show that in addition to energy waste, larger FTQs increase the wrong-path-caused pollution.

Departing from these observations, we add lightweight support for mitigating wrong-path pollution in the state-of-the-art Entangling instruction prefetcher and propose two optimizations to further improve its performance. This work makes a solid step towards including the L1I prefetcher in future chip designs by: (1) demonstrating its effectiveness on top of FDP, (2) reducing wrong-path pollution by 99%, (3) improving performance by 1.8% over a wrong-path aware version of the best up-to-date L1I prefetcher, and (4) with a moderate storage overhead of 304 bytes.

Acknowledgments

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 819134), MCIN/AEI/10.13039/501100011033/ and the "ERDF A way of making Europe", EU (grant PID2022-136315OB-I00), MCIN/AEI/10.13039/501100011033 and the European Union NextGenerationEU/PRTR (grant TED2021-130233B-C33), and the Ramón y Cajal Research Contract RYC2018-025200-I.

References

[1] "McPAT 1.3," https://github.com/HewlettPackard/mcpat, Sep. 2015.

[2] "The 1st Championship Value Prediction (CVP-1)," https://www.microarch.org/cvp1/cvp1/index.htm, Jun. 2018.

[3] "The 1st Instruction Prefetching Championship (IPC1)," https://research.ece.ncsu.edu/ipc, May 2020.

[4] "A look at the AMD Zen 2 core," https://fuse.wikichip.org/news/2458/a-look-at-the-amd-zen-2-core, May 2021.

[5] "PCACTI," https://sportlab.usc.edu/downloads/packages, 2021.

[6] "AMD's Zen 4 part 1: Frontend and execution engine," https://chipsandcheese.com/2022/11/05/amds-zen-4-part-1-frontend-and-execution-engine, Nov. 2022.

[7] AMD, *Software Optimization Guide for AMD EPYC™ 7003 Processors*, AMD, No. 56665, revision 3.00, Nov. 2020.

[8] A. Ansari, F. Golshan, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Mana: Microarchitecting an instruction prefetcher," in *The 1st Instruction Prefetching Championship (IPC1)*, May 2020.

[9] A. Ansari, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Divide and conquer frontend bottleneck," in *47th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2020, pp. 65–78.

[10] D. N. Armstrong, H. Kim, O. Mutlu, and Y. N. Patt, "Wrong path events: Exploiting unusual and illegal program behavior for early misprediction detection and recovery," in *37th Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2004, pp. 119–128.

[11] G. Ayers, N. P. Nagendra, D. I. August, H. K. Cho, S. Kanev, C. Kozyrakis, T. Krishnamurthy, H. Litz, T. J. Moseley, and P. Ranganathan, "Asmdb: Understanding and mitigating front-end stalls in warehouse-scale computers," in *46th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2019, pp. 462–473.

[12] R. Bhargava, L. K. John, and F. Matus, "Accurately modeling speculative instruction fetching in trace-driven simulation," in *Int'l Performance Computing and Communications Conference (IPCCC)*, Feb. 1999, pp. 65–71.

[13] J. Feliu, A. Perais, D. Jimenez, and A. Ros, "Rebasing microarchitectural research with industry traces," in *Int'l Symp. on Workload Characterization (IISWC)*, Oct. 2023.

[14] M. Ferdman, C. Kaynak, and B. Falsafi, "Proactive instruction fetch," in *44th Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2011, pp. 152–162.

[15] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Temporal instruction fetch streaming," in *41th Int'l Symp. on Microarchitecture (MICRO)*, Nov. 2008, pp. 1–10.

[16] N. Gober, G. Chacon, L. Wang, P. V. Gratz, D. A. Jimenez, E. Teran, S. Pugsley, and J. Kim, "The championship simulator: Architectural simulation for education and competition," *CoRR*, vol. abs/2210.14324, Oct. 2022.

[17] B. Grayson, J. Rupley, G. D. 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 *47th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2020, pp. 40–51.

[18] Intel, "Intel® 64 and ia-32 architectures optimization reference manual," www.intel.com, Jun. 2016.

[19] Y. Ishii, J. Lee, K. Nathella, and D. Sunwoo, "Re-establishing fetch-directed instruction prefetching," in *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2021, pp. 172–182.

[20] C. Kaynak, B. Grot, and B. Falsafi, "SHIFT: Shared history instruction fetch for lean-core server processors," in *46th Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2013, pp. 272–283.

[21] ——, "Confluence: Unified instruction supply for scale-out servers," in *48th Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2015, pp. 166–177.

[22] T. A. Khan, A. Sriraman, J. Devietti, G. Pokam, H. Litz, and B. Kasikci, "I-spy: Context-driven conditional instruction prefetching with coalescing," in *53rd Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2020, pp. 146–159.

[23] A. Kolli, A. G. Saidi, and T. F. Wenisch, "RDIP: Return-address-stack directed instruction prefetching," in *46th Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2013, pp. 260–271.

[24] R. Kumar, B. Grot, and V. Nagarajan, "Blasting through the front-end bottleneck with shotgun," in *23rd Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Mar. 2018, pp. 30–42.

[25] R. Kumar, C.-C. Huang, B. Grot, and V. Nagarajan, "Boomerang: A metadata-free architecture for control flow delivery," in *23rd Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2017, pp. 493–504.

[26] C. J. Lee, H. Kim, O. Mutlu, and Y. N. Patt, "Performance-aware speculation control using wrong path usefulness prediction," in *14th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2008, pp. 39–49.

[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 *42nd Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2009, pp. 469–480.

[28] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques," in *2011 Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov. 2011, pp. 694–701.

[29] O. Mutlu, H. Kim, D. N. Armstrong, and Y. N. Patt, "Understanding the effects of wrong-path memory references on processor performance," in *3rd Workshop on Memory Performance Issues (WMPI)*, Jun. 2004, pp. 56–64.

[30] A. Perais, R. Sheikh, L. Yen, M. McIlvaine, and R. D. Clancy, "Elastic instruction fetching," in *25th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2019, pp. 478–490.

[31] J. Pierce and T. N. Mudge, "Wrong-path instruction prefetching," in *29th Int'l Symp. on Microarchitecture (MICRO)*, Dec. 1996, pp. 165–175.

[32] A. Ramirez, O. J. Santana, J. L. Larriba-Pey, and M. Valero, "Fetching instruction streams," in *35th Int'l Symp. on Microarchitecture (MICRO)*, Nov. 2002, pp. 371–382.

[33] G. Reinman, T. M. Austin, and B. Calder, "A scalable front-end architecture for fast instruction delivery," in *26th Int'l Symp. on Computer Architecture (ISCA)*, May 1999, pp. 234–245.

[34] G. Reinman, B. Calder, and T. Austin, "Fetch directed instruction prefetching," in *32nd Int'l Symp. on Microarchitecture (MICRO)*, Dec. 1999, pp. 16–27.

[35] G. Reinman, B. Calder, and T. M. Austin, "Optimizations enabled by a decoupled front-end architecture," *IEEE Transactions on Computers (TC)*, vol. 50, no. 4, pp. 338–355, Apr. 2001.

[36] A. Ros and A. Jimborean, "A cost-effective entangling prefetcher for instructions," in *48th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2021, pp. 99–111.

[37] O. J. Santana, A. Ramirez, and M. Valero, "Enlarging instruction streams," *IEEE Transactions on Computers (TC)*, vol. 56, no. 10, pp. 1342–1357, Oct. 2007.

[38] A. Seznec, "Don't use the page number, but a pointer to it," in *23rd Int'l Symp. on Computer Architecture (ISCA)*, Jun. 1996, pp. 104–113.

[39] ——, "A 64-Kbytes ITTAGE indirect branch predictor," in *2nd JILP Workshop on Computer Architecture Competitions (JWAC-2): Championship Branch Prediction*, Jun. 2011.

[40] ——, "TAGE-SC-L branch predictors again," in *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, Jun. 2016.

[41] N. K. Soundararajan, P. Braun, T. A. Khan, B. Kasikci, H. Litz, and S. Subramoney, "Pdede: Partitioned, deduplicated, delta branch target buffer," in *54th Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2021, pp. 779–791.

[42] V. Srinivasan, E. S. Davidson, G. S. Tyson, M. J. Charney, and T. R. Puzak, "Branch history guided instruction prefetching," in *7th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Jan. 2001, pp. 291–300.

[43] D. Tarjan and K. Skadron, "Merging path and gshare indexing in perceptron branch prediction," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 2, no. 3, pp. 280–300, Sep. 2005.

[44] T.-Y. Yeh and Y. N. Patt, "A comprehensive instruction fetch mechanism for a processor supporting speculative execution," in *25th Int'l Symp. on Microarchitecture (MICRO)*, Nov. 1992, pp. 129–139.

[45] Y. Zhang, S. Haga, and R. Barua, "Execution history guided instruction prefetching," in *16th Int'l Conf. on Supercomputing (ICS)*, Jun. 2002, pp. 199–208.

IX. Biography Section

Alberto Ros is full professor in the Computer Engineering Department at the University of Murcia, Spain. Funded by the Spanish government to conduct the PhD studies he received the PhD in computer science from the University of Murcia in 2009. He held postdoctoral positions at the Universitat Politècnica de València and Uppsala University. He received an European Research Council Consolidator Grant in 2018 to improve the performance of multicore architectures. Working on cache coherence, memory hierarchy designs, memory consistency, and processor microarchitecture, he has co-authored more than 100 peer-reviewed articles. He has been inducted into the ISCA Hall of Fame and MICRO Hall of Fame. He is IEEE Senior member.

Alexandra Jimborean is a Ramón y Cajal researcher at the University of Murcia, since 2020. Her research focuses on compile-time and run-time code analysis and optimization for performance, energy efficiency, and security and on software-hardware co-designs. She obtained her PhD from the University of Strasbourg, France in 2012, held a postdoctoral fellowship in Uppsala University, Sweden and continued as Assistant Professor and Associate Professor.