2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA) | 978-1-6654-3333-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/ISCA52012.2021.00017

A Cost-Effective Entangling Prefetcher for Instructions

1st Alberto Ros
Computer Engineering Department
University of Murcia
Murcia, Spain
aros@ditec.um.es

2nd Alexandra Jimborean
Computer Engineering Department
University of Murcia
Murcia, Spain
alexandra.jimborean@um.es

***Abstract*—Prefetching instructions in the instruction cache is a fundamental technique for designing high-performance computers. There are three key properties to consider when designing an efficient and effective prefetcher: timeliness, coverage, and accuracy. Timeliness is essential, as bringing instructions too early increases the risk of the instructions being evicted from the cache before their use and requesting them too late can lead to the instructions arriving after they are demanded. Coverage is important to reduce the number of instruction cache misses and accuracy to ensure that the prefetcher does not pollute the cache or interacts negatively with the other hardware mechanisms.**

This paper presents the *Entangling* Prefetcher for Instructions that entangles instructions to maximize timeliness. The prefetcher works by finding which instruction should trigger the prefetch for a subsequent instruction, accounting for the latency of each cache miss. The prefetcher is carefully adjusted to account for both coverage and accuracy. Our evaluation shows that with 40KB of storage, Entangling can increase performance up to 23%, outperforming state-of-the-art prefetchers.

***Index Terms*—Instruction prefetching, caches, entangling, correlation, latency**

I. Introduction

As software-as-a-service and Cloud computing become increasingly popular, server and Cloud applications exhibit notoriously large instruction sets that do not fit in the first level instruction cache (L1I), leading to high L1I miss rates and therefore stalls. This causes significant performance degradation, in addition to wasteful energy expenditure and underutilization of resources.

Until now, processors have been traditionally designed for scientific and desktop applications, with very different characteristics, making them inefficient for the large and ever-increasing instruction footprint of server applications. According to a study conducted by Google [25] over three years on one of their Warehouse Scale Computing (WSC) live centers with tens of thousands of server machines running workloads and services used by billions of users, processors do useful work for only 10-20% of the time, stalling for more than 80% of the time. This analysis demonstrates that one of the main reasons of stalling is that instructions are not available for execution when running server applications, making the cache and memory systems of server processors a prime optimization target. In particular, this study identifies a significant and growing problem with L1I bottlenecks, due to the instruction footprint of server applications growing at a much higher rate per year than the size of the L1I, conclusions reinforced by several recent studies [7], [27]. They demonstrate that instruction fetching represents a considerable fraction of the memory stalls, together with data accesses, and underline the importance of prefetching for data centers.

Indeed, as memory latency has been recognized as a critical factor for performance, a plethora of prefetching techniques have been proposed over the last decades [10], [13], [15], [19], [22]–[24], [36], [39], [46], [48]–[50], [52]. However, the research community focused predominantly on data prefetching and there is relatively little research done on instruction prefetching, despite its increasing importance with more and more applications being served through the Cloud.

Basic prefetch mechanisms include simple next line instruction prefetchers [11] and *next line prefetchers of arbitrary sizes* [37], [43], but more advanced ones have been proposed, from prefetchers guided by branch prediction [10], [26], [30], [31], [36], [38], [39], [46], [48] and execution history [51] to prefetchers that use idle hardware resources to fetch instructions (e.g. run-ahead helper threads [4], [49], [52]).

One common technique for prefetching employs *correlation* [13], [15], [19], [23], [29], [35], namely, building correlations between a memory reference and a previous event, such as memory reference streams, instruction addresses, or branch history, by exploiting temporal or spatial patterns. *Temporal prefetchers* are a class of correlation-based prefetchers that record sequences of cache misses and predict future misses by replaying the history, reaching higher coverage and accuracy than their predecessors [13], [15], but incurring impractical storage costs.

Other types of prefetchers interact with hardware structures [6], [26], [30], [31], [38], such as the branch predictor (e.g. BTB directed), to gain insights into the program's execution ahead of time, however they require intrusive changes in the processor design.

Typically, prior work in prefetching has adopted look-ahead mechanisms [5], [6], [13], [15], [26], [29]–[31], [35], [44] to

This work was supported by the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement No 819134) and by the Ramón y Cajal Research Contract (RYC2018-025200-I).

978-1-6654-3333-4/21/$31.00 ©2021 IEEE
DOI 10.1109/ISCA52012.2021.00017

address both coverage and accuracy. Look-ahead prefetchers follow the execution path *n* steps in advance and prefetch the corresponding block (instruction, cache line, etc.). The *steps* may refer to instructions, branches, function calls, etc, while *n* is typically referred to as the *look-ahead distance*. Nevertheless, by employing a fixed look-ahead distance, such prefetchers are rigid and cannot timely serve all instruction misses: a long look-ahead distance would bring in the instruction too early, unnecessarily polluting the cache if the instruction is evicted before its use; a short look-ahead distance may prefetch the instruction too late, after it has been demanded by the processor. Yet, look-ahead is a popular technique, if a "good-enough" distance is identified through careful tuning.

Inspired by previous proposals [33], [40], this work builds on the observation that different instructions have different fetch-latency and thus require different look-ahead distances for a timely and useful prefetch. Figure 1 illustrates the fraction of timely prefetches given a fixed look-ahead distance, over a selection of server workloads (see Section IV). The look-ahead distance represents the number of taken branches (discontinuities), akin to previous proposals [29]. To generate this figure, we used a baseline without any prefetching and tracked the L1I misses and their latency using dedicated structures (see Section III for details). For each L1I miss, we computed how many discontinuities in advance a prefetch should be issued not to be late. This can be seen as an oracle to identify the optimal look-ahead distance for each miss and the percentage of total misses each distance would cover. Figure 1 shows the fraction of timely prefetches for distances between 1 and 10, indicating that the remaining fraction of L1I misses are covered with prefetching distances larger than 10. While for this study the look-ahead distance was fixed statically, it is akin to determining a suitable look-ahead distance dynamically, e.g. based on observations performed during a warm-up phase.

The first remark is that a look-ahead distance fixed for all misses is sub-optimal, as different misses require different distances, or even the same miss can require different distances depending on the execution path. In our proposal, we support several distances simultaneously (even for the same address).

Second, there is no fixed look-ahead distance to work well across all benchmarks. A look-ahead distance of 1 may prefetch 70% of the L1I misses in a timely manner for one application, but only 20% of the misses for another. At the other end, large look-ahead distances (10+) serve a considerable number of misses (up to 15%) and cannot be neglected in the design of an an effective prefetcher. Complementing Figure 1, Figure 2 emphasizes prefetching pollution caused by wrong or early prefetchers (lack of accuracy) if a fixed look-ahead distance is used. While some applications can tolerate an increase in the look-ahead distance without loosing accuracy, other (see top lines) can reduce accuracy by 10% when moving from a distance of 1 to 10.

The departure point of this proposal is *timeliness*, as a key metric for instruction prefetching. Figures 1 and 2 demonstrate that a fixed look-ahead distance leads to both low coverage (only few of the misses are served in a timely manner) and low accuracy (only few of the issued prefetchers are useful).

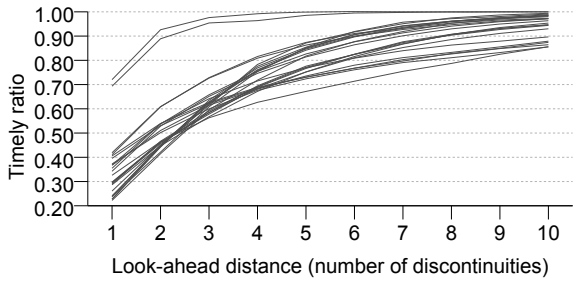

Fig. 1. Fraction of timely prefetches with respect to the look-ahead distance.

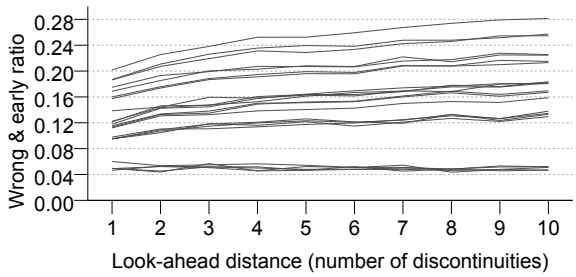

Fig. 2. Fraction of useless (wrong or early) prefetches with respect to the look-ahead distance.

To approach the performance of an ideal instruction cache (no L1I misses), we propose the *Entangling* Prefetcher for Instructions, or *Entangling* I-Prefetcher,[1][2] which, in contrast to its predecessors, is designed around the notion of timeliness. Entangling computes the latency of cache misses and *entangles* them with the cache accesses that should trigger the prefetch to ensure the timely arrival of the requested instructions. In this way, Entangling is robust and effective, agnostic to the application characteristics and achieves a 97.6% L1I hit rate, approaching the perfect L1I.

This paper makes the following contributions:

- Makes the observation that a significant fraction of L1I misses cannot be *timely* prefetched by employing a fixed look-ahead distance.
- Proposes an instruction prefetcher whose core design point is *timeliness* and demonstrate its effectiveness over a wide selection of benchmarks.
- Proposes *Entangling* as a mechanism to identify the right prefetch time for each cache line – rather than a fixed look-ahead distance. Instead of learning the look-ahead distance, which is central to many state-of-the-art methods, our prefetcher builds time aware correlations between past L1I events (handles) and L1I misses and learns which event the prefetch should be associated with so that it is timely.

[1]A performance-oriented version of the Entangling Instruction Prefetcher [41], [42] won the 1st Instruction Prefetch Championship (IPC1).

[2]The code of the Entangling prefetcher proposed in this paper is available at https://github.com/alberto-ros/EntanglingInstructionPrefetcher

- The results demonstrate that Entangling triggers timely prefetches delivering high coverage (88.2%) and accuracy (71.5%), and close to ideal L1I performance (97.6% L1I hit rate), offering the best area vs performance trade-off.
- We designed a compression scheme and a clever encoding and table organization, that yields Entangling compact, with low storage demands, such that it makes best use of the allotted hardware budget.

II. The Entangling I-Prefetcher

The key conceptual contribution of the Entangling I-prefetcher is the entanglement (or pairing) of instructions, namely, the instruction upon whose execution should be triggered a timely prefetch for the instruction. In a more concise representation, we define as *src-entangled* the cache line (also referred to as source) that should trigger the prefetch of the *dst-entangled* cache line (destination) such that the requested line arrives timely.

To ensure timeliness, we compute the latency of each cache miss by subtracting the time the requested cache line enters the cache to a recorded timestamp of the cache miss. Once we know the latency of a miss, we can pair the missing cache line to a previous accessed cache line that took place at least *latency* cycles before the missing access, as shown in Figure 3. To this end, the Entangling I-prefetcher records a history of the last L1I accessed cache lines. We track back the recorded history looking for the *src-entangled* cache line fitting our criteria and entangle it with the *dst-entangled* cache line that missed in cache. Next time the *src-entangled* line accesses the cache, it will trigger a timely prefetch for the *dst-entangled* line, transforming the previous miss into a hit.

As tracking all pairs of entangled cache lines in a program would require considerable storage requirements, the Entangling I-prefetcher only entangles heads of basic blocks, defined as follows. A basic block represents the set of consecutive cache lines, where consecutive refers to the program order of instructions, grouped in cache lines [11]. The head of a basic block is therefore the first non-consecutive cache line that accesses the cache and the size of the basic block is the number of following consecutive lines.[3] Recording only the basic block head and its size suffices for efficient prefetching of contiguous cache lines. In order to further reduce the number of entangled lines, the Entangling I-prefetcher merges "almost" consecutive basic blocks, entangling only the head of the first basic block.

The prefetching engine is then triggered upon every cache access to a head of a basic block, and prefetches the entire basic block of the current cache line and of each of its *dst-entangled* cache lines.

The entangling mechanism is versatile and by design can easily adapt to different (or multiple) execution paths or variations in latency. First, a *src-entangled* cache line can have multiple *dst-entangled* cache lines, such that they are all served timely. In turn, a *dst-entangled* cache line can

[3]Note that this is a dynamic view of a basic block and it can change depending on the execution path.

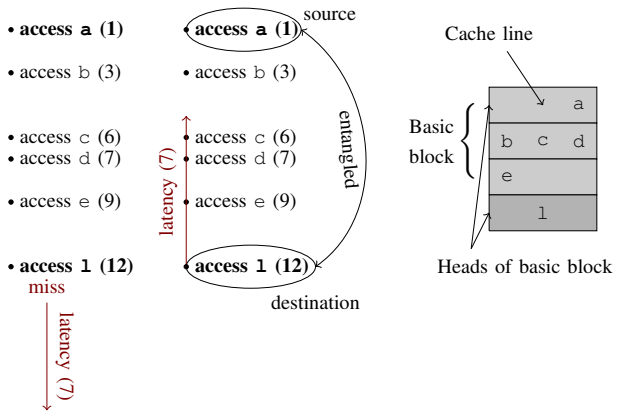

Fig. 3. A running example: The first basic block, BB1, consists in accesses a, b, c, d, e and the second block, BB2 in access l. Basic block heads are marked with bold. The first column illustrates the history of accesses together with the cycle when each access executed (shown in brackets) and the occurrence of access l that misses in L1I and its $latency$. The second column illustrates that in order to prefetch BB2, we travel back in history $latency$ cycles from access l and find the instruction that executed $latency$ cycles ahead (access b, part of BB1). We then entangle BB2's head (access l) with the head of BB1 (access a). The third column shows the organization of the cache lines in basic blocks.

be linked to multiple *src-entangled* cache lines, thus it can timely prefetched regardless of the execution path. Second, to deal with fluctuations in latency, the Entangling prefetcher constantly creates new entangled pairs on misses and discards pairs that are no longer useful. Note that small latency fluctuations are often accommodated by default, given that the source of the entanglement is a basic block head and not the instruction that precisely matches the latency of fetching the target instruction.

III. A Cost-Effective Implementation

This section describes a cost-effective implementation of the Entangling I-prefetcher. We start by presenting a basic implementation and continue with advanced techniques to improve accuracy and reduce storage overhead. Finally, we offer further implementation details and considerations.

A. Basic implementation

As mentioned, the Entangling prefetcher triggers prefetches on each access to a source-entangled address, prefetching both the next lines within the current basic block and the whole basic block of each of its destination-entangled addresses. In what follows, we detail the hardware mechanisms that identify basic blocks and, for each L1I miss, the position of a *timely* prefetch (i.e. the source-entangled line). The required hardware and the interaction between the new structures is depicted in Figure 4.

*1) **Computing basic block sizes**:* To compute the size of each *dynamic* basic block fetched by the processor front-end, we constantly track both the *head* (first) cache line of the current basic block being fetched and its *size* (see right-top part, *Basic block*, in Figure 4). A simple comparison of the

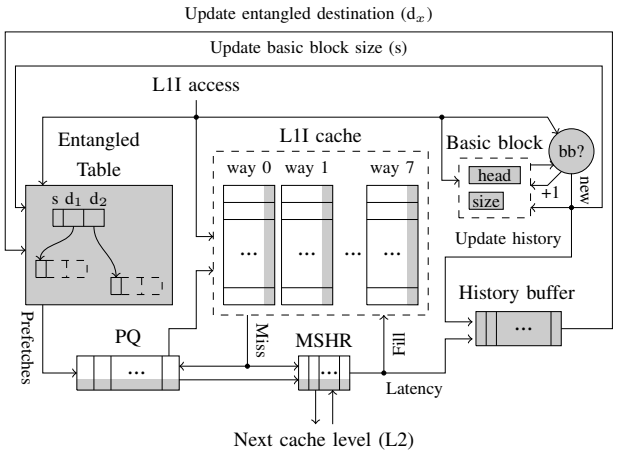

Fig. 4. Overview of the Entangling I-prefetcher with hardware extensions shown in gray. The Basic block registers *head* and *size* keep track of the current basic block. *History* is a small circular queue and each entry records basic block information: head, size and timestamp of the first L1I access. *Entangled* is a cache-like structure and each entry consists in a src-entangled cache line, its basic block size, and a compressed array of dst-entangled cache lines (for the advanced optimization techniques, a confidence counter is associated to each destination). The L1I, PQ and MSHR are extended with information on timing (the timestamp when the request was issued) and on the src-entangled (position of the source in the *Entangled table* and an access bit indicating if the access stems from a demand access or a prefetch).

current access with the head address plus the current size (*bb?*) indicates whether the new access is for the next line (same basic block) or a non-consecutive cache line. In the first case (*+1*), the size of the basic block is increased by one. In the second case (*new*), the basic block ends and we start tracking a new block.

When detecting a new basic block, we first store the size of the basic block that just completed in the *Entangled table*, the core structure of our proposal that records the necessary information to issue prefetches (see left part of Figure 4). If the basic block is already recorded in the *Entangled table*, we update its size to the maximum of the already stored size and the new size. This decision increases the coverage of the prefetcher at the cost of having extra false positives. Finally, we start tracking the new basic block, by updating the *head* register with the current cache line and resetting the *size* to 0.

This mechanism populates the *Entangled table* with basic blocks that will act as sources of entangled pairs.

2) ***Building entangled pairs for timely prefetching****:* Finding a suitable source-entangled cache line for each L1I miss requires two steps. First, we need to compute the latency of each L1I miss (or L1I prefetch) and second, to identify the cache line executed (at least) $latency$ cycles before the miss. The head of its parent basic block will be the source and the address causing the L1I miss will be the destination of the entangled pair.

To find potential *src-entangled* cache lines, we store the recent history of basic block heads together with the timestamp of their first access to L1I in a small circular queue called the *History buffer* (right-bottom part of Figure 4).

To compute the latency of a demand L1I miss, we require the start and end timestamps. For the start timestamp, we record the time of the demand miss along with the entry allocated by default in the miss status holding register (MSHR). Additionally, two other fields are added to each MSHR entry: an access bit (set to one for demand misses) and a pointer to the position of that access in the *History buffer* (if the access is a basic block head). Similarly, we keep track of the latency of the prefetches in order to compute the actual latency on late prefetches (a miss for an already prefetched cache line takes place). We extend the prefetch queue (PQ) such that, when a prefetch is issued, it stores the current time along with the currently allocated prefetch entry. An access bit is also added to the PQ, initialized to 0. If the prefetch misses in cache, it is automatically handled as a regular cache miss. Additionally, we ensure that the information held in the PQ is transferred to the MSHR entry allocated by the corresponding cache miss. Otherwise, the information of the PQ entry is discarded. Subsequent demand misses that find the MSHR entry allocated by a prefetch, i.e. with the access bit unset, simply enable the access bit. This indicates that the prefetch was late, as the access resulted in an L1I miss, despite the preceding prefetch.

The end timestamp corresponds to the time of the cache fill. Upon each cache fill, we check if the access bit in the MSHR is set, indicating that there was either a demand miss or a late prefetch. If in addition the entry has a valid pointer to the *History buffer*, we know that the miss corresponds to a basic block head. Under these conditions, the Entangling I-prefetcher attempts to find a *src-entangled* cache line for the newly cached line. The latency of the current memory access is computed by subtracting from the time of the cache fill (end) the timestamp recorded with the MSHR entry (start of the L1I miss). The source is then selected among the accesses that took place at least *latency* cycles before the miss and is identified by parsing backwards the *History buffer*, starting from the position of the current access in *History*. For misses without pointers to the *History buffer*, no *src-entangled* is searched for, as such misses will be covered by prefetching the full basic block starting from the head.

Once a *src-entangled* cache line is found, the *Entangled table* is updated and the corresponding *src-entangled* entry is paired with the newly cached line, which acts as *dst-entangled*. A *src-entangled* entry can have several entangled destinations. In the same way, a *dst-entangled* address can be paired with multiple *src-entangled* entries. This way, prefetching a cache line reached from different execution paths is automatically supported. The first and fourth column in Figure 5 illustrates when an entangled pair is added to the *Entangled table*.

3) ***Triggering the prefetches****:* The *Entangled table* is checked on cache accesses. In case of a hit (1) the entire basic block starting with the accessed cache line is prefetched (i.e., *size* lines starting from the second line in the basic block); (2) for each *dst-entangled*, the entire basic block starting from *dst-entangled* is prefetched (for finding its size, the *Entangled table* is parsed again using the *dst-entangled* address).

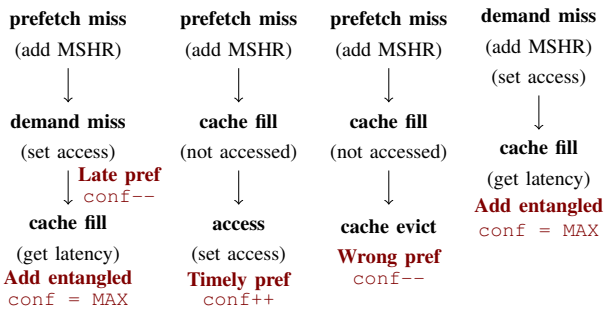

Fig. 5. Actions taken on prefetch misses and cache accesses, misses, fills, and evictions. Entanglements are added for misses and late prefetches by searching for a source in the *History buffer*. The confidence counter is increased on prefetch hits and decreased on late and wrong prefetches.

B. Advanced techniques and optimizations

1) ***Adding confidence to deal with latency variation****:* The latency of a cache miss may vary depending on whether the cache line is fetched from L2, LLC, or main memory, or even due to contention when accessing the structures or routing the packets through the interconnection network. Therefore, an entangled pair that is timely once may not be timely on the next occurrence. To adapt to such variations and trigger timely prefetches we add a *confidence* counter (a two-bit saturated counter) to each entangled pair.

New entangled pairs are added to the *Entangled table* with the *confidence* set to the maximum value, as they are expected to be timely. The counter is decreased by one upon late or wrong prefetches and increased by one upon timely prefetches. When *confidence* reaches 0, the entangled pair is considered invalid. When adding a new entangled pair, if the array of destinations is full, the *dst-entangled* with the lowest *confidence* is replaced.

In order to update the confidence, we need to know (1) if the prefetch is late, timely, or wrong, and (2) which is the entangled pair that triggered the prefetch. The first piece of information is provided by the access bit which is copied from the PQ to the MSHR when a prefetch misses in the L1I, and from the MSHR to the L1I on a cache fill. Late prefetches are detected when a cache miss finds the access bit unset in the MSHR (or the PQ). Timely prefetches are detected on a cache hit that finds the access bit unset. Wrong and early prefetches are detected upon a cache line eviction with the access bit unset (the line was unnecessarily brought, i.e., not accessed before being evicted). Figure 5 depicts these scenarios. The second piece of information just requires actually to know the *src-entangled* address, since the *dst-entangled* address is the prefetched one. Hence, the *src-entangled* address is also stored in the PQ, and moved accordingly to the MSHR and L1I.

We implemented a version of the Entangling prefetcher adding context information in order to increase accuracy. The source was replicated for each different context, resulting in an overloaded Entangled table, that suffers from frequent evictions and consequently achieves lower performance. Moreover, context-based predictions may not always be correct, missing opportunities. Other studies also report little benefit when adding context information (e.g., Markov Predcitors [23], end of Section.2.1).

TABLE I
COMPRESSION MODES OF *dst-entangled* BLOCKS

Mode	Destinations	$signifB$ bits	Size (bits)
1	1	[29, 58]	$(58+2)\times 1 = 60$
2	2	[19, 28]	$(28+2)\times 2 = 60$
3	3	[14, 18]	$(18+2)\times 3 = 60$
4	4	[11, 13]	$(13+2)\times 4 = 60$
5	5	[9, 10]	$(10+2)\times 5 = 60$
6	6	[1, 8]	$(8+2)\times 6 = 60$

2) ***Merging spatio-temporal basic blocks****:* Reducing the number of entangled basic blocks in the *Entangled table* is key for keeping storage overhead manageable. To this end, we perform a merge of quasi-consecutive (in time) basic blocks whose addresses overlap or are consecutive (in space). Merging is critical when our prefetcher employs a low-budget *Entangled table*.

Merging is also aimed to address scenarios such as the sequence of accessed cache lines: $ABCXCD$, in which a basic block head C always hits in the cache because it was prefetched as part of another basic block (ABC) and was not evicted. However, D may lead to a substantial number of misses that would not be covered by the Entangling I-prefetcher since it is not a basic block head.

To address this issue, the size of each basic block is added to the *History buffer*. The *History buffer* is inspected on each computed basic block, and if the basic block can be merged with one of the previous basic blocks in the history (i.e., they are consecutive or have overlapping addresses), the size of the previous basic block is updated (the size of the basic block starting with A would become 4, i.e., $ABCD$) and the merged basic block (starting with C) is not recorded in the *History*).

Since we dedicate 6 bits to store the size of a basic block, merging is not performed if the resulting basic block size is larger than 64 cache lines.

3) ***Compressing destinations****:* The *Entangled table* uses different modes for encoding the array of *dst-entangled* line address and *confidence* on 63 bits, as follows: 3 bits for the mode + 60 bits of the *dst-entangled* line address and the *confidence*. The destination bits encode the least significant bits ($signifB$) of the *dst-entangled* line, starting from the most significant bit that differs from the *src-entangled*. The most significant bits can be inferred from the source. Since the distance between *src-entangled* and *dst-entangled* is typically small, destinations can be highly compressed.

The mode is a value between 1 and 6 which indicates how many destinations can be kept in the 60 bits of the array of *dst-entangled* cache lines and the associated *confidence*. Depending on how many significant bits are required, the number of destinations can vary. If one destination is encoded, the full virtual address of the cache line is stored. For the

confidence we always use a 2-bit saturated counter. Table I details the available modes.

All entries of the same *dst-entangled* array must be represented in the same mode. Hence, every time a new *dst-entangled* entry is inserted, we compute the maximum between its mode and the mode of the previously recorded destinations. To improve compression, upon the eviction of a *dst-entangled* we re-compute the mode, to ensure that it is not unnecessarily set to a restricting value due to a destination that no longer exists.

Finally, to maximize the utilization of the *Entangled table* if the selected src-entangled line has not free destination entries, the prefetcher looks for a second *src-entangled* line, namely a cache line with the timestamp earlier than the one searched for. If the second line has not a free destination entry, the first line is inserted by evicting an old entry.

C. Further implementation details

1) ***Dealing with wrong-path execution****:* There are two problems arising from wrong-path execution: (i) polluting the cache with prefetches triggered on the wrong path and (ii) training the prefetcher with entangled pairs computed on the wrong path.

The solution to avoid issuing wrong-path prefetches is to trigger the prefetches when instructions retire, such that they are only issued on the correct path. In *Entangled*, the latency can be computed by accounting for the time the instruction takes to commit, in addition to the cache miss latency. However, even if the prefetch-on-retire solution is disregarded, the performance degradation may not be significant as the L1I is usually plenty of dead cache lines, that could be evicted without increasing miss rate [16]. Since the Entangling prefetcher commonly entangles the destination with the most recent timely source, it tolerates L1I evictions better than fixed look-ahead alternatives.

To avoid polluting the tables with wrong-path information, *Entangled* keeps the speculatively computed pairs in a separate structure until the destination instruction commits and then updates the *Entangled table*.

2) ***Timing constraints****:* When triggering the prefetches, on a hit in the *Entangled table*, a maximum of 6 extra searches (average of 2.5, as presented in Section IV-D) are performed in the *Entangled table* to find the basic block sizes of the entangled destinations. The *Entangled table* is indexed with a simple XOR operation of the different bits of the address, and the 16 ways are searched in parallel. The time needed to retrieve the prefetching information is accounted for in the latency of the prefetches so that they are timely, regardless the latency of accessing the *Entangled table*. All other updates are done out of the critical path of issuing prefetches.

3) ***Memory Overhead****:* The *History buffer* is a 16-entry circular queue, with a 58-bit tag field, a 20-bit timestamp field, and a 6-bit basic block size field. A 4-bit register points to the head of the queue. The maximum basic block size is therefore 63 cache lines. The total memory required by this structures is 167 bytes.

The *timing and src-entangled information* is stored along with PQ (32 entries), MSHR (10 entries) and L1I cache (512 entries). The timing information consists of the time the request was issued (12 bits) and the position of the access in the *History buffer* (4 bits). The *src-entangled* information includes the position of the source in the *Entangled table* (4 bits for the way since we model 16-way entangled tables and 7, 8, or 9 bits for the set, depending of the size of the *Entangled table*: 2K, 4K or 8K entries, respectively) and an access bit. Once the miss is resolved, the timing information is no longer necessary, thus the L1I cache only records the *src-entangled* information. The total memory required to store the timing and *src-entangled* information is about 1KB (915 bytes, 984.25 bytes, and 1053.5 bytes for the 2K, 4K, and 8K entries configurations, respectively).

The *Entangled table* is a large set-associative cache that stores sources along with their maximum basic block size and destinations. It employs an enhanced FIFO replacement policy, in which the information in the entry selected for eviction can be reallocated to another entry that does not hold any entangled pair. It has 128, 256, or 512 sets (2K, 4K, and 8K entries configurations, respectively) and 16 ways per set. The tags are encoded using 10 bits, the basic block is encoded with 6 bits, and the format, destinations, and confidence bits are encoded on a total of 63 bits. This is the largest structure employed by our prefetcher and requires 19.81KB (2K entries), 39.63KB (4K entries), or 76.25KB (8K entries). Storing basic block sizes and entangled pairs in different structures is an alternative to a unified *Entangled table*, likely beneficial for low-storage configurations. We leave this study for future work.

4) ***Physical addresses****:* Recent ARM-based architectures employ large virtual L1 caches and as consequence can efficiently train the L1 prefetcher with virtual addresses [18]. However, for x86 cores employing smaller virtually-indexed physically-tagged caches [20], placing the prefetcher in the physical address space is a likely alternative. Otherwise L1 accesses performed by the prefetches would add critical pressure to the translation look-ahead buffer (TLB). Although we have described our design for virtual addresses, it can perfectly work on physical address space, even reducing the storage requirements, e.g., for a 48-bit physical address space.

If the prefetcher is trained with physical addresses, the compression mechanism can be adapted using 46 bits for (*dst-entangled* line addresses and *confidence*) as follows: 2 bits for the mode + 44 bits of the *dst-entangled* line address and the *confidence*. The mode takes a value between 1 and 4 which indicates how many destinations can be kept in the 44 bits of the array of *dst-entangled* cache lines and the associated *confidence*, as detailed in Table II. Additionally, the History buffer holds cache line addresses represented on 42 bits instead of 58 bits for virtual. This way, our L1I Entangling prefetcher requires 16.59KB, 32.21KB, and 63.40KB for the versions with 2K, 4K, and 8K entries, respectively.

TABLE II
COMPRESSION MODES USING PHYSICAL ADDRESSES

Mode	Destinations	$signifB$ bits	Size (bits)
1	1	[21, 42]	$(42+2)\times 1 = 44$
2	2	[13, 20]	$(20+2)\times 2 = 44$
3	3	[10, 12]	$(12+2)\times 3 = 42$
4	4	[1, 9]	$(9+2)\times 4 = 44$

IV. EVALUATION

A. Methodology

We evaluate the Entangled I-prefetcher using a modified version of the ChampSim simulator [1] employed for the 1st Instruction Prefetching Championship (IPC-1). The ChampSim version used for IPC-1 modeled a simple front-end [21]. Our modified version extends the current model of the develop branch in ChampSim [2], which implements a more realistic decoupled front-end modeling Fetch-Directed Prefetching [38], a Branch Target Buffer (BTB), a Target Cache to predict the target of indirect branches [9], and a return address stack (RAS). Prefetches issued by the Fetch-Directed Prefetching engine are considered demand accesses, hence our baseline does not report any prefetch request. We extended it to model an out-of-order processor with a seven-stage pipeline as described by González et al. [17] and different branch misprediction penalty (number of stages to be flushed) depending on the stage the misprediction is detected. The processor and memory hierarchy parameters aim to resemble the latest Intel's Sunny Cove machine. The main configuration parameters of our baseline system are shown in Table III. The energy consumption of the cache hierarchy, taking into consideration the energy expenditure of tag accesses, reads, and writes to caches, has been modeled with CACTI-P [32] for a 22nm process technology.

The version of ChampSim employed for the IPC-1 trained the L1I prefetchers with virtual addresses. However, as L1I prefetchers can also work with physical addresses, we also provide results training the L1I prefetchers with physical addresses. In that scenario, there is no guarantee that two consecutive virtual memory pages are consecutive in the physical space, slightly reducing the prefetcher's coverage.

ChampSim does not simulate wrong-path execution, and therefore no wrong-path prefetches are issued. Consequently, in a more realistic implementation, the accuracy of the prefetchers may be reduced. As we explain in Section III-C1, the Entangling prefetcher can avoid wrong-path pollution. All prefetchers evaluated in this work benefit from not modeling the wrong path. We leave for future work examining in detail the implications of wrong-path execution.

We test our prefetcher on the *large secret* traces provided in the 1st and 2nd Championship Value Prediction (CVP) [3] and created by Qualcomm Datacenter Technologies.[4] The traces, which include a set of integer (compute_*int*), floating point (compute_*fp*), cryptography (*crypto*), and server (*srv*) workloads, have been ported to the ChampSim format. We selected the 959 workloads that showed at least 1 MPKI (miss per kilo-instruction) at the L1I in our baseline configuration. The complete analysis has been performed with this set of benchmarks. Additionally, to evaluate our prefetcher on a larger variety of benchmarks with different behaviours, we include performance results on applications from the Cloud-Suite [12] that exhibit at least 1 MPKI in the L1I. Workloads run until the end after a short 20M-instruction warm-up.

[4]IPC-1 used a subset of these traces for evaluating the prefetchers.

TABLE III
BASELINE SYSTEM CONFIGURATION

Processor decoupled front-end	
Width	6 instructions
Fetch queue	64 entries
Decode queue	32 entries
Dispatch queue	32 entries
Branch target buffer	8K entries
Target cache	4K entries
Return address stack	64 entries
Branch penalty	2 cycles (decode stage)
Branch predictor	Hashed perceptron
Processor back-end	
Execute width	4 instructions
Retire width	5 instructions
Branch penalty	7 cycles (execute stage)
Re-order buffer	352 entries
Load, store queue	128, 72 entries
Memory hierarchy	
L1I cache	32KB, 8-way, 4 hit cycles, no prefetcher
L1-D cache	48KB, 12-way, 5 hit cycles, next-line
L2 cache	512KB, 8-way, 10 hit cycles, spp-dev [28]
L3 cache	2MB, 16-way, 20 hit cycles, no prefetcher
DRAM	4 GB, one 8-byte channel, 1600MT/s

B. Evaluated prefetchers

We evaluate in detail several state-of-the-art prefetching strategies:

- *Next-line* [8]: A pure next-line prefetcher that always prefetches the next cache line given the current access. It adds no area overhead.
- *SN4L* [6] is a memory-efficient proposal that implements a 16K-bit vector, where the next four cache lines of the current access are prefetched if the corresponding bit is set, that is, if prefetching that cache line is expected to be useful. It requires only 2.06KB of storage.
- *MANA* [5] is a refinement of SN4L-Dir-BTB [6] that uses an 8-bit vector for consecutive prefetchers (previously proposed by PIF [14]). It offers a good performance-area trade-off and it is representative of state-of-the-art BTB-directed instruction prefetchers. We evaluate the two low-cost configurations described by Ansari *et al.* [5]: A 2K-entry MANA table (9KB) and a 4K-entry MANA table (17.25KB). We also show geometric IPC for an 8K-entry MANA table that requires 74.18KB.
- *RDIP* [29], [34] is a RAS-directed instruction prefetcher. It records the return address stack and its context as signatures which are then consulted upon each call and return

operations to trigger prefetching. We evaluate a 4K-entry miss table with 3 trigger prefetchers for discontinuities and an 8-bit vector for consecutive cache lines. The total storage is 63KB.

We also evaluate the three first ranked proposals in IPC-1:

- *D-JOLT* [35] is a refinement of RDIP. First, it implements more accurate context-based signatures. Second, it uses a dual look-ahead distance mechanism to generate prefetches. We evaluate an 8KB entry miss table, which gives a total storage of 125KB.
- *FNL-MMA* [44] combines a Footprint Next Line (FNL) prefetcher and a Multiple Miss Ahead (MMA) prefetcher. FNL is an enhanced next line prefetcher that estimates if a cache line is worth to prefetch, while MMA selects the look-ahead distance. We evaluate an 8K entry miss table which gives a total storage of 97KB.
- *EPI* [41] a performance-oriented and hardly implementable –as the previous two version– of the Entangled prefetcher. It models highly associative structures (e.g., a +1000-entry history buffer, and a 34-way *Entangled table* which gives +8K entries). Its total storage requirements are 127.9KB.

In addition, we also evaluate three different configurations of our cost-effective Entangling prefetcher:

- *Entangling* is our proposed Entangling prefetcher. We model three different sizes for the *Entangled table*: 2K, 4K, and 8K entries. We perform a more aggressive merging of basic blocks in the low-budget configurations, considering merging distances of basic blocks recorded in the history buffer of 15, 6 and 5 for the 2K, 4K, and 8K configurations, respectively. The area requirements as computed in Section III-C3 are 20.87KB, 40.74KB, and 77.44KB, respectively.

Finally, we also show the effect on increasing the cache size, instead of using the budget for the prefetching mechanism:

- *L1I-64KB* and *L1I-96KB* increase the associativity of the L1I from 8 ways to 16 ways and 24 ways, respectively, while keeping the L1I access latency to 4 cycles.
- An *Ideal* instruction prefetcher where the L1I cache always returns a hit [34]. It issues all necessary prefetches to the next level cache, thus modeling the pollution entailed by the L1I cache.

C. Performance results

We evaluate our Entangling prefetcher and compare it with state-of-the-art prefetchers, presenting a number of different metrics: coverage (percentage of L1I misses covered by prefetching), accuracy (percentage of useful prefetches with respect to the total number of prefetches issued), the L1I miss ratio, and instructions per cycle (IPC) as an indication of performance. Since we evaluate for a large number of proposals and applications (10000+ simulations), we first offer the geometric mean of the instructions per cycle of all the evaluated schemes and then focus on the prefetching techniques that require less that 64KB of storage.

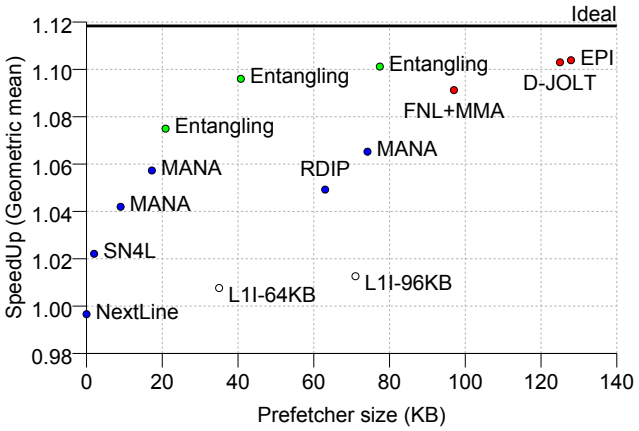

Fig. 6. IPC vs memory requirements.

*1) **Performance vs storage**:* Figure 6 summarizes the performance of all the prefetchers (geometric mean of the IPC obtained for the 959 CVP workloads, normalized with respect to our baseline) along with their storage requirements. The medium- and low-budget configurations are shown in blue, the high-budget configurations presented in IPC-1 in red, our Entangling prefetcher in green, the large L1I configurations in white, and the Ideal prefetcher as a black line. The Entangling prefetcher achieves 10.1% speedup with respect to the baseline configuration when using 77.44KB (Entangling-8K), while an Ideal L1I cache would offer 11.8% speedup. More interestingly, with 40.74KB overhead, Entangling-4K offers a good area-performance balance achieving 9.60% performance improvements, on par with other area demanding proposals: FNL+MMA 9.12% with 97KB overhead, D-JOLT 10.3% with 125KB, and EPI 10.4% with 127.9KB). For a low-budget configuration of 20.87KB, Entangling-2K offers good performance improvements of 7.50%. Entangling also outperforms all low-budget configurations of MANA and furthermore, the low-budget Entangling version outperforms the high-budget version of MANA.

*2) **IPC**:* Figure 7 shows the IPC normalized to a configuration without any L1I prefetcher across the CVP workloads. The normalized IPCs have been individually ordered from lower to higher for each configuration. Both the low- and medium-budget configurations of Entangling outperforms the other state of the art prefetchers. The medium-budget configuration of the Entangled prefetcher (4K) is very close to the ideal for many workloads, and only for a few of them, an ideal prefetcher gets significant improvements with respect to our proposal. More importantly, the Entangling prefetcher never gets performance degradation with respect to not using any prefetcher, as it clearly happens with a NextLine prefetcher, and other techniques.

*3) **L1I miss rate**:* Figure 8 shows the L1I miss ratio for the CVP workloads, again individually ordered from lower to higher for each configuration. The line labeled as *no* is the baseline configuration without a dedicated L1I prefetcher. The

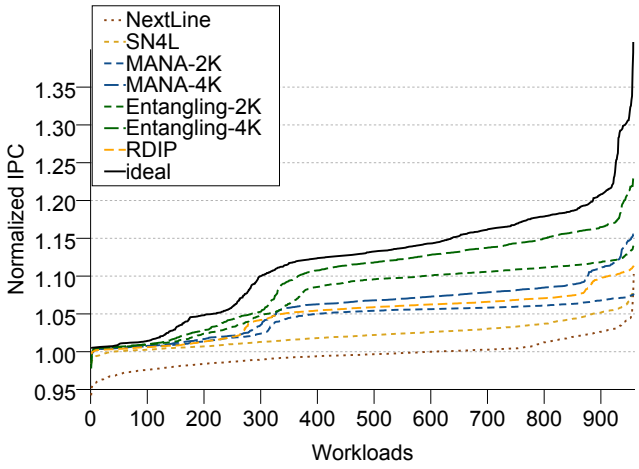

Fig. 7. IPC normalized to our baseline configuration.

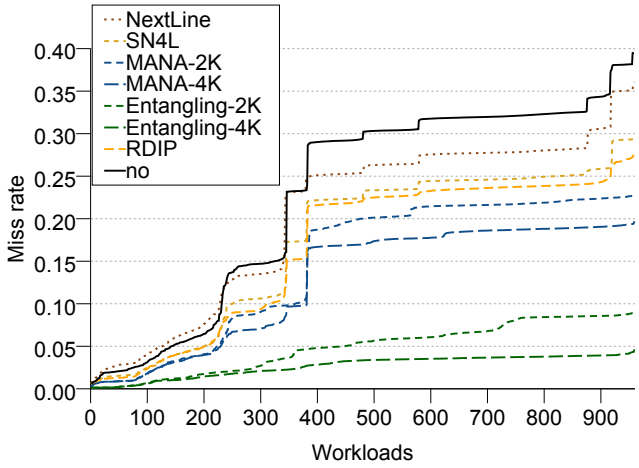

Fig. 8. Miss rate of the instruction prefetchers.

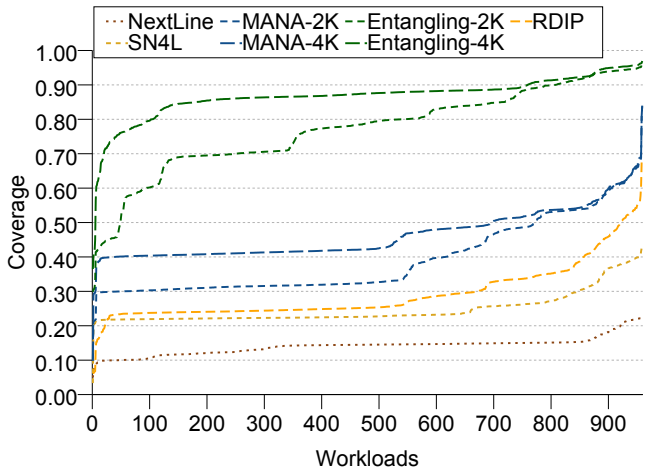

Fig. 9. Coverage of the instruction prefetchers.

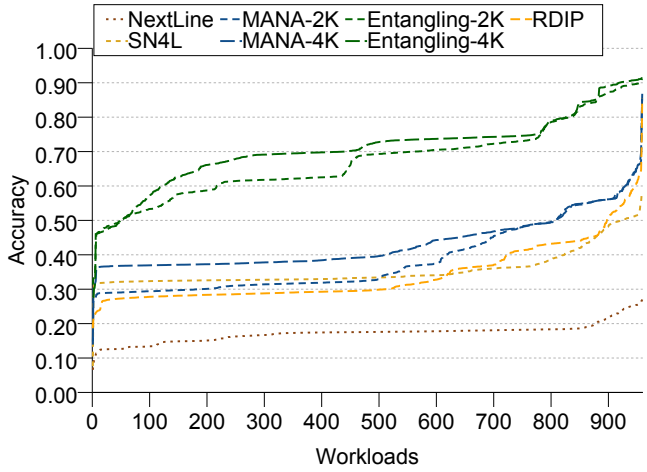

Fig. 10. Accuracy of the instruction prefetchers.

Entangling prefetcher significantly outperforms its competitors across all benchmarks, reducing drastically the miss rate. In the worst case, the Entangling prefetcher reduces the miss rate to just 10% when using 2K entries and to 5% miss rate when using 4K entries. The other evaluated prefetchers report a worst case of 20% miss rate or more. This way, the Entangling prefetcher approaches an ideal L1I cache.

*4) **L1I prefetcher coverage**:* Figure 9 shows the coverage (ratio of misses that became hits) of all prefetchers across all workloads, individually ordered from lower to higher. Mimicking the miss rate figure, the Entangling prefetcher shows a much higher coverage than the state-of-the-art prefetchers. For most workloads, Entagling-4K shows a coverage around 90%. Entangling-2K offers a coverage higher than 68% for most workloads. In contrast, the other prefetchers offer a coverage below 50% for most workloads.

*5) **L1I prefetcher accuracy**:* Figure 10 shows the accuracy (ratio of useful prefetches) across all workloads, individually ordered from lower to higher. Entangling achieves highest accuracy, being above 50% for most workloads, and reaching 90% accuracy for almost 10% of the workloads. RDIP is below 50% accuracy for more than 90% of the workloads, and MANA is below 50% for more than 80% of the workloads. The high accuracy of Entangling indicates that it is the most energy efficient prefetcher in terms of prefetches issued to the higher cache levels (L2, LLC, Memory), as it generates less useless traffic.

*6) **Energy consumption**:* Accuracy is a representative indicator of the energy expenditure, as a 100%-accurate prefetcher generates no extra traffic to the higher memory levels with respect to no prefetching. A non-accurate L1I prefetcher would pollute not only the L1I with accesses and cache lines, but also the L2 and LLC caches with unnecessary requests. Table IV shows the energy expenditure for the caches employed in our system configuration. The Entangling prefetcher has the highest accuracy among the studied prefetchers, thus reducing the energy consumption at the L2 and LLC considerably (38.6% on average compared to NextLine). When accounting

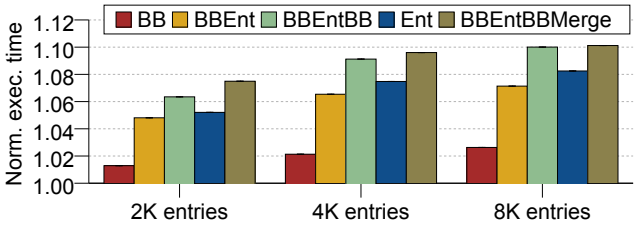

Fig. 11. Breakdown of the contributions to performance.

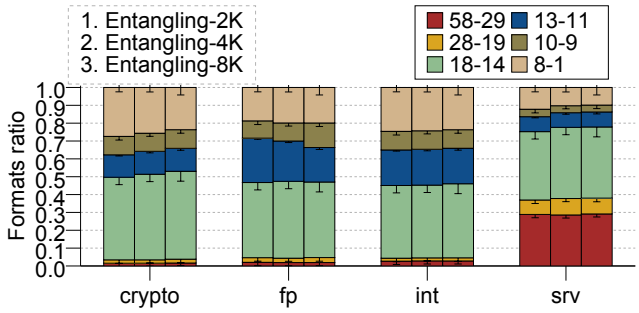

Fig. 12. Compressed format for each destination inserted in the *Entangled table*.

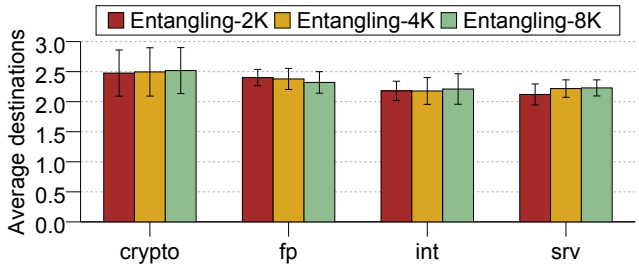

Fig. 13. Average number of entangled destinations.

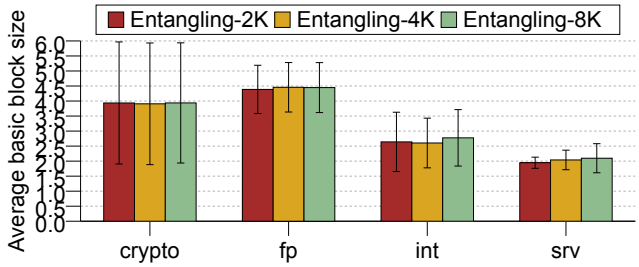

Fig. 14. Average basic block size for accesses hitting in the *Entangled table*.

also for the extra L1I accesses generated by the prefetches, the Entangled prefetcher still reduces the overall energy consumption of the memory hierarchy by 2.97%, 3.35%, and 3.39% for the 2K, 4K, and 8K configurations, respectively. In contrast, the most energy-efficient technique, RDIP, adds very few prefetches, and therefore few new accesses, however, many of these prefetches are late.

D. Analyzing the Entangling Prefetcher

We first detail the average performance obtained by isolating the proposed techniques for the three analyzed configurations of the Entangling prefetcher (Figure 11). *BB* prefetches the whole basic block on the first access to its head (source). *BBEnt* extends *BB* and prefetches each *dst-entangled* cache line, while *BBEntBB* extends *BB* by prefetching each *dst-entangled* basic block. *Ent* does not track basic blocks and entangles all cache lines missing in the cache. Finally, *BBEntBB-Merge* is our proposal, which extends *BBEntBB* with the mechanism for merging basic blocks. The key improvements come from entangling pairs of cache lines in a timely manner. Prefetching the *dst-entangled* basic block (*BBEntBB*) contributes to some extent to the improvements as less entangled pairs are required to be tracked with respect to *Ent*. Merging is more relevant for the smaller sizes of the prefetcher, since compression is essential for fitting in the reduced storage budget.

We also analyze the compression ratio of *dst-entangled* cache lines by showing in Figure 12 in which format are they represented. We present for each category of workloads provided in CVP (*crypto*, *int*, *fp*, and *srv*) the arithmetic mean and standard deviation of their workloads. We can observe that almost all destinations can be highly compressed in *crypto*, *fp* and *int* workloads. However, in *srv* workloads there is a non-negligible fraction of destination that cannot be compressed. Still the compression rate is quite high in *srv* workloads, and the majority of the destinations are stored using just 18 bits. The fraction of destinations compressed using just 8 bits ranges from ≈25% in *crypto* and *int* to ≈10% in *srv*. Overall, the average number of destinations found on a hit in the *Entangled table* ranges from 2.5 for *crypto* workloads to 2.2 for *srv* workloads (Figure 13).

Finally, we compute the number of prefetches issued on each hit in the *Entangled table*. Figure 14 and Figure 15 show the average number of cache lines of the currently accessed basic block (omitting the first cache line) and the number of cache lines for the basic blocks of the entangled destinations (omitting the first cache line), respectively. Then, we can compute the average number of triggered prefetches with the following formula: bbsize + destinations * (1 + bbsize_destination). This results in a number of prefetches ranging form ≈9 in *srv* workloads to ≈17 in *fp* workloads. Although this number is not dramatically high (our Entangling prefetcher shows the best accuracy among the evaluated prefetchers – Figure 10), our prefetcher would benefit from a larger prefetch queue (32 entries employed in our evaluation), as less prefetches would be discarded.

E. Working with physical addresses

We evaluated all the state-of-the-art prefetchers trained for physical addresses. Our results on the CVP workloads show that the Entangling prefetcher outperforms its competitors, achieving an IPC improvement (geometric mean) over our no-prefetch baseline of 5.62%, 8.10%, and 8.87% when using 2K,

TABLE IV
AVERAGE ENERGY CONSUMED AT EACH CACHE LEVEL (IN NJ) AND GEOMETRIC MEAN OF NORMALIZED ENERGY

	no	NextLine	SN4L	MANA-2K	MANA-4K	Entangling-2K	Entangling-4K	RDIP
Average L1I energy (nJ)	129222	144473	155196	162745	166312	185197	191868	144604
Average L1D energy (nJ)	329322	328837	327665	326637	325902	324513	323331	326236
Average L2C energy (nJ)	138535	125477	115300	107774	99581	78754	73942	76449
Average LLC energy (nJ)	78039	92826	78105	69385	65995	63543	60117	62095
Geomean (norm.)		1.0250	1.0043	0.9914	0.9786	0.9703	0.9665	0.9082

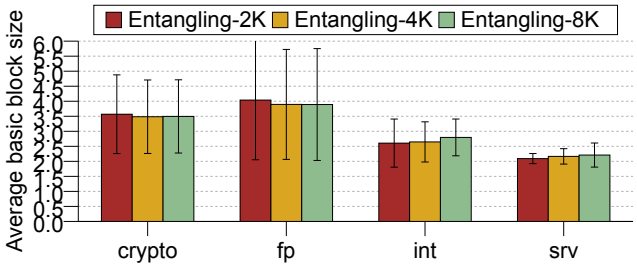

Fig. 15. Average basic block size of entangled destinations.

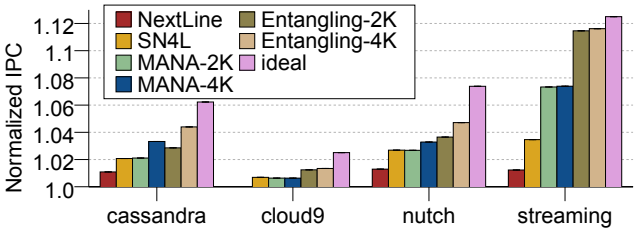

Fig. 16. Normalized IPC for CloudSuite applications.

4K and 8K entries in the *Entangled table*, respectively. The trends are similar to the ones observed for virtual addresses.

F. Other applications: the CloudSuite

Finally, we evaluate the prefetchers on a new class of benchmarks specifically designed to represent Cloud applications, with behaviors that may differ from the ones observed in the previous set of workloads. Figure 16 shows the performance improvements of our Entangled prefetcher for the applications of CloudSuite [12] that show more than 1 MPKI in L1I. Again, we can observe that the Entangling prefetcher outperforms the other state-of-the-art prefetchers we have evaluated.

V. RELATED WORK

Driven by their impact on performance, prefetchers have evolved from simple *next line prefetchers* [8], [45], to more complex techniques, as described below.

A common prefetching technique is the *look-ahead*, which follows the execution path *n* steps in advance and prefetches the corresponding block (instruction, cache line, etc.). One of the most recent and competitive look-ahead I-prefetchers is the FNL+MMA [44], which combines a Footprint Next Line (FNL) prefetcher and a Multiple Miss Ahead (MMA) prefetcher. FNL is an enhanced next line prefetcher that estimates if a block is worth to prefetch, while MMA identifies a "good-enough" look-ahead distance (*n*) and combines it with a technique to predict the *n*th next L1I miss. Yet, as we have shown, a fixed look-ahead distance impacts the prefetcher's accuracy and efficiency. Techniques to adjust the look-ahead distance dynamically using heuristics have also been proposed [28], [47], however, the look-ahead still remains the same (fixed) for all cache misses during certain execution windows.

By entangling cache misses, the Entangling prefetcher goes along the lines of correlation-based prefetchers [13], [15], [19], [23], [29], [35]. Markov-based prefetchers [23] use probabilities to predict and prefetch the next cache miss with a fixed look-ahead distance and select to prefetch all or some of the following predicted misses. TIFS [15] records the history of L1I misses and predicts the next miss and the number of cache lines to be cached, thus being more accurate and timely than simple, next-line prefetchers. The Proactive Instruction Fetch (PIF) prefetcher [13] improves performance by capturing the cache lines accessed by the committed instructions and instructions from handlers for OS interrupts. PIF operates on the correct-path, retire-order instruction stream, and records the exact instruction fetch sequence which is then used to compute spatial locality. This technique results in a 99.5% I-hit rate on the evaluated benchmarks, but incurs substantial storage overhead (beyond the limits considered in our evaluation). To capture the context of a miss caused by a function call, Return-address stack-directed instruction prefetching (RDIP) [29] records the return address stack and its context as signatures which are then consulted upon each call and return operations to trigger prefetching. RDIP approaches the performance of PIF (within 2%) with significantly lower storage demands, while Entangling significantly outperforms RDIP (8%). A refined solution, D-JOLT [35] builds more accurate context-based signatures and further improves performance over its predecessor (RDIP), but it entails a large memory overhead to reach the performance reported by our prefetcher.

While Entangling and other correlating prefetchers (e.g. Markov, look-ahead based prefetchers such TIFS, PIF, etc) share similarities (events are correlated), the main difference consists in the way these correlations are built: based on *distance* (number of branches, accesses, misses, etc) vs. correlations (entanglings) based on *timeliness* (latency expressed in cycles). Previous correlation-based prefetchers use a fixed look-ahead distance, which we show in our motivation figures 1 and 2 that it cannot timely serve all misses.

For increasing accuracy, prefetchers that interact with the branch prediction mechanism have been proposed [10], [26], [30], [31], [36], [39], [46], [48], [50]. For instance, Kumar et al [30] leverage the branch target buffer (BTB) and simultaneously prefill the BTB with the branch instructions of each decoded block of instructions, in order to avoid BTB misses. This is achieved by leveraging the information of the I-prefetcher without adding any BTB storage overhead. Generally, instructions prefetchers that depend on the BTB (i.e. BTB-directed-prefetches) are considerably hindered by BTB misses and require significant changes in the processor [31]. Even attempts to prefill the BTB [26], [30] suffer of high BTB miss rates for applications with very large instruction footprints (e.g. server workloads, the same that usually incur L1I misses). Shotgun [30] dedicates a significant fraction of the BTB to unconditional (U-)branches and a smaller fraction to conditional (C-)branches, plus a third fraction for return instructions. While the U-branches are handled well thanks to the large dedicated storage, Shotgun remains ineffective for workloads with high C-branches miss rates, due to its mechanism to reactively pre-fill such branches. More recently, Ansari et al [6] propose SN4L-Dis-BTB, a lightweight prefetcher that reduces storage demands. SN4L-Dis-BTB classifies the misses in three categories and provide tailored solutions for each: (1) an enhanced N4L prefetcher to detect worthy-to-prefetch blocks dedicated to cover sequential misses, (2) a discontinuity prefetcher – based on the observation that discontinuities are introduced by branches – designed with extra care for storage and aimed to cover the remaining misses, and finally (3) a Confluence [26]-like solution that pre-fills the BTB to avoid BTB misses. While this proposal is competitive for very low storage budgets, it cannot fully leverage a larger storage to offer high-performance. MANA [5], a follow-up version designed for higher budgets, brings some performance improvements, but is still less competitive than our Entangling prefetcher. Overall, the BTB-guided prefetchers are highly sensitive to BTB misses and branch prediction accuracy. While our Entangling prefetcher operates on basic block heads – which can be seen as branch targets, Entangling is not hindered by BTB misses, being based on cache events correlations, rather than following the branches to issue the prefetch. By learning and building correlations on all (or most frequent) execution paths, Entangling is not sensitive to the accuracy of the branch predictor and has its own mechanisms to deal with changes in execution paths (multiple sources for the same destination, confidence counters, etc). This approach based on correlating cache misses instead of focusing on the prediction of next branch target addresses yields Entangling more robust and less sensitive to predictions, compared to other state-of-the-art techniques.

Other designs used stream buffers [24] as additional hardware structures to prefetch sequences of successive cache lines starting at the miss target, akin to the basic block heads used by our prefetcher.

Rather than primarily targeting coverage, then accuracy, then latency, Entangling makes a bold step and targets timeliness first, using a novel approach that proves to be highly effective, thus achieving higher accuracy than its predecessors.

VI. Conclusions

The Entangling prefetcher for instructions offers an alternative prefetching direction driven by timeliness. Entangling estimates the latency of the cache missing operations and *entangles* them with the instructions that should trigger the prefetch to ensure the timely arrival of the requested instructions. It pairs sequences of mostly sequential cache lines: the source sequence gets associated with one or more destination sequences. On a cache access to the first cache line in the source sequence, Entangling generates prefetches for all lines in the source sequence as well as for all lines in the associated destination sequences (if confident). The pairing between source and destinations is done by measuring the cache miss latency of the first line in a sequence, then using an auxiliary structure to locate an earlier sequence that started more than *latency* cycles ahead of the current sequence, and adding the current sequence to the destination sequence list.

By considering the sequences of cache lines granularity, Entangling subsumes the next line prefetching. It also uses a novel compression scheme, depending on the distance relationship between the source and destinations, and a clever encoding and table organization, which keeps storage at bay. The design does not require access to the branch prediction structures, does not add contention to the critical structures, and does not entail large associative searches. Thus, the implementation of the Entangling prefetcher is highly efficient without being intrusive in the processor design. It is robust and effective, agnostic to the application characteristics and achieves a 97.6% L1I hit rate, approaching a perfect L1I, clearly outperforming state-of-the-art proposals and offering a good area-performance trade-off.

Acknowledgment

We would like to thank Ali Ansari, Gino Chacon, Nathan Gober, Daniel Jiménez, Tomoki Nakamura, Seth Pugsley, and André Seznec for their contributions to the ChampSim ecosystem, help in evaluating related work, and discussions.

References

[1] "ChampSim simulator," http://github.com/ChampSim/ChampSim, May 2020.

[2] "ChampSim simulator, develop branch," https://github.com/ChampSim/ChampSim/tree/develop, Nov. 2020.

[3] "The Second Championship Value Prediction," https://www.microarch.org/cvp1/, Nov. 2020.

[4] T. M. Aamodt, P. Chow, P. Hammarlund, H. Wang, and J. P. Shen, "Hardware support for prescient instruction prefetch," in *11th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2004, pp. 84–95.

[5] 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.

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

[7] 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.

[8] J.-L. Baer, *Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors*, 1st ed. Cambridge University Press, 2009.

[9] P.-Y. Chang, E. Hao, and Y. N. Patt, "Target prediction for indirect jumps," in *24th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 1997, pp. 274–283.

[10] I.-C. K. Chen, C.-C. Lee, and T. N. Mudge, "Instruction prefetching using branch prediction information," in *1997 Int'l Conf. on Computer Design (ICCD)*, Oct. 1997, pp. 593–601.

[11] B. Falsafi and T. F. Wenisch, *A Primer on Hardware Prefetching*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2014.

[12] M. Ferdman, A. Adileh, Y. O. Koçberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *17th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Mar. 2012, pp. 37–48.

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

[14] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi, "Cuckoo directory: A scalable directory for many-core systems," in *17th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2011, pp. 169–180.

[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, D. A. Jiménez, and P. Gratz, "Temporal ancestry prefetcher," in *The 1st Instruction Prefetching Championship (IPC1)*, May 2020.

[17] A. González, F. Latorre, and G. Magklis, *Processor Microarchitecture: An Implementation Perspective*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011.

[18] 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.

[19] Z. Hu, M. Martonosi, and S. Kaxiras, "Tcp: Tag correlating prefetchers," in *9th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2003, pp. 317–326.

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

[21] Y. Ishii, J. Lee, K. Nathella, and D. Sunwoo, "Rebasing instruction prefetching: An industry perspective," *IEEE Computer Architecture Letters*, Oct. 2020.

[22] A. Jimborean, K. Koukos, V. Spiliopoulos, D. Black-Schaffer, and S. Kaxiras, "Fix the code. don't tweak the hardware: A new compiler approach to voltage-frequency scaling," in *12th Int'l Symp. on Code Generation and Optimization (CGO)*, Feb. 2014, pp. 262—-272.

[23] D. Joseph and D. Grunwald, "Prefetching using markov predictors," in *24th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 1997, pp. 252–263.

[24] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *17th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 1990, pp. 364–373.

[25] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *42nd Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2015, p. 158–169.

[26] C. Kaynak, B. Grot, and B. Falsafi, "Confluence: Unified instruction supply for scale-out servers," in *48th Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2015, pp. 166–177.

[27] 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.

[28] J. Kim, S. H. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in *49th Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2016, pp. 60:1–60:12.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] P. Michaud, "Best-offset hardware prefetching," in *22nd Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Mar. 2016, pp. 469–480.

[34] T. Nakamura, "ChampSim," https://github.com/tomokinex/ChampSim, Nov. 2020.

[35] T. Nakamura, T. Koizumi, Y. Degawa, H. Irie, S. Sakai, and R. Shioya, "D-jolt: Distant jolt prefetcher," in *The 1st Instruction Prefetching Championship (IPC1)*, May 2020.

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

[37] A. Ramirez, O. J. Santana, J. L. Larriba-Pey, and M. Valero, "Fetching instruction streams," in *35th Int'l Symp. on Microarchitecture (MICRO)*, Nov. 371–382, pp. 3–14.

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

[39] 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.

[40] A. Ros, "Berti: A per-page best-request-time delta prefetcher," in *The 3rd Data Prefetching Championship*, Jun. 2019.

[41] A. Ros and A. Jimborean, "The entangling instruction prefetcher," in *The 1st Instruction Prefetching Championship (IPC1)*, May 2020.

[42] ——, "The entangling instruction prefetcher," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 84–87, Jul. 2020.

[43] 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.

[44] A. Seznec, "The fnl+mma instruction cache prefetcher," in *The 1st Instruction Prefetching Championship (IPC1)*, May 2020.

[45] A. Smith, "Sequential program prefetching in memory hierarchies," *IEEE Computer*, vol. 11, no. 12, pp. 7–21, Dec. 1978.

[46] L. Spracklen, Y. Chou, and S. G. Abraham, "Effective instruction prefetching in chip multiprocessors for modern commercial applications," in *11th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2005, pp. 225–236.

[47] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *13th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2007, pp. 63–74.

[48] 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.

[49] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, "Slipstream processors: Improving both performance and fault tolerance," in *9th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Nov. 2000, pp. 257–268.

[50] A. V. Veidenbaum, "Instruction cache prefetching using multilevel branch prediction," in *1997 High Performance Computing, Int'l Symp. (ISHPC)*, Nov. 1997, pp. 51–70.

[51] 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.

[52] C. B. Zilles and G. S. Sohi, "Execution-based prediction using speculative slices," in *28th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2001, pp. 2–13.