The Entangling Instruction Prefetcher

Alberto Ros, *Senior Member, IEEE,* Alexandra Jimborean

Abstract—Prefetching instructions 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 an essential property, as bringing instructions too early increases the risk of the instructions being evicted from the cache before their use while requesting them too late can lead to the instructions arriving past their designated execution time. Coverage is important to reduce the number of instruction cache misses (there is enough prefetching), and accuracy to ensure that the prefetcher does not pollute the cache or interacts negatively with the other hardware mechanisms (there is not too much prefetching).
This paper presents the Entangling instruction prefetcher that entangles instructions to provide 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 the Entangling I-prefetcher increases performance by 29.3% on average, with a coverage of 94.9% and accuracy of 77.4%.

Index Terms—Instruction prefetcher, timely prefetching, performance.

1 Introduction

Instruction fetch stalls block the processor pipeline, causing significant performance degradation. In particular, applications with large working instruction sets that do not fit in the first level cache, such as server applications or applications designed to run in the Cloud, exhibit large instruction-cache miss rates and thus incur more stalls. In such cases, instruction fetching represents a considerable fraction of the memory stalls, together with data accesses.

As memory latency has been recognized as a critical factor for performance, prefetching techniques have emerged to install the data or instructions in the cache ahead of time, ready to be used when demanded by the processor [1]. Driven by their impact on performance, prefetchers have evolved from simple *next line prefetchers*, to complex techniques, such as the Proactive Instruction Fetch prefetcher [2] captures the blocks accessed by the committed instructions and instructions from handlers for OS interrupts. The Return-address stack-directed instruction prefetching (RDIP) [3] captures the context of a miss caused by a function call as signatures which are then consulted upon each call and return operations to trigger prefetching. More recently, Ansari et al [4] propose a lightweight prefetcher that reduces storage demands. We propose the Entangling Instruction Prefetcher,[1] which, in contrast to its predecessors, is designed around the notion of timeliness. The Entangling I-prefetcher 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. In this way, the Entangling I-prefetcher is robust and effective, agnostic to the application characteristics and achieves a 99.3% I-hit rate, approaching the perfect L1-I.

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

1. A performance-oriented version of the Entangling Instruction Prefetcher won the 1st Instruction Prefetch Championship (IPC1) [5].

2 The Entangling I-Prefetcher

The Entangling I-prefetcher entangles distant and unrelated operations, which, intuitively, translates to pairing two instructions, the instruction i_{src} upon whose execution should be triggered the prefetch for the instruction i_{dst}. More precisely, we define as *src-entangled* the cache line that should trigger the prefetch of the *dst-entangled* cache line such that the requested line arrives timely.

To ensure timeliness, we first compute the latency of each cache miss. To this end, the Entangling I-prefetcher starts by recording the history of L1-I accesses and in-flight misses which are kept in a condensed form in dedicated data structures, as explained below. For each L1-I miss, we compute the latency of fetching the requested cache line by subtracting the timestamp of the cache miss from the time the requested cache block enters the cache. Next, we track back in the recorded history the instruction which was executed at least $latency$ number of cycles earlier than the requested instruction and entangles the cache lines corresponding to the source and destination instructions.

As tracking each pair of entangled cache lines would require considerable storage space, 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 [1]). The head of a basic block is therefore the first non-consecutive cache line that is accessed. The size of the basic block is the number of consecutive lines being accessed. Furthermore, in order to reduce the number of entangled lines, the Entangling I-prefetcher merges "almost" consecutive basic blocks (see Subsec. 2.6) and entangles only the head of the first block.

The prefetching engine is then triggered upon every cache access and prefetches the entire basic block of the current line and of the entangled destinations (see Subsec. 2.4).

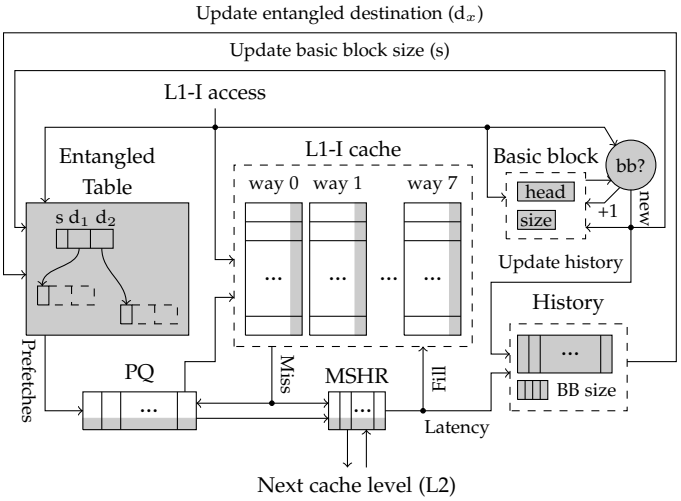

Fig. 1. Overview of the Entangling I-prefetcher

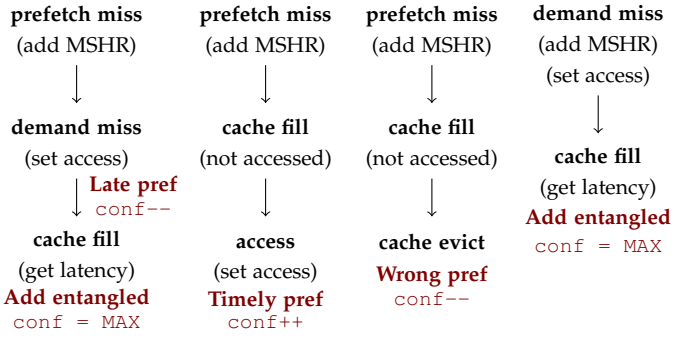

Fig. 2. Actions taken on various cache events.

2.1 Design

Figure 1 shows the data structures employed by the Entangling I-prefetcher, with the hardware extensions marked as gray. The right-top part (*Basic block*) computes the *size* of the current basic block and records its *head* (first) cache line. It also indicates when the basic block ends (*new*).

History buffer, on the right-bottom part of Figure 1, is a small circular queue that records the history of basic block heads together with the timestamp of their first access to L1-I. This table is parsed to identify potential *src-entangled* cache lines for each L1-I miss. *Basic block size table* (*BB size*) is a small structure that records the size of the youngest basic blocks in the *History buffer*. This table is employed to merge consecutive and overlapping basic blocks.

Timing and src-entangled information are stored along with the prefetch queue (PQ), the miss status holding register (MSHR), and the L1-I cache (shown as gray areas in these structures). The timing information is a timestamp stored only in the PQ and MSHR, holding the initial trigger time for in-flight prefetches (if they miss in the L1-I cache) and L1-I misses. When the requested data is stored in the cache, the latency of the miss is computed and a *src-entangled* cache line from the *History buffer* is selected. The *src-entangled* information consists of an access bit which indicates whether the line has been accessed and the corresponding *src-entangled* line (when applicable). Each request moves from one structure to another as the request progresses (PQ→MSHR→L1-I). The *src-entangled* information serves to add confidence to the entangled source-destination pair.

Entangled table, depicted on the left part of Figure 1, is the core structure of this proposal and records the entangled basic block heads used for deciding which cache lines to prefetch. Note that in an effort to keep the structure within a limited size, entangling is rather coarse-grain, i.e. only head basic blocks are entangled, not all cache lines. An entry contains the *src-entangled* cache line, its basic block size, a compressed array of *dst-entangled* cache lines (up to 6 destinations), and their associated *confidence*. Each *dst-entangled* is associated a *confidence* field initialized to the maximum value (since it was just computed prior to inserting it in the table and therefore expected to be accurate). The *confidence* is increased by timely prefetches and decreased by late and wrong prefetches. When *confidence* reaches 0, the *dst-entangled* becomes invalid (no prefetch is triggered).

2.2 Updating the basic block size

When a non-consecutive cache line accesses the cache, we start tracking a new basic block. We store in the *Entangled table* the size of the previous basic block (that has just completed). If the block to be added is already recorded in the table, we update its size to the maximum between the old size (of the already stored basic block) and the new size.

2.3 Adding dst-entangled cache lines

To populate the *Entangled table*, we insert each new basic block *head* together with its *size*, and add destinations as explained below. Figure 2 illustrates the actions taken upon each cache event and how the tables are populated.

Prefetches. Upon each prefetch issued, the *PQ* stores along with the currently allocated prefetch entry the current time and the *src-entangled* cache line (if the prefetch is for a basic block head). The access bit is not set. If the prefetch misses in cache, this information is transferred to the MSHR entry allocated by the cache miss. Otherwise the information in the PQ entry is discarded.

Demand cache misses. Upon a demand cache miss, an entry is allocated in the MSHR. The PQ is first checked for a matching prefetch and, if found (i.e. the prefetch was not timely), the timing and *src-entangled* information of the PQ entry is transferred to the MSHR entry. Otherwise, the current time is set in the MSHR entry. Demand misses also set the MSHR access bit and a pointer to the entry for that access in the *History buffer* (if it is a basic block head).

Cache fills. Upon a cache fill, if the access bit is set, it indicates that a miss happened before the line was prefetched (either there was no prefetch altogether or the prefetch was late). To fix this, 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 based on the timestamp of the MSHR. If the entry has a valid pointer to the *History buffer*, we know when the actual access for this cache line took place and that it is a basic block head. The source is then selected among the accesses that took place at least *latency* cycles before. If there is no pointer stored, no action is taken. Such misses will be covered by prefetching the full basic block starting from the head, as explained in the summary for triggering the prefetch.

TABLE 1
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]	$(8+2)\times 6 = 60$

TABLE 2
Baseline System Configuration

Processor width	6 fetch, 6 decode, 6 execute, 4 retire
ROB, LQ, and SQ	352 entries, 128 entries, and 72 entries
L1I cache	32KB, 8-way, 4 hit cycles, no prefetcher
L1D cache	48KB, 12-way, 5 hit cycles, next-line prefetcher
L2 cache	512KB, 8-way, 10 hit cycles, spp-dev prefetcher
L3 cache	2MB, 16-way, 20 hit cycles, no prefetcher
DRAM	4 GB, one 8-byte channel, 1600MT/s

Once a *src-entangled* cache line is found, the *Entangled table* is updated by adding to the *src-entangled* entry the corresponding *dst-entangled* with the *confidence* set to the maximum value. If the array of destinations is full, the *dst-entangled* with the lowest *confidence* is replaced.

If in the time window between issuing the prefetch and the corresponding cache fill there has been no demand access to the cache line, it means that the prefetch is either timely or wrong and no entangled pair needs to be added.

When the MSHR entry is removed, the *src-entangled* information is transferred to the corresponding L1-I entry.

Demand cache hits. Cache hits may find the access bit in the L1-I unset, if the cache line was brought by a prefetch. This is the ideal scenario indicating a timely prefetch. The access bit becomes set upon the hit.

Cache line evictions. Upon a cache line evict, the *src-entangled* entry is checked. If this is non-empty, it indicates that the evicted cache line has been brought through an entangled prefetch. Depending on the access bit:

- If it is not set, the line was unnecessarily brought to the cache, which indicates a wrong prefetch (early or unnecessary). The *Entangled table* is updated by decreasing the *confidence* of the *dst-entangled* corresponding to the evicted cache line.
- If it is set, it indicates a timely prefetch and in consequence the *confidence* of the *dst-entangled* corresponding to the evicted cache line is increased.

2.4 Triggering the prefetches

For every cache access we check the *Entangled table*. If the current cache line is recorded in the *Entangled table* (1) the entire basic block that starts with that cache line is prefetched.(i.e., *size-1* lines starting from the second line in the basic block); (2) for each *dst-entangled* with *confidence* $>$ 0, prefetch the entire basic block starting from *dst-entangled*.

2.5 Compressing destinations

The *Entangled table* uses different modes for encoding the array of *dst-entangled* entries (*dst-entangled* block and *confidence*) on 63 bits, as follows: 3 bits for the mode + 60 bits of the *dst-entangled* block 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, the destinations can be highly compressed.

The mode is a value between 1 and 6 which indicating how many destinations can be kept in the 60 bits of the array of *dst-entangled* blocks and the associated *confidence*. Depending on how many significant bits are required, the number of destinations can vary. For the confidence we always use a 2-bit saturated counter. Table 1 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* we first try to fill the *dst-entangled* arrays for the sources that are already inserted. More precisely, if the selected src-entangled is not present in the *Entangled table*, the prefetcher looks for the next best *src-entangled* entry, namely a cache line with the timestamp earlier than the one searched for. Up to six entries are checked and if there is no free destination entry, one is evicted.

2.6 Merging spatio-temporal basic blocks

Finally, to further reduce the number of entangled blocks in the *Entangled table* we perform a merge of quasi-consecutive basic blocks and prefetch the entire block. Merging is also aimed to address scenarios such as the sequence of accessed cache lines: $ABCECD$, 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 D is not a basic block head. To address this issue, we inspect the last four recorded basic block sizes, and if the current block can be merged with one of the previous blocks (i.e. they are consecutive or overlapping), then we update the size of the previous basic block (size of block starting with A would become 4 and prefetch $ABCD$) and do not record the current basic block (starting with C in the *History buffer*).

3 Evaluation

We evaluate our instruction prefetcher using the ChampSim simulator [6]. We model an out-of-order processor and a memory hierarchy similar to the latest Intel's Sunny Cove machine. The main configuration parameters of our baseline system are shown in Table 2.

We evaluate our prefetcher on the public traces provided by the 1st Instruction Prefetching Championship, which

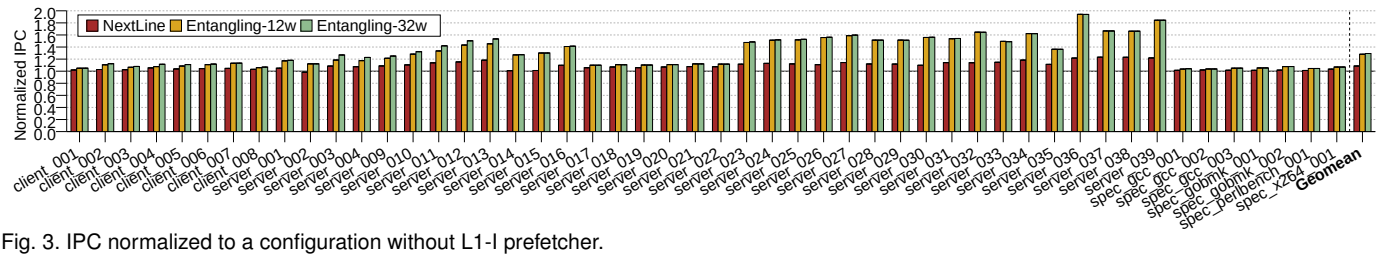

Fig. 3. IPC normalized to a configuration without L1-I prefetcher.

includes a set of client and server traces as well as applications from the SPEC CPU 2017 benchmark suite (*gcc*, *gobmk*, *perlbench*, and *x264*). Applications run for 50M instructions after a 50M instructions warm-up.

3.1 Configurations and Memory Overhead

The *History buffer* is a 16-entry circular queue, with a 58-bit tag field and a 20-bit timestamp field. A 4-bit register points to the head of the queue. The *Basic Block Size table* keeps the size of the last four basic blocks inserted in the *History buffer*. The basic block size is represented on 7 bits, allowing a maximum block size of 127 cache lines. The total memory required by both structures is 160 bytes.

Timing and src-entangled information is stored along with PQ (64 entries), MSHR (10 entries) and L1-I 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* (8 bits for the set and 4 bits for the way) and an access bit. Once the miss is resolved, the timing information is no longer necessary, thus the L1-I cache only records the *src-entangled* information. The total memory required to store the timing and *src-entangled* information is about 1KB (1100.25 bytes).

The *Entangled table* is a large set-associative cache that stores sources along with their maximum basic block size and destinations. It employs a FIFO replacement policy. It has 256 sets and 12 ways per set (next section performs a sensitivity analysis with respect to the number of ways). The tags are encoded using 34 bits, the basic block is encoded with 7 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 39.1KB.

3.2 Performance Results

We compare our prefetcher to a baseline system without any L1-I prefetcher (*No-IPref*) and a system with an L1-I pure next-line prefetcher (*NextLine*) [7]. We evaluate several configurations of the *Entangling I-prefetcher* with an *Entangled table* of 256 sets and the number of ways ranging from 4 (*Entangling-4w*) to 32 (*Entangling-32w*).

Figure 3 shows the instructions per cycle (IPC) normalized to *No-IPref*. Table 3 offers a sensitivity analysis of the number of ways of the *Entangled table* presenting the memory requirements of the prefetcher, the geometric mean of the normalized IPC, and the arithmetic mean for coverage (ratio of misses that became hits), accuracy (ratio of useful prefetches), and percentage of L1-I misses, across all applications. The Entangling I-prefetcher achieves 29.3% speedup with respect to the baseline configuration when using $105.56KB$ (*Entangling-32w*). More interestingly, with just $40.36KB$ overhead, *Entangling-12w* offers a good area-performance balance obtaining 28.1% speedup and reducing the average cache miss rate from 32% to just 1%.

TABLE 3
Sensitivity analysis and detailed results

Prefetcher	Size (KB)	Norm. IPC	Coverage	Accuracy	Misses (%)
No-IPref	0.00	1.000	0.000	0.000	23.055
NextLine	0.00	1.085	0.244	0.307	17.754
Entangling-4w	14.15	1.155	0.639	0.736	8.575
Entangling-8w	27.25	1.253	0.850	0.764	2.540
Entangling-12w	40.36	1.281	0.921	0.773	1.084
Entangling-16w	53.36	1.288	0.941	0.774	0.805
Entangling-20w	66.46	1.290	0.946	0.774	0.736
Entangling-24w	79.46	1.291	0.947	0.774	0.717
Entangling-28w	92.46	1.292	0.949	0.774	0.705
Entangling-32w	105.56	1.293	0.949	0.774	0.697

4 Conclusions

The entangled prefetcher for instructions offers an alternative prefetching direction driven by timeliness. The Entangling I-prefetcher entangles source cache lines that trigger prefetches for destination cache lines in a timely manner. 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 Entangling I-prefetcher's implementation is highly efficient without being intrusive in the processor design.

Acknowledgments

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).

References

[1] B. Falsafi and T. F. Wenisch, *A Primer on Hardware Prefetching*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2014.
[2] M. Ferdman, C. Kaynak, and B. Falsafi, "Proactive instruction fetch," in *44th Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2011, pp. 152–162.
[3] 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.
[4] 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.
[5] A. Ros and A. Jimborean, "The entangling instruction prefetcher," in *The 1st Instruction Prefetching Championship (IPC1)*, May 2020.
[6] "ChampSim simulator," May 2020. [Online]. Available: http://github.com/ChampSim/ChampSim
[7] J.-L. Baer, *Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors*, 1st ed. Cambridge University Press, 2009.

Alberto Ros is associate professor in the Computer Engineering Department at the University of Murcia, Spain. He was funded by the Spanish government to conduct the PhD studies and received the PhD in computer science from the University of Murcia in 2009. He hold postdoctoral positions at the Universitat Politècnica de València and at Uppsala University. He has coauthored more than 70 research papers in international journals and conferences. He received a Consolidator Grant from the European Research Council (ERC). His research interests include cache coherence, memory hierarchy designs, memory consistency, and processor microarchitecture.

Alexandra Jimborean 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 in the same university. In recognition of excellent research, she was awarded the Google Anita Borg Memorial Scholarship, along with more than 30 distinctions, awards and grants.