The Entangling Instruction Prefetcher

Alberto Ros
University of Murcia
aros@ditec.um.es

Alexandra Jimborean
University of Murcia
alexandra.jimborean@um.es

ABSTRACT

Prefetching instructions in the instruction cache is a fundamental technique for designing high-performance computers. To achieve maximum performance, there are three key properties one needs to consider in designing an efficient and effective prefetcher: (1) timeliness, (2) coverage, and (3) accuracy. Timeliness is an essential property of a prefetcher. 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 past their designated execution time. Coverage is essential to effectively 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 Prefetcher for Instructions (EPI) 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 prefetch. The prefetcher is carefully adjusted to account for both coverage and accuracy. Our evaluation shows that EPI increases performance by 29.5% on average, with a coverage of 95.6% and accuracy of 77.0%.

1. INTRODUCTION AND MOTIVATION

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], [2]. Driven by their impact on performance, prefetchers have evolved from simple *next line prefetchers* [1], to complex techniques, such as Temporal Instruction Fetch Streaming (TIFS) [3]. TIFS records the history of L1-I misses and predicts the next miss and the number of blocks to be cached, thus being more accurate and timely than traditional prefetchers. To capture the context of a miss (e.g. caused by a function call), Return-address stack-Directed Instruction Prefetching (RDIP) [4] records the return address stack and its context as signatures which are then consulted upon each call and return operations to trigger prefetching. More recently, Proactive Instruction Fetch [5] increases RDIP's performance by capturing the blocks accessed by the committed instructions and instructions from handlers for OS interrupts. Hence, it operates on the correct-path, retire-order instruction stream, and records the exact instruction fetch sequence which is then used to compute spatial locality.

We propose a prefetcher (EPI) oblivious to the control flow path and the instabilities introduced by predicting the correct or wrong path, by function calls or system interrupts. It works by estimating the latency of the cache missing operations and *entangling* them with the instructions that should trigger the prefetch to ensure the timely arrival of the requested instructions. In this way, EPI is robust and effective, agnostic to the application characteristics and achieves a 99.6% I-hit rate, approaching the perfect L1-I.

2. THE ENTANGLING I-PREFETCHER

The key contribution of this proposal is the entanglement of operations, which, intuitively, consists in pairing two instructions, the instruction i_{source} upon whose execution should be triggered the prefetch for the instruction $i_{destination}$. In a more concise representation, we define as *source-entangled* the cache line that should trigger the prefetch of the *destination-entangled* cache line such that the requested line arrives timely.

In order to ensure the timeliness of the prefetcher, we must first compute the latency of each cache miss. To this end, EPI 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, EPI computes 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, EPI tracks 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, EPI 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 [2]). The head of a basic block is therefore the first non-consecutive cache line. The size of the basic blocks is the number of consecutive lines. Furthermore, in order to reduce the number of entangled lines, EPI merges "almost" consecutive basic blocks (as explained below) and entangles only the head of the first block.

The prefetching engine is then triggered upon every cache access and if already tracked it will fetch the entire basic block of the current cache line, all the *destination-entangled* cache lines, and their corresponding basic blocks.

2.1 Implementation

We proceed by describing the registers and data structures employed by EPI.

Registers.

Current represents the first cache line (head) of the current basic block.

Counter indicates the current size of the basic block, that is, the number of consecutive cache lines accessed from *current*.

Tables.

History buffer records the history of basic blocks heads (i.e. the first non-consecutive cache line), together with the timestamp of the triggering instruction (i.e. the instruction whose execution led to installing the block in the cache).

Basic block size table is associated to the *History buffer*. We keep a small structure that records the size of the youngest basic blocks in the *History buffer*. This table is employed to merge previous basic blocks with the current one.

Timing table records I-cache misses and prefetches. An entry consists of the tag of the cache line, the timestamp when the miss/prefetch is triggered, an access bit (indicating if it has been a demand access for this line), a valid bit (indicates whether the entry is used or the table is underutilized), and the corresponding *source-entangled* cache line (when applicable). The access bit and the destination are used to update accordingly the confidence in the entangled pair source-destination.

Cache extension models the addition of two extra fields to each cache entry. In addition to the line address and the valid bit, it records an access bit which indicates whether the line has been accessed or not and the corresponding *source-entangled* line (when applicable). Upon a cache fill, the access bit and the entangled source-destination pair is moved from the *Timing table* to the *Cache extension*.

Entangled table is the core structure of this proposal and encodes the entangled basic block heads. 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 *source-entangled* cache line, its basic block size, a compressed array of *destination-entangled* cache lines (currently up to 6 destinations), and their associated *confidence*. Each *destination-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 prefetchers and decreased by late and wrong prefetchers. When *confidence* reaches 0, the *destination-entangled* becomes invalid (no prefetch is triggered).

Extended prefetch queue stores prefetches that cannot be sent due to a full prefetch queue, in a compressed manner.

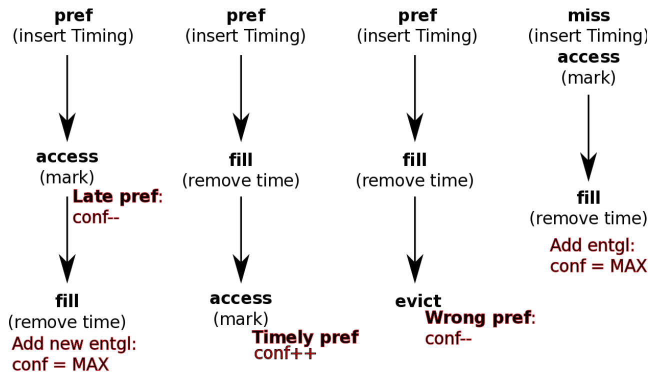

Figure 1: Actions taken on prefetches (pref), cache accesses, cache fills, and cache evictions. Entanglements are added for misses and late prefetches by looking for the source in the *History buffer*. The confidence counter is increased on prefetch hits and decreased on late and wrong prefetches.

2.1.1 *Populating the tables*

To populate the *Entangled table*, we insert each new basic block head (*Current*) together with its size (*Counter*), and add destinations as explained in the following. Figure 1 illustrates the actions taken upon each cache event and how the tables are populated.

Step 1. Upon a cache miss or prefetch issued, insert in the *Timing table* the address of the requested cache line and the timestamp of the triggering instruction. Misses mark the entry as accessed.

Step 2. Upon a cache fill, find the corresponding entry in the *Timing table* and compute the latency of the current memory access.

Step 3. Based on the latency, find the *source-entangled* cache line for misses and late prefetches, i.e. when should the prefetch be triggered such that the data is brought in time.

Step 3.1. For misses (demand accesses), search in the *History buffer* whether the cache line was recorded as the head of a basic block.

- If found, subtract from the timestamp of that basic block its latency and find the timestamp corresponding to the *source-entangled* cache line. Remove the entry from the *Timing table* and move it to *Cache extension*. Update the *Entangled table* by adding to the *source-entangled* entry the corresponding *destination-entangled* and its *confidence*. If the array of destinations is full, the *destination-entangled* with the lowest *confidence* is replaced.
- If not found, 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.

Step 3.2. For prefetches: if in the time window between inserting the prefetch in the *Timing table* and the corresponding cache fill there is a cache miss (*Acc*) to the same cache line (i.e. the prefetch was not timely), there is an attempt to insert *Acc* in the *Timing table* upon the miss. Since the address of the corresponding cache line is already present in the *Timing table*, the only action is to change the access bit from 0 to 1 (i.e. from prefetch to demand access). When the cache fill happens, the latency of the fill is calculated with respect to its timestamp in the *Timing table*. If the line has been accessed, then the corresponding *source-entangled* cache line is identified in the *History buffer*. The *source-entangled* entry and the corresponding *destination-entangled* are stored in the *Entangled table* with the *confidence* set to the maximum value. Again, if the array of destinations is full, the *destination-entangled* with the lowest *confidence* is replaced.

If in the time window between insterting the prefetch in the *Timing table* and the corresponding cache fill there has been no other access to the same line, it means that the prefetch is either timely or wrong and no entangled pair needs to be added. We simply remove the entry from the *Timing table* and move it to the *Cache extension*.

Step 4. Upon a cache hit, set the access bit.

Step 5. Upon a cache line evict, check the corresponding *source-entangled* entry. If this is non-empty, it indicates that the evicted cache line has been brought through a prefetch. Next, check the access bit.

- If the access bit is not set, the line was unnecessarily brought to the cache, which indicates a wrong prefetch (early or unnecessary). Update the *Entangled table* decreasing the *confidence* of the *destination-entangled* corresponding to the evicted cache line.

- If the access bit is set, it indicates a timely prefetch and in consequence we update the *Entangled table* increasing the *confidence* of the *destination-entangled* corresponding to the evicted cache line.

2.1.2 Updating the basic block size

When a non-consecutive cache line accesses the cache, we start tracking a new basic block. Before that we store its basic block size in the *Entangled table*. 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.

2.1.3 Triggering the prefetches

For every cache access we check the *Entangled table*. If the current cache line is recorded in the *Entangled table* (1) fetch the entire basic block that starts with that cache line (fetch *size* lines starting from the basic block head); (2) for each *destination-entangled* with *confidence* > 0, prefetch the entire basic block starting from *destination-entangled* (for finding its size we parse the *Entangled table* one more time).

Prefetches are stored in a extended prefetch queue that is drained when the processor prefetch queue has space for new prefetches.

2.1.4 Compression mechanisms

While several compression mechanisms have been employed in EPI, we describe in what follows the ones that are less standard and specialized for its data structures.

Furthermore, the *Entangled* table uses different modes for encoding the array of *destination-entangled* entries (*destination-entangled* block and *confidence*) on 63 bits, as follows: 3 bits for the mode + 60 bits of the *destination-entangled* block and the *confidence*. The bits reserved for the destination one need to encode the less significant bits ($signifB$) that vary between *source-entangled* and *destination-entangled* cache lines. The most significant bits can be taken from the source. Since the distance between *source-entangled* and *destination-entangled* is typically small, the 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 *destination-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. Next we detail the used modes:

Mode 1: $signifB$ requires up to 58 bits $\rightarrow (58+2)\times 1$
Mode 2: $signifB$ requires between 19 and 28 $\rightarrow (28+2)\times 2$
Mode 3: $signifB$ requires between 14 and 18 $\rightarrow (18+2)\times 3$
Mode 4: $signifB$ requires between 11 and 13 $\rightarrow (13+2)\times 4$
Mode 5: $signifB$ requires between 9 and 10 $\rightarrow (10+2)\times 5$
Mode 6: $signifB$ requires between 1 and 8 $\rightarrow (8+2)\times 6$

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

Finally, we aim to maximize the utilization of the *Entangled table* by first trying to fill the *destination-entangled* arrays for the sources that are already inserted. More precisely, if the selected *source-entangled* is not present in the *Entangled table*, we look for the next best *source-entangled* entry, namely a cache line with the timestamp earlier than the one searched for. We try up to six times and if there is no free destination entry, we evict one.

3. MEMORY REQUIREMENTS

Table 1 shows the memory requirements of EPI (less than 128KB of memory).

The *History buffer* is a circular queue. The cache line address is represented using 58 bits. The timestamp is stored as the difference with respect to the time of the previous entry in the buffer, using 20 bits. It also needs a pointer to the head (11 bits) and the timestamp of the last entry in the buffer (64 bits), in order to compute the timestamp of each entry in the buffer.

The *Basic Block Size table* stores the size of the last basic blocks inserted in the *History buffer*. Therefore, it does not

Table 1: Memory requirements

Structure	Number of entries	bits per entry	Other	Total (bits)
History buffer	1072	$58+20$	$11+64$	83691
Basic block size buffer	4	7	2	30
Timing table	42	$1+42+58+12+1$		4788
Cache extension	512	$1+36+58+1$		49152
Entangled table	34×256	$34+3+60+7$	6×256	906752
Extended prefetch queue	32	$58+58+7$	6	3942
Global registers	1		$58+7+7$	72
EPI				$(< 128KB)$ 1048425

need to store the address of the block. The maximum block size allowed is 127 (7 bits for storing the basic block size).

The *Timing table* stores the in-flight misses and prefetches and it can be implemented along with the miss status holding register (MSHR). The cache line address is represented using its 42 least significant bits. The table uses 12 bits to record the time the request was sent in order to compute its latency when it is resolved. It also stores an access bit and the source of entanglement (58 bits), in order to update the confidence. This is a fully associative structure.

The *Extended cache* keeps memory blocks that are in the cache and copies the access bit and the source of entanglement from the *Timing table* upon a cache fill. It can be implemented either as part of the cache or as a separate structure. The cache line address is represented using its 36 bits, as it is not required to store the 6 least significant bits of the cache line address that are used to index the cache.

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 (6 bits per set). It has 256 sets and 34 ways per set. The destinations and the confidence bits are encoded on a total of 60 bits. The cache line address is stored using 34 bits. The format is represented with 3 bits and the basic block size using 7 bits.

The *Extended Prefetch queue* stores prefetches that do not fit in the prefetch queue and acts as a spill structure. It stores the first address to prefetch (58 bits), its *source-entangled* (58 bits), and the total number of consecutive lines to prefetch (7 bits).

Finally, there are three global registers that keep the head of the current basic block (56 bits), a count of the number of consecutive lines seen (7 bits), and the new basic block size if the basic block will merge (7 bits).

4. EVALUATION RESULTS

We evaluate our prefetcher on the provided applications, executing on a single core. Applications run for 50M instructions after a 50M instructions warm-up.

The baseline is the original configuration, without any instruction prefetcher (No-IPref). We also compare our prefetcher with a next-line prefetcher (NextLine) [1]. Table 2 shows the three versions (No-IPref, NextLine, and EPI) and reports the number of instructions per cycle (IPC) as the geometric mean normalized to the baseline, and the arithmetic mean for Coverage, Accuracy, and I-HitRate, across all applications. EPI achieves 29.5% speedup with respect to the baseline configuration without instruction prefetching and increases the cache hit ratio up till 0.996 with a 95.6% coverage.

Table 2: Evaluation results

Prefetcher	Norm. IPC	Coverage	Accuracy	HitRate
No-IPref	1.000	0	0	0.813
NextLine	1.069	–	–	0.848
EPI	1.295	0.956	0.770	0.996

5. DISCUSSION AND FUTURE WORK

As the championship rules prevent accessing cache and MSHR information, both the *Timing table* and the *Cache extension* are standalone in the code implementation. However, in a real implementation these structures can be coupled with the MSHR and the L1I cache respectively.

The largest associative structure is the *History buffer* and it is iterated entry by entry to compute the timestamp. Other implementations could store the actual time and hence be searched more efficiently.

Finally, EPI does not use confidence counters for merging the basic blocks, as other I-prefetchers based on branch predictors [2]. We will consider adding confidence counters for basic block merging to increase the accuracy of the prefetcher.

6. ACKNOWLEDGMENTS

This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 819134).

7. REFERENCES

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

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

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

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

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