PIPS: Prefetching Instructions with Probabilistic Scouts

Pierre Michaud

PIPS: Prefetching Instructions with Probabilistic Scouts

Pierre Michaud
Inria, Univ Rennes, CNRS, IRISA
pierre.michaud@inria.fr

Abstract

A new instruction prefetching method is proposed, called probabilistic scouting, based on the concept of Control Flow Graph (CFG). Each node of the CFG is a distinct memory line, and a directed edge from node X to node Y means that line Y is a possible successor of line X. Each edge is annotated with the probability for the edge to be taken. The hardware discovers the CFG progressively while the program executes, by monitoring instructions retiring from the processor in program order. CFG information is stored in a Line History Table (LHT). Starting from the line currently being fetched, aka front line, the prefetcher sends scouts to explore the CFG. A scout goes from node to node in a semi-random fashion, according to the edge probabilities stored in the LHT. The scout prefetches the nodes encountered along the followed path. A scout dies after a certain number of steps. After a scout dies, a new scout is sent from the front line. This process repeats indefinitely. For higher prefetch coverage and timeliness, several parallel scouts may be sent to explore the CFG simultaneously.

This paper describes probabilistic scouting and a prefetcher using it, called PIPS, which is submitted to the championship. PIPS is tuned for the championship's simulated microarchitecture and public traces. Besides the 128 KB storage limit, the LHT of PIPS is ideal, as allowed by the championship rules.

1 Probabilistic scouting

1.1 Some definitions

Line. In this paper, by a convenient abuse of language, the term *line* may have three different meanings: it can mean (1) the information content of a cache line (aka cache block), or (2) the line's address, or (3) the range of byte addresses corresponding to the line. Hopefully, when the meaning is not made explicit, it should be easy to disambiguate it from the context.

Front line. The *front line* is the line where the program counter (PC) currently lies in. While the PC has a clear definition in the instruction-set architecture, its embodiment in a pipelined microarchitecture is more equivocal. One may define distinct microarchitectural PCs at distinct pipeline stages, leading to different definitions of the front line. In this paper, the front line is the line most recently read by the instruction fetch stage.[1]

Control-flow graph (CFG). What is called *control-flow graph* (CFG) in this paper is a graph whose nodes are *distinct* lines (i.e., different line addresses) and where a directed edge $X \rightarrow Y$ from line X to line Y means that Y is a possible *successor* of X, that is, line Y might be fetched and executed just after line X. A line cannot be its own successor, that is, Y is distinct from X. For example, if line X contains a branch instruction whose target lies in another line Y, then Y is a possible successor of X. A sufficient (but not necessary) condition for line $X+1$ to be a successor of X is that line X do not contain any unconditional jump.

Each edge $X \rightarrow Y$ of the CFG is annotated with a probability $P(Y|X)$ which is the probability that the line fetched and executed after an instance of X is line Y. For each node, the sum of all outgoing edges probabilities is equal to 1.

In the remaining, the terms *line* and *node* are used interchangeably. In particular, the *front node* is the node corresponding to the front line.

Line-history table (LHT). The hardware discovers the CFG progressively, at run time, by monitoring the addresses of retiring instructions, in program execution order, accumulating information about the CFG in a *Line History Table* (LHT). That is, the LHT provides a dynamic approximation of the CFG.

Edge probability can be estimated dynamically with frequency counts. When the previous and current retired instructions belong to distinct lines X and Y respectively, Y is recorded as a possible successor of X in the LHT entry for X (if this is the first occurrence of $X \rightarrow Y$) and the frequency count associated with Y in that entry is incremented.

Frequency counts are coded with a limited number of bits, e.g., 4 bits per counter. When, for a given node, one frequency count cannot be incremented because it has reached the maximum counter value (15 for 4-bit counters), then all the frequency counts for that node are halved.

[1] At the moment of writing this, it is not clear to me how to best define the front line. A microarchitecture simulator modeling wrong-path instruction fetching is needed to answer this question.

1.2 Probabilistic scouting

As the program executes, the PC moves in the CFG, from node to node. Without any other information but edges probabilities, the exact path that the PC will follow in the near future is not known with 100% certainty in general. Nevertheless, the PC is less likely to take edges that have a low probability. The probability that a given path is taken is the product of the probabilities of all the edges on that path. One possible prefetching method would be to prefetch all the nodes on paths whose probability exceeds a fixed threshold, for instance by a depth-first traversal of the CFG, starting from the front node. However, such method would be complex to implement in hardware. Probabilistic scouting emulates that prefetching method, but with relatively simple hardware.

Probabilistic scouting consists in sending *scouts* to explore the CFG, taking into account edges probabilities. As a scout follows a path in the CFG, it prefetches the lines encountered along the path. Concretely, a scout is just a line address. The scout starts from the front node. At each step, the scout accesses the LHT with the address of the node X where it currently is, retrieves the possible successors $Y_1,...,Y_n$ of X, with their frequency counts, selects *one* successor Y_k with probability $P(Y_k|X)$, moves to node Y_k and prefetches the corresponding line. This step is repeated until the scout *dies*. The scout dies when a certain ending condition is met, e.g., after a fixed number of step.

When the scout dies, a new scout is sent from the front node. The new scout will perhaps take a different path in the CFG than the previous scout. The process of sending a new scout when one dies repeats indefinitely. If the LHT provides enough bandwidth, it is possible to have multiple scouts exploring the CFG simultaneously.

For prefetching to be effective, the number of LHT entries should be greater (if possible, much greater) than the instruction cache capacity in number of lines. Hence in practice, the LHT is a relatively large table. However, some workloads may exceed the LHT capacity. In case of LHT miss for node X, the scout moves to node $X+1$ (and prefetches it) provided X is not too far from the front node F, say $X \in [F,F+3]$.

1.3 Scouting cache (SCC)

Scouting should be done fast enough for high prefetch coverage and timely prefetches. The LHT is relatively large, and its latency may limit scouting speed. This problem can be mitigated by introducing a *scouting cache* (SCC). An SCC entry is identical to an LHT entry, However the SCC is much smaller, hence faster than the LHT.

The SCC works like a cache: the scout first accesses the SCC, and accesses the LHT only upon an SCC miss. Upon an SCC miss, the missing node is copied from the LHT to the SCC, evicting one SCC entry. The SCC's effectiveness rests on temporal locality, which mostly comes from loops. When a scout moves onto a node, there is some chance that this node has been visited recently by the same or another scout.

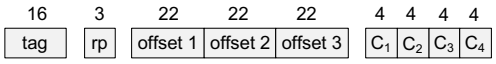

Figure 1: One LHT entry of PIPS. The entry size is 101 bits, counting the tag (16 bits) and the replacement policy state (rp, 3 bits). The SSC entry is identical to the LHT entry, but with 2 rp bits instead of 3.

The SCC brings two other benefits besides accelerating scouting. One benefit is that the SCC can serve as a filter to reduce redundant prefetches. Indeed, an SCC hit means that the scout is on a node that has already been visited recently. If the SCC capacity is less than that of the instruction cache (in number of lines), a scout needs issuing prefetches only upon SCC misses.

Another benefit of the SCC is that it allows an alternative method for selecting a node's successor, easier to implement than probabilistic selection. A possible algorithm for probabilistic selection is to sum the node's frequency counts C_i, i.e., $S = \sum_{i=1}^{n} C_i$, take a uniformly distributed random integer $R \in [0,S)$, and select the successor Y_k such that $\sum_{i=1}^{k-1} C_i \leq R < \sum_{i=1}^{k} C_i$. This algorithm is probably cumbersome to implement in hardware.

Instead, a pseudo-probabilistic method can be used, which is to select the successor with the greatest count C_k in the SCC entry, that is, $C_k \geq C_i$ for all $i \in [1,n]$. If C_k is null, the scout dies. Otherwise, successor Y_k is selected, and C_k is decremented ($C_k := C_k - 1$) *only in the SCC*. It should be noted that, once all the counts in an SCC entry have been decremented down to zero, the corresponding node kills any scout reaching it, i.e., it is a barrier. However, the node's LHT entry still holds the up-to-date frequency counts. The node remains a barrier until it is evicted from the SCC and is reaccessed, which creates a new SCC entry with replenished frequency counts. Simulations show that pseudo-probabilistic selection yields practically the same IPC as probabilistic selection while reducing over-prefetching.

2 Submitted prefetcher

The prefetcher submitted to the championship, called PIPS (see the paper title), is one possible implementation of probabilistic scouting, taking into account the championship rules and the characteristics of the championship's simulator and public traces. As the championship rules do not require to simulate the latency of accessing large hardware arrays, I made the LHT as large as possible (within the 128 KB budget) for good prefetching performance on workloads having a large instruction footprint.

The LHT and the SCC are both organized like set-associative caches. The LHT entry is depicted in Figure 1. The maximum number n of recorded successors for a node X is set to $n = 4$, one of the four successors being $X + 1$. For successors Y other than $X + 1$, which correspond to branch targets, a signed target offset is calculated as $Y - X$ and truncated to 22 bits.[2] If the offset cannot be represented in 22 bits, it is ignored. Note that, as a node cannot be its own successor, $Y - X$ cannot be null. So a null offset field means that the offset field is currently empty. When the offset corresponding to Y is not already present in the LHT entry, a null offset field is searched to record the new offset. If there is no null offset field, one of the 3 offset fields is overwritten: the smallest frequency count among C_1, C_2 and C_3 is selected (C_4 is for $X+1$), and the corresponding offset is the victim. When a new offset is recorded, its frequency count C_i is initialized to one. Otherwise, if the offset is already present or if the successor is $X+1$, the corresponding C_i is incremented. If C_i cannot be incremented because it is already equal to the maximum value (15), then C_1, C_2, C_3 and C_4 are all divided by two. Tags, offsets and frequency counts are initially set to zero at the start of the simulation.

The LHT has 1024 sets and 10 ways of associativity (10,240 entries). The set index is obtained as the 10 least significant bits of the line address. The next 16 bits of the line address constitute the tag.[3] The replacement policy is a 3-bit SRRIP [8]. The total storage used by the LHT is $10{,}240 \times 101 = 1{,}034{,}240$ bits, i.e., 126.25 KB.

The SCC has 32 sets and 4 ways (128 entries). The set index is obtained as the 5 least significant bits of the line address. The next 16 bits of the line address constitute the tag. The replacement policy is a 2-bit SRRIP. The total storage used by the SCC is $128 \times 100 = 12{,}800$ bits, i.e., 1.56 KB.

PIPS maintains 4 parallel scouts. Each scout moves to a new node every clock cycle. When the front node changes, one scout is killed (chosen in a round-robin fashion) and a new scout is sent from the front node. The ending condition is that the prefetch queue contain more than 7 not-yet-issued prefetch requests. That is, if the prefetch queue occupancy exceeds 7, all four scouts die.

As shown in Figure 2, the speedup obtained with PIPS approaches that of the ideal prefetcher. On the 50 public traces, inside the region of interest (i.e., after warmup), PIPS's over-prefetching varies between +7% and +400% (median +85%) more cache insertions than when instruction prefetching is disabled.[4] On average, about 90% of instruction cache insertions are prefetch requests, and 60% of these prefetch requests are useful, i.e., prefetch hits (between 40% and 85% useful prefetches depending on traces).

[2] On the public traces, 20 offset bits are enough. I use 22 bits as allowed by the 128 KB storage budget, just in case some of the private traces need slightly more than 20 offset bits.

[3] It is a partial tag. False hits happen but are rare.

[4] Over-prefetching can be reduced by reducing the number of parallel scouts and/or by using a more restrictive ending condition.

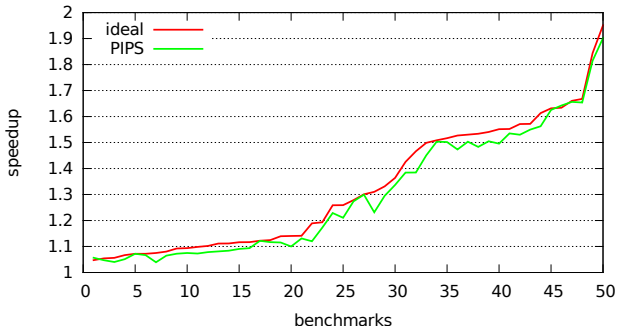

Figure 2: Speedup for the 50 public traces, relative to no prefetching. Ideal-prefetch speedup is obtained by turning each instruction-cache miss into a hit, but sending the miss request to the level-two cache anyway. Traces are sorted by increasing ideal-prefetch speedup.

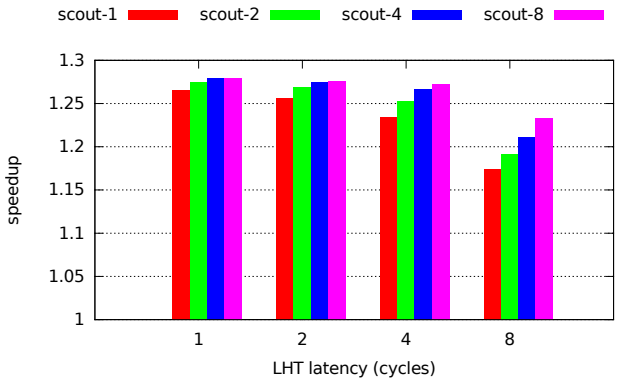

Figure 3: Effect of LHT latency and bandwidth on speedup (geometric mean on the 50 traces). LHT bandwidth is assumed proportional to the number of parallel scouts.

3 LHT latency and bandwidth

As permitted by the championship rules, the LHT in PIPS is ideal in the sense that its latency and bandwidth are the same as those of the SCC. However, in a real hardware implementation, the LHT would be slower than the SCC.

Figure 3 shows how LHT latency and bandwidth impacts IPC speedup on the championship simulated microarchitecture. The number of parallel LHT accesses equals the number of parallel scouts, i.e., LHT bandwidth is proportional to the number of scouts. As expected, the IPC drops as LHT latency increases. The IPC drop can be compensated, to a certain extent, by having more parallel scouts. However, when the LHT latency exceeds 4 cycles, the IPC drop is significant, even with 8 parallel scouts.

4 Related work

This section mentions some hardware instruction-prefetching methods.[5]

Smith and Hsu proposed in 1992 to combine next-line prefetching and target prefetching (using a BTB-like structure) [15]. In particular, both directions of conditional branches are prefetched. Pierce and Mudge proposed that the BTB-like structure in Smith and Hsu's prefetcher be removed. Instead, next-line prefetching is used at the fetch stage and target prefetching is deferred to the decode stage [13]. Spracklen et al. described a prefetcher combining next-line prefetching and target prefetching, but working at line granularity and prefetching deeper than Smith's and Pierce's prefetchers [16]. More precisely, Spracklen's prefetcher prefetches the next few sequential lines after the current PC, and the target line of each of these sequential lines (plus a few sequential lines for each target line).

Another approach to instruction prefetching, called branch-predictor-directed (BPD) prefetching [5], consists in having a branch predictor featuring a *large* branch target buffer (BTB) and working in an autonomous fashion. In this way, instruction-cache misses can be pipelined, which alleviates the cache miss penalty when the predictor stays on the correct path [4, 14, 12]. The BTB size requirement can be reduced by prefetching BTB entries [3, 2, 9, 12]. The prefetcher proposed by Annavaram et al. is a sort of BPD prefetcher, using a coarse-grained control-flow predictor predicting calls and returns only, next-line prefetching being used for function bodies [1].

Another approach to instruction prefetching is to observe instruction-cache accesses and to associate one or several accesses to a triggering event. The information is stored in a history table and the accesses are replayed (as prefetch requests) when the same triggering event reoccurs. This approach might be called *replay* prefetching. The triggering event can be a branch [17], an instruction-cache access [7, 6], a hash of the call stack [10], for instance. Kumar et al. proposed an instruction prefetcher combining replay prefetching and BPD prefetching [11].

To the best of my knowledge, probabilistic scouting is a new idea. Its general philosophy puts it in the lineage of Smith, Pierce and Spracklen [15, 13, 16]. It should be noted that, unlike some instruction prefetchers such as [6, 11], PIPS does not exploit spatial locality.

5 Conclusion

In the submitted prefetcher, the front line is the line currently being fetched, which, in the championship's trace-driven simulator, is always on the correct path. Understanding how to best define the front line requires a simulator modeling wrong-path fetching.

The submitted PIPS prefetcher is somewhat idealized and does not take into account the LHT latency and bandwidth. Some research is needed to understand how probabilistic scouting can be implemented without losing the potential speedup offered by an ideal LHT. Exploiting spatial locality might help here.

[5] Some data-prefetching methods can be adapted for prefetching instructions (see [5] for an overview of hardware prefetching). Nevertheless, instruction prefetching is a distinct problem.

References

[1] M. Annavaram, J. M. Patel, and E. S. Davidson. Call graph prefetching for database applications. In *HPCA*, 2001.

[2] J. Bonanno, A. Collura, D. Lipetz, U. Mayer, B. Prasky, and A. Saporito. Two level bulk preload branch prediction. In *HPCA*, 2013.

[3] I. Burcea and A. Moshovos. Phantom-BTB: a virtualized branch target buffer design. In *ASPLOS*, 2009.

[4] I-C. K. Chen, C.-C. Lee, and T. N. Mudge. Instruction prefetching using branch prediction information. In *ICCD*, 1997.

[5] B. Falsafi and T. F. Wenisch. *A primer on hardware prefetching*. Morgan & Claypool Publishers, 2014.

[6] M. Ferdman, C. Kaynak, and B. Falsafi. Proactive instruction fetch. In *MICRO*, 2011.

[7] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Temporal instruction fetch streaming. In *MICRO*, 2008.

[8] A. Jaleel, K. B. Theobald, S. C. Steely, and J. Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *ISCA*, 2010.

[9] C. Kaynak, B. Grot, and B. Falsafi. Confluence: unified instruction supply for scale-out servers. In *MICRO*, 2015.

[10] A. Kolli, A. Saidi, and T. F. Wenisch. RDIP: return-address-stack directed instruction prefetching. In *MICRO*, 2013.

[11] R. Kumar, B. Grot, and V. Nagarajan. Blasting through the front-end bottleneck with Shotgun. In *ASPLOS*, 2018.

[12] R. Kumar, C. C. Huang, B. Grot, and V. Nagarajan. Boomerang: a metadata-free architecture for control flow delivery. In *HPCA*, 2017.

[13] J. Pierce and T. Mudge. Wrong-path instruction prefetching. In *MICRO*, 1996.

[14] G. Reinman, B. Calder, and T. Austin. Fetch directed instruction prefetching. In *MICRO*, 1999.

[15] J. E. Smith and W.-C. Hsu. Prefetching in supercomputer instruction cache. In *Supercomputing*, 1992.

[16] L. Spracklen, Y. Chou, and S. G. Abraham. Effective instruction prefetching in chip multiprocessors for modern commercial applications. In *HPCA*, 2005.

[17] V. Srinivasan, E. S. Davidson, G. S. Tyson, M. J. Charney, and T. R. Puzak. Branch history guided instruction prefetching. In *HPCA*, 2001.