Run-Jump-Run: Bouquet of Instruction Pointer Jumpers for High Performance Instruction Prefetching

Vishal Gupta, Neelu Shivprakash Kalani, and Biswabandan Panda
Dept. of Computer Science and Engineering, Indian Institute of Technology, Kanpur
{vishal,neeluk,biswap}@cse.iitk.ac.in

***Abstract*—Large instruction working sets are common with modern client and server workloads. These working sets often fit in the large last-level cache (LLC). However, the L1 instruction cache (L1-I) suffers from a high miss rate blocking the instruction supply to the front-end of the processor. Instruction prefetching is a latency hiding technique that can bring instructions from the LLC into the L1-I. We propose a bouquet of instruction pointer (IP) jumpers, named JIP. JIP is a high-performance L1-I prefetcher that uses different prefetching techniques by classifying instructions into the following categories: (i) a non-branch, (ii) a branch that jumps to a single target IP on all instances, and (iii) a branch that jumps to different target IPs on different instances. Compared to a baseline with no instruction prefetching, averaged across 50 traces, JIP provides a prefetch coverage of 91.33% (as high as 99.99%), which leads to a performance improvement of 27.75% (as high as 93%). JIP makes a strong case for instruction prefetching as the performance gap between the perfect L1-I and JIP is just 7.49%. JIP demands a hardware overhead of 127.8 KB.**

I. Introduction

Modern client and server workloads operate on large code footprints that do not fit in a small L1 instruction cache (L1-I). L1-I prefetchers play an essential role in improving system performance by mitigating the front-end bottleneck. Existing instruction prefetching techniques [1]–[8] mitigate the front-end bottleneck up to a great extent.

Opportunity: To understand the impact of instruction prefetching, we simulate the perfect L1-I (miss rate of 0%). Compared to a baseline system with no prefetching, next-line (NL) and next-two-line (N2L) prefetchers, provide performance improvement of 6.92% and 9.99%, respectively. In contrast, the perfect L1-I provides a performance improvement of 35.24%. This trend shows that there is ample opportunity available for high-performance instruction prefetching.

Challenges: The code footprints of most applications usually fit into the on-chip caches like L2 and the last-level cache (LLC). The challenge is to perform inter-cache prefetching and prefetch the instruction cache lines into the L1-I from L2 and LLC, on time. If prefetch responses come late, then demand requests do not get L1-I hits. So, prefetch timeliness is the key. Further, the prefetcher's performance is dependent on the frequency of control flow jumps.

Our goal and approach: We aim to bridge the gap between the perfect L1-I and N2L prefetcher by *on time* instruction prefetching of control flow jumps (target-IPs). Our prefetching framework takes an IP as input (trigger-IP) and prefetches the cache lines containing the target-IPs. We classify trigger-IPs into three different classes: (i) a non-branch, (ii) a branch that jumps to a single target-IP on all instances (mostly `direct-jump/call` and `conditional` IPs)[1], and (iii) a branch that jumps to different target-IPs on different instances (mostly `indirect-jump/call` and `return` IPs). We name our framework JIP, a bouquet of IP jumpers. We are motivated by the bouquet of IPs for data prefetching [9]. For each IP class, JIP provides a prefetching technique that is most suitable for the IPs belonging to that class. On average, JIP covers 91.33% of the L1-I misses by prefetching on-time 96.9% of the time, and provides a performance improvement of 27.75% (closer to the perfect L1-I with a gap of just 7.49%).

II. Bouquet of IP jumpers

In this section, we make a case for a bouquet based instruction prefetching framework. At a 10,000 feet view, our framework falls into the category of branch-predictor directed instruction prefetching, where the effectiveness of the branch predictor is the key. The championship infrastructure provides a highly accurate hashed-perceptron branch predictor. The outcomes of the branch predictor, along with the branch targets are available at the L1-I, providing the illusion of a perfect Branch Target Buffer (BTB). Note that, the infrastructure provides a target-IP of zero in two cases: if the branch predictor predicts that (i) the branch is not taken or (ii) the branch is taken, but the prediction is wrong. We leverage branch prediction and branch target information in designing our prefetching framework.

A. JIP Prefetching framework

Our prefetching framework comprises of three components: a runner and two IP jumpers, where the input to the runner/jumper is a trigger-IP, and the output is a target-IP. JIP also uses two supporting components for improving prefetch timeliness and storage density.

The Runner: For non-branch IPs, the framework uses a runner that continues to prefetch the next cache line following the sequential control flow (next sequential IP). Whenever

[1]Self-modifying codes are one of the exceptions.

SJT (7800 entries, Fully Associative)

Trigger IP	Target IP	NRU
25 bits	25 bits	1 bit

MJT-I (1024 entries, Direct Mapped)

Tag	Target IPs	Array of Targets	Target Confidence
15 bits	3 targets x 25 bits	8 indices x 2 bits	3 targets x 2 bits

MJT-II (512 entries, Direct Mapped)

Tag	Target IPs	Array of Targets	Target Confidence
16 bits	8 targets x 25 bits	16 indices x 3 bits	8 targets x 2 bits

Mapper Table (512 entries, Fully Associative)

Uncompressed IP	Compressed IP
Upper 48 bits	9 bits

Temporal Table (7150 entries, Fully Associative)

Leader IP	Follower IP
25 bits	25 bits

Fig. 1. Hardware tables with the JIP prefetching framework. NRU: Not recently used.

the championship infrastructure provides a branch target of zero, the runner component assumes that the control flow is sequential. For a branch IP, the framework uses two different jumpers to predict the target-IPs.

Jumper-I (Single target jumper): Single target jumper handles branches that jump to the same target-IPs on all instances. We maintain the trigger-IP and target-IP relationship using a hardware table that we call Single Target Jump Table (SJT). SJT mostly deals with `direct-calls/jumps`, with a few exceptions. This table also stores the trigger-IP and target-IP pairs for `conditional-jumps`. These jumps have only two branch outcomes i.e., either the branch is taken (in which case SJT stores the target-IP), or the branch is not taken. In the latter case, the program continues with the sequential control flow on the basis of the runner component of our prefetching framework. SJT issues a prefetch request for the cache line containing the target-IP if the trigger-IP is present in SJT. On average, across 50 client, server, and SPEC CPU traces, SJT handles 85% of the total branches.

Jumper-II (Multiple targets jumper): In contrast to branches that have the same target-IPs for all instances, branches of type `indirect-jump/call` and `return` can have different target-IPs for different instances of a given trigger-IP. The target-IPs recur forming a target-IP sequence. For an indirect branch with trigger-IP A, suppose the target-IP for A_1 is B, and for A_2 is C (where A_x is the x^{th} instance of a trigger-IP A). It is likely that this sequence will recur in the future i.e., for trigger-IP A_x, if the target-IP is B, then for trigger-IP A_{x+1}, the target-IP will be C. So we store multiple target-IPs for a given trigger-IP using a direct mapped hardware table called Multiple Targets Jump Table (MJT). We also store the temporal sequence of last n target-IPs in an array called array-of-targets. We prefetch on the basis of the target-IP information present in MJT. Before prefetching, for a given trigger-IP, we compare the recent k target-IPs with the array-of-targets, k at a time. In case of a perfect match of k target-IPs with the array-of-targets from index i to j, we use the target at $j + 1^{th}$ index as the target-IP for the next instance of the trigger-IP.

We also store a confidence-counter per target-IP to select amongst the multiple target-IPs if the recent k target-IPs do not get a perfect match with the array-of-targets. In such cases, MJT selects the target-IP with the highest confidence as the target-IP for the next instance. We observe that some trigger branch IPs have fewer unique target-IPs than others. So, we use two MJTs (MJT-I and MJT-II) to store four and eight target-IPs per trigger-IP. We use (8,4) and (16,4) as the values of (n, k), for MJT-I and MJT-II, respectively. We store two and three bits indices in the array-of-targets that point to target-IPs of MJT-I and MJT-II, respectively. The array-of-targets for MJT-I and MJT-II store the last eight instances of three target-IPs and last 16 instances of eight target-IPs, respectively.

Migration among the jumpers: We classify a trigger-IP's branch behaviour on the basis of the number of unique target-IPs it jumps to on different instances.

We do not insert non-branch IPs into SJT and MJTs. Initially, we insert every branch trigger-IP into SJT. On a future instance of the same trigger-IP, we index into SJT and compare the current target-IP with the target-IP in SJT. If the target-IPs are different then we speculate that the trigger-IP can jump to different target-IPs in the future instances, and migrate that entry into MJT-I. Further, if the number of unique target-IPs for a trigger-IP crosses the number of target-IPs that MJT-I stores then we migrate it to MJT-II. However, if the number of unique target-IPs crosses the number of target-IPs that MJT-II stores then we use a replacement policy. The policy replaces the oldest target on the basis of the temporal order of targets.

Improving prefetch timeliness with a temporal table: We observe that runner and jumpers are accurate in terms of predicting the target-IPs (with an average accuracy of more than 65%). However, more than 16% of the prefetch requests are late, on average. To improve the timeliness of prefetch requests, we store the IPs corresponding to the recent n L1-I accesses in an n-entry Recent Access Queue (RAQ).

We observe that the temporal sequence in which an application accesses IPs usually recurs. So, we correlate the last n^{th} L1-I access IP (head of the RAQ) with the current L1-I access IP, if the current L1-I access is a miss. We refer to the former as leader IP and the latter as follower IP. We store such [leader IP, follower IP] pairs in a table we call the temporal table, and whenever we observe an access to the leader IP, we prefetch the cache line containing the follower IP, too. The temporal table uses a random replacement policy. With the temporal table, the prefetcher's lateness reduces to 3.1%.

Improving storage density with a mapper table: We find that storing IPs in a 64-bit format demands a huge amount of storage for our framework. To make our design practical, we use an IP mapper. We observe that the number of unique values that the upper 48 bits of IPs represent is much less than 2^{48}. So, we use a mapper that maps unique upper 48 bits of an IP to unique nine bits, and stores the mappings in a fixed size, fully associative mapper table. Note that if the number of unique values of the upper 48 bits crosses 2^9, then we expect the accuracy of the prefetcher to drop. We use the first-in-first-out (FIFO) replacement policy when the mapper table is full.

Figure 1 shows all the hardware tables of interest, including temporal and mapper tables. Note that, to index into the direct

mapped MJTs, we right shift the 25-bit compressed IP by two bits and then use the lsbs for indexing (10 and nine for MJT-I and MJT-II, respectively) and use the rest of the bits as tag.

III. Design and Implementation

Access flow: Figure 2 shows the flow of our JIP framework. On every L1-I access, we use IP mapper to generate the compressed trigger-IP that we use to index into JIP tables (①). If there is a hit in JIP tables, we get the target-IP from JIP tables. If there is a miss in JIP tables, we prefetch the cache line containing the next IP in the sequential control flow assuming that the IP is a non-branch IP. Note that this may not always be the case as JIP tables can not store all the branch IPs to abide by the storage budget. We continue the lookahead from the prefetch IP until we reach the lookahead-depth or the prefetch-degree (②). We use a Recent Prefetch Queue (RPQ) as a filter to avoid making redundant prefetch requests (③). We use the reverse IP mapper to decompress the prefetch IP and issue the prefetch request (④).

Whenever an L1-I access occurs, JIP uses the access IP (trigger-IP) to lookup JIP Tables and find the target-IP for the given trigger-IP. Suppose, for trigger-IP A; JIP finds that the target-IP is B. JIP prefetches B, begins another table lookup with B as the trigger-IP, and finds the target-IP for B. We define maximum lookahead-depth as the number of times JIP looks up JIP tables with a trigger-IP. We define prefetch-degree as the maximum number of prefetch requests JIP makes to unique cache lines during the lookahead process.

JIP uses a lookahead-depth of 260 IPs and prefetch-degree of seven cache lines on every L1-I access. Note that the lookahead stops if we reach the limit of either lookahead-depth or prefetch-degree. On average, across 50 traces, we find that JIP uses a degree of 2.2, which means it generates a similar amount of prefetch traffic as an N2L prefetcher.

The extended lookahead process: If there are no L1-I accesses for two cycles after the last L1-I access, we decide to perform lookahead beyond the look-ahead depth (extended lookahead). We store the last cycle in which we make a prefetch request in a register (last-prefetch-cycle). If we perform the extended lookahead the cycle after the last-prefetch-cycle, and another L1-I access comes at the next cycle, then the extended lookahead prefetch request may cause contention. It can block the prefetch requests from the actual L1-I access lookahead path as the L1-I prefetch queue is a FIFO. So, we wait till two cycles after the last-prefetch-cycle to begin the extended lookahead in case another L1-I access occurs.

We distribute the extended lookahead process across the subsequent three clock cycles. To perform the extended lookahead for three cycles, we use a counter remaining-lookahead-cycles, which we initialize to three before starting the extended lookahead and decrement it in every cycle in which we perform the extended lookahead. Also, we use a prefetch-degree of one in the subsequent clock cycles as it is possible that the extended lookahead may be following the incorrect control flow path. If another L1-I access occurs before the extended lookahead process is complete, then we stop the extended lookahead and begin a new lookahead starting with the latest L1-I access.

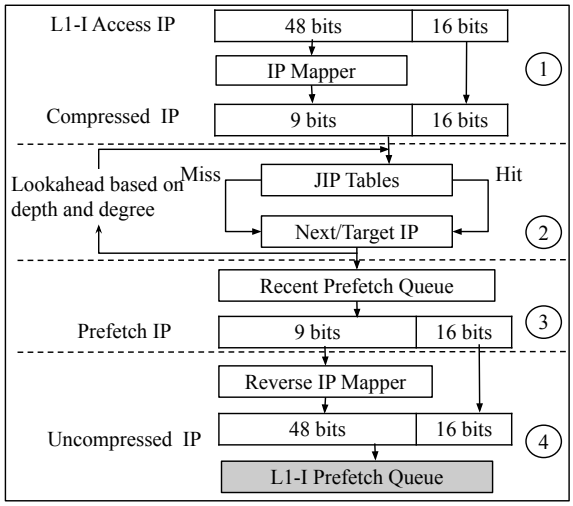

Fig. 2. Access flow of JIP framework. IP mapper and reverse IP mapper use the same mapper table.

Lookahead path selector: While performing the extended lookahead in the subsequent cycles, we may have multiple control flow paths to select from: (i) starting from the last-prefetch-IP and (ii) another starting from the last-temporal-table-target-IP (if the trigger-IP gets a hit in the temporal table). We use a 9-bit saturating LAP (lookahead path)-confidence-counter that we initialize to 256. The counter selects between these two control flow paths. We modify the LAP-confidence-counter for every accurate request that JIP makes during the extended lookahead, increment it by two or decrement it by one when we perform the extended lookahead starting from the last-temporal-table-target-IP or the last-prefetch-IP, respectively. Note that we favor the temporal table target path over the last prefetch IP path. We reset the LAP-confidence-counter to 256 after every 256 L1-I accesses. This requires a counter (L1-I-accesses) to track the number of L1-I accesses.

To track the accuracy of both the extended lookahead paths, JIP uses two 64-entry queues that we call lookahead prefetch request queues (LAPRQs). LAPRQs store the cache line addresses for two paths: one for the path following last-temporal-table-target-IP and another for the path following the last-prefetch-IP. Whenever JIP observes an L1-I hit for an entry in one of the LAPRQs, it modifies the LAP- confidence-counter to favor the path corresponding to that LAPRQ.

Practical Implementation of JIP: For the sake of competition, we use fully-associative tables. However, we find making the fully-associative tables direct-mapped, causes an average performance degradation of just 1.31%. As it is impractical to access JIP tables lookahead-depth times in a single cycle, we can implement a Bloom filter [10] to classify each IP as branch or non-branch. As we only store branches in the JIP tables, we need to access the tables only if the IP is a branch IP which is only about 20% of the instructions on average. Further, if JIP predicts the following branch path for trigger

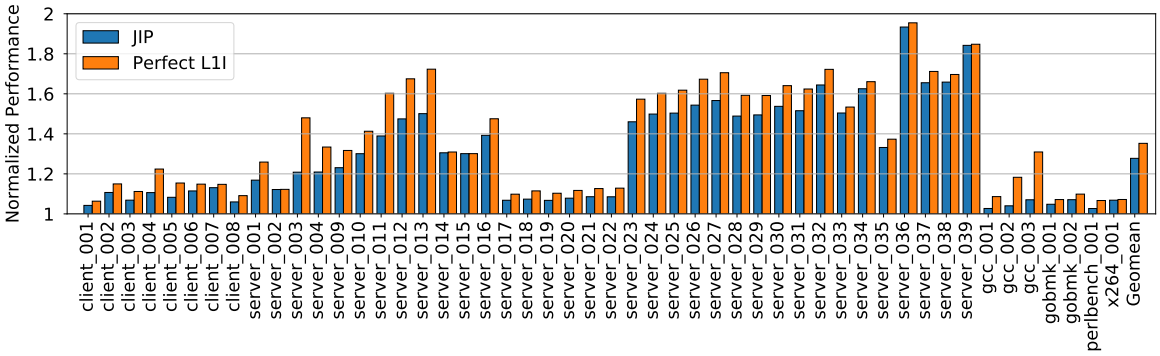

Fig. 3. Speedup normalized to no prefetching with JIP and the Perfect L1-I.

TABLE I
HARDWARE OVERHEAD. REFER FIGURE 1 FOR THE DETAILS ABOUT DIFFERENT HARDWARE TABLES.

	#Entries (A)	Entry size (B bits)	A×B (bits)
Hardware Tables			
SJT	7800	51	397800
MJT-I	1024	112	114688
MJT-II	512	280	143360
Temporal Table	7150	50	357500
Mapper Table	512	57	29184
Queues			
RPQ	64	19	1216
RAQ	25	25	625
LAPRQ (for two paths)	64×2	19	2432
Counters and Registers			
Degree, Depth, L1-I-accesses, LAP-confidence-counter			3+9+8+9=29
Last-prefetch-IP, Last-temporal-table-target-IP			25+25 = 50
Last-prefetch-cycle, Remaining-lookahead-cycles			64+3 = 67
Total			127.8 KB

IP A: (B, C, D, E, F, G, H), then we can store this control flow and verify the temporal access order. So if we observe an L1-I access to IP B, we know that JIP tables predict the same path from IP C for trigger IP B. So, we can avoid accessing the JIP tables until we see L1-I accesses for say IP F or G, or we get to know that JIP's prediction is wrong in which case we need to access the JIP tables immediately to predict a new branch path. This can help us in reducing the number of accesses to JIP tables by four to five times on the basis of JIP's accuracy.

Hardware overhead: A self-contained Table I shows the hardware overhead with the JIP prefetching framework.

IV. EVALUATION

Performance: Figure 3 shows speedup with JIP normalized to a system with no instruction prefetching. On average, JIP provides a speedup of 27.75%. The gap between the speedup of JIP and the perfect L1-I is less than 5% for 25 out of the 50 traces. `gcc` is one of the benchmarks where we fail to bridge the performance gap when we compare it with the perfect L1-I. Recent lightweight prefetcher return-directed instruction prefetching [4] provides performance improvement of 21% with a 128KB storage budget. Our SJT component provides closer to 24% with around 50KB of storage budget (refer Figure 4). Temporal table provides a competitive advantage to JIP, and pushes performance closer to 28%.

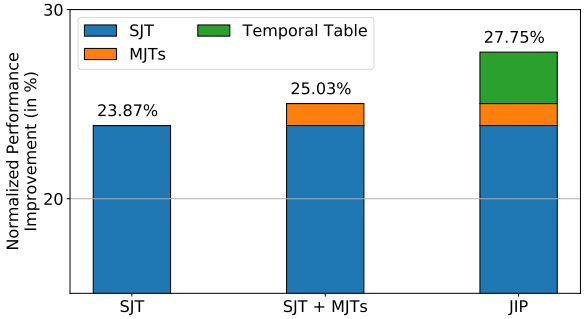

Fig. 4. Utility of different components in our JIP framework. Runner is an integral part in all three cases.

Prefetch coverage and timeliness: JIP achieves a prefetch coverage of 91.33% (as high as 99.99%). To quantify the prefetch timeliness, we use the following metric: 1-$\frac{late-prefetch}{late-prefetch+useful-prefetch}$. JIP is a timely prefetcher with average timeliness of 96.9%.

V. SUMMARY

We proposed a bouquet of instruction pointer (IP) jumpers (JIP) that uses runner and jumpers, to predict the control flow target IPs for non-branch and branch instructions, respectively. JIP covers 91.33% of the L1-I misses by prefetching on time 96.9% of the time, resulting in a high performance improvement of 27.75%.

VI. ACKNOWLEDGEMENTS

We would like to thank all the anonymous reviewers for their helpful comments and suggestions.We would also like to thank members of CARS research group for their feedback on the initial draft. This work is supported by the SRC grant SRC-2922.001.

REFERENCES

[1] Reinman *et al.*, "Fetch directed instruction prefetching," in *MICRO*, 1999.
[2] Ferdman *et al.*, "Temporal instruction fetch streaming," in *MICRO*, 2008.
[3] Kaynak *et al.*, "SHIFT: shared history instruction fetch for lean-core server processors," in *MICRO*, 2013.
[4] Kolli *et al.*, "RDIP: return-address-stack directed instruction prefetching," in *MICRO*, 2013.
[5] Ferdman *et al.*, "Proactive instruction fetch," in *MICRO*, 2011.
[6] Kaynak *et al.*, "Confluence: unified instruction supply for scale-out servers," in *MICRO*, 2015.
[7] Kumar *et al.*, "Boomerang: A metadata-free architecture for control flow delivery," in *HPCA*, 2017.

[8] Kumar *et al.*, "Blasting through the front-end bottleneck with shotgun," in *ASPLOS*, 2018.
[9] S. Pakalpati and B. Panda, "Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching," in *ISCA*, 2020.
[10] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," in *ACM Communications*, 1970.