Bouquet of Instruction Pointers: Instruction Pointer Classifier-based Spatial Hardware Prefetching

Samuel Pakalapati
Intel Technology Private Limited
*Birla Institute of Technology and Science, Pilani**
Hyderabad, India
samuel.pakalapati@intel.com

Biswabandan Panda
Dept. of Computer Science and Engineering
Indian Institute of Technology Kanpur
Kanpur, India
biswap@cse.iitk.ac.in

***Abstract*—Hardware prefetching is one of the common off-chip DRAM latency hiding techniques. Though hardware prefetchers are ubiquitous in the commercial machines and prefetching techniques are well studied in the computer architecture community, the "memory wall" problem still exists after decades of micro-architecture research and is considered to be an essential problem to solve. In this paper, we make a case for breaking the memory wall through data prefetching at the L1 cache.**

We propose a bouquet of hardware prefetchers that can handle a variety of access patterns driven by the control flow of an application. We name our proposal Instruction Pointer Classifier based spatial Prefetching (IPCP). We propose IPCP in two flavors: (i) an L1 spatial data prefetcher that classifies instruction pointers at the L1 cache level, and issues prefetch requests based on the classification, and (ii) a multi-level IPCP where the IPCP at the L1 communicates the classification information to the L2 IPCP so that it can kick-start prefetching based on this classification done at the L1. Overall, IPCP is a simple, lightweight, and modular framework for L1 and multi-level spatial prefetching. IPCP at the L1 and L2 incurs a storage overhead of 740 bytes and 155 bytes, respectively.

Our empirical results show that, for memory-intensive single-threaded SPEC CPU 2017 benchmarks, compared to a baseline system with no prefetching, IPCP provides an average performance improvement of 45.1%. For the entire SPEC CPU 2017 suite, it provides an improvement of 22%. In the case of multi-core systems, IPCP provides an improvement of 23.4% (evaluated over more than 1000 mixes). IPCP outperforms the already high-performing state-of-the-art prefetchers like SPP with PPF and Bingo by demanding 30X to 50X less storage.

***Index Terms*—Hardware Prefetching, Caching**

I. Introduction

Improved hardware prefetchers at the different levels of cache hierarchy translate to performance gain by reducing the off-chip costly DRAM accesses. Hardware prefetchers such as next-line (NL) and stride based on instruction pointer (IP-stride) [18] are some of the simple, efficient, and light-weight data prefetchers employed at the L1 level. Well-established and recent *spatial* L2 prefetchers (prefetchers that prefetch within a spatial region) [33], [13], [14], [38], [11], [45] have pushed the limits of data prefetching. Apart from these spatial prefetchers, there are temporal prefetchers [54], [55], [24], [12], [59], [58] that target irregular but temporal accesses. In general, spatial prefetchers demand less storage (closer to tens of KBs, except spatial memory streaming (SMS) [47] and Bingo [11]) as compared to the temporal ones (closer to hundreds of KBs). In the 3rd Data Prefetching Championship (DPC-3) [3], variations of these proposals were proposed[1].

It is well understood that the prefetchers at L1 and L2 would need to be different as the access patterns at the L2 are different from those at the L1 (filtered by the L1). The primary reason being, identifying access patterns at the L2 is not trivial as the L1 prefetcher may cover a few demand misses or may trigger additional inaccurate prefetch requests jumbling the access pattern at the L2. *Note that, most of the recent spatial prefetchers are L2 based with prefetchers like NL and IP-stride dominating the space of L1 data prefetching.*

The opportunity: One of the key objectives behind designing hardware prefetchers is to break the memory wall by hiding the costly off-chip DRAM accesses. An ideal solution to the memory wall problem would be an L1-D cache (L1-D) hit rate of 100%, with permissible access latency. One of the ways to achieve the same is through L1-D prefetching. Prefetching at the L1-D provides the following benefits (i) unfiltered memory access pattern, (ii) prefetched blocks can get filled into all the levels of cache hierarchy (more importantly, the L1-D), (iii) an ideal L1-D prefetcher can make the L2 prefetcher superfluous.

The challenges: The benefits mentioned above come with the following challenges. (i) Hardware overhead: an L1-D prefetcher should be light-weight. (ii) Prefetch address generation should meet the lookup latency requirement of L1-D. (iii) An L1-D prefetcher should not probe the L1-D on every prefetch access (to make sure that the address is already not present in the L1 cache) as L1-D is bandwidth starved. (iv) Aggressive hardware prefetching may not be possible at the L1-D because of limited entries at the supporting hardware resources such as prefetch queue (PQ) and miss-status-holding-registers (MSHRs). For example, typically, the #entries in the PQ and MSHR of L1-D is one-half of L2's. (v) An L1-D prefetcher with low accuracy can pollute the small L1-D.

The problem: State-of-the-art spatial prefetchers [45] [33], [11], [14], [13] are designed specifically for L2's access patterns.

* A major part of the work was done through a remote internship, while the author was at BITS Pilani.

[1] A preliminary version of bouquet of prefetchers won the 3rd data prefetching championship.

978-1-7281-4661-4/20/$31.00 ©2020 IEEE
DOI 10.1109/ISCA45697.2020.00021

Prefetchers like SMS [47], [49] and Bingo [11] are capable of prefetching at the L1-D. However, both SMS and Bingo demand hardware overhead closer to 100KB.

Our goal is to propose a lightweight spatial L1-D prefetcher that can overcome the challenges and seize the opportunities mentioned above without compromising the prefetch accuracy and prefetch coverage.

Our approach: We propose a prefetching framework in the form of a bouquet of prefetchers based on instruction pointer (IP) classification (driven by the control flow of an application). We cover a wide variety of memory access patterns like (i) only control flow, (ii) control flow predicted data flow, and (iii) control flow coupled with data flow. We find that each IP can be classified into unique IP-classes based on its access patterns, and the resulting classification could be used for better prefetching. We perform IP classification at the L1-D and use it for L1-D prefetching. We also extend our framework to the L2 prefetcher by communicating the IP classification information from the L1-D prefetcher.

Overall, we make the following key contributions:

- We find that spatial access patterns can be correlated with the IPs and motivate the need for lightweight spatial L1-D prefetchers (Section III).
- We propose Instruction Pointer Classification based spatial Prefetching (IPCP) that classifies IPs into three classes and design a tiny prefetcher per class (Section IV). These tiny prefetchers cover more than 60%, 70%, and 80% of the L1, L2, and last-level cache (LLC) demand misses for memory-intensive (LLC MPKI$\geq$1) SPEC CPU 2017 applications, respectively (refer Figure 10).
- We propose a bouquet of tiny prefetchers that work in harmony with each other through per-class throttling and hierarchical priority (Section V).
- We also communicate the classification information from L1-D to L2, facilitating multi-level prefetching using the common theme. Overall, IPCP is an extremely lightweight framework that demands **895 bytes** (Section V).
- On average, compared to no prefetching, IPCP provides a performance improvement of 45.1%, 22%, and 23.4% for single-core memory-intensive applications, single-core all applications, and multi-core mixes with 4 and 8-cores, respectively. IPCP provides this performance with 30X to 50X lower hardware overhead when compared to state-of-the-art spatial prefetchers (Section VI).

II. Related Work

Spatial prefetchers: Spatial prefetchers predict strides, streams, or complex strides within a spatial region providing competitive coverage. Prefetchers like variable length delta prefetching (VLDP) [45] and signature path prefetching (SPP) [33] are well known delta prefetchers. VLDP stores the history of deltas to predict future deltas. SPP is a state-of-the-art delta prefetcher that predicts the non-constant (irregular) strides (commonly known as deltas). SPP works based on the signatures (hash of deltas) seen within a physical OS page to index into a prediction table that predicts future delta. It dynamically controls the prefetch aggressiveness based on the success probability of future deltas.

Spatial Memory Streaming (SMS) [47], [49] is a spatial prefetcher that exploits the relationship between the IP of a memory request, and access pattern within a spatial region based on the IP and the first offset within that region. SMS incurs huge storage overhead, which is larger than the L1-D size. A recent work called Bingo [11], uses multiple signatures (like IP, IP+Offset, and memory region) and fuses them into a single hardware table. Bingo provides better coverage than SMS. However, Bingo incurs similar overhead as SMS (around 119KB). There are component prefetchers like division of labor (DOL) [35] that target specific program semantics (like pointer chains, loops, etc.) for prefetching by getting the information of interest from the processor core.

Offset prefetchers: Offset based prefetchers such as Best-offset Prefetcher (BOP) [38] and Sandbox [42] prefetcher explore multiple offsets. An offset of k means the cache block that is distanced by k cache blocks. These prefetchers choose the offset that provides the maximum likelihood of future use. BOP continues to prefetch with a particular offset till a new offset performs better than the current offset. Multi-Look-ahead Offset Prefetcher (MLOP) [44] is an extension of BOP that considers several lookaheads for each offset, and finds the best offset for each look-ahead. MLOP is motivated by Jain's Ph.D. thesis that proposed an Aggregate Stride Prefetcher (ASP) [23].

Temporal Prefetchers: Temporal prefetchers like temporal streaming [55], Irregular Stream Buffer (ISB) [24], and Domino [12] track the temporal order of accesses. Usually, temporal prefetchers demand hundreds of KBs. Recently, Managed ISB (MISB) [59] and Triage [58] have optimized the hardware overhead without compromising coverage.

Prefetch filters/throttlers: To further improve the effectiveness of hardware prefetchers, prefetch filters like Perceptron-Prefetch-Filter (PPF) [14] and Evicted-Prefetch-Filter (EPF) [43] have been proposed. Apart from filters, there are aggressiveness controllers (throttlers) [16], [20], [30], [31], [39]–[41], [50] that control the prefetch degree and prefetch distance based on prefetch metrics like accuracy, coverage, LLC pollution, and DRAM bandwidth. Dual Spatial Pattern Prefetcher (DSPatch) [13] is an adjunct spatial prefetcher that works like a throttler. It improves the effectiveness of SPP based on DRAM bandwidth utilization.

III. Motivation

Unique and persistent IP behavior: One of the major insights that drive IPCP is that each IP has a specific behavior associated with it. For example, here is an L1-D access pattern (in terms of cache-line aligned addresses) of an $\texttt{IP}_A$ from SPEC CPU 2017 benchmark named `bwaves: C0,C3,C6,C7,C9`. This IP follows a constant stride of three for the most part. A simple IP-stride prefetcher can provide high prefetch coverage, in this case. Here is another access pattern for $\texttt{IP}_B$ from a benchmark named `mcf: C0,C1,C3,C4,C6,C7`. The stride pattern of this IP is 1,2,1,2,1. In this case, an IP-stride prefetcher provides zero coverage as the prefetcher fails to get high

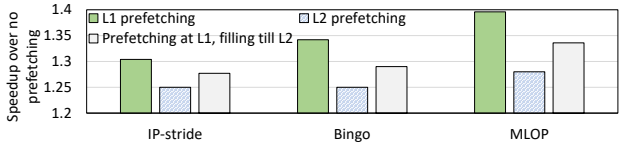

Fig. 1: Utility of L1-D prefetching.

confidence for either of the strides. Another access pattern that is common in streaming benchmarks like `lbm` and `gcc` is the following: $\texttt{IP}_C\texttt{(C0,C2,C1)}$, $\texttt{IP}_D\texttt{(C3,C6,C4,C5)}$, and $\texttt{IP}_E\texttt{(C9,C8,C7)}$. It is a global stream. Looking at the global pattern, we can see that all the accesses are contiguous and limited to a small memory region. However, their access pattern is a bit jumbled based on the program order. In this case, many IPs like IP_C,IP_D, and IP_E follow the global stream. It is clear that IPs are unique and can be classified into different classes based on its access patterns. Note that a particular IP can move from one access pattern to another and it can stay active with one or more access patterns.

Utility of L1 prefetching: To understand the importance of prefetching into the L1-D, we perform a simple experiment with prefetchers like IP-stride, Bingo, and MLOP over 46 memory-intensive SPEC CPU 2017 traces [8] [9] (refer Table II for the simulated systems parameters). Figure 1 shows the utility of prefetching into the L1. On average, compared to a baseline with no prefetching, prefetching into the L1 provides 6% to 13% additional speedup over L2 prefetching. The reasons for the performance difference is obvious: (i) in case of L2 prefetching, prefetched blocks are brought till L2 only, and (ii) access patterns learned at the L2 are noisy because of the L1 filtered accesses. To improve the learning, we also perform experiments where L1 prefetchers learn at L1 but prefetch till the L2. It brings the performance gap to 3-7%. However, there are traces (e.g., `gcc-2226B`) that show a performance difference of more than 73% for Bingo and MLOP. Out of 46 memory-intensive traces, only one trace (`bwaves-2931B`) shows prefetching till L2 is better and by a marginal 1.4%, making a strong case for L1 prefetching.

Dearth of spatial L1-D prefetchers: Note that state-of-the-art spatial prefetchers like SMS, VLDP, SPP, Bingo, and DSPatch are designed for L2 or the LLC. VLDP, SPP, and DSPatch that are specifically designed for L2, provide better performance when employed at the L2 only. SMS and Bingo at L1 do a good job but demand too much storage (∼100KB) for an L1-D prefetcher. Bingo provides better performance density (speedup/KB) over SMS. Figure 1 shows Bingo's performance with a 48KB L1 (our L1-D is of size 48KB, same as the upcoming Intel Ice Lake's L1 [7]).

Key observations: The persistent behavior of IPs demands for an IP classification. Each class can be handled by one prefetcher. It is in contrast to global access based prefetchers where a huge monolithic prefetcher is expected to learn and predict all the memory access patterns. Also, as the state-of-the-art spatial prefetchers are not designed for L1-D or are heavy-weight, combining them at the L1-D to cater a variety

IP Table

IP-tag	Last-vpage	Last-line-offset	Stride	Confidence

Fig. 2: Hardware table for the CS class.

of access patterns is not a promising direction to improve performance. With IPCP, we seek for *high performance with simplicity and modularity using the minimum silicon area*. We use extremely simple and tiny L1-D prefetchers enabling a highly practical design. On top of that, a new access pattern can be added to the existing classes as a new class seamlessly, thus enabling *modularity*.

IV. IP Classifier

We propose a *spatial* IPCP that classifies an IP into three classes. We do not prefetch crossing the page boundary as IPCP is a simple spatial prefetcher that prefetches within a small region (2KB and 4KB)[2].

A. Constant stride (CS):only control flow

IPs that show constant stride in terms of cache line aligned addresses belong to this class. It is a common pattern seen by IPs and can be prefetched using an IP-stride prefetcher.

Figure 2 shows an IP table for prefetching based on the constant strides (CS). For the CS class, an IP table is tagged and indexed by an IP. Each entry in the table has a `stride` field that corresponds to the stride seen by the IP. A 2-bit confidence counter `confidence` is incremented every time the same stride is seen, and decremented otherwise. It is used to determine whether to prefetch using the constant stride or not. The entry also stores the `last-vpage` (last two least significant bits (lsbs) of the last-virtual-page), and the last cache-line-offset (`last-line-offset`) within a page. In the virtual address space, pages are mostly contiguous and a change in the last two lsbs is sufficient to detect a page change (previous page or the next page) seen by the IP. For a 4KB page and 64B cache lines, offset can vary from 0 to 63). The last-line-offset, along with the last-virtual-page, is used to calculate the stride between two accesses from the same IP. The virtual page information is used for learning and calculating the stride when a new page is seen. For example, a change from an offset 63 to 0, with page change in the forward direction, would be (0-63) + 64 = stride of one. It is a small addition to the IP-stride prefetcher.

Training phase: An IP goes through training till it gains enough confidence (counter value greater than one) to prefetch.

Trained phase: Once an IP gains confidence, it is termed as trained, and it starts prefetching as follows: prefetch address = (current cache-line-address) + $k \times$ (learned-stride), where k varies from one to the prefetch degree. Note that a learned IP stops prefetching in case of low confidence and starts prefetching again after gaining confidence.

[2]Exploring IPCP as a light-weight spatio-temporal prefetcher like STeMS [48] along with a synergistic TLB prefetcher is a promising direction of research and we leave the exploration to future work.

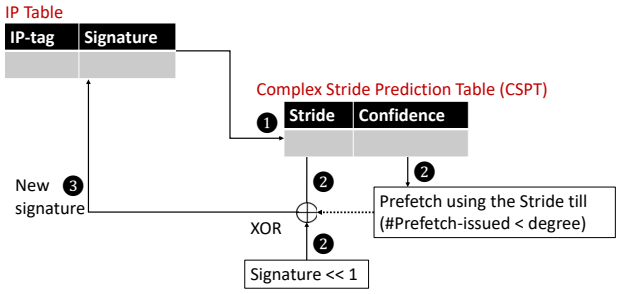

Fig. 3: Hardware table(s) for the CPLX class.

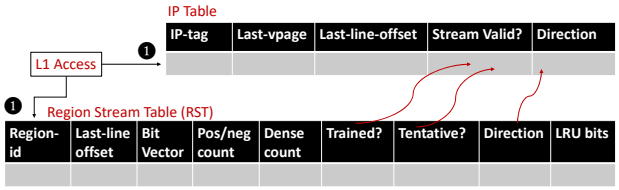

Fig. 4: Hardware table(s) for the GS class.

B. Complex stride (CPLX):control flow coupled with data flow

For access patterns like `C, C+3, C+6, C+10, C+13, C+16`, and `C+20` with strides of 3,3,4,3,3,4, a CS class prefetcher would provide 66% coverage since it would be unable to predict stride 4. Also, if the stride pattern is 1,2,1,2,1,2, a CS class prefetcher would lack the confidence to prefetch any stride since the two strides compete for the same entry in the IP table. In this case, coverage would be zero. We call these patterns as complex strides and create a complex stride class (CPLX) for the corresponding IPs.

We create an n-bit `signature` of strides seen by an IP and use it to index into a complex stride prediction table (CSPT) that predicts future complex strides. An n-bit `signature` captures the last n strides seen by an IP by hashing. The IP table of CPLX class is also tagged and indexed by an IP. The IP table of CPLX class stores the IP-tag and the `signature` that points to the previous stride(s) predicted by the IP. CSPT stores the next predicted stride pointed to by a signature and a 2-bit confidence counter (similar to the CS class). Figure 3 shows the IP table of CPLX class and the CSPT table.

Training phase: An IP with its signature field finds the stride at the CSPT. Every time it sees the same stride the confidence counter is incremented by one and decremented otherwise. This stride is hashed with the existing `signature`, and the CSPT is looked up again to issue prefetch requests. The stride obtained previously is added to the signature according to the equation: *signature = (signature << 1) ^ stride*. Note that we shift the signature by a single bit so that we can accommodate a highly complex stride pattern. Thus a pattern can produce many signatures, but we do not observe too many collisions in the CSPT because there are not many CPLX IPs at the same point of time.

Trained phase: Every time the signature points to the stride, and if the confidence is high enough ($\geq$ one in our case), the complex stride is added to the cache line to produce the prefetch address. This look-ahead continues until the prefetch degree count is reached (❷, ❸, and ❶). If the confidence value is zero, then the stride is added to the signature using the above equation to predict the next stride (❸) and no prefetching is done.

CPLX and SPP: Fundamentally, CPLX class is different from SPP. The latter uses a memory region (an OS page) and captures the deltas observed within a page. However, CPLX uses an IP and there is a difference in the access patterns captured by CPLX. We find, there are cases where IP driven complex strides hold the key. (i) The memory accesses (for a given IP) are sometimes not in the powers of two (memory layout in data structures across cache lines), causing an non-constant stride pattern. For example, consider a cache line of 8 bytes, and if every 12th byte is accessed, the accesses create strides as follows: byte addresses: 0, 12, 36, 48, 72; cache line aligned addresses: 0, 1, 3, 4, 6; strides: 1, 2, 1, 2. (ii) Another case is where the accesses are made by loops at various levels. An outer loop could make constant stride accesses (can be easily captured by the CS class). However, an inner loop could make different stride accesses (depending on the strides of the outer loop), thus causing *bumps* in the stride pattern. An IP based CPLX can exploit this pattern.

Also, CPLX class focuses on local order of complex strides (capturing control and data flow) unlike the global order (data flow) seen by SPP. Note that, SPP is a high performing prefetcher designed for L2 and CPLX alone cannot match SPP's effectiveness (apples vs oranges). CPLX's implementation is extremely lightweight since it is an L1-D prefetcher and has the added benefit of reduced latency on the critical path of issuing a prefetch at the L1 (SPP has to calculate confidence by using logic or lookup tables).

C. Global stream (GS):control flow predicted data flow

A global stream is a set of cache aligned accesses (within a small memory region) that usually follow a bursty pattern, and these accesses can come from different IPs. Prefetching based on the global stream makes more sense as it preserves the global order of accesses (data flow within a region) and results in much better timeliness. We propose a *new* prefetching technique to prefetch global streams. Figure 4 shows the prefetch tables of interest for the GS class.

Training phase: GS class prefetcher uses an IP table (tagged and indexed in the same way as the previous classes) and an IP corresponds to the GS class based on a `stream-valid` bit with a `direction` of the stream. The IP table gets this information from a Region Stream Table (RST).

RST keeps track of regions and their denseness (#accesses). Each region is of size 2KB (bigger size regions take more time to train and provide marginal performance improvement) and it maintains a 32-bit bit-vector (for tracking 32 cache lines). When a new region is accessed, we allocate an entry in RST. If a cache line within that region is accessed for the first time, we set the corresponding bit in the bit-vector and

increment a saturating counter called `dense-count`. The `last-line-offset` within the region is also stored. Note that the width of last-line-offset in the IP table is 6 bits whereas in RST it is 5 bits. If `dense-count` counter crosses a GS threshold (75% of the cache blocks accessed within a region), then the region is a dense region contributing to the GS, and all the IPs accessing this region are classified as GS IPs. Also, the `trained bit` of the corresponding RST entry is set. Note that if a bit in the bit-vector is already set, the counter is not incremented.

RST also uses an n-bit saturating counter (`pos/neg count`) to determine the direction of the stream. Note that this counter does not start from zero. It is initialized to $\frac{2^n}{2}$. The direction is calculated by finding out the difference between two consecutive cache accesses (the difference between the last-cache-line-offset and current access-offset within a region). The `pos/neg count` gets incremented on positive direction and gets decremented on negative direction. Depending on the most significant bit (msb) of the `pos/neg count`, the direction of a GS IP is determined.

When a GS IP encounters a new region, we look at the previous region it had accessed (using `last-vpage` and the msb of `last-line-offset` of the IP table). If the region had already been trained as dense, i.e., the `trained` bit is set in the RST, we assume the new region to be dense, *tentatively* (control flow predicted data flow). The `tentative` bit in the RST entry of the new region is set. If the `trained` bit is not set in the previous region, it may mean that the GS nature is no longer exhibited by the IPs and the `tentative` bit is not set. This feature is designed to prevent locking of behavior due to initial conditions. The reason we are using this scheme is because it takes some time for the region to be trained as dense, and we may not be able to issue GS prefetches during this time. Hence we correlate the training information from the previous region to tentatively issue GS prefetches in the new region.

Trained phase: On a demand access, we check the trained and tentative bits in the RST entry. If either of the bits is set, we call the corresponding IP a GS IP and set the `stream-valid` and `direction` bits in the IP table. Note that, through this scheme, all IPs that access a dense region become GS IPs. Once trained, a GS IP prefetcher just prefetches the next k cache lines based on the trained direction (positive/negative direction), where k is the prefetch degree.

D. A case for tentative NL (tentative data flow)

In case a demand access does not fall into any of the three classes (CS, CPLX, and GS), we use the NL prefetcher. However, the usage of NL prefetcher can be detrimental to performance, especially in case of irregular access patterns. So, we make it tentative. We calculate the L1 misses per kilo instructions (MPKI) per core. Two counters are used, one to count the number of L1 misses and the other to count the number of retired instructions (if this information is unavailable then misses per kilo cycles can also be used and it is equally effective). Since we cannot afford useless prefetches when the MPKI is too high, we turn off NL prefetching at the L1. Based on the MPKI values, a *tentative-NL* bit is set for each cache level when the MPKI is low (50, chosen empirically based on average MPKI when prefetching turned off). NL prefetching is ON only when *tentative-NL* is set.

V. Bouquet of Prefetchers

Based on the classification done in the previous Sections, we design a single IP table *shared* by all three classes as four fields of the IP table are used by all the classes. We have auxiliary tables like CSPT and RST for CPLX and GS class, respectively. Figure 5 shows the IPCP as a framework. On L1 access, IPCP uses the corresponding IP-tag bits to compare entries with the IP table. Our IP table is a direct-mapped, 64 entry table. We get marginal performance improvements with a 128 and 256 entry IP tables, corroborating with recent works [38] and [13] that use IP-stride at the L1 with 64 entries. We use a 128 entry direct-mapped CSPT table that captures a signature of width seven (seven strides).

Since a replacement policy would add latency into the critical path, we use a direct-mapped implementation instead. All the confidence counters are 2-bit wide. We use an eight entry RST to keep track of eight recent regions and maintain LRU order among the regions. As the IP table is shared among the classes, IPCP learns the constant and complex strides by sharing the `IP-tag`, `last-vpage`, and `last-line-offset` fields. GS class use `last-vpage`, and the msb of `last-line-offset` to index into the RST as mentioned in Section IV-C. Now, both CS and CPLX learn their respective strides when they see a new page, as mentioned in Section IV-A.

IP table and hysteresis: As the IP table is direct-mapped and tagged, it is a challenge to decide which IP to keep for prefetching, as there can be collisions between IPs matching to the same table entry, we add an additional field `valid` bit to maintain hysteresis (Figure 5). When an IP is encountered for the first time, it is recorded in the IP table and the `valid` bit is set. When another IP maps to the same entry, the `valid` bit is reset, but the previous entry remains active. If the `valid` bit is reset when a new IP is seen then the table entry is allocated to the new IP and the `valid` bit is set again, ensuring that at least one of the two competing IPs is tracked in the IP table. Note that, `valid` bit is also shared by all the classes.

Priority of classes: In case of an IP table hit (❶ of Figure 5), IPCP checks all three classes concurrently (❷). *Note that, at a given point of time, an IP can be a part of no class, one class or multiple classes.* RST is checked concurrently for its training. In step ❷, IPCP finds out if the IP belongs to CS or GS class. IPCP prioritizes GS over CS if an IP gets a *tie* between GS and CS (primarily for better timeliness and global order). So, at the end of step ❷, IPCP either prefetches based on GS or CS. In step ❸, IPCP goes for the CPLX class (it means the IP does not belong to CS or GS class) by indexing into the CSPT, and if it gets a low-confidence in the CSPT, then it goes for the tentative NL class by looking at the MPKI. In a nutshell, IPCP uses the following *hierarchical* priority:

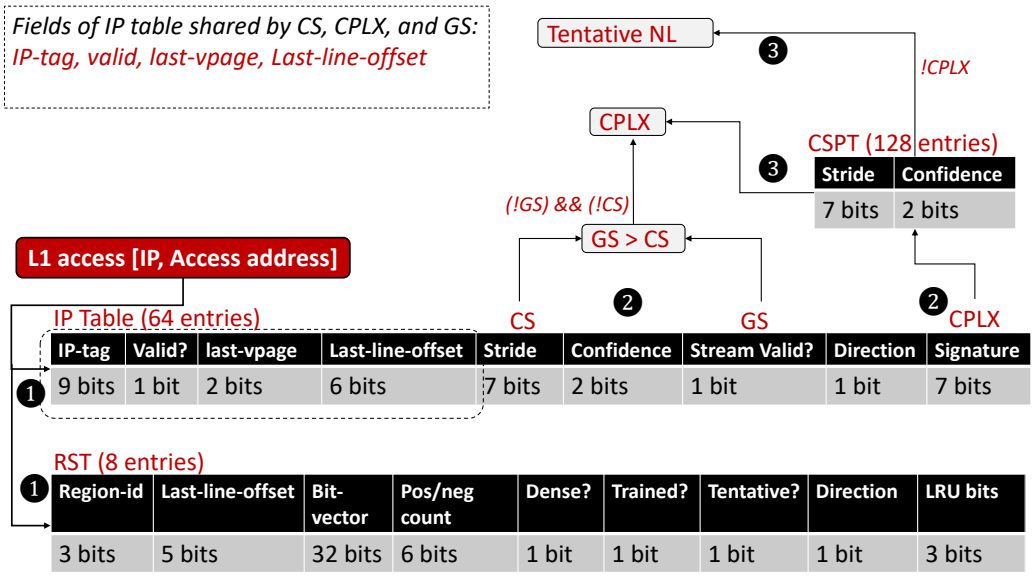

Fig. 5: IPCP as a bouquet of IP classes.

GS, CS, CPLX, and then NL. If an IP belongs to multiple IP classes, then this priority order is used. Note that IPCP does not access the table multiple times to find out the class of particular demand access, because all the information is stored as part of a single entry. IPCP checks all the classes concurrently and finally selects the highest priority class, in case an IP belongs to multiple (or all) classes. We discuss the utility of priority orders in Section VI (Figure 13 (b)).

Lookup latency: The latency incurred during the issue of a prefetch request is three cycles (cycle one: IP-table-lookup, cycle two: prefetch based on CS or GS class as per the priority, and CSPT table lookup, and cycle three: prefetch based on CSPT if confidence is high else tentative-NL prefetching). Since an L1-D lookup is around 5 cycles (48KB L1-D), a prefetch can be issued by the time the corresponding demand request is serviced. In case an L1-D reads two requests per cycles (which is the case in our simulation framework and also in the commercial machines), we go for a pipelined IPCP. Now, the second request is pipelined with the first request's CSPT access, such that the second request's prefetch can be issued at the 4th cycle. We synthesize IPCP (at the RTL level by using VHDL code) with the help of a Design Compiler for 7nm technology, and verify the latency, clocked at 4GHz.

If critical path of L1-D latency is an issue then for CSPT table lookups, the prefetch distance can be increased. For example, if CPLX generates the following prefetch addresses: 10, 25, and 30 then instead of prefetching from address 10, CPLX would start prefetching from address 25. Note that this applies only to the CPLX class.

Coordinated Prefetch Throttling: IPCP issues prefetch requests with a default prefetch degree of three for CS and CPLX classes and six for the GS class at L1. One of the primary reasons for an aggressive GS is, once an IP becomes GS, it means more than 75% of the cache blocks will be accessed within that region. For coordination among the classes, we use an epoch based prefetch accuracy driven throttling mechanism to control the prefetch degrees of the various classes at the L1-D. Each cache line at the L1 contains two bits to indicate the class of the issued prefetch. Two counters are assigned to every class, one to count the number of issued prefetches and the other for the number of useful prefetches.

For each class, once in 256 per-class prefetch fills, we measure the accuracy. We use two watermark thresholds (based on empirical studies) in terms of prefetch accuracy: prefetch accuracy of 0.75 (high water mark) and 0.40 (low water-mark). We do not throttle degrees if the accuracy of a class lies in between 0.40 to 0.75. If the accuracy is greater than 0.75, IPCP keeps on increasing the degree till it reaches the default degree for that class. Similarly, if the accuracy is lower than 0.40, IPCP throttles down the prefetcher until it reaches a prefetch degree of one. With this throttling, IPCP allows other classes like CS and CPLX to prefetch if the accuracy of the high priority class (GS) is too low. For example, if the accuracy of the GS class is below 0.4, then IPCP prefetches based on GS class with the throttled degree and also explores prefetching using CS and CPLX.

L1-D bandwidth and Recent Request Filter: An L1-D prefetcher probes the L1-D before issuing prefetch requests to make sure that it is generating a prefetch address that is not present in the L1-D. However, as mentioned in Section I, L1-D is bandwidth-starved and heavily ported, and adding an additional port, only for a prefetcher is costly, and may not be feasible. To solve this problem, we use a small (32 entry) recent-request filter (RR filter) [15], that keeps track of recently seen tags (partial tag is sufficient) in the demand access and recent prefetch addresses generated by the IPCP. These tags

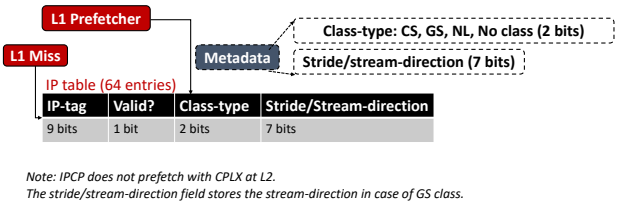

Fig. 6: IPCP at the L2.

TABLE I: Hardware Overhead with IPCP at L1 and L2.

	entry-size in bits × entries	overhead
IPCP at L1	IP table (36×64) + CSPT (9×128) + RST (53×8) + 2 class-bits per line ×64 sets×12 ways (48KB L1) + RR filter (12 bit tag×32 entries)	5800 bits
Others	1 bit: tentative-NL + 8×4 bits: prefetch-issued/class + 8×4 bits: prefetch-hits/class + 10 bits: miss-counter + 10 bits: instruction-counter + 7×4 bits: per-class accuracy registers and one 7-bit MPKI register	113 bits
IPCP at L2	IP table (19 × 64) + 1 bit: tentative-NL + 10 bits: miss counter + 10 bits: instruction counter	1237 bits
	Total overhead: **740 bytes at L1 + 155 bytes at L2**	**895 bytes**

are most likely to stay in the L1-D or the L1 MSHRs (for the misses). Before generating a prefetch request, IPCP probes this filter and in case there is a hit, the prefetch request is dropped.

Multilevel Holistic IPCP: We implement IPCP at two cache levels: L1 and L2. We do not implement it at the LLC, as we do not see any considerable benefit. The prefetch requests issued into L2 and L1 are also filled into the LLC. The access stream at the L2 is now jumbled since it consists of prefetch requests and demand misses from the L1. Thus we cannot train on the L1 misses since some of the misses are converted to hits due to L1 prefetching. This corruption of the stream makes pattern matching at the L2 difficult.

Another alternative is to train the prefetcher at the L1 but to fill it till L2. As IPCP at the L1 is already aggressive, if we issue further prefetch requests (just to fill till the L2), PQ (a FIFO) becomes full and starts dropping prefetch requests, frequently and creates indirect throttling at the L1, affecting both coverage and timeliness. For example, if IP-A and IP-B are accessing the L1 concurrently and if we prefetch for IP-A at the L1 and also for fills till L2 on top of IPCP at the L1 (by prefetching additional requests) then PQ will become full, frequently and prefetch requests for IP-B will be dropped. Note that even if the PQ is not full and the PQ occupancy is high all the time, it affects the timeliness of L1 prefetch requests. *Hence we use the L1 prefetch requests to communicate the IP classification information to the L2 prefetcher by transmitting lightweight metadata along with the prefetch requests. With our communication, we prefetch deep based on the L1 access stream but from L2 and till L2, only. Note that L2 has relatively more resources (PQ=16 entries and MSHR=32 entries) for aggressive prefetching.*

The IP table at the L2 (Figure 6) is only used for book-keeping purposes. IPCP at the L2 does not issue prefetch requests for the CPLX class. CPLX at the L2 does not yield any benefits. It even causes performance degradation for some of the benchmarks when used on top of IPCP at the L1. For the benchmarks that we use, CPLX with prefetch degree of three at the L1 provides a sweet-spot In terms of prefetch coverage and accuracy. CPLX helps IPCP mostly for high MPKI applications with irregular stride patterns. With degree 4 and above, CPLX degrades the performance for high MPKI benchmarks. As we find no utility of deep CPLX prefetching using higher degrees (in contrast to CS and GS classes where we go for deep prefetching), we drop the idea of using CPLX at the L2. Note that, with SPP, large depth (more degree) works well as SPP works on a global access pattern whereas CPLX works on a local per-IP access pattern.

IPCP at the L2 uses an IP table of 64 entries (similar to L1 IP table entries). Each entry contains an IP tag, an IP-valid bit, a 2-bit class-type field (based on the metadata information there are four possibilities, three classes along with the case of no-class), 7-bit stride or the direction of the stream.

On demand access at the L2, the L2 prefetcher issues prefetch requests by consulting the L2 IP table that is populated based on the metadata information communicated by the L1 prefetcher. The stride values of the classes are passed down to the L2 through the metadata only when the accuracy of the respective classes is greater than 75, preventing the L2 from learning and issuing low accuracy prefetches. For the NL class, similar to the L1 level, we use the L2 MPKI threshold (40, chosen empirically, as per the guideline discussed for L1-D) for tentative-NL prefetching. This threshold is essential for high MPKI applications like `mcf`. Note that, if the L2 sees a prefetch request from L1-D with class NL, it simply prefetches NL at the L2. At the L2, for the CS class, IPCP uses a prefetch degree four. We use a higher prefetch degree at L2 because of presence of additional MSHR and PQ entries.

Metadata Decoding at L2: When a prefetch request is issued from L1, IPCP communicates the metadata containing the class type along with its *stride/stream-direction* (in total 9 bits, only). Modern caches use L1 to L2 bus width of 128-bits, 256-bits [32] or more for carrying information for various micro-architecture optimizations. Usually 20 to 30 bits are unused and IPCP can leverage it. However, a hardware vendor may not consider metadata transfer in case the bus width is already utilized, fully. We discuss the utility of metadata transfer in Section VI-B2. The metadata does not contain the IP because the IP of the request is passed to the L2. IP information has been used extensively at the L2 and the LLC [14] [56], [60], [25], [29], [11], [24], [59]. However, if a hardware vendor does not wish to communicate IP information to L2 (as mentioned in [34]), then IPCP provides a simple alternative. As both L1 and L2 IPCP tables are of the same size and use IP-tag, IPCP can communicate the IP-tag and the IP-index, which take up to 15 bits together instead of a 64-bit IP.

Storage overhead: A self-contained Table I shows the storage demand of IPCP at L1 and IPCP as a framework. IPCP is extremely light-weight and tiny with a storage demand of 740 bytes at the L1 and 895 bytes for the entire cache hierarchy. This is a significant improvement compared to lightweight versions of MLOP [44], SPP+PPF [14], and Bingo [11] that

TABLE II: Simulated System parameters.

Core	One to eight cores, 4GHz, 4-wide, 256-entry ROB
TLBs	64 entries ITLB, 64 entries DTLB, 1536 entry shared L2 TLB
L1I	32KB, 8-way, 3 cycles, PQ: 8, MSHR: 8, 4 ports
L1D	48KB, 12-way, 5 cycles, PQ: 8, MSHR: 16, 2 ports
L2	512KB, 8-way, 10 cycles, PQ: 16, MSHR: 32, 2 ports
LLC	2MB/core, 16-way, 20 cycles, PQ: 32×#cores, MSHR: 64×#cores
DRAM	4GB 1 channel/1-core, 8GB 2 channels/multi-core, 1600 MT/sec

demands 10X, 30X, and 50X more storage, respectively. Note that, IPCP at L1 uses virtual address for its training as our L1 is virtually-indexed and physically-tagged. In case of physical-indexed and physically-tagged L1, IPCP demands around 2KB of storage.

A. IPCP and DOL

DOL [35] is a recent prefetching framework that uses component prefetchers (similar to IPCP). However, there are subtle differences that manifest in the performance difference (refer Figure 7). The key differences are as follows: (i) DOL uses a prefetcher that identifies loops and calculates strides to prefetch. Similarly it identifies pointer chains. IPCP does not depend on such application semantics and is only concerned with accesses observed at the memory side. (ii) A component prefetcher called C1 is similar to our GS class, but the former does not have a mechanism to declassify stream-based IPs once they stop being dense. Also, we measure the direction of the global order, whereas DOL randomly prefetches all the cache lines in a region into the L2. (iii) DOL has a P1 component which is a prefetcher for linked data structures (LDSs) that can be integrated into our framework. However, IPCP focusses on spatial prefetching only. Overall, DOL uses narrow component prefetchers that are tightly coupled with the core's 256 entry loop predictor, register files, 32 entry return address stack (RAS), and 192 entry ROB. DOL also demands 32 MSHR (too large for an L1-D) as the components do not have an upper limit on the prefetch degree. DOL's performance is tightly coupled with the core's parameters whereas IPCP is independent of the dynamics of the processor core. The above points are the primary reasons for DOL's poor performance compared to state-of-the-art spatial prefetchers [11], [13], [14].

VI. Evaluation

We evaluate IPCP with an extensively modified ChampSim [4] that faithfully models the entire memory system, including the virtual memory system. ChampSim was used for the 2nd and 3rd data prefetching championships (DPC-2 and DPC-3) [2], [3]. The simulation framework is enhanced with multi-level prefetching for DPC-3. ChampSim is an effective framework to compare the recent cache replacement and prefetching techniques as the fine-tuned source codes of the state-of-the-art techniques are available on the public domain. Recent prefetching proposals [11], [14], [33] have also been coded and evaluated with ChampSim, helping the community for a fair comparison of techniques. Table II shows the simulation parameters. We simulate single-core, 4-core, and 8-core simulations. For single-core, we warm-up caches for 50M sim-point instructions and report the performance (*normalized to no prefetching*) for the next 200M sim-point instructions. For four-core and eight-core simulations, we warm-up caches for 50M instructions per core and then report the normalized weighted-speedup ($\sum_{i=0}^{i=N-1} \frac{IPC_{together}(i)}{IPC_{alone}(i)}$) compared to a baseline with no prefetching for the next sim-point 200M instructions. For each mix, we simulate the benchmarks until each benchmark has executed at least 200M instructions. If a benchmark finishes fast, it gets replayed until all the benchmarks finish their respective 200M instructions. $IPC_{together}(i)$ is the instructions per cycle (IPC) of core i when it runs along with other N–1 applications on an N-core system. $IPC_{alone}(i)$ is the IPC of core i when it runs alone on a multi-core system of N cores.

A. Benchmarks and workloads

We use the SPEC CPU 2017 [8], [9] and CloudSuite [5], [17] (four-core mixes spread across six phases [1], [5]) benchmarks. We also use a set of Convolutional Neural Networks (CNNs) and a Recurrent Neural Network (RNN) [19], [21], [22], [36], [37], [46] that are commonly used in applications like object recognition and image classification. We evaluate IPCP on the entire SPEC CPU 2017 suite based on the sim-point traces provided by DPC-3 [9]. However for brevity, we discuss in detail only the memory-intensive ones (46 traces with LLC MPKI $\geq$ 1). For multi-core (4-core and 8-core) simulations, we simulate homogeneous mixes and heterogeneous mixes. In case of homogeneous mixes, we simulate 92 (46 for 4-core and 46 for 8-core) memory-intensive mixes where a mix contains the same memory-intensive traces, for all the cores. For heterogeneous mixes, we simulate 1000 mixes: 500 random mixes (includes the entire SPEC CPU 2017 suite) and 500 mixes containing only the memory-intensive traces.

Evaluated Prefetching Techniques: We compare the effectiveness of IPCP with L1 prefetchers like NL, Stream [51], BOP, VLDP, SPP, DSPatch, MLOP, TSKID, DOL, SMS, and Bingo. Table III provides the details of top four multi-level prefetching combinations based on their performance on single-core and multi-core mixes. Note that for all the prefetchers, we have used their highly tuned equivalent by sweeping through all of the possible parameter space including prefetch table sizes (from 0.5KB onwards). Also, we sweep through all the possible combinations of L1, L2, and LLC prefetchers. We do not implement Bingo at the LLC as it provides low performance (the reason being, Bingo [11] is implemented with 37.5GBps DRAM bandwidth, fixed latency DRAM. So with 12GBps it is unable to perform to its peak). Also, for L2 prefetching, SPP with PPF [14] and DSPatch [13] (SPP+Perceptron+DSPatch) provides better performance than SPP+PPF and SPP+DSPatch[3].

[3]The SPP+PPF code available at DPC-3 is buggy. The authors of SPP shared the bug-free SPP+PPF (P. Gratz, Personal Communication, October 28, 2019). Applying DSPatch on top of SPP+PPF has mixed utility. So we decided to use both PPF and DSPatch as DSPatch provides additional coverage.

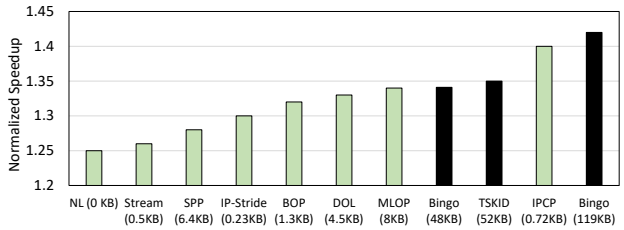

Fig. 7: L1 prefetchers for memory-intensive traces.

B. Single-core Results

1) IPCP as an L1 only prefetcher: Figure 7 shows the performance of competing prefetchers (for memory-intensive SPEC CPU 2017 traces) when employed at the L1 (L2 and LLC prefetchers are turned off). We do not show the performance of VLDP and DSPatch as on average, SPP performs better than VLDP and DSPatch, at the L1. Similarly, Bingo performs better than SMS [47] with relatively less storage demand. Note that Bingo demands 119KB at the L1-D. On average, Bingo provides similar performance with 44KB to 72KB hardware overhead. So, we tune Bingo to make it same as the L1-D size (48KB). We simulate Bingo with 119 KB, too. We also relax the lookup latency bottleneck at the L1-D for all the competing prefetchers. This helps us in understanding the best performance that we can get with the ideal L1-D implementations. Clearly, IPCP outperforms all others except Bingo with 119KB. As expected, SPP does not perform well at the L1-D. One of the reasons for this is a region based global order driven prefetching. Same applies to VLDP and DSPatch. Note that, these prefetchers improve performance when employed at L2.

Next, we try to see the effects of L2 and LLC prefetchers on top of these prefetchers. We sweep through all the prefetchers and their possible combinations at different cache levels and find that the combinations proposed at the DPC-3 are the best multi-level prefetching options (Table III). Bingo at L1 with 48KB and a restrictive NL at L2 and LLC (NL on demand accesses only) provides similar effectiveness as 119KB at L1.

Note that, we find, if the L1 prefetcher is high performing then L2 and LLC prefetchers bring marginal utility. This is surprising and counter-intuitive. To understand this statement, we simulate IPCP at the L1 with various L2 prefetchers (SPP+Perceptron+DSPatch, BOP, VLDP, MLOP, IP-Stride, and Bingo) and find that the utility of L2 prefetchers is negligible (less than 1.7%). SPP+Perceptron+DSPatch is the best L2 prefetcher. Normally, L2 prefetchers should provide additional performance on top of an L1-D prefetcher by prefetching deep into the access stream based on the L1-D's prefetch accesses at the L2. However, with an aggressive IPCP at L1, the opportunity for the other L2 prefetchers is limited. This observation applies to other high performing L1 prefetchers like MLOP and Bingo. We observe a trend that is same as the DPC-3 [3] where prefetchers used at L2 and LLC on top of a high performing L1-D prefetcher are NL prefetchers. This observation opens up an interesting dimension on the *multi-level prefetching* where we need L2 prefetchers that can complement L1-D prefetchers.

TABLE III: Combinations for multi-level prefetching.

Combination name	Prefetchers at L1, L2, and L3
SPP+Perceptron +DSPatch	SPP+Perceptron+DSPatch(L2), throttled-NL(L1) [10], NL(LLC) = **32KB at L2 + 0.6KB at L1**
MLOP	MLOP(L1) and NL(L2+LLC) = **8KB (L1)**
Bingo	Bingo(L1) and NL(L2+LLC): 6K entry history table = **48KB (L1)**
TSKID	TSKID(L1) and SPP(L2): 52KB at L1 + 6.4KB at L2 = **58.4KB**
IPCP	IPCP(L1+L2): 740 bytes at L1 + 155 bytes at L2 = **895B**

TABLE IV: Prefetch Coverage and prefetch accuracy for different Combinations of multi-level prefetching.

Combination name	Coverage	Accuracy
SPP+Perceptron+DSPatch	0.50 at L1, 0.75 at L2,and 0.83 at L3	0.75 at L2
MLOP	0.59 at L1, 0.72 at L2, and 0.78 at L3	0.64 at L1
Bingo	0.54 at L1, 0.72 at L2,and 0.80 at L3	0.79 at L1
TSKID	0.67 at L1, 0.72 at L2, and 0.80 at L3	0.60 at L1
IPCP	0.60 at L1, 0.79 at L2, and 0.83 at L3	0.80 at L1

2) Performance with multi-level prefetching: Due to space limitations, we compare (Figure 8), in detail, IPCP with the top three prefetching combinations (in terms of performance) as mentioned in Table III. The effectiveness of Bingo goes down in case of multi-level prefetching as two other combinations use the state-of-the-art SPP at their respective L2s (Bingo does not perform well with SPP at L2 as discussed in Section VI-B). MLOP at L1 complements well with an NL at L2. Our observations for Bingo are same as the trend observed at the DPC-3 (refer slide no.4 [6]). For multicore workloads, the trend changes as Bingo joins the league of top-performing prefetchers. IPCP at the L2 improves performance on top of L1 because of the holistic semantics of IPCP, both at the L1 and L2. DOL [35] at L1 and L2 fails to outperform the top four prefetchers. IPCP performs better than DOL for the reasons mentioned in Section V-A.

Detailed Performance: Figure 8 shows the effectiveness of IPCP along with the next top three prefetchers (in terms of performance) for a set of 46 memory-intensive traces. On average, IPCP provides 45.1% improvement, where the rest three prefetchers perform equally well (improvements $\sim$ 42.5%). We also evaluate the entire SPEC CPU 2017 suite (a collection 98 traces) where on average, IPCP provides an average improvement of 22% whereas the next top three provide performance in the range of 18.2% to 18.8%. Note that there is only one benchmark named `623.xalancbmk` (not shown in Figure 8 as it is not memory-intensive) where all the prefetchers fail to improve performance for traces that start after 325 billion instructions [9]. IPCP outperforms other prefetchers for all the traces (or provide the same level of effectiveness) except for `cactusBSSN` and `fotonik`. For `cactusBSSN`, TSKID and MLOP outperform all the prefetchers at the L1-D. `cactusBSSN` has many IPs whose reuse distance is more than 1024. So in an extreme case, we need a 1024 associative table, which is practically not feasible at the L1. When we simulate with a 1024 associative table, we get performance closer to MLOP but not TSKID. Also, the prefetched blocks, even though correct, are prefetched too early and are replaced by other loads before they are used

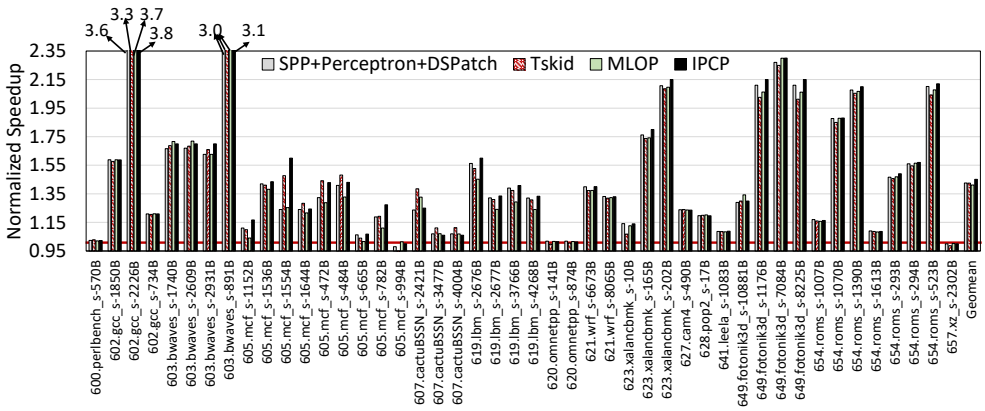

Fig. 8: Normalized performance compared to no prefetching.

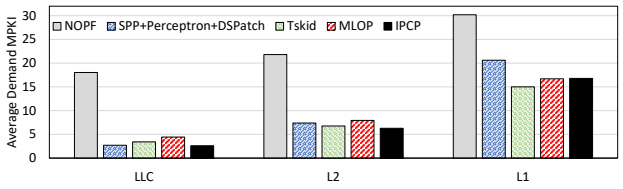

Fig. 9: Reduction in demand MPKI for all the prefetchers.

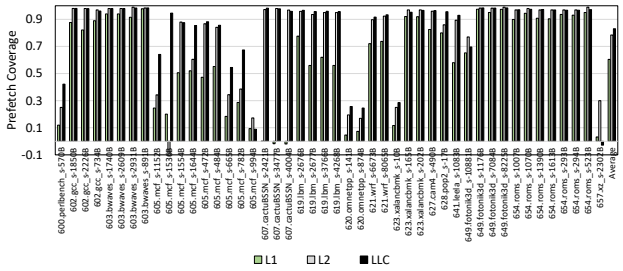

Fig. 10: Demand misses covered by IPCP at L1, L2, and LLC.

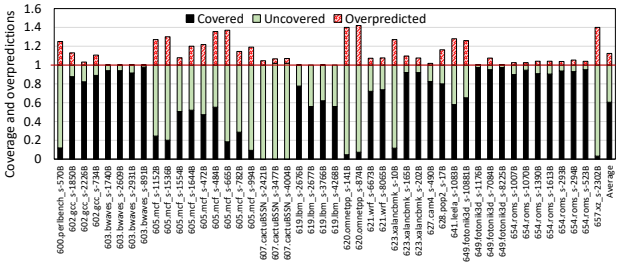

Fig. 11: Coverage and accuracy with IPCP at the L1.

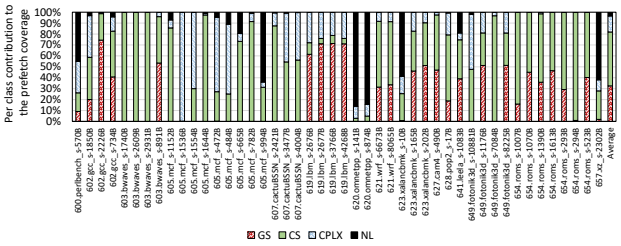

Fig. 12: Contribution of each class on L1 prefetch coverage.

(small L1-D). TSKID takes care of that by prefetching at the right time, but by consuming more than 50KB at L1-D. Overall, for the entire SPEC CPU 2017 suite, the maximum performance improvement with IPCP is 380% while the minimum is a 2% degradation (only for post-325 billion `xalancbmk` traces).

Prefetch coverage: Figure 9 shows the reduction in demand MPKI for the competing prefetchers at all the cache levels. To better understand the MPKI improvements, Figure 10 shows the demand misses that are covered by IPCP at all the levels of the cache hierarchy. On average, IPCP covers 60%, 79.5%, and 83% of the demand misses at L1, L2, and the LLC, respectively. For some of the irregular traces of benchmarks like `mcf` and `omnetpp`, IPCP provides poor coverage. This trend is well-known, and state-of-the-art spatial prefetchers, including IPCP, fail to cover a majority of misses for these two benchmarks. TSKID provides the best L1 coverage of 67%, and MLOP covers 59% of the L1 misses.

Note that for `cactusBSSN`, IPCP provides zero or less than zero coverage at the L1. IPCP does not incur cache pollution at L1 and L2 that can impact performance for all the traces (except for `mcf`, `cactusBSSN`, and `omnetpp`). At the L2 and LLC, IPCP covers 4.5% to 8% more misses compared to SPP+Perceptron+DSPatch, TSKID, and MLOP. SPP+Perceptron+DSPatch provides a coverage of 75% while the rest provide a coverage of 72%. Table IV provides the details about prefetch coverage and prefetch accuracy of all the multi-level prefetching combinations.

Predictions, over-predictions, and utility of classes: Figure 11 shows the demand misses that are covered, uncovered, and over-predicted with IPCP, at the L1. The trend remains similar for the L2 IPCP prefetcher (except for `cactusBSSN`), with no contribution from the CPLX class. Figure 12 digs deep into the prefetch coverage numbers at the L1 and shows which class contributes how much to the prefetch coverage. On average, GS and CS classes contribute

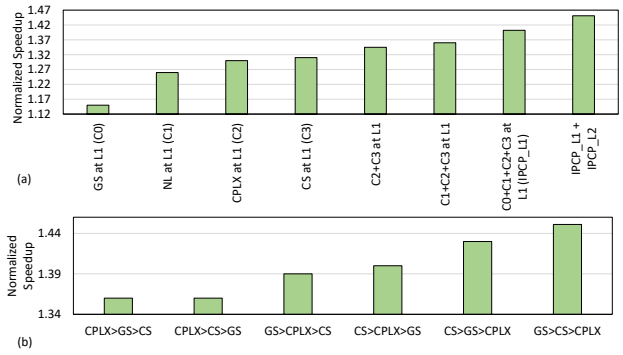

Fig. 13: (a) Utility of IPCP classes and (b) class priority.

30% and 46.7% of the total coverage, respectively. CPLX and NL cover some of the complex and irregular strides in the irregular benchmarks like `mcf` and `omnetpp`. Streaming benchmarks like `lbm`, and `gcc` benefit from GS class the most, and in case the GS class fails, the CS class does a good job to cover most of the misses. Some of the phases of `mcf` are regular (like the trace `mcf-1152B`) and are covered by the CS class. However, `mcf-1536B` has irregular accesses and gets the coverage from CPLX class only. NL helps in covering some of the irregular accesses. However, in the process, brings in-accurate blocks too. The impact of NL's inaccurate prefetch is low as it is not called often and is only triggered when none of the classes deliver. So, in terms of absolute prefetch count, NL contributes marginally.

DRAM bandwidth: Compared to no prefetching, IPCP demands additional bandwidth of 16.1% for performance improvement of 45.1%. SPP+Perceptron+DSPatch and MLOP demand 28% additional bandwidth, whereas TSKID demands bandwidth of 38% (with a maximum of 692% for `mcf-994B`).

Utility of IPCP classes and metadata: Figure 13 (a) shows the utility of each class when used in isolation, and when used as a bouquet in the form of IPCP. It is interesting to see that GS class at the L1 alone is unable to provide more than 15% performance improvement. CS and CPLX are the top performers in isolation, providing more than 30% when employed at the L1. CS+CPLX crosses 34%, and with the help of tentative NL, it covers some of the irregular accesses providing a performance improvement of 36%. GS class alone is not effective. However, when added to the bouquet, it improves the effectiveness of IPCP to 40%. IPCP at the L2 provides an additional 5.1% performance improvement by prefetching deep based on the L1 metadata.

To understand the utility of IPCP class priority, we perform prefetching with different combinations of priorities. Figure 13 (b) shows the utility of different priority orders. Prioritizing an aggressive GS over others provides the maximum benefit showing the effectiveness of IPCP's priority order. If we change the priority order, then there is a performance gap of 9%. IPCP without meta-data transfer sees a performance drop of 3.1%. So, if a hardware vendor cannot use metadata then there will be a performance loss of 3.1% on memory-intensive applications. Metadata helps in synergistic IPCP prefetching at the L2.

Differentiating factor: Compared to the competing prefetchers, the differentiating factor with IPCP is (i) the usage of the GS class, (ii) its aggressiveness, and (iii) prioritization order among the classes. Note that with GS, IPCP gets better coverage and timeliness at the expense of accuracy. In the process, the GS class covers misses that are hard to cover by the competing prefetchers.

In summary, a lightweight IPCP covers a majority of demand misses throughout the hierarchy resulting in 45.1% improvement for the memory-intensive traces and 22% improvement on the entire SPEC CPU 2017 suite. This improvement comes with 16% additional DRAM traffic and **895B** storage overhead.

C. Sensitivity studies

Effect of LLC replacement policies: We study the effectiveness of IPCP with a variety of replacement policies [25]–[29], [56], [57], [60]. IPCP is resilient to the underlying replacement policies with marginal performance difference (less than 1%). However, with multiperspective placement, promotion, and bypass (MPPPB) [29], all the prefetchers see an average performance drop of 3%. TSKID goes down by 6% with HAWKEYE [25], [26], which is mostly attributed to the inaccurate prefetch requests.

Effect of cache sizes and hierarchies: Based on the recent industry trends, we quantify the effect of IPCP with different combinations of cache sizes (32KB and 48KB of L1; 256KB, 512 KB, and 1MB of L2; and 1MB, 2MB, and 4MB of LLC). IPCP is resilient to all the combinations with a maximum performance difference of 1.05%. We also test IPCP with an extremely small LLC (512KB/core) and find that IPCP outperforms other prefetchers by the same margin as in the case of 2MB/core. However, in terms of an absolute performance improvement, it goes down by 3%, which is the case for all competing prefetchers too.

Sensitivity to DRAM bandwidth: To understand the effect of DRAM bandwidth, we perform experiments with low DRAM bandwidth of 3.2GBps and high DRAM bandwidth of 25GBps (dual channel DDR4-1600). With 3.2GBps, benchmarks like `mcf`, `fotonik3d`, and `omnetpp` see a performance degradation for all the prefetchers. On average, IPCP beats the second-best prefetcher (MLOP in this case) by 1%. With 25GBps, all prefetchers (except MLOP) improve their performance by 2% to 3%. SPP+Perceptron+DSPatch does a good job with high DRAM bandwidth and IPCP beats it by 1.5%. MLOP's performance does not scale well with the increase in DRAM bandwidth and the performance does not improve.

PQ and MSHR entries: The effectiveness of an aggressive IPCP at the L1 is limited by the #entries in the MSHR and PQ. So far, we have evaluated IPCP with PQ of eight entries and MSHR of 16 entries. We sweep through the following pair of (PQ,MSHR) entries: (2,4), (4, 8), and (16, 32) and evaluate

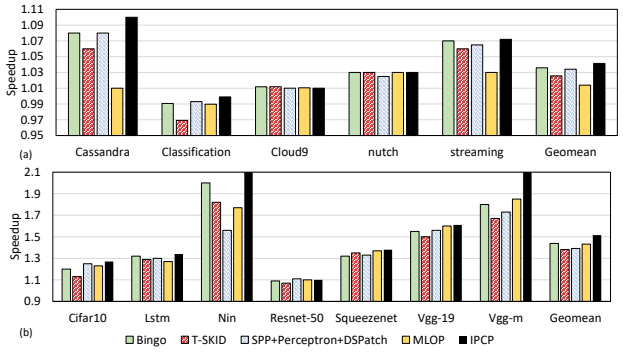

Fig. 14: Speedup for (a) CloudSuite and (b) CNNs/RNN.

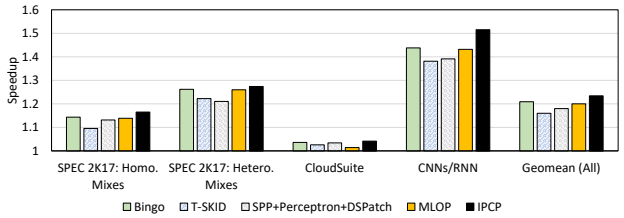

Fig. 15: Summary of multi-core performance.

the effectiveness of IPCP. Compared to the baseline pair (8,16), (2,4) sees a performance drop of 2.7% (maximum drop for `gcc-2226B` that drops from 380% to 323%). We observe marginal performance drop (around 1.5%) for applications like `lbm` and `bwaves`. On average, applications with high memory level parallelism (MLP) get affected with the (2,4) pair.

Sensitivity to prefetch table size: IPCP, in its current form is extremely light-weight and only brings marginal average improvement (0.7%) when we increase the size of IP table, RST, and the CSPT, two to 100 times. This is not a surprising trend. With 895 bytes, IPCP is able to capture the IPs that are sufficient for competitive spatial prefetching, for SPEC CPU 2017, CloudSuite, and CNNs/RNN. Note that, for large code footprints or in case of outliers like `cactusBSSN`, size of the tables should be increased for better performance.

D. Multicore results

For the evaluation of multi-core systems, we compare IPCP with one additional prefetcher named Bingo that performs well in the multi-core system. One of the primary reasons is that Bingo uses an NL prefetcher at the LLC that covers some of the costly misses, and it complements well with Bingo.

Homogeneous memory-intensive mixes: For homogeneous mixes, on average, IPCP provides a 16.5% improvement, whereas Bingo provides 14% improvement, and SPP+Perceptron+DSPatch and MLOP provide just above 13%. For homogeneous mixes of `omnetpp`, `xalancbmk`, and `xz`, it degrades performance by 1%, whereas for mix of `605.mcf-994B`, IPCP degrades the performance by 4%, and it is the only mix where it fails. IPCP provides maximum improvement of 66% for the mix containing `fotonik.1176B`. TSKID degrades performance significantly (67% for `mcf`), whereas others (SPP+Perceptron+DSPatch, MLOP, and Bingo) degrade performance in the range of 10 to 14%. One of the primary reasons for this trend across all the prefetchers, including IPCP is, contention at the LLC and the DRAM bandwidth. It is specific to these mixes as all cores run the same applications. We find that DRAM bandwidth is the major limiting factor compared to the LLC contention.

Heterogeneous mixes: For heterogeneous mixes, we use both memory-intensive and non-intensive traces and evaluate on 1000 mixes as discussed in Section VI-A. On average, Bingo, MLOP, and IPCP perform similarly. IPCP provides an improvement of 27.4%, whereas Bingo and MLOP improve performance by 26.1% and 25.9%. Note that, this trend is a bit different with the DPC-3 trend for multi-core mixes. One of the key changes that help IPCP's performance is the accuracy based coordinated throttling, which is crucial for heterogeneous mixes with non-uniform DRAM bandwidth demands.

Differentiating factor: Except TSKID, all prefetchers perform on a similar scale for the majority of the mixes. The differentiating factor is the performance improvement in the case of mixes that are not well predicted by all the prefetchers, including IPCP. Some examples of these kinds of mixes are mixes that contain memory-intensive applications. A mix containing `605.mcf-1536B`, `605.mcf-1554B`, `605.mcf-1644B`, and `605.mcf-994` is one such mix where the competing prefetchers lose performance in the scale of 50 to 70%, whereas IPCP degrades by 9% thanks to coordinated throttling. Improving performance for these mixes is not trivial. Turning off all the prefetchers is the obvious approach. We believe there is a need of future research for these kinds of mixes. Overall, we find that SPP+Perceptron+DSPatch's coverage is highly dependent on the accuracy of throttling decisions and some of the thresholds used are too strict (threshold of 90 in the scale of 100). MLOP uses coverage and timeliness to select prefetch offsets, providing better performance.

Performance for CloudSuite and Neural Networks: Figures 14 (a) and (b) show performance improvement with CloudSuite benchmarks and some of the applications from the world of CNNs and RNN. `Classification` is one of the benchmarks where all the prefetchers fail. On average, SPP+Perceptron+DSPatch, Bingo, and IPCP perform on the same scale. It is well known that spatial prefetchers fail to improve performance for server workloads like CloudSuite [14], [33], [53], [58], [59] and additional prefetchers [12], [24], [52], [58], [59] can be used on top of IPCP to improve the performance. As IPCP demands less than 900 bytes, all the temporal prefetchers can use IPCP as their spatial counter-part. For the neural networks, IPCP outperforms the rest of the prefetchers primarily because these applications are mostly streaming in nature.

Figure 15 summarizes multi-core results spanning across SPEC CPU 2017 homogeneous and heterogeneous mixes, CloudSuite, and neural networks. On average, IPCP provides

performance improvement of 23.4% while the next best- Bingo and MLOP provide 20.9% and 20%, respectively.

VII. Summary

In this paper, we make a case for tiny high-performing prefetchers at the L1-D level. We find that a high-performing L1-D prefetcher can bring more performance and make the utility of L2 prefetching marginal. To achieve high performance with extremely low storage overhead, we propose an IP Classifier based spatial Prefetching framework (IPCP). IPCP classifies IPs into three classes that cover the majority of access patterns and they work in harmony. We extend IPCP to the L2 level by communicating the classification information, making IPCP an attractive prefetching framework for the multi-level cache hierarchy. In summary, IPCP, as a framework, incurs a hardware overhead of 895 bytes only, and outperforms state-of-the-art spatial prefetchers and multi-level prefetching combinations. Based on the observations and insights, we believe IPCP opens up new directions of research on multi-level prefetching, which are as follows: (i) designing a high-performing L2 prefetcher that can cover the misses that are not covered by the L1 prefetcher and (ii) enhancing IPCP with a temporal component for covering temporal and irregular accesses.

VIII. Availability

The source code is available at https://www.cse.iitk.ac.in/users/biswap/ipcp.html.

IX. Acknowledgements

We would like to thank all the anonymous reviewers for their helpful comments and suggestions. Special thanks to Nilay Shah for helping us in running multi-core experiments. We would also like to thank members of CARS research group, Andre Seznec, Pierre Michaud, R. Govindarajan, Rahul Bera, and Nayan Deshmukh for their feedback on the initial draft. This work is supported by the SRC grant SRC-2922.001.

References

[1] "2nd cache replacement championship." [Online]. Available: https://crc2.ece.tamu.edu/

[2] "2nd data prefetching championship." [Online]. Available: http://comparch-conf.gatech.edu/dpc2/

[3] "3rd data prefetching championship." [Online]. Available: https://dpc3.compas.cs.stonybrook.edu/?final_programs

[4] "Champsim simulator." [Online]. Available: https://github.com/ChampSim/ChampSim

[5] "Cloudsuite traces." [Online]. Available: https://www.dropbox.com/sh/pgmnzfr3hurlutq/AACciuebRwSAOzhJkmj5SEXBa/CRC2_trace?dl=0&subfolder_nav_tracking=1

[6] "Dpc-3 closing remarks." [Online]. Available: https://dpc3.compas.cs.stonybrook.edu/slides/dpc3_closing.pdf

[7] "Intel ice lake." [Online]. Available: https://en.wikipedia.org/wiki/Ice_Lake_(microprocessor)

[8] "Spec cpu 2017." [Online]. Available: https://www.spec.org/cpu2017/

[9] "Spec cpu 2017 traces (spec speed: 6xx numbered)." [Online]. Available: http://hpca23.cse.tamu.edu/champsim-traces/speccpu/

[10] "Throttled {NL at L1 with spp+ppf." [Online]. Available: https://dpc3.compas.cs.stonybrook.edu/src/enhancing.zip

[11] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Bingo spatial data prefetcher," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2019, pp. 399–411.

[12] M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Domino temporal data prefetcher," in *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*, 2018, pp. 131–142. [Online]. Available: https://doi.org/10.1109/HPCA.2018.00021

[13] R. Bera, A. V. Nori, O. Mutlu, and S. Subramoney, "Dspatch: Dual spatial pattern prefetcher," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019.*, 2019, pp. 531–544. [Online]. Available: https://doi.org/10.1145/3352460.3358325

[14] E. Bhatia, G. Chacon, S. H. Pugsley, E. Teran, P. V. Gratz, and D. A. Jiménez, "Perceptron-based prefetch filtering," in *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*, 2019, pp. 1–13. [Online]. Available: https://doi.org/10.1145/3307650.3322207

[15] M. Chaudhuri and N. Deshmukh, "Sangam: A multi-component core cache prefetcher," in *3rd Data Prefetching Championship*, 2019.

[16] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, "Coordinated control of multiple prefetchers in multi-core systems," in *42st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42 2009), December 12-16, 2009, New York, New York, USA*, 2009, pp. 316–326. [Online]. Available: https://doi.org/10.1145/1669112.1669154

[17] 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 *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, 2012, pp. 37–48. [Online]. Available: https://doi.org/10.1145/2150976.2150982

[18] J. W. C. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, ser. MICRO 25. Los Alamitos, CA, USA: IEEE Computer Society Press, 1992, pp. 102–110. [Online]. Available: http://dl.acm.org/citation.cfm?id=144953.145006

[19] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016, pp. 770–778.

[20] W. Heirman, K. D. Bois, Y. Vandriessche, S. Eyerman, and I. Hur, "Near-side prefetch throttling: Adaptive prefetching for high-performance many-core processors," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '18. New York, NY, USA: ACM, 2018, pp. 28:1–28:11. [Online]. Available: http://doi.acm.org/10.1145/3243176.3243181

[21] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997. [Online]. Available: http://dx.doi.org/10.1162/neco.1997.9.8.1735

[22] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size," *arXiv:1602.07360*, 2016.

[23] A. Jain, "Exploiting long-term behavior for improved memory system performance," in *Ph.D. dissertation*. Austin TX, USA, 2016.

[24] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in *The 46th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-46, Davis, CA, USA, December 7-11, 2013*, 2013, pp. 247–259. [Online]. Available: https://doi.org/10.1145/2540708.2540730

[25] A. Jain and C. Lin, "Back to the future: Leveraging belady's algorithm for improved cache replacement," in *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*, 2016, pp. 78–89. [Online]. Available: https://doi.org/10.1109/ISCA.2016.17

[26] A. Jain and C. Lin, "Rethinking belady's algorithm to accommodate prefetching," in *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018*, 2018, pp. 110–123. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00020

[27] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, and J. Emer, "Adaptive insertion policies for managing shared caches," in *2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Oct 2008, pp. 208–219.

[28] A. Jaleel, K. B. Theobald, S. C. S. Jr., and J. S. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *37th International Symposium on Computer Architecture (ISCA 2010), June 19-23, 2010, Saint-Malo, France*, 2010, pp. 60–71. [Online]. Available: https://doi.org/10.1145/1815961.1815971

[29] D. A. Jiménez and E. Teran, "Multiperspective reuse prediction," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2017, Cambridge, MA, USA, October 14-18, 2017*, 2017, pp. 436–448. [Online]. Available: https://doi.org/10.1145/3123939.3123942

[30] V. Jiménez, A. Buyuktosunoglu, P. Bose, F. P. O'Connell, F. J. Cazorla, and M. Valero, "Increasing multicore system efficiency through intelligent bandwidth shifting," in *21st IEEE International Symposium on High Performance Computer Architecture, HPCA 2015, Burlingame, CA, USA, February 7-11, 2015*, 2015, pp. 39–50. [Online]. Available: https://doi.org/10.1109/HPCA.2015.7056020

[31] V. Jiménez, R. Gioiosa, F. J. Cazorla, A. Buyuktosunoglu, P. Bose, and F. P. O'Connell, "Making data prefetch smarter: adaptive prefetching on POWER7," in *International Conference on Parallel Architectures and Compilation Techniques, PACT '12, Minneapolis, MN, USA - September 19 - 23, 2012*, 2012, pp. 137–146. [Online]. Available: https://doi.org/10.1145/2370816.2370837

[32] H. Kim and P. V. Gratz, "Leveraging unused cache block words to reduce power in CMP interconnect," *Computer Architecture Letters*, vol. 9, no. 1, pp. 33–36, 2010. [Online]. Available: https://doi.org/10.1109/L-CA.2010.9

[33] J. Kim, S. H. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*, 2016, pp. 60:1–60:12. [Online]. Available: https://doi.org/10.1109/MICRO.2016.7783763

[34] J. Kim, E. Teran, P. V. Gratz, D. A. Jiménez, S. H. Pugsley, and C. Wilkerson, "Kill the program counter: Reconstructing program behavior in the processor cache hierarchy," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, 2017, pp. 737–749. [Online]. Available: https://doi.org/10.1145/3037697.3037701

[35] S. Kondguli and M. Huang, "Division of labor: A more effective approach to prefetching," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, June 2018, pp. 83–95.

[36] A. Krizhevsky, "Learning multiple layers of features from tiny images," *University of Toronto*, 05 2012.

[37] M. Lin, Q. Chen, and S. Yan, "Network In Network," *arXiv e-prints*, p. arXiv:1312.4400, Dec 2013.

[38] P. Michaud, "Best-offset hardware prefetching," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, March 2016, pp. 469–480.

[39] B. Panda, "SPAC: A synergistic prefetcher aggressiveness controller for multi-core systems," *IEEE Trans. Computers*, vol. 65, no. 12, pp. 3740–3753, 2016. [Online]. Available: https://doi.org/10.1109/TC.2016.2547392

[40] B. Panda and S. Balachandran, "CAFFEINE: A utility-driven prefetcher aggressiveness engine for multicores," *TACO*, vol. 12, no. 3, pp. 30:1–30:25, 2015. [Online]. Available: https://doi.org/10.1145/2806891

[41] B. Panda and S. Balachandran, "Expert prefetch prediction: An expert predicting the usefulness of hardware prefetchers," *Computer Architecture Letters*, vol. 15, no. 1, pp. 13–16, 2016. [Online]. Available: https://doi.org/10.1109/LCA.2015.2428703

[42] S. H. Pugsley, Z. Chishti, C. Wilkerson, P. Chuang, R. L. Scott, A. Jaleel, S. Lu, K. Chow, and R. Balasubramonian, "Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers," in *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*, 2014, pp. 626–637. [Online]. Available: https://doi.org/10.1109/HPCA.2014.6835971

[43] V. Seshadri, S. Yedkar, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Mitigating prefetcher-caused pollution using informed caching policies for prefetched blocks," *TACO*, vol. 11, no. 4, pp. 51:1–51:22, 2014. [Online]. Available: https://doi.org/10.1145/2677956

[44] M. Shakerinavaet, M. Bakhshalipour, P. L. Kamran, and H. Sarbazi-Azad, "Multi-lookahead offset prefetching," in *3rd Data Prefetching Championship*, 2019.

[45] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*, 2015, pp. 141–152. [Online]. Available: https://doi.org/10.1145/2830772.2830793

[46] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv 1409.1556*, 09 2014.

[47] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *33rd International Symposium on Computer Architecture (ISCA'06)*, June 2006, pp. 252–263.

[48] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-temporal memory streaming," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 69–80. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555766

[49] S. Somogyi, T. F. Wenisch, M. Ferdman, and B. Falsafi, "Spatial memory streaming," *J. Instruction-Level Parallelism*, vol. 13, 2011. [Online]. Available: http://www.jilp.org/vol13/v13paper8.pdf

[50] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *13st International Conference on High-Performance Computer Architecture (HPCA-13 2007), 10-14 February 2007, Phoenix, Arizona, USA*, 2007, pp. 63–74. [Online]. Available: https://doi.org/10.1109/HPCA.2007.346185

[51] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy, "Power4 system microarchitecture," *IBM Journal of Research and Development*, vol. 46, no. 1, pp. 5–25, Jan 2002.

[52] J. Wang, R. Panda, and L. K. John, "Selsmap: A selective stride masking prefetching scheme," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 4, pp. 42:1–42:21, Oct. 2018. [Online]. Available: http://doi.acm.org/10.1145/3274650

[53] J. Wang, R. Panda, and L. K. John, "Prefetching for cloud workloads: An analysis based on address patterns," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2017, Santa Rosa, CA, USA, April 24-25, 2017*, 2017, pp. 163–172. [Online]. Available: https://doi.org/10.1109/ISPASS.2017.7975288

[54] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos, "Temporal streams in commercial server applications," in *2008 IEEE International Symposium on Workload Characterization*, Sep. 2008, pp. 99–108.

[55] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos, "Practical off-chip meta-data for temporal memory streaming," in *15th International Conference on High-Performance Computer Architecture (HPCA-15 2009), 14-18 February 2009, Raleigh, North Carolina, USA*, 2009, pp. 79–90. [Online]. Available: https://doi.org/10.1109/HPCA.2009.4798239

[56] C. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. S. Jr., and J. S. Emer, "Ship: signature-based hit predictor for high performance caching," in *44rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2011, Porto Alegre, Brazil, December 3-7, 2011*, 2011, pp. 430–441. [Online]. Available: https://doi.org/10.1145/2155620.2155671

[57] C. Wu, A. Jaleel, M. Martonosi, S. C. S. Jr., and J. S. Emer, "Pacman: prefetch-aware cache management for high performance caching," in *44rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2011, Porto Alegre, Brazil, December 3-7, 2011*, 2011, pp. 442–453. [Online]. Available: https://doi.org/10.1145/2155620.2155672

[58] H. Wu, K. Nathella, J. Pusdesris, D. Sunwoo, A. Jain, and C. Lin, "Temporal prefetching without the off-chip metadata," in *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: ACM, 2019, pp. 996–1008. [Online]. Available: http://doi.acm.org/10.1145/3352460.3358300

[59] H. Wu, K. Nathella, D. Sunwoo, A. Jain, and C. Lin, "Efficient metadata management for irregular data prefetching," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: ACM, 2019, pp. 449–461. [Online]. Available: http://doi.acm.org/10.1145/3307650.3322225

[60] V. Young, C.-C. Chou, A. Jaleel, and M. Qureshi, "Ship++ : Enhancing signature-based hit predictor for improved cache performance," in *2nd Cache Replacement Championship*, 2017.