Analysis and Optimization of the Memory Hierarchy for Graph Processing Workloads

Abanti Basak*, Shuangchen Li*, Xing Hu*, Sang Min Oh*, Xinfeng Xie*, Li Zhao†, Xiaowei Jiang†, Yuan Xie*
*University of California, Santa Barbara** *Alibaba, Inc.*†
{*abasak,shuangchenli,xinghu,sangminoh,xinfeng,yuanxie*}*@ucsb.edu**, {*li.z,xiaowei.jx*}*@alibaba-inc.com*†

***Abstract*—Graph processing is an important analysis technique for a wide range of big data applications. The ability to explicitly represent relationships between entities gives graph analytics a significant performance advantage over traditional relational databases. However, at the microarchitecture level, performance is bounded by the inefficiencies in the memory subsystem for single-machine in-memory graph analytics. This paper consists of two contributions in which we analyze and optimize the memory hierarchy for graph processing workloads.**

First, we perform an in-depth data-type-aware characterization of graph processing workloads on a simulated multi-core architecture. We analyze 1) the memory-level parallelism in an out-of-order core and 2) the request reuse distance in the cache hierarchy. We find that the load-load dependency chains involving different application data types form the primary bottleneck in achieving a high memory-level parallelism. We also observe that different graph data types exhibit heterogeneous reuse distances. As a result, the private L2 cache has negligible contribution to performance, whereas the shared L3 cache shows higher performance sensitivity.

Second, based on our profiling observations, we propose DROPLET, a Data-awaRe decOuPLed prEfeTcher for graph applications. DROPLET prefetches different graph data types differently according to their inherent reuse distances. In addition, DROPLET is physically decoupled to overcome the serialization due to the dependency chains between different data types. DROPLET achieves 19%-102% performance improvement over a no-prefetch baseline, 9%-74% performance improvement over a conventional stream prefetcher, 14%-74% performance improvement over a Variable Length Delta Prefetcher, and 19%-115% performance improvement over a delta correlation prefetcher implemented as a global history buffer. DROPLET performs 4%-12.5% better than a monolithic L1 prefetcher similar to the state-of-the-art prefetcher for graphs.

I. Introduction

Graph processing is widely used to solve big data problems in multiple domains such as social networks, web searches, recommender systems, fraud detection, financial money flows, and transportation. The high potential of graph processing is due to its rich, expressive, and widely applicable data representation consisting of a set of entities (vertices) connected to each other by relational links (edges). Numerous vendors such as Oracle [1], Amazon (AWS) [2], and Microsoft [3] provide graph processing engines for enterprises. Moreover, companies such as Google [4], Facebook [5], and Twitter [6], [7] have built customized graph processing frameworks to drive their products. Graph technology is also predicted to be the driver for many emerging data-driven markets, such as an expected $7 trillion worth market of self-driving cars by 2050 [8].

Prior work in graph analytics primarily spans hardware platforms such as distributed systems [4], [5], [9]–[11], out-of-core systems [12]–[16], and customized accelerators [17]–[21]. However, many common case industry and academic graphs have recently been reported to fit comfortably in the RAM of a single high-end server [22]–[24], given the availability and affordability of high memory density[1]. Consequently, single-machine in-memory CPU platform has become an attractive choice for many common-case graph analytics scenarios. As opposed to distributed systems, a scale-up big-memory system does not need to consider the challenging task of graph partitioning among nodes and avoids network communication overhead. Programming frameworks developed for this kind of platform in both academia [22], [26], [27] and industry [6], [7], [28] have shown excellent performance while significantly reducing the programming efforts compared to distributed systems. Both out-of-core systems and accelerators require expensive pre-processing to prepare partitioned data structures to improve locality. Single-machine in-memory graph analytics can avoid costly pre-processing which has been shown to often consume more time than the algorithm execution itself [29], [30].

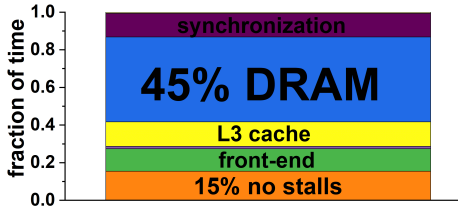


Figure 1: Cycle stack of PageRank on orkut dataset

However, the performance of single-machine in-memory graph analytics is bounded by the inefficiencies in the memory subsystem, which make the cores stall as they wait for data to be fetched from the DRAM. As shown in the cycle stack [31] in Fig. 1 for one of the benchmarks used

[1]A quad-socket Intel Xeon machine with 1.5TB RAM costs about $35K as of 2017 [25].

in our evaluation, 45% of the cycles are DRAM-bound stall cycles whereas the core is fully utilized without stalling in only 15% of the cycles.

The goal of this paper is to solve the memory inefficiency issue for single-machine in-memory graph analytics. We adopt a two-phase approach to achieve our goal. In the first phase, we develop an in-depth understanding of the memory-bound behavior observed in Fig. 1 by characterizing two features on a simulated multi-core architecture: (1) the memory-level parallelism (MLP) [32] in an out-of-order (OoO) core and (2) the request reuse distance in cache hierarchy. In the second phase, we use the findings from our characterization to propose an architecture design for an application-specific prefetcher.

In the first phase, we extend or fill the gaps in prior characterization work [33]–[37] in two aspects. First, we perform a data-aware profiling which provides clearer guidelines on the management of specific data types for performance optimization. Second, with the flexibility of a simulated platform, we vary the instruction window and cache configuration design parameters to explicitly explore their performance sensitivity. Beyond prior profiling work, our key findings include:

- Load-load dependency chains that involve specific application data types, rather than the instruction window size limitation, make up the key bottleneck in achieving a high MLP.
- Different graph data types exhibit heterogeneous reuse distances. The architectural consequences are (1) the private L2 cache shows negligible impact on improving system performance, (2) the shared L3 cache shows higher performance sensitivity, and (3) the graph property data type benefits the most from a larger shared L3 cache.

In the second phase, we use the guidelines from our characterization to design DROPLET, a Data-awaRe decOuPLed prEfeTcher for graphs. DROPLET is a physically decoupled but functionally cooperative prefetcher co-located at the L2 cache and at the memory controller (MC). We adopt a decoupled design to overcome the serialization due to the dependency between different graph data types. Moreover, DROPLET is data-aware because it prefetches different graph data types differently according to their intrinsic reuse distances. DROPLET achieves 19%-102% performance improvement over a no-prefetch baseline, 9%-74% performance improvement over a conventional stream prefetcher, 14%-74% performance improvement over a Variable Length Delta Prefetcher (VLDP) [38], and 19%-115% performance improvement over a delta correlation prefetcher implemented as a global history buffer (GHB) [39]. DROPLET performs 4%-12.5% better than a monolithic L1 prefetcher similar to the state-of-the-art prefetcher for graphs [40].

II. Background

In this section, we first discuss the graph data layout and the data type terminology used in the rest of the paper. Next, we briefly review the concepts of MLP, caches, and prefetchers, which are the three latency tolerance techniques addressed in this paper.

A. Graph Data Types and Data Layout

One of the most widely used graph data layouts is the Compressed Sparse Row (CSR) representation because of its efficient memory space usage. As shown in Fig. 2, the CSR format consists of three main components: the offset pointers, the neighbor IDs, and the vertex data. Each entry in the offset pointer array belongs to a unique vertex V and points to the start of the list of V's neighbors in the neighbor ID array. In the case of weighted graphs, each entry in the neighbor ID array also includes the weight of the corresponding edge. The vertex data array stores the property of each vertex and is indexed by the vertex ID. In the rest of the paper, we use the following terminology:

- **Structure data**: the neighbor ID array.
- **Property data**: the vertex data array.
- **Intermediate data**: any other data.

Property data is indirectly indexed using information from the structure data. Using the example in Fig. 2, to find the property of the neighbors of vertex 6, the structure data is first accessed to obtain the neighbor IDs (59 and 78), which in turn are used to index the property data.

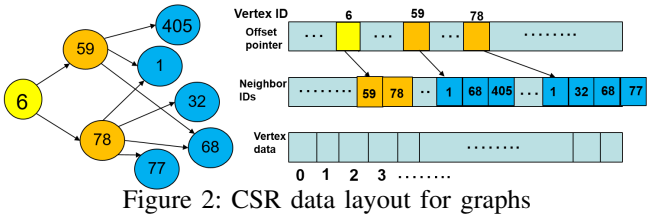


Figure 2: CSR data layout for graphs

B. Latency Tolerance in Modern CPUs

Three key latency tolerance techniques used in modern CPUs are MLP arising from OoO execution, on-chip caches, and prefetching. First, OoO execution relies on the reorder buffer (ROB) to look ahead down the instruction stream and it can support multiple in-flight memory requests with the help of load/store queues, non-blocking caches, and parallelism in the DRAM. For memory-intensive applications, a larger instruction window exposes more MLP and improves performance by overlapping the long latencies of multiple DRAM requests [41]. Second, caches reduce off-chip access latencies by locating data closer to the core to leverage spatial and temporal locality. A typical modern cache hierarchy consists of multiple levels of both private and shared caches, which makes up about 50% of the total chip area [42]. Third, hardware prefetchers located in the cache hierarchy monitor simple access patterns, such as

streams or strides with low overhead, and fetch data into the caches ahead of the demand access [43].

III. Experiment Setup

In this section, we present the experimental platform and the benchmark used for our characterization.

A. Profiling Platform

Profiling experiments have been done using SNIPER simulator, an x86 simulator based on the interval simulation model [44]. We selected SNIPER over other simulators because it has been validated against Intel Xeon X7460 Dunnington [44] and, with an enhanced core model, against Intel Xeon X5550 Nehalem [45]. Moreover, a recent study of x86 simulators simulating the Haswell microarchitecture has shown that SNIPER has the least error when validated against a real machine [46]. Cache access timings for different cache capacities were extracted using CACTI [47]. The baseline architecture is described in Table I. We used fewer cores than typically present in a server node because previous profiling work has shown that resource utilization for parallel and single-core executions are similar [33]. Hence, we do not expect the number of cores to change our observations. We marked the region of interest (ROI) in the application code. We ran the graph reading portion in cache warm-up mode and, upon entering the ROI, collected statistics for 600 million instructions across all the cores.

Table I: Baseline architecture

core	4 cores, ROB = 128-entry, load queue = 48-entry, store queue = 32-entry, reservation station entries = 36, dispatch width = issue width = commit width = 4, frequency = 2.66GHz
caches	3-level hierarchy, inclusive at all levels, writeback, least recently used (LRU) replacement policy, data and tags parallel access, 64B cacheline, separate L1 data and instruction caches
L1D/I cache	private, 32KB, 8-way set-associative, data access time = 4 cycles, tag access time = 1 cycle
L2 cache	private, 256KB, 8-way set-associative, data access time = 8 cycles, tag access time = 3 cycles
L3 cache (LLC)	shared, 8MB, 16-way set-associative, data access time = 30 cycles, tag access time = 10 cycles
DRAM	DDR3, device access latency = 45ns, queue delay modeled

B. Benchmark

We use the GAP benchmark [48] which consists of optimized multi-threaded C++ implementations of some of the most representative algorithms in graph analytics. For our profiling, we select GAP over a software framework to rule out any framework-related performance overheads and extract the true hardware bottlenecks[2]. We use five algorithms from GAP, which are summarized in Table II. A summary of the datasets is shown in Table III (size = unweighted/weighted).

[2]Previous study shows a 2-30X slowdown of software frameworks compared to hand-optimized implementations [49].

Table II: Algorithms

Algorithm	Description
Betweenness Centrality **(BC)**	Measure the centrality of a vertex, i.e., the number of shortest paths between any two other nodes passing through it
Breadth First Search **(BFS)**	Traverse a graph level by level
PageRank **(PR)**	Rank each vertex on the basis of the ranks of its neighbors
Single Source Shortest Path **(SSSP)**	Find the minimum cost path from a source vertex to all other vertices
Connected Components **(CC)**	Decompose the graph into a set of connected subgraphs

Table III: Datasets

Dataset	vertices	edges	Size	Description
kron [48]	16.8M	260M	2.1GB/2GB*	synthetic
urand [48]	8.4M	134M	1.1GB/2.1GB	synthetic
orkut [50]	3M	117M	941MB/1.8GB	social network
livejournal [50]	4.8M	68.5M	597MB/1.1GB	social network
road [48]	23.9M	57.7M	806MB/1.3GB	mesh network

* Weighted graph is smaller due to generation from a smaller degree for a manageable simulation time.

IV. Characterization

To understand the memory-bound behavior in Fig. 1, we analyze both the core and the cache hierarchy [51].

A. Analysis of the Core and the MLP

Observation#1: *Instruction window size is not the factor impeding MLP.* In general, a larger instruction window improves the hardware capability of utilizing more MLP for a memory-intensive application [41]. In addition, previous profiling work on a real machine concludes that the ROB size is the bottleneck in achieving a high MLP for graph analytics workloads [33]. However, by changing the design parameters in our simulator-based profiling, we observe that even a 4X larger instruction window fails to expose more MLP. As shown in Fig. 3a, for a 4X instruction window, the average increase in memory bandwidth utilization is only 2.7%. Fig. 3b shows the corresponding speedups. The average speedup is only 1.44%, which is very small for the large amount of allotted instruction window resources.

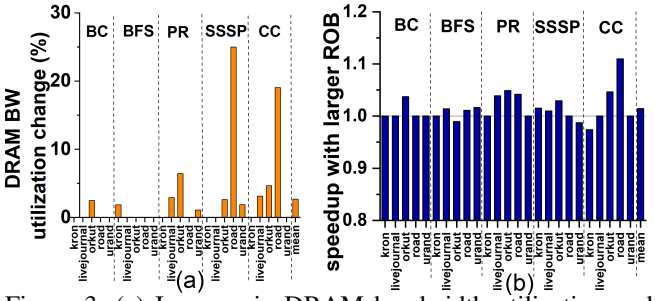


Figure 3: (a) Increase in DRAM bandwidth utilization and (b) overall speedup from a 4X larger ROB

Observation#2: *Load-load dependency chains prevent achieving high MLP.* To understand why a larger ROB does not improve MLP, we track the dependencies of the load instructions in the ROB and find that the MLP is bounded

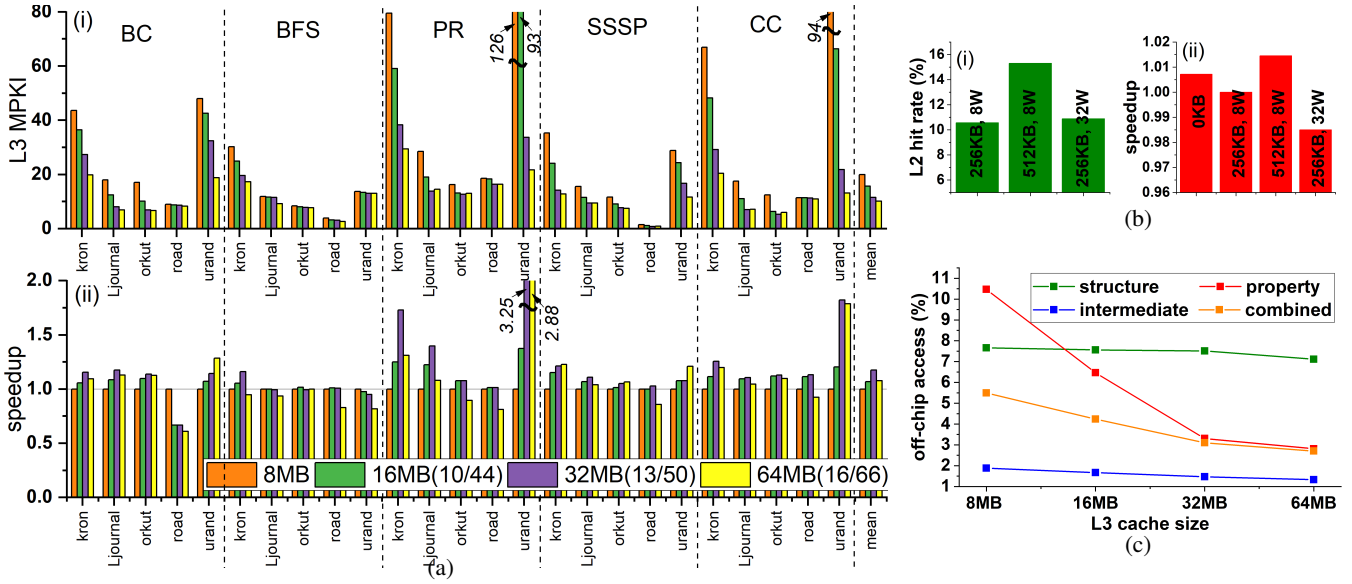


Figure 4: (a) Sensitivity of i) L3 MPKI and ii) system performance to shared L3 cache size ((X/Y)=access times for (tags/data) in cycles); (b) Sensitivity of (i) L2 cache hit rate and (ii) system performance to private L2 cache configurations (average across all benchmarks); (c) Effect of larger L3 cache on off-chip accesses of different data types (average across all benchmarks).

by an inherent application-level dependency characteristic. For every load, we track its dependency backward in the ROB until we reach an older load instruction. We call the older load a producer load and the younger load a consumer load. We find that short producer-consumer load dependency chains are inherent in graph processing and can be a serious bottleneck in achieving a high MLP even for a larger ROB. The two loads cannot be parallelized as they are constrained by true data dependencies and have to be executed in program order. Fig. 5 shows that, on average, 43.2% of the loads are part of a dependency chain with an average chain length of only 2.5, where we define chain length as the number of instructions in the dependency chain.

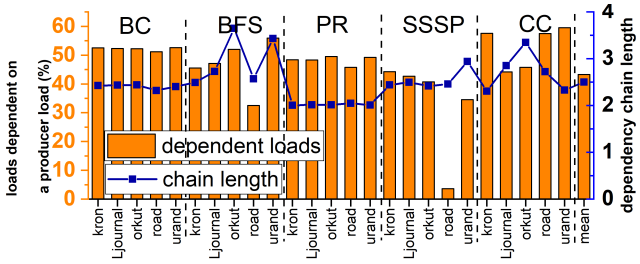


Figure 5: load-load dependency in ROB

Observation#3: *Graph property data is the consumer in a dependency chain.* To identify the position of each application data type in the observed load-load dependency chains, we show the breakdown of producer and consumer loads by data type in Fig. 6. On average, we find that the graph property data is mostly a consumer (53.6%) rather than a producer (5.9%). Issuing graph property data loads is delayed and cannot be parallelized because it has to depend on a producer load for its address calculation. Fig. 6 also

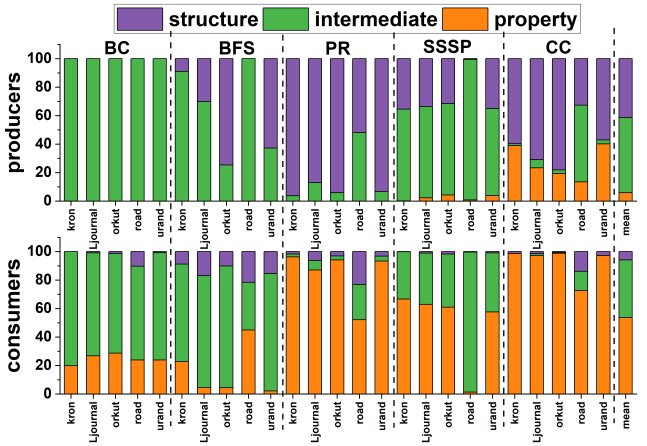


Figure 6: Breakdown of producer and consumer loads by application data type

shows that, on average, graph structure data is mostly a producer (41.4%) rather than a consumer (6%).

B. Analysis of the Cache Hierarchy

Observation#4: *The private L2 cache shows negligible performance sensitivity, whereas the shared LLC shows higher performance sensitivity.* As shown in Fig. 4a, we vary the LLC size from 8MB to 64MB and find the optimal point of 17.4% (max 3.25X) performance improvement for a 4X increase in the LLC capacity. The mean LLC MPKI (misses per kilo instructions) is reduced from 20 in the baseline to 16 (16MB) to 12 (32MB) to 10 (64MB). The corresponding speedups are 7%, 17.4%, and 7.6%. The optimal point is a balance between a reduced miss rate and a larger LLC access latency.

Table IV: Prefetch decisions based on profiling observations

Profiling Observation	Prefetch Design Question	Answer
Negligible impact of L2 (Observation#4).	In which cache level to put prefetched data?	L2 cache, because (1) no risk of cache pollution and (2) technique to make the under-utilized L2 cache useful.
Intermediate data are mostly cached, structure/property data are not (Observation#6).	What to prefetch?	Structure and property data.
• Structure data exhibits a large reuse distance with a pattern: a cacheline missed in the L1 cache is almost always serviced by the DRAM (Observation#6). • Property data exhibits a randomly large reuse distance which is larger than the L2 stack depth, leading to heavy LLC and DRAM accesses (Observation#6). • In load-load dependency chains, property data is mostly the consumer, whereas structure data is mostly the producer (Observation#3).	How to prefetch?	• Prefetch structure data from the DRAM in a streaming fashion. • Prefetch property data using explicit address calculation due to difficult-to-predict large reuse distance. • Let the calculation of target property prefetch addresses be guided by structure data. • Address calculation of target property prefetches is a serialized process. Decoupling the prefetcher will help break the serialization.
Load-load dependency chains are short (Observation#2).	When to prefetch?	If property prefetches are guided by structure *demand* data, they could be late since dependency chains are short. Hence, property prefetches should be guided by structure *prefetches*.

Fig. 4b(i) shows that the L2 hit rate (which is already very low at 10.6% in the baseline) increases to only 15.3% after a 2X increase in the capacity while a 4X increase in set associativity has no impact (hit rate rises to only 10.9%). Fig. 4b(ii) shows that the system performance exhibits little sensitivity to different L2 cache configurations (in both capacity and set associativity). The leftmost bar represents an architecture with no private L2 caches and no slowdown compared to a 256KB cache. Therefore, an architecture without private L2 caches is just as fine for graph processing.

Observation#5: *Property data is the primary beneficiary of LLC capacity.* To understand which data type benefits from a larger LLC, Fig. 4c shows, for each data type, the percentage of memory references that ends up getting data from the DRAM. We observe that the most reduction in the number of off-chip accesses comes from the property data. Structure and intermediate data benefit negligibly from a higher L3 capacity. Intermediate data is already accessed mostly in on-chip caches since only 1.9% of the accesses to this data type is DRAM-bound in the baseline. On the other hand, structure data has a higher percentage of off-chip accesses (7.5%), which remains mostly irresponsive to a larger LLC capacity.

Observation#6: *Graph structure cacheline has the largest reuse distance among all the data types. Graph property cacheline has a larger reuse distance than that serviced by the L2 cache.* To further understand the different performance sensitivities of the L2 and L3 caches, we break down the memory hierarchy usage by application data type as shown in Fig. 7. In most benchmarks, accesses to the structure data are serviced by the L1 cache and the DRAM, which indicates that a cacheline missed in L1 is one that was referenced in the distant past such that it has been

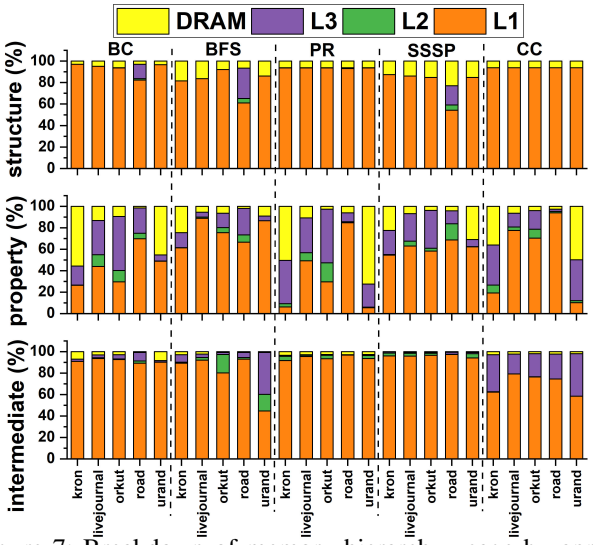


Figure 7: Breakdown of memory hierarchy usage by application data type

evicted from both the L2 and L3 caches. The fact that the reuse distance is beyond the servicing capability of the LLC explains why a larger LLC fails to significantly reduce the proportion of off-chip structure accesses in Fig. 4c. On the other hand, most of the property data loads missed in the L1 cache cannot be serviced by the L2 cache but can be serviced by the LLC and the DRAM. Overall, the LLC is more useful in servicing property accesses rather than structure accesses. Thus, the property cacheline has a comparatively smaller reuse distance that is still larger than that captured by the L2 cache. Finally, Fig. 7 provides evidence that the accesses to intermediate data are mostly on-chip cache hits in the L1 cache and the LLC. The reuse distances of the three data types explain why the private L2 cache fails to service

memory requests and shows negligible benefit.

C. Summary and Opportunities

The memory-bound stalling behavior in graph analytics arises due to two issues:

- Heterogeneous reuse distances of different data types leads to intensive DRAM accesses to retrieve structure and property data.
- Low MLP due to load-load dependency chains, limiting the possibility of overlapping DRAM accesses.

Our profiling motivates the adoption of prefetching as a latency tolerance technique for graph analytics. A good prefetcher can fix the first issue by locating data in on-chip caches ahead of the demand accesses. Prefetching can also bring an additional benefit by mitigating the effect of low MLP. A dependency chain with a consumer property data means serialization in the address calculation and a delay in issuing the property data load. However, once issued, prefetching ensures that the property data hits on-chip, mitigating low-MLP induced serially exposed latency. However, due to heterogeneous reuse distances of the different graph data types, it is challenging to employ simple hardware prefetchers that can efficiently prefetch all data types. To address this challenge, we leverage our observations from the profiling to make prefetch decisions for DROPLET architecture, as summarized in Table IV.

V. DROPLET ARCHITECTURE

In this section, we first provide an overview of our proposed prefetcher. Next, we discuss the detailed architecture design of the components of DROPLET.

A. Overview and Design Choice

We propose a physically decoupled but functionally cooperative prefetcher co-located at the L2 cache level and at the MC. Specifically, DROPLET consists of two data-aware components: the L2 streamer for prefetching structure data (Fig. 8 ❶) and the memory controller based property prefetcher (MPP) (Fig. 8 ❷) which is guided by the structure stream requests from the L2 streamer. Fig. 8 shows the entire prefetching flow. The blue path shows the flow of prefetching the structure data using the data-aware L2 streamer. The stream prefetcher is triggered only by structure data and the structure prefetch request is almost always guaranteed to go to the DRAM as observed in our profiling. On the refill path from the DRAM, the prefetched structure cacheline is transferred all the way to L2. Also, a copy of the structure data is forwarded to the MPP, which triggers the property data prefetching. The green path shows the prefetching flow of the property data. The MPP reacts to the prefetched structure cacheline by scanning it for the neighbor node IDs, computing the virtual addresses of the target property prefetches using those neighbor IDs, and performing virtual-physical address translation. To avoid unnecessary DRAM accesses for property data that may already be on-chip, the physical prefetch address is used to check the coherence engine. If not on-chip, the property prefetch address is queued in the MC. Upon being serviced by the DRAM, the cacheline is sent to the LLC and the private L2 cache. Instead, if the to-be-prefetched property cacheline is detected to be on-chip, it is further copied from the inclusive LLC into the private L2 cache.

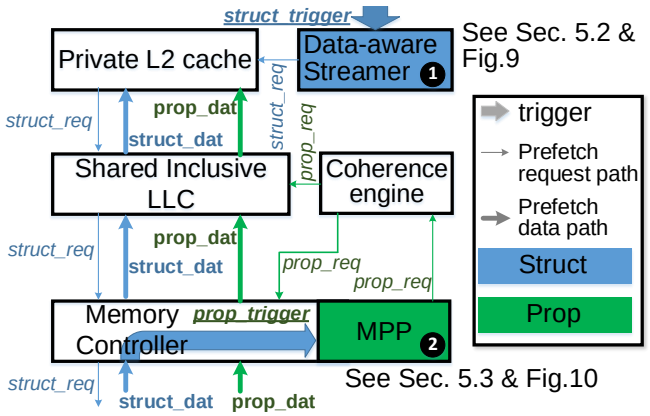


Figure 8: Overview of DROPLET

Our design choices are based on our profiling observations as summarized in Table IV. The L2 cache is the location in which the L2 streamer brings in structure prefetches and the MPP sends property prefetches. Despite this fact, we design a decoupled property prefetcher in the MC in order to break the serialization arising from consumer property data in a dependency chain. A single prefetcher at the L2 cache would have to wait until the structure prefetch returns to the L2 cache before the property address can be calculated and issued for prefetching. By decoupling the prefetcher, the property address can be calculated as soon as the producer structure prefetch arrives on the chip at the MC. This overlaps the return of the structure prefetch on the refill path through the caches and the issuing of the subsequent property prefetch, breaking the serialization. A previous work, which *dynamically offloads* and *accelerates* dependent load chains in the MC, shows a 20% lower latency when a dependent load is issued from the MC rather than from the core side [52]. We use this insight in our *prefetching* scheme to achieve better timeliness by aggressively issuing property prefetch requests directly from the MC.

B. L2 Streamer for Structure Data

First, we discuss the difficulties posed by a conventional streamer for graph workloads. Next, to address these challenges, we propose the design of a data-aware L2 streamer.

*1) **Shortcomings of a Conventional Streamer**:* A conventional L2 streamer, as shown in Fig. 9(a), can snoop all cacheblock addresses missed in the L1 cache and can track multiple streams, each with its own tracker (tracking address) [53]. Once a tracker has been allocated, the stream

requires two additional miss addresses to confirm a stream before starting to prefetch [53]. In graph processing, such a streamer is detrimental to both structure and property data in terms of prefetch accuracy and coverage. First, due to the streamer's ability to snoop all L1 miss addresses, many trackers in the streamer get wastefully assigned to track pages corresponding to the property and the intermediate data types. These trackers are wasteful due to one of the two reasons: (1) often, a stream is not successfully detected due to large reuse distances, which results in small property and intermediate prefetch volumes and low coverage; (2) due to random accesses, these trackers get the wrong signals, i.e., random streams are detected, leading to mostly useless prefetches (low prefetch accuracy). The direct consequence of property and intermediate data trackers is the reduction in the effective number of trackers available for structure data (the data type which actually shows stream behavior) at any given time. This reduces the volume of structure prefetches and the coverage for structure data.

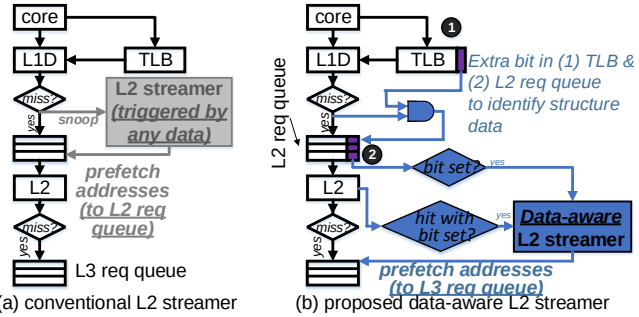


Figure 9: L2 streamer for prefetching structure data. Extra bits in purple and newly added cache controller decision-making in blue.

*2) **Data-aware Streamer in DROPLET**:* To address the problems of a conventional streamer, our proposed streamer is data-aware, i.e., it operates only on structure data to bring in a high volume of accurate structure prefetches. Fig. 9(b) shows how this is achieved. We use a variant of *malloc* to allocate graph structure data (see Section VI), which updates page table entries with an extra bit ("1"=structure data). Upon address translation at the L1D cache, this extra bit in the TLB entry (Fig. 9(b) ❶) is copied to the L1D cache controller. When an access misses in the L1D cache, the L1D controller buffers this extra bit, together with the miss address, into the L2 request queue. Hence, the L2 request queue is augmented with an additional bit (Fig. 9(b) ❷) to identify whether a request address corresponds to the structure data. The L2 cache controller in our design preferentially guides into the streamer only the addresses in the request queue which belong to structure data (as recognized by the extra bit being "1"). Another input to the streamer is a feedback from the L2 cache. When a L2 hit occurs to an address corresponding to structure data, the address is sent to the streamer. Finally, the L2 streamer buffers its target prefetch addresses in the L3 request queue and stores the prefetched data in the L2 cache.

In contrast to a conventional streamer, as shown in Fig. 9(a), our design contains two fundamental modifications guided by our observations in Section IV. First, our data-aware L2 streamer is triggered only by *structure data* to overcome the shortcomings of a conventional streamer mentioned in Section V-B1. Second, as opposed to a conventional L2 streamer that buffers the target prefetch addresses in the L2 request queue, we buffer them in the L3 request queue. This design choice is guided by our observation that new structure cachelines are serviced by a lower level in the memory hierarchy. Later in Section VII, we quantitatively analyze the superiority of our data-aware structure streamer over the conventional design in terms of both performance and bandwidth savings.

C. Memory Controller (MC)

First, the memory request buffer (MRB) is used to give the MC the knowledge of (1) whether a cacheline coming from the DRAM corresponds to a structure prefetch and (2) which core sent the structure prefetch request. Second, we incorporate the MPP unit.

*1) **Memory Request Buffer (MRB)**:* When cachelines are on their refill path from the DRAM, the MC needs to filter the ones corresponding to structure prefetches to transfer their copies to the MPP for property prefetching. To enable this, we reinterpret the *C*-bit (criticality bit) in the baseline MRB, as shown in Fig. 10 (in the lower left corner), which differentiates a prefetch request from a demand request for priority-based scheduling purposes in modern MCs [54]. If the *C*-bit is set, in addition to indicating a prefetch request, it specifically indicates a *structure* prefetch since it comes from the L2 streamer, which only sends structure prefetch requests. We also add a core ID field to give the MPP the knowledge of which core sent the structure prefetch request so that it can later send the property prefetches into that core's private L2 cache.

*2) **MC based Property Prefetcher (MPP)**:* As shown in Fig. 10, the MPP consists of a property address generator (PAG), a FIFO virtual address buffer (VAB), a near-memory TLB (MTLB), and a FIFO physical address buffer (PAB). If a cacheline from the DRAM is detected to be a structure prefetch (see Section V-C1), its copy is transferred to the MPP.

The prefetched structure cacheline is propagated to the PAG which is shown in detail in Fig. 10. The PAG scans the cacheline in parallel for neighbor IDs that are indices to the property data array. To calculate the target virtual address of the property prefetch from a scanned neighbor ID, we use the following equation:

$$\textbf{property address} = \textbf{base} + \textbf{4} \times \textbf{neighbor ID} \quad (1)$$

where *base* is the base virtual address of the property array and the granularity of the property data is 4B. The

PAG receives two pieces of information from the software (see Section VI): (1) the *base* in Equation 1 and (2) the granularity at which the structure cacheline needs to be scanned (4B for unweighted graphs and 8B for weighted graphs). One structure cacheline can generate 8 and 16 property addresses per cycle for weighted and unweighted graphs, respectively.

The virtual addresses generated by the PAG, together with the corresponding core ID, are buffered in the VAB for translation. The translation happens through a small MTLB (see Section V-C3), upon which the resulting physical address is buffered in the PAB. The physical address is then used to check the coherence engine and determine the subsequent prefetch path of the property data, as described in Section V-A.

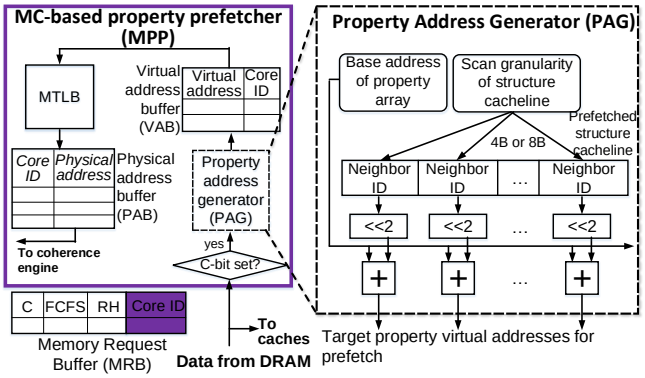


Figure 10: MC-based Property Prefetcher (MPP)

*3) **Virtual Address Translation in MTLB**:* The MTLB caches mappings of the property data pages to allow virtual-physical translation of the property prefetch addresses in the MPP. A MTLB miss is handled by page-walking, but a property prefetch address encountering a page fault is simply dropped. Like core-side TLBs, the MTLB should also be involved in the TLB shootdown process so that the MPP does not continue using stale mappings to prefetch wrong data. To optimize the coherency maintenance between the core-side TLBs and the MTLB, and to reduce the number of invalidations to the MTLB, we leverage (1) the extra bit in the TLBs used to differentiate between structure and non-structure data and (2) the fact that the MTLB caches only property mappings. Due to these two features, during a TLB shootdown, MTLB entries are invalidated using only the core-side TLB invalidations for entries that have an extra bit with the value "0" (indicating non-structure data).

D. Hardware Overhead

DROPLET incurs negligible area overhead. The data-aware streamer in DROPLET requires storing extra information in the page table and the L2 request queue entries. In the hierarchical paging scheme in x86-64, each paging structure consists of 512 64-bit entries, leading to a 4KB storage [55]. The addition of an extra bit in each page table entry to identify structure data results in a 64B (1.56%) storage overhead in the paging structure. Assuming a 32-entry L2 request queue [56], with each entry containing the miss address and the status [57], the addition of an extra bit results in a 4B (1.54%) queue storage overhead. In addition, property prefetching in DROPLET introduces the MPP and requires additional storage in the MRB. Using McPAT integrated with SNIPER, we find the area of the baseline chip to be 188 mm^2 in a 45nm technology node. The area of the MPP, with its parameters shown in Table V, is 0.0654 mm^2, resulting in a 0.0348% area overhead with respect to the entire chip. With a 7.7KB storage, the VAB, the PAB, and the MTLB comprise 95.5% of the total MPP area. For the MRB, the additional storage required for the core ID field for our quad-core system is only 64B, if assuming a 256-entry MRB.

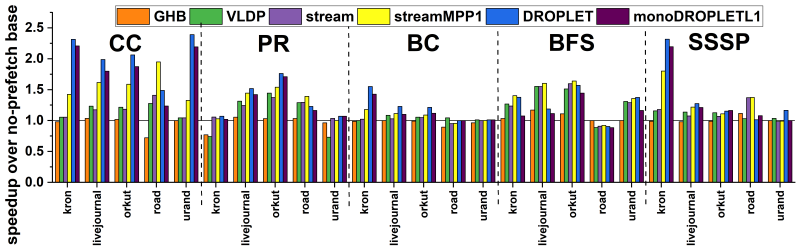
VI. Discussion

System support for address identification. DROPLET is data-aware in two aspects. First, the L2 streamer is only triggered by structure data. Second, the MPP prefetches only property data with the knowledge of the base address of the property array and the scan granularity of the structure cacheline. To obtain this information, DROPLET needs help from the operating system and the graph data allocation layer of graph processing frameworks. Similar to prior work [58], [59], we develop a specialized *malloc* function which can label all allocated pages with an extra bit in the page table to help identify structure data during address translation at the L1D cache (the extra bit being "1" indicates structure data). The specialized *malloc* function can also pass in some information to be written into the hardware registers in the MPP. When allocating the structure data, the *malloc* function writes the scan granularity of the structure cacheline in the MPP. The *malloc* function for the property data writes the base address of the property array in the MPP. We add a special store instruction to provide an interface for the processor to write these two registers in the MPP [3].

User Interface. DROPLET requires the above *malloc* modification only at the framework level and is transparent to user source code. This is because, in the system stack for graph processing frameworks, the layer of graph data allocation and management (the layer at which we introduce a specialized *malloc*) is separate from user code [59]. Hence, our modification can be handled "behind the scenes" by the data management layer without modifying the API for users.

Applicability to different programming models and data layouts. In this work, we assume a CSR data layout and a programming abstraction focused on the vertex (see Section II-A). DROPLET is easily extensible to other abstractions because a common aspect across most abstractions is that they maintain data in two distinct parts which can be mapped to our data type terminology (structure and property

[3]The configuration registers of the MPP are part of the context and are backed up and restored when switching between graph analytics processes.



(a)

Scheme	CC	PR	BC	BFS	SSSP
GHB	0.94	0.96	0.97	1.06	1.01
VLDP	1.16	1.06	1.04	1.28	1.1
stream	1.16	1.19	1	1.29	1.13
streamMPP1	1.57	1.26	1.06	**1.36**	1.27
DROPLET	**2.02**	**1.30**	**1.19**	1.26	**1.32**
monoDROPLETL1	1.82	1.25	1.12	1.12	1.27

(b)

Figure 11: (a) Performance improvement of the six configurations; (b) Performance summary for Fig. 11a (each entry is the geomean of the speedups across all the five datasets).

data). For example, in edge-centric graph processing [12], [29], structure data would be equivalent to a sorted or unsorted edge array which is typically streamed in from slower to faster memory [12], [29]. DROPLET can prefetch these edge streams and use them to trigger a MPP located near the slower memory to prefetch property data. Moreover, DROPLET can easily handle multi-property graphs by using the information from prefetched structure data to index one property array (in case the property data layout is an array of vectors) or multiple property arrays (in the case of separate arrays for each property).

Applicability to larger graphs. DROPLET is based on the observations obtained from profiling graph processing workloads on a simulated multi-core architecture. As opposed to a real machine, a simulated platform provides much higher flexibility for data-aware profiling and for studying the explicit performance sensitivities of the instruction window and the cache hierarchy. However, the trade-off is the restriction to somewhat small datasets to achieve manageable simulation times. However, our profiling conclusions (hence the effectiveness of our proposed DROPLET design) still hold for larger graphs because of two aspects. First, we use small datasets that are still larger than that fully captured by the on-chip caches, which stresses the memory subsystem sufficiently. Second, we explain the observed architecture bottlenecks in terms of inherent data type features such as dependency characteristics and reuse distances, which are independent of the data size.

Multiple MCs. Our experimental platform consists of a single MC in a quad-core system. However, platforms with more cores may contain multiple MCs and data may be interleaved across DRAM channels connected to different MCs. It is possible that a property prefetch address generated by the MPP of one MC may actually be located in a DRAM channel connected to another MC. In such a case, we adopt a technique similar to prior work [52]. The property prefetch request, together with the core ID, is directly sent to the MRB of the destination MC for prefetching.

VII. DROPLET Evaluation

In this section, we discuss the experimental setup and the evaluation results for DROPLET.

A. Prefetch Experiment Setup

Beyond Table I, the features of the prefetchers used for evaluation are shown in Table V. Evaluation is performed for six prefetcher configurations listed below.

GHB [39]: This is a G/DC (Global/Delta Correlation) L2 prefetcher.

VLDP [38]: This is a L2 prefetcher which, unlike GHB, can use longer and variable delta histories to make more accurate prefetch predictions.

stream: This is the conventional L2 streamer which can snoop all L1 miss addresses.

streamMPP1: This is a conventional L2 streamer together with a MPP. This configuration shows the benefit of letting property prefetching be guided by structure streaming. However, since the L2 streamer is not data-aware, relying only on the *C*-bit in the MRB to identify structure data does not work. So, we use MPP1 which is equivalent to MPP equipped with the ability to recognize structure data.

DROPLET: Unlike stream and streamMPP1, the L2 streamer is structure-only. When compared to streamMPP1, DROPLET shows the additional benefit by restricting the streamer to work on structure data only.

monoDROPLETL1: This prefetcher is a data-aware streamer + MPP1[4] implemented monolithically at the L1 cache, an arrangement close to the state-of-the-art graph prefetching proposition of Ainsworth et al. [40]. Although [40] uses a worklist-triggered recursive prefetcher instead of a data-aware streamer + MPP1, monoDROPLETL1 imitates their design philosophies of 1) a monolithic prefetcher and 2) bringing all prefetches into the L1 cache. Hence, we select this design point to evaluate if DROPLET provides any performance benefit over a [40]-like **monolithic L1** prefetcher.

B. Performance Results

Fig. 11a and 11b show the performance improvement of the six configurations normalized to a no-prefetch baseline. Among all of them, DROPLET provides the best performance for CC (102%), PR (30%), BC (19%), and SSSP (32%). In these four algorithms, the only exception is the *road* dataset (CC-*road*, PR-*road*, and SSSP-*road*) where streamMPP1 is the best performer (see Section VII-C1). For

[4]The MPP1 in monoDROPLETL1 can reuse the core-side TLB and does not require a MTLB.

Table V: Prefetchers for evaluation

L2 GHB	index table size = 512, buffer size = 512
L2 VLDP	implemented using code from [56], last 64 pages tracked by DHB, 64-entry OPT, 3 cascaded 64-entry DPTs
L2 streamer	implemented as described in section 2.1 of [53], prefetch distance=16, number of streams=64, stops at page boundary
MPP	address generation latency in PAG = 2 cycles, 512-entry VAB and PAB, 128-entry MTLB, 2 64-bit registers, coherence engine checking overhead=10 cycles
MPP1	MPP augmented with the ability to identify a prefetched structure cacheline

BFS, DROPLET provides a 26% performance improvement, but the best prefetcher is streamMPP1 with an average improvement of 36% (see Section VII-C1). DROPLET could easily be extended to adaptively turn off the streamer's data-awareness to convert it into the streamMPP1 design. In that case, our design would be no worse than streamMPP1 for BFS and the *road* dataset.

Compared to a conventional stream prefetcher, DROPLET provides performance improvements of 74% for CC, 9% for PR, 19% for BC, and 16.8% for SSSP. These improvements come from both the MPP and the data-aware streamer, as shown by the progressive speedup from stream to DROPLET. For BFS, streamMPP1 outperforms the conventional streamer by 5.4% but DROPLET slightly degrades performance by 2.3%.

Compared to monoDROPLETL1 (close to [40]), DROPLET provides performance improvements of 11% for CC, 4% for PR, 6% for BC, 12.5% for BFS, and 4% for SSSP. DROPLET differs from monoDROPLETL1 by 1) adopting a physically decoupled design and 2) prefetching into the L2 instead of the L1. DROPLET performs better because it achieves better prefetch timeliness by decoupling the structure and property prefetching (Section V-A). The benefit of decoupling is more so because computation per memory access is very low in graph processing. In addition, DROPLET does not pollute the more useful L1 cache. Instead, it prefetches into the L2 cache which is poorly utilized in graph processing (Section IV-B). In addition to the performance benefit, DROPLET offers two critical advantages over [40] in terms of practicality and simplicity. First, [40] triggers its prefetcher with an *implementation specific* data structure called the worklist. However, many all-active algorithms are the simplest to implement without having to maintain a worklist [48]. For [40], these algorithms have to re-written to integrate a worklist data structure. In contrast, DROPLET does not depend on any worklist, leverages the intrinsic reuse distance feature of graph structure data to trigger the prefetcher, and introduces a special *malloc* which is transparent to user code (Section VI). Second, unlike [40], we do not require additional hardware to ensure prefetch timeliness. We achieve timeliness by using a decoupled design to break the serialization between structure and property data.

DROPLET outperforms G/DC GHB by 115% for CC, 35% for PR, 23% for BC, 19% for BFS, and 31% for SSSP. GHB is overall the least performing prefetcher in our evaluation. This is because it is hard to identify consistent correlation patterns in graph processing due to the heterogeneous reuse distances of different data types.

Compared to VLDP, DROPLET provides performance improvements of 74% for CC, 23% for PR, 14% for BC, and 20% for SSSP. For BFS, streamMPP1 outperforms VLDP by 6% but DROPLET slightly degrades performance by 1.6%. On average, the performance improvement of DROPLET compared to VLDP is similar to its improvement compared to the conventional stream prefetcher. Due to complex reuse distances of different data types, delta histories may not always make effective prefetch predictions.

C. Explaining DROPLET's Performance

In this section, we zoom in on DROPLET to demystify its performance benefits. Since DROPLET is an enhancement upon the L2 streamer to include a data-aware streamer and a MPP, we compare to stream and streamMPP1 configurations to show the additional effect of each component.

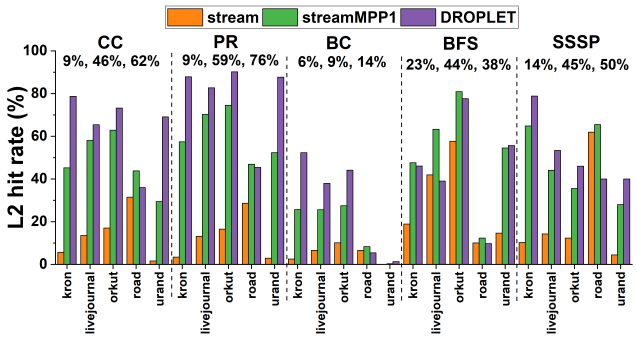


Figure 12: L2 cache performance. The set of numbers shows the average for the three prefetch configurations across the five datasets.

1) Benefit to Demand Accesses: **L2 cache performance:** Fig. 12 shows that DROPLET converts the seriously under-utilized L2 cache (Fig. 4b) into a higher performance resource by increasing the L2 hit rate to 62%, 76%, 14%, 38%, and 50% for CC, PR, BC[5], BFS, and SSSP, respectively. The L2 hit rate for DROPLET is the highest on average for CC, PR, BC, and SSSP. Fig. 12 also explains why streamMPP1 is the ideal prefetcher for the *road* dataset and most BFS benchmarks. For these benchmarks, the conventional streamer provides a comparatively higher cache performance, indicating that the streamer may also be effective at capturing some property prefetches. Hence, by making the streamer structure-only in DROPLET, we lose the streamer-induced

[5]The value is relatively lower due to BC-*road* and BC-*urand* exceptions which do not benefit highly from any prefetcher configuration. However, DROPLET causes no slowdown for them (Fig. 11a).

property prefetches, leading to a lower L2 cache hit rate when using DROPLET over streamMPP1.

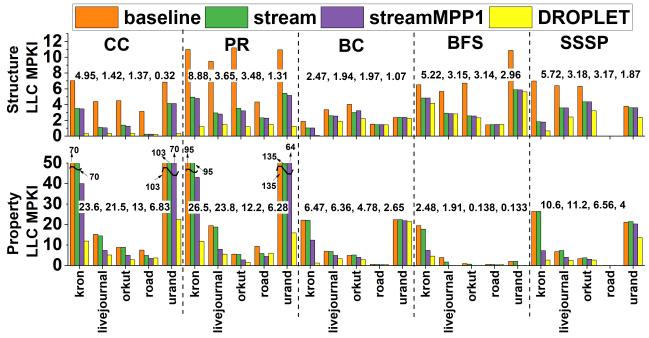


Figure 13: Off-chip demand accesses

Off-chip demand accesses: To understand how the two components of DROPLET (MPP and data-aware streamer) affect the off-chip demand accesses to structure and property data, Fig. 13 breaks down the *demand* MPKI from the LLC by data type. Below, we summarize our observations on the additive benefit of each prefetch configuration:

(1) Compared to the no-prefetch baseline, stream reduces structure demand MPKI in all cases (71.3%, 58.9%, 21.5%, 39.7%, and 44.4% for CC, PR, BC, BFS, and SSSP, respectively), but it mostly fails in significantly reducing the property demand MPKI because of its inability to capture such difficult-to-predict large reuse distances. The only exception is BFS, in which the property MPKI is reduced by 23%, corroborating the earlier observation that stream is comparatively better at capturing property prefetches for most BFS benchmarks.

(2) Compared to stream, streamMPP1 significantly reduces the property MPKI for all algorithms (39.5%, 48.7%, 24.8%, 92.8%, and 41.4% for CC, PR, BC, BFS, and SSSP, respectively), but it does not significantly reduce the structure MPKI. The property MPKI reduction comes from the MPP, which follows structure prefetches to bring in a high volume of useful property prefetches. Structure MPKI does not benefit because the streamers in stream and streamMPP1 are the same.

(3) Compared to streamMPP1, DROPLET further reduces structure demand MPKI (76.6%, 62.4%, 45.7%, 5.73%, and 41% for CC, PR, BC, BFS, and SSSP, respectively) because a structure-only streamer leads to a larger volume of structure prefetches by dedicating all the trackers to structure memory regions. Correspondingly, the property MPKI is also reduced since property prefetches accompany the structure prefetches (47.5%, 48.5%, 44.6%, 3.6%, and 39% for CC, PR, BC, BFS, and SSSP, respectively). However, we observe that the benefit for BFS is small compared to other algorithms. Hence, the small reduction in the LLC MPKI, together with a lower L2 cache performance, further explains why streamMPP1 rather than DROPLET is the ideal prefetcher for the BFS algorithm.

Prefetch accuracy: Fig. 14 shows the prefetch accuracies

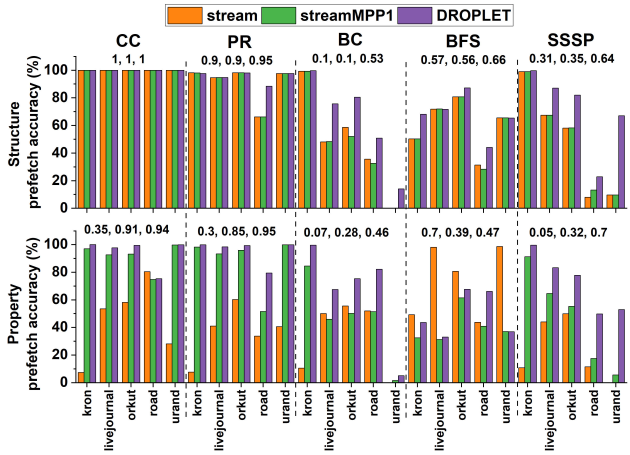


Figure 14: Prefetch accuracy

for the three prefetch configurations and the two data types. For structure data, DROPLET has the highest prefetch accuracy for all the algorithms. The accuracies are 100% for CC, 95% for PR, 53% for BC, 66% for BFS, and 64% for SSSP. For property data, DROPLET's prefetch accuracy is the highest for CC (94%), PR (95%), BC (46%), and SSSP (70%). For BFS, the property prefetch accuracy for DROPLET (47%) is lower than that of stream (70%). Overall, CC and PR have a higher prefetch accuracy for structure and property data than BC, BFS, and SSSP. This is because the former two algorithms process vertices in very sequential order. On the other hand, the BC, BFS, and SSSP algorithms depend on intermediate data structures, such as bins or worklists, when processing vertices. Consequently, the access pattern for structure data in BC, BFS, and SSSP consists of random starting points from which data can start being streamed.

2) Overheads of Prefetching: One of the side effects of prefetching is additional bandwidth consumption due to inaccurate prefetches, which may offset performance gains. DROPLET incurs low extra bandwidth consumption. Fig. 15 shows the extra bandwidth consumption for the three configurations measured in BPKI (bus accesses per kilo instructions). Compared to a no-prefetch baseline, DROPLET requires 6.5%, 7%, 11.3%, 19.9%, and 15.1% additional bandwidth for CC, PR, BC, BFS, and SSSP, respectively. The bandwidth requirement for CC and PR is smaller due to the comparatively higher structure and property prefetch accuracies.

VIII. Related Work

Graph characterization. Prior characterization on real hardware [33]–[37] provides insights such as high cache miss rates [35], under-utilization of memory bandwidth [33], [34], and limited instruction window size [33]. Our profiling fills in the gaps in previous work through its data-aware feature and an explicit performance sensitivity analysis of the instruction window and the caches.

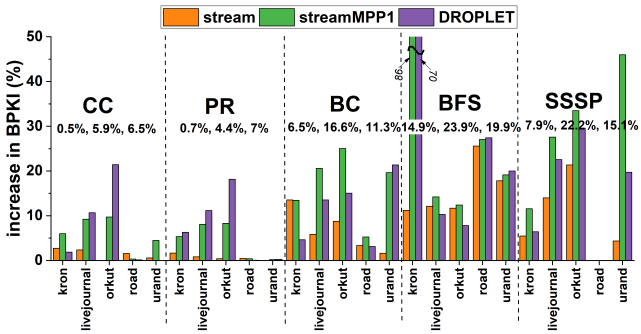


Figure 15: Additional off-chip bandwidth consumption

Prefetching. Although prefetching is a well-studied topic [38], [39], [60]–[69], our work shows that many state-of-the-art prefetchers are not adequate for graph processing (Section VII-B). Our contributions lie in the in-depth analysis of the data type behavior in graph analytics and in the observation-oriented prefetching design decisions (data-aware and decoupled prefetcher).

IMP [70] is a hardware-only L1 prefetcher for indirect memory references. Unlike IMP, we use the system support for data awareness to avoid long and complex prefetcher training. A data-aware prefetcher removes the need for IMP's strict and not-widely-applicable assumption of very long streams to be eventually able to train the prefetcher. In addition, we show that, for graph processing, a decoupled architecture provides performance benefits over a monolithic prefetcher (the design adopted by IMP) (Section VII-B).

Many prefetchers have been proposed for linked-list type data structures [60]–[63]. However, they prefetch at multiple levels of recursion (over-prefetch), whereas the structure-to-property indirect array access in graph analytics represents only one level of dependency. Runahead execution [66] uses a stalled ROB to execute ahead in the program path but can only prefetch independent cache misses. Dependence graph precomputation [67] is not data-aware and it requires large hardware resources to dynamically identify dependent load chains for precomputation.

Near-memory prefetchers and accelerators. There exist near-memory prefetchers that perform next-line prefetching from the DRAM row buffer [71] or prefetch linked-list data structures [61], [72]. To the best of our knowledge, our work is the first to propose a data-aware prefetcher in the MC for indirect array accesses in graph analytics. In addition, our near-memory MPP is significantly different from previous approaches [61], [71], [72] in how it is guided by a data-aware L2 streamer.

Hashemi et al. [52] propose, for SPEC CPU2006 benchmarks, dynamically offloading dependent chains predicted to be cache misses to the MC for *acceleration*. However, in graph analytics, such a scheme could lead to high overheads from very frequent offloading since cache miss rates are much higher. Instead, we use the concept of issuing requests directly from the MC to decouple our prefetcher and to achieve aggressive and timely property *prefetching* (Section V-A). Note that acceleration and prefetching are two orthogonal techniques, and can be combined to gain benefit from both.

IX. Conclusion

This paper analyzes and optimizes the memory hierarchy for single-machine in-memory graph analytics. First, we characterize the MLP and the cache hierarchy and find that 1) load-load dependency chains, with consumer graph property data, form the bottleneck in achieving a high MLP and 2) the L2 cache has little impact on the system performance, whereas the LLC exhibits a higher performance sensitivity. Next, using the profiling insights, we propose the architecture design for an application-specific prefetcher called DROPLET. DROPLET is data-aware and prefetches graph structure and property data in a physically decoupled but functionally cooperative manner. DROPLET achieves 19%-102% performance improvement over a no-prefetch baseline, 9%-74% performance improvement over a conventional stream prefetcher, 14%-74% performance improvement over VLDP, and 19%-115% performance improvement over a G/DC GHB. DROPLET performs 4%-12.5% better than a monolithic L1 prefetcher similar to the state-of-the-art prefetcher for graphs.

X. Acknowledgments

We thank Onur Mutlu, Bowen Huang, Yungang Bao, and the anonymous reviewers for their valuable feedback. This work was supported in part by NSF 1730309, 1719160, 1500848 and by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation program sponsored by DARPA.

References

[1] "Spatial and graph anaytics with oracle database 18c," tech. rep., Oracle, 2018.

[2] R. Dillet, "Amazon introduces an aws graph database service called amazon neptune." https://techcrunch.com/2017/11/29/amazon-introduces-an-aws-graph-database-service-called-amazon-neptune/, 2017.

[3] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *International Conference on Management of Data*, pp. 505–516, ACM, 2013.

[4] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *SIGMOD*, pp. 135–146, ACM, 2010.

[5] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at facebook-scale," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1804–1815, 2015.

[6] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh, "Wtf: The who to follow service at twitter," in *International conference on World Wide Web*, pp. 505–514, ACM, 2013.

[7] A. Sharma, J. Jiang, P. Bommannavar, B. Larson, and J. Lin, "Graphjet: real-time content recommendations at twitter," *Proceedings of the VLDB Endowment*, vol. 9, no. 13, pp. 1281–1292, 2016.

[8] N. Ismail, "Graph databases lie at the heart of $7tn self-driving car opportunity." http://www.information-age.com/graph-databases-heart-self-driving-car-opportunity-123468309/, Nov 2017.

[9] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs.," in *OSDI*, vol. 12, p. 2, ACM, 2012.

[10] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: a framework for machine learning and data mining in the cloud," *VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[11] V. Kalavri, V. Vlassov, and S. Haridi, "High-level programming abstractions for distributed graph processing," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 2, pp. 305–324, 2018.

[12] A. Kyrola, G. E. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc," USENIX, 2012.

[13] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu, "Turbograph: a fast parallel graph engine handling billion-scale graphs in a single pc," in *International conference on Knowledge discovery and data mining*, pp. 77–85, ACM, 2013.

[14] X. Zhu, W. Han, and W. Chen, "Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning.," in *USENIX Annual Technical Conference*, pp. 375–386, 2015.

[15] M. Zhang, Y. Wu, Y. Zhuo, X. Qian, C. Huan, and K. Chen, "Wonderland: A novel abstraction-based out-of-core graph processing system," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 608–621, ACM, 2018.

[16] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim, "Mosaic: Processing a trillion-edge graph on a single machine," in *European Conference on Computer Systems*, pp. 527–543, ACM, 2017.

[17] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *ISCA*, pp. 105–117, IEEE, 2015.

[18] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *MICRO*, pp. 1–13, IEEE, 2016.

[19] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "Graphr: Accelerating graph processing using reram," in *International Symposium on High Performance Computer Architecture*, pp. 531–543, IEEE, 2018.

[20] J. Zhou, S. Liu, Q. Guo, X. Zhou, T. Zhi, D. Liu, C. Wang, X. Zhou, Y. Chen, and T. Chen, "Tunao: A high-performance and energy-efficient reconfigurable accelerator for graph processing," in *International Symposium on Cluster, Cloud and Grid Computing*, pp. 731–734, May 2017.

[21] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," in *International Symposium on Computer Architecture*, pp. 166–177, IEEE, 2016.

[22] Y. Perez, R. Sosič, A. Banerjee, R. Puttagunta, M. Raison, P. Shah, and J. Leskovec, "Ringo: Interactive graph analytics on big-memory machines," in *International Conference on Management of Data*, pp. 1105–1110, ACM, 2015.

[23] Z. Shang, F. Li, J. X. Yu, Z. Zhang, and H. Cheng, "Graph analytics through fine-grained parallelism," in *International Conference on Management of Data*, pp. 463–478, ACM, 2016.

[24] J. Lin, "Scale up or scale out for graph processing?," *IEEE Internet Computing*, vol. 22, pp. 72–78, May 2018.

[25] "System memory at a fraction of the dram cost," tech. rep., Intel Corporation, 2017.

[26] J. Leskovec and R. Sosič, "Snap: A general-purpose network analysis and graph-mining library," *ACM Trans. Intell. Syst. Technol.*, vol. 8, pp. 1:1–1:20, July 2016.

[27] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *ACM Sigplan Notices*, vol. 48, pp. 135–146, ACM, 2013.

[28] C. Eksombatchai, P. Jindal, J. Z. Liu, Y. Liu, R. Sharma, C. Sugnet, M. Ulrich, and J. Leskovec, "Pixie: A system for recommending 3+ billion items to 200+ million users in real-time," *CoRR*, vol. abs/1711.07601, 2017.

[29] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Symposium on Operating Systems Principles*, pp. 472–488, ACM, 2013.

[30] J. Malicevic, B. Lepers, and W. Zwaenepoel, "Everything you always wanted to know about multicore graph processing but were afraid to ask," in *USENIX Annual Technical Conference*, pp. 631–643, 2017.

[31] W. Heirman, T. E. Carlson, S. Che, K. Skadron, and L. Eeckhout, "Using cycle stacks to understand scaling bottlenecks in multi-threaded workloads," in *International Symposium on Workload Characterization*, pp. 38–49, IEEE, 2011.

[32] Y. Chou, B. Fahs, and S. Abraham, "Microarchitecture optimizations for exploiting memory-level parallelism," in *International Symposium on Computer Architecture*, pp. 76–87, IEEE, 2004.

[33] S. Beamer, K. Asanovic, and D. Patterson, "Locality exists in graph processing: Workload characterization on an ivy bridge server," in *IISWC*, pp. 56–65, IEEE, 2015.

[34] A. Eisenman, L. Cherkasova, G. Magalhaes, Q. Cai, and S. Katti, "Parallel graph processing on modern multi-core servers: New findings and remaining challenges," in *MASCOTS*, pp. 49–58, IEEE, 2016.

[35] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin, "Graphbig: understanding graph computing in the context of industrial solutions," in *SC-International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, IEEE, 2015.

[36] A. Eisenman, L. Cherkasova, G. Magalhaes, Q. Cai, P. Faraboschi, and S. Katti, "Parallel graph processing: Prejudice and state of the art," in *International Conference on Performance Engineering*, pp. 85–90, ACM, 2016.

[37] L. Jiang, L. Chen, and J. Qiu, "Performance characterization of multi-threaded graph processing applications on many-integrated-core architecture," in *International Symposium on Performance Analysis of Systems and Software*, pp. 199–208, April 2018.

[38] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *International Symposium on Microarchitecture*, pp. 141–152, IEEE, 2015.

[39] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," in *Software, IEEE Proceedings-*, pp. 96–96, IEEE, 2004.

[40] S. Ainsworth and T. M. Jones, "Graph prefetching using data structure knowledge," in *International Conference on Supercomputing*, p. 39, ACM, 2016.

[41] Y. Kora, K. Yamaguchi, and H. Ando, "Mlp-aware dynamic instruction window resizing for adaptively exploiting both ilp and mlp," in *International Symposium on Microarchitecture*, pp. 37–48, IEEE, 2013.

[42] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," in *International Symposium on High Performance Computer Architecture*, pp. 481–492, IEEE, 2017.

[43] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, 2016.

[44] W. Heirman, T. Carlson, and L. Eeckhout, "Sniper: Scalable and accurate parallel multi-core simulation," in *ACACES*, pp. 91–94, HiPEAC, 2012.

[45] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *TACO*, vol. 11, no. 3, p. 28, 2014.

[46] A. Akram and L. Sawalha, "×86 computer architecture simulators: A comparative study," in *International Conference on Computer Design*, pp. 638–645, IEEE, 2016.

[47] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP laboratories*, 2009.

[48] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *CoRR abs/1508.03619*, 2015.

[49] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey, "Navigating the maze of graph analytics frameworks using massive graph datasets," in *International conference on Management of data*, pp. 979–990, ACM, 2014.

[50] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection." http://snap.stanford.edu/data, June 2014.

[51] A. Basak, X. Hu, S. Li, S. M. Oh, and Y. Xie, "Exploring core and cache hierarchy bottlenecks in graph processing workloads," *IEEE Computer Architecture Letters*, vol. 17, pp. 197–200, July 2018.

[52] M. Hashemi, Khubaib, E. Ebrahimi, O. Mutlu, and Y. N. Patt, "Accelerating dependent cache misses with an enhanced memory controller," in *ISCA*, pp. 444–455, IEEE Press, 2016.

[53] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *International Symposium on High Performance Computer Architecture*, pp. 63–74, IEEE, 2007.

[54] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt, "Prefetch-aware dram controllers," in *Proceedings International Symposium on Microarchitecture*, pp. 200–209, IEEE Computer Society, 2008.

[55] *Intel® 64 and IA-32 Architectures Software Developer's Manual*, vol. 3A. 9 2016.

[56] T. 2nd Data Prefetching Championship (DPC2). http://comparch-conf.gatech.edu/dpc2/, 2015.

[57] F. Liu and R. B. Lee, "Random fill cache architecture," in *International Symposium on Microarchitecture*, pp. 203–215, Dec 2014.

[58] V. Seshadri, T. Mullins, A. Boroumand, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Gather-scatter dram: in-dram address translation to improve the spatial locality of non-unit strided accesses," in *Proceedings International Symposium on Microarchitecture*, pp. 267–280, ACM, 2015.

[59] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "Graphpim: Enabling instruction-level pim offloading in graph computing frameworks," in *International Symposium on High Performance Computer Architecture*, pp. 457–468, IEEE, 2017.

[60] R. Cooksey, S. Jourdan, and D. Grunwald, "A stateless, content-directed data prefetching mechanism," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 279–290, ACM, 2002.

[61] C.-L. Yang and A. R. Lebeck, "Push vs. pull: Data movement for linked data structures," in *International Conference on Supercomputing*, pp. 176–186, ACM, 2000.

[62] A. Roth, A. Moshovos, and G. S. Sohi, "Dependence based prefetching for linked data structures," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 115–126, ACM, 1998.

[63] E. Ebrahimi, O. Mutlu, and Y. N. Patt, "Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems," in *International Symposium on High Performance Computer Architecture*, pp. 7–17, IEEE, 2009.

[64] D. Joseph and D. Grunwald, "Prefetching using markov predictors," *IEEE transactions on computers*, vol. 48, no. 2, pp. 121–133, 1999.

[65] J. Kim, S. H. Pugsley, P. V. Gratz, A. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in *International Symposium on Microarchitecture*, p. 60, IEEE Press, 2016.

[66] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *International Symposium on High-Performance Computer Architecture*, pp. 129–140, IEEE, 2003.

[67] M. Annavaram, J. M. Patel, and E. S. Davidson, "Data prefetching by dependence graph precomputation," in *ISCA*, pp. 52–61, IEEE, 2001.

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

[69] Y. Ishii, M. Inaba, and K. Hiraki, "Access map pattern matching for high performance data cache prefetch," *Journal of Instruction-Level Parallelism*, vol. 13, pp. 1–24, 2011.

[70] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "Imp: Indirect memory prefetcher," in *International Symposium on Microarchitecture*, pp. 178–190, ACM, 2015.

[71] P. Yedlapalli, J. Kotra, E. Kultursay, M. Kandemir, C. R. Das, and A. Sivasubramaniam, "Meeting midway: Improving cmp performance with memory-side prefetching," in *International conference on Parallel architectures and compilation techniques*, pp. 289–298, IEEE Press, 2013.

[72] C. J. Hughes and S. V. Adve, "Memory-side prefetching for linked data structures for processor-in-memory systems," *Journal of Parallel and Distributed Computing*, vol. 65, no. 4, pp. 448–463, 2005.