A. M. Kaushik, G. Pekhimenko, and H. Patel, "Gretch: A Hardware Prefetcher for Graph Analytics," ACM Transactions on Architecture and Code Optimization (TACO), pp. 1–25, Nov. 2020.

Gretch: A Hardware Prefetcher for Graph Analytics

ANIRUDH MOHAN KAUSHIK, University of Waterloo, Canada
GENNADY PEKHIMENKO, University of Toronto, Canada
HIREN PATEL, University of Waterloo, Canada

Data-dependent memory accesses (DDAs) pose an important challenge for high-performance graph analytics (GA). This is because such memory accesses do not exhibit enough temporal and spatial locality resulting in low cache performance. Prior efforts that focused on improving the performance of DDAs for GA are not applicable across various GA frameworks. This is because (1) they only focus on one particular graph representation, and (2) they require workload changes to communicate specific information to the hardware for their effective operation.

In this work, we propose a hardware-only solution to improving the performance of DDAs for GA across multiple GA frameworks. We present a hardware prefetcher for GA called Gretch, that addresses the above limitations. An important observation we make is that identifying certain DDAs without hardware-software communication is sensitive to the instruction scheduling. A key contribution of this work is a hardware mechanism that activates Gretch to identify DDAs when using either in-order or out-of-order instruction scheduling. Our evaluation shows that Gretch provides an average speedup of 38% over no prefetching, 25% over conventional stride prefetcher, and outperforms prior DDAs prefetchers by 22% with only 1% increase in power consumption when executed on different GA workloads and frameworks.

CCS Concepts: • **Computer systems organization → Architectures**; • **General and reference → Performance**;

Additional Key Words and Phrases: Hardware prefetching, graph analytics, data-dependent memory accesses

ACM Reference format:
Anirudh Mohan kaushik, Gennady Pekhimenko, and Hiren Patel. 2021. Gretch: A Hardware Prefetcher for Graph Analytics. *ACM Trans. Archit. Code Optim.* 18, 2, Article 18 (February 2021), 25 pages.
https://doi.org/10.1145/3439803

1 INTRODUCTION

Graph analytics (GA) is an important and emerging domain for machine learning applications [51], image recognition [60], recommendation systems [23, 43], social networks [27, 38], and security threat analysis [48, 65]. The importance of GA has resulted in several works that profile GA on modern computing platforms to better understand the performance bottlenecks of GA [14, 26, 39, 45, 55]. These works make two key observations. First, GA only spends a *marginal* amount of time

New article, not an extension of a conference paper.

Authors' addresses: A. M. Kaushik, University of Waterloo, 200 University Avenue West, Waterloo, Ontario, Canada, N2L 3G1; email: anirudh.m.kaushik@uwaterloo.ca; G. Pekhimenko, University of Toronto, 27 King's College Circle, Toronto, Ontario, Canada M5S 1A1; email: pekhimenko@cs.toronto.edu; H. Patel, University of Waterloo, Waterloo, Canada; email: hiren.patel@uwaterloo.ca.

1544-3566/2021/02-ART18
https://doi.org/10.1145/3439803

A. M. Kaushik, G. Pekhimenko, and H. Patel, "Gretch: A Hardware Prefetcher for Graph Analytics," ACM Transactions on Architecture and Code Optimization (TACO), pp. 1–25, Nov. 2020.

on computing, while spending most of the time on data movement through the memory hierarchy. Second, data movement in GA renders levels of the conventional cache hierarchy that are closer to the cores (L1, L2) *ineffective* toward improving their performance [14, 55]. This is because the data movement in GA largely comprise of *data-dependent* accesses (DDAs) that exhibit insufficient spatial and temporal locality for the cache levels closer to the cores to exploit. DDAs are pairs of memory instructions where the data accessed by the first instruction (referred as *producer*) is used to compute the memory address accessed by a following instruction (referred as *consumer*). As a result, these cache levels register low cache hit rates for GA workloads, which in turn limits GA workload performance.

Recent GA-specific micro-architectural techniques mainly focused on improving the performance of DDAs [4, 12, 53, 67]. Although these prior techniques delivered performance speedups for different graph algorithms, their applicability across different GA frameworks is limited for the following reasons.

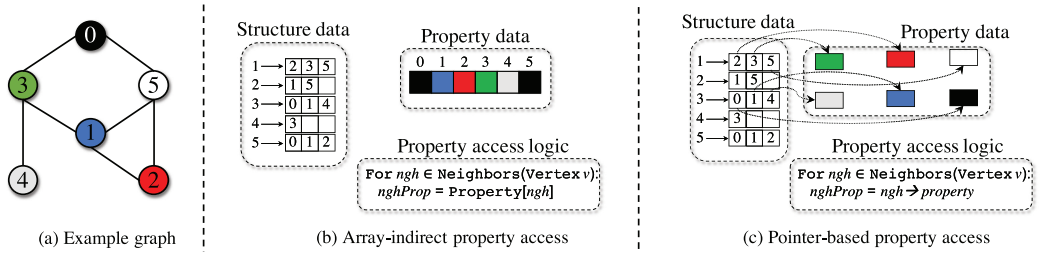
First, prior works only focused on a *single type of graph representation*–the compressed sparse row (CSR) representation. CSR uses a combination of multiple array data structures. However, different GA frameworks use different graph representations. For example, graph databases [1, 55] use a combination of arrays and pointers, which is different from CSR. Pointers are key for supporting dynamic graphs by enabling graph modifications in the form of additions/deletions of nodes and edges [1, 55]. As a result, different graph representations exhibit *different DDAs*. Therefore, prior efforts optimizing solely for one type of DDAs cannot improve performance of other types of DDAs. As a result, prior works have *limited* applicability across different GA frameworks.

Second, **all** prior works that focus on improving GA performance through micro-architectural techniques *require some hardware-software interaction from the GA framework* to guide their operation [4, 12, 53, 67]. Specifically, the software enables the hardware accelerator when it encounters DDAs. This requires workload changes to communicate the beginning of regions that access DDAs to the accelerators. Consequently, to use these prior works for other GA frameworks, one would need to identify the correct locations to insert the necessary hardware-software interactions, which requires a detailed understanding of the GA framework implementations. This is an additional impediment in applying prior works to other GA frameworks.

It is important to note that devising a method that detects DDAs in hardware without such hardware-software interaction poses a critical challenge. Notably, the method to detect certain DDAs must change based on the core's instruction scheduling capability. For instance, in-order instruction scheduling simplifies the identification of DDAs, but, that same approach is not applicable to out-of-order instruction scheduling. To the best of our knowledge, there does not exist an approach to effectively identify DDAs from GA workloads when the cores use out-of-order instruction scheduling. Most prior works on improving DDAs sidestepped this challenge either due to the hardware-software interaction supported by their specific GA framework [4, 12, 53, 67] or by optimizing primarily for in-order cores [66].

In this work, we design a hardware DDA prefetcher for GA, Gretch, that directly addresses the two aforementioned limitations. As a result, Gretch (1) is applicable across different graph representations and identifies different DDAs; and, (2) it does not require **any** interaction from the GA framework to identify these access patterns. Gretch uses a novel approach to address the challenge posed by out-of-order instruction scheduling. This is accomplished by exploiting the interaction between memory accesses in GA workloads and the behaviour of the conventional stride prefetcher [21]. Hence, Gretch implements a purely hardware approach to detect DDAs for GA frameworks that works for both in-order and out-of-order instruction scheduling. Gretch is positioned next to the level 1 data cache (L1-D), and prefetches data from the lower memory hierarchy levels (L2 and memory) into the L1-D cache.

A. M. Kaushik, G. Pekhimenko, and H. Patel, "Gretch: A Hardware Prefetcher for Graph Analytics," ACM Transactions on Architecture and Code Optimization (TACO), pp. 1–25, Nov. 2020.



(a) Example graph (b) Array-indirect property access (c) Pointer-based property access

Fig. 1. Graph representations.

We summarize our main contributions as follows.

(1) We describe the challenge in identifying certain DDAs in the presence of out-of-order instruction scheduling. We describe one technique that addresses this challenge by using the interaction between the GA's memory access patterns and conventional stride prefetcher [21].
(2) We propose a new hardware L1-D cache prefetcher for GA called Gretch. Gretch uses a unified set of hardware structures to implement an approach to identify different DDAs simultaneously. This approach leverages the identification technique from contribution (1) without needing any interaction from the GA frameworks.
(3) We evaluate Gretch across GA frameworks Ligra [64] and GraphBig [55], and the mainstay GA benchmark Graph-500 [54]. We also compare against stride prefetching, and various DDA prefetchers. Our evaluation shows that Gretch delivers an average 25% (up to 89%) performance improvement over stride prefetching, and 20% (up to 89%) over the next best DDA prefetcher.

2 BACKGROUND

2.1 GA Frameworks

GA frameworks are software frameworks that allow implementing different graph algorithms (GA workloads). GA frameworks must provide features to meet the demands imposed by their use in a variety of domains. This has resulted in a variety of GA frameworks. Some examples include graph databases [1, 25, 55], shared-memory graph processing frameworks [37, 64], and graph benchmark frameworks [2, 15, 54]. We evaluate our proposed hardware prefetcher using three different GA frameworks taken from different domains: (1) GraphBIG [55], which is inspired by IBM SystemG's graph database [19]; (2) Ligra [64], a highly optimized shared-memory graph processing framework; and (3) Graph-500 [54], a high-performance graph benchmark for ranking supercomputers. Table 1 describes the GA workloads executed on these GA frameworks.

2.2 Data-dependent Memory Accesses in GA

GA frameworks typically have graph representations with graph structure data (nodes and edges), and node/edge property data. Structure data stores information about nodes and edges. Property data, however, embeds semantic information relevant to nodes and edges that is relevant to the application domain. GA workloads use structure data to *explore* the graph, and *compute* using property data.

Figure 1 (a) shows an example of an undirected graph with its structure data being the connections, and property being the node color. A GA workload explores a graph by traversing a node's edges. Most graph representations use arrays to store the node's edges to benefit from spatial locality. The data in the node's edges encode information about the neighbor node. This is used to

A. M. Kaushik, G. Pekhimenko, and H. Patel, "Gretch: A Hardware Prefetcher for Graph Analytics," ACM Transactions on Architecture and Code Optimization (TACO), pp. 1–25, Nov. 2020.

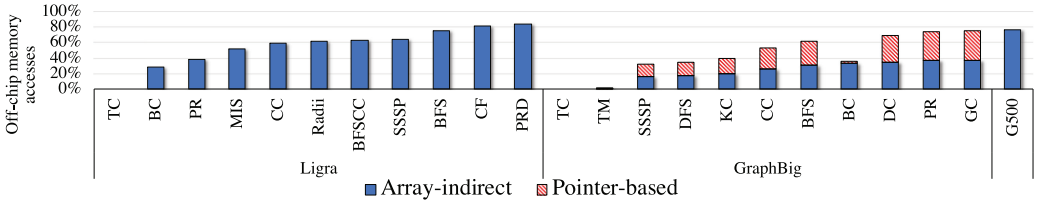


Fig. 2. Contribution of data-dependent accesses to off-chip main-memory accesses.

extract the property of the neighbor node. Hence, extracting neighbors' properties in GA workloads exhibit DDAs. Consider the neighbor node identifier as an unsigned integer. This neighbor node identifier is used as an *index* into an array of node properties as shown in Figure 1(b) resulting in an *array-indirect DDA*. Another example is a pointer to an encapsulated data structure that stores the neighbor node's connections and property as shown in Figure 1(c). Accessing the property of the neighbor node requires *dereferencing* the pointer to the structure, and accessing the property field resulting in a *pointer-based DDA*.

3 MOTIVATION

Prior works have focused on addressing these DDAs from GA to improve GA performance [4, 12, 53, 67]. In the following subsections, we list two observations regarding DDAs that have received less attention from these prior works. These observations limit the applicability and performance benefits of prior works on different GA frameworks. The key design novelties of Gretch are built on these observations.

3.1 Diversity in Graph Representations

Our first key observation is that *different GA frameworks employ different graph representations, and these different graph representations in turn exhibit different DDAs*. Figure 2 shows a breakdown of the contribution of DDAs due to node property accesses to the off-chip main-memory accesses across different GA frameworks and workloads. We make two observations from Figure 2. First, DDAs are a significant contributor to the off-chip main-memory accesses across most GA workloads (as high as 82% in PRD in Ligra, 45% on average). As a result, addressing the performance impact of DDAs is key toward achieving high-performance GA. Second, GA workloads on Ligra and Graph-500 exhibit array-indirect DDAs whereas GA workloads on GraphBIG exhibit *both* array-indirect and pointer-based DDAs. This is because Ligra and Graph-500 use the CSR graph representation whereas the graph representation in GraphBIG uses a combination of both arrays and pointers. In GraphBIG, an array-indirect access first retrieves a pointer to the neighbor node's data structure, and then a pointer dereference retrieves the neighbor node's property. As a result, property accesses in GraphBIG incur both array-indirect and pointer-based DDAs.

Prior hardware prefetchers for DDAs identify *only one type of DDAs* [22, 24, 61, 66]. Expectedly, these prefetchers are unable to deliver performance benefits across GA frameworks that have different DDAs. Prior GA-specific micro-architectural accelerators also work for one type of DDAs as they are designed for the CSR graph representation DDAs [4, 12, 53, 67].

In summary, *prior techniques to improve DDAs from GA workloads deliver sub-optimal performance benefits for other graph representations*. While we acknowledge CSR's importance and popularity, we find that focusing on improving the performance for one particular type of graph representation is *restrictive*. An important contribution of Gretch is that it is designed to improve GA performance across different graph representations by identifying the common characteristics of

A. M. Kaushik, G. Pekhimenko, and H. Patel, "Gretch: A Hardware Prefetcher for Graph Analytics," ACM Transactions on Architecture and Code Optimization (TACO), pp. 1–25, Nov. 2020.

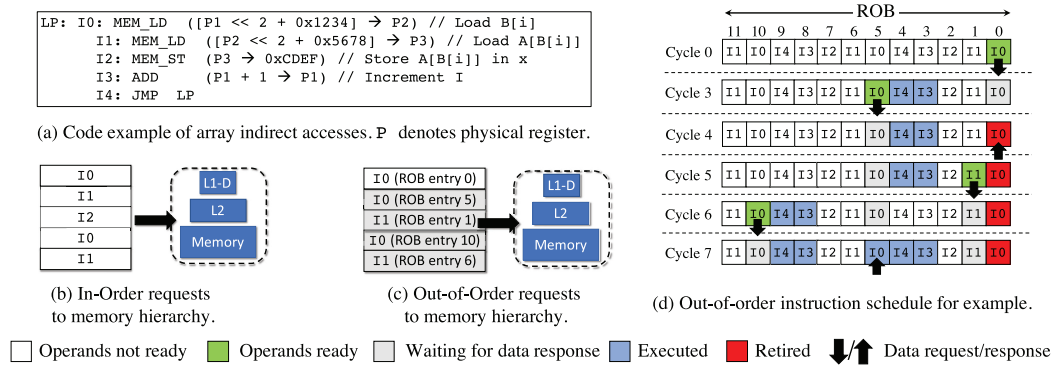


Fig. 3. Effect of instruction scheduling on identification of data-dependent accesses.

DDAs in GA workloads. We evaluate Gretch on different graph representations, and show that Gretch improves GA performance for workloads deployed on different graph representations.

3.2 Identifying Data-dependent Accesses

Our second key observation is that *prior on-chip micro-architectural accelerators for GA relied on hardware-software interaction from the GA framework resulting in application changes.* It was necessary for this hardware-software interaction to identify the start of DDAs in the GA workload to deliver good performance. In this work, we focus on a *purely hardware approach* that (1) identifies DDAs in GA workloads and (2) activates Gretch to train for DDAs without any interaction from the workloads. Thus, application changes are *not* needed when employing Gretch.

We observe that designing a purely hardware approach for identifying DDAs is *sensitive* to the instruction scheduling. In particular, identifying *array-indirect DDAs* in the presence of out-of-order scheduling to the best of our knowledge, remains a challenge (see Figure 3).

Illustrative example. Figure 3(a) shows a code example that generates array-indirect DDAs where instruction `I1` uses the data loaded by `I0` to generate its memory address. The index i is assumed to be in physical register `P1`, and arrays `A` and `B` hold 4-byte elements. The base addresses of `A` and `B` are `0x1234` and `0x5678`, respectively. For this example, we use the following equation to infer array-indirect DDAs [66]:

$$\text{Address } A[B[i]] = A\text{'s base address} + \text{Offset size} \times B[i]. \tag{1}$$

Figure 3(b) shows the order of memory accesses issued by an in-order core. For this access order, using the data accessed by one execution of `I0` and memory address of the following access issued by `I1` in the above equation can establish the DDA relationship between `I0` and `I1`. Hence, for in-order cores, a hardware mechanism can activate Gretch on any access as every memory access from `I0` is followed by a corresponding DDA from `I1`.

Figure 3(c) shows the order of memory accesses issued by an out-of-order core to the memory hierarchy, and Figure 3(d) shows a timeline of the memory accesses issued by instructions in the reorder buffer (ROB). We denote an instruction `I` at ROB entry i as `I` @ i. We assume that a memory request completes in 4 cycles. Unlike the in-order memory access schedule, multiple memory accesses from `I0` can precede the DDA from `I1`. This is shown in Figures 3(c) and 3(d), where memory accesses scheduled by `I0` @ 0 and `I0` @ 5 precede the memory access sheduled by `I1` @ 1. This is because of two reasons: (1) `I0` is not dependent on any instruction for its address computation, and (2) multiple accesses from `I0` can be in flight based on the available memory

A. M. Kaushik, G. Pekhimenko, and H. Patel, "Gretch: A Hardware Prefetcher for Graph Analytics," ACM Transactions on Architecture and Code Optimization (TACO), pp. 1–25, Nov. 2020.

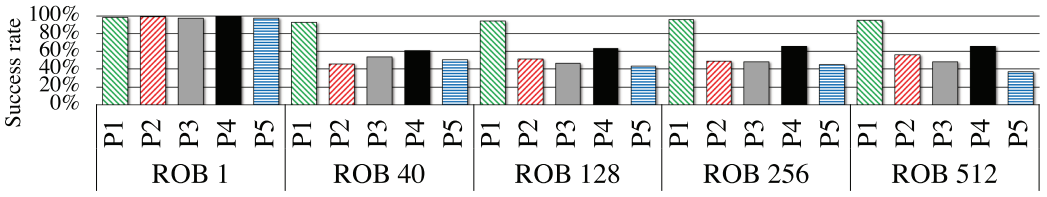


Fig. 4. Success rate of $B[i]$ choice with ROB sizes.

level parallelism (MLP). Hence, the mechanism described for in-order instruction schedule *cannot* be applied for out-of-order instruction schedule as it will result in inferring incorrect DDAs. For example, using the data accessed by memory access issued by `I0` @ 5 and the memory address of the subsequent memory access issued by `I0` @ 1 results in inferring incorrect DDA as the memory address issued by `I0` @ 1 uses the data accessed by `I0` @ 0.

From our evaluation, we observed that this interaction of array-indirect DDAs and out-of-order instruction scheduling is prevalent in GA workloads. This is because of two key reasons. (1) In a graph representation such as CSR, which exhibits array-indirect DDAs, neighbors of nodes are represented as array data structures. As real-world graphs are large and comprise of millions of nodes, these neighbor arrays of nodes are dynamically allocated. Hence, the neighbor array of one node is not contiguous with the neighbor array of another node. As a result, initial memory accesses to neighbors ($B[i]$ accesses in Figure 3) are typically cache misses[1] resulting in an execution scenario that is similar to Figure 3(d). (2) A conventional stride prefetcher can eventually capture the strided accesses to the nodes' neighbor arrays, and prefetch $B[i]$ into the cache hierarchy. As a result, stride prefetching of $B[i]$ can expedite the virtual address computation of neighbor property accesses ($A[B[i]]$). However, these accesses typically result in address translation misses in the translation look-aside buffer (TLB). This is because the data $B[i]$ that are used to construct the addresses of $A[B[i]]$ do not follow a particular pattern, which results in limited page locality [14]. Hence, a cache miss on a neighbor property access results in first fetching the appropriate address translation, and then fetching the data from the requested address from the cache hierarchy or off-chip memory. Out-of-order instruction scheduling can issue multiple independent memory accesses during these long latency neighbor property misses to improve instruction throughput. In particular, multiple accesses of $B[i]$ can be scheduled during a TLB miss of $A[B[i]]$ resulting in an execution scenario similar to Figure 3(d). In summary, the identification of array-indirect DDAs is a challenge in the presence of out-of-order instruction scheduling.

One technique that we explore in this work to address this challenge is to identify the *first* producer access. For GA workloads where accessing nodes' properties incur array indirect DDAs (Figure 1(b)), this means identifying the first neighbor access of a node. The central rationale behind this technique is that the memory address of the following consumer access is dependent on the data returned by the first producer access. Hence, the first producer access and the following consumer access can be used to correctly infer the array indirect DDAs in the presence of out-of-order instruction scheduling. From our evaluation, we find that this observation strongly exhibits only for the first producer access.

To highlight the impact of the above observation, Figure 4 shows the success rate in identifying array-indirect DDAs using different producer accesses for the PR workload in Ligra. Recall that most DDAs in GA workloads executed in Ligra are array-indirect DDAs (Figure 2). For this experiment, we performed a detailed instruction level analysis of the PR workload in Ligra, and identified the producer ($B[i]$) and consumer memory instructions ($A[B[i]]$) that resulted in array

[1]This is due to stride deviation observed by a stride prefetcher. We discuss this in detail in Section 4.

A. M. Kaushik, G. Pekhimenko, and H. Patel, "Gretch: A Hardware Prefetcher for Graph Analytics," ACM Transactions on Architecture and Code Optimization (TACO), pp. 1–25, Nov. 2020.

indirect DDAs. We also recorded the offset size (size of $B[i]$) and the base address (address of array A). During program execution, we used information about the memory address and accessed data for the marked producer and consumer instructions, and applied Equation (1) to compute the base address of A, and compared it with the recorded base address. A successful array-indirect DDA identification is when the computed and recorded base addresses match.

We use the notation Pi to denote the ith producer access used for identifying array-indirect DDAs. Therefore, $P1$ denotes the first producer access, $P2$ denotes the second producer access and so on. We define success rate as the ratio of successful array-indirect DDA identifications and the total number of identification attempts. For a producer access choice Pi, a 100% success rate means that for each graph node's exploration by the workload, the ith neighbor node's access (producer) and the following node property access (consumer) correctly identifies the array indirect DDA. We vary the ROB size in the cores, which controls how far the producer can execute ahead of the consumer.

For a ROB size of 1, *any* producer access can be used to correctly infer array-indirect DDAs with high success rate (>90%). This is shown in Figure 4. A ROB size of 1 is an in-order core, and forces the producer and consumer to issue memory accesses to the cache hierarchy such that a producer access is followed by the corresponding consumer access (Figure 3(b)). However, for ROB sizes greater than 1, all producer access choices *with the exception of* $P1$ begin to show lower success rates in identifying array-indirect DDAs. A low success rate translates to missed opportunities in identifying and optimizing DDAs, and hence, low performance benefits. Larger ROBs allow the producer to execute further ahead than the consumer. As a result, for a producer access other than $P1$, the following consumer access may not be dependent on it; rather the consumer access may be dependent on a much prior producer access. This is because for producer accesses other than $P1$, there is little to no guarantee that the following consumer access is dependent on the data accessed by the preceding producer access. However, $P1$ continues to register a high success rate (> 90%) even with higher ROB sizes. From our evaluation across GA frameworks and workloads, including PR, we observe that the ROB contents prior to a node's exploration does not have pending memory accesses from the previous node's exploration. As a result, consumer accesses after the first neighbor access of a node are not from a previous node's exploration. We attribute this to one of two reasons: (1) ROB flushes due to branch mis-predictions, (2) execution of routines between exploration of nodes that allows any pending memory accesses from the previous node exploration to complete before the start of the next node exploration. Branch mis-predictions can be either due to mis-predicting the degree of the node (number of node's neighbors) currently being explored or mis-predicting the node property value.

In the following section, we show how we use this intuition to build a hardware mechanism in Gretch to identify array-indirect DDAs in the presence of instruction scheduling. Identifying pointer-based DDAs do not encounter the same challenge as that of array-indirect DDAs. This is because pointer-based DDAs can be identified using simple hardware based on producer-consumer tag matching as done in Reference [61] irrespective of the instruction scheduling. Note that while we explicitly mark the producer and consumer instructions in the GA workload for the above experiment, Gretch identifies the producer and consumer instructions in hardware without any assistance from the GA workload and changes to the GA workload.

4 GRETCH: KEY DESIGN IDEAS

We describe four key design ideas of Gretch using Figure 5 that shows a high-level design of our approach.

Stride prefetching ①. Gretch works with a conventional stride prefetcher (SP) that identifies and optimizes strided accesses. Across multiple graph representations, we find that a node's neighbors

A. M. Kaushik, G. Pekhimenko, and H. Patel, "Gretch: A Hardware Prefetcher for Graph Analytics," ACM Transactions on Architecture and Code Optimization (TACO), pp. 1–25, Nov. 2020.

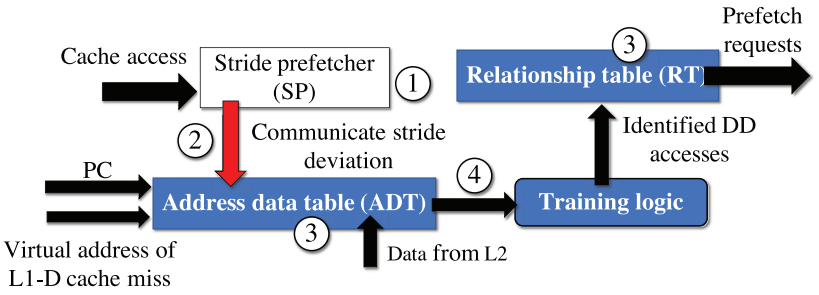


Fig. 5. High-level design of Gretch.

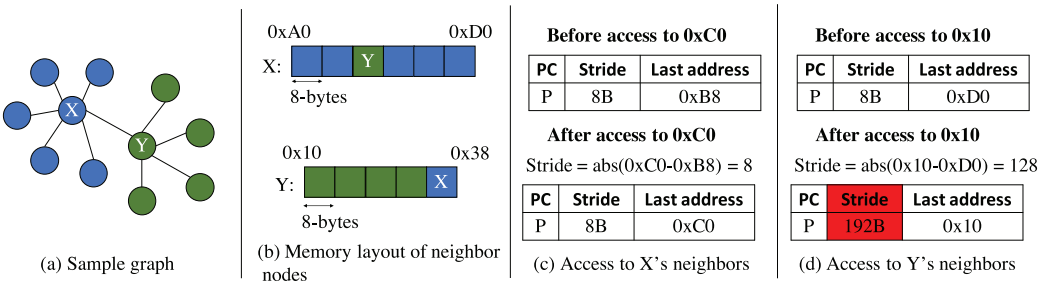


Fig. 6. Illustrative example of hardware approach for activating Gretch.

typically use an array data structure to benefit from spatial locality offered by arrays. Hence, exploration of a node's neighbors exhibits strided access patterns that can be captured and optimized by SP. From our evaluation, SP speeds up GA workload execution by up to 41% over no prefetching (average 10%). Hence, GA workloads benefit from SP. In this work, we use the reference prediction table-based SP [21] that identifies strided accesses on a per-memory instruction granularity.

Hardware approach for activating Gretch ②. We explain the key intuition behind the purely hardware mechanism that activates Gretch to begin training for DDAs for out-of-order and in-order instruction scheduling using Figure 6. Figure 6(a) shows a graph with two highlighted nodes Ⓧ and Ⓨ, and Figure 6(b) shows the memory layout of the neighbors of Ⓧ and Ⓨ.

GA workloads typically execute the following sequence of operations: (1) explore a node's neighbors, (2) modify the node's property or the node's neighbors' properties, and (3) mark a neighbor to be explored next based on the workloads' requirements. For the example in Figure 6(a), assume that a GA workload explores the neighbors of Ⓧ, marks Ⓨ to be explored, and then the GA workload explores the neighbors of Ⓨ. The neighbors of Ⓧ and Ⓨ are stored in two different arrays as shown in Figure 6(b). GA frameworks typically layout the identifiers of a node's neighbors in contiguous memory locations to benefit from spatial locality [25, 37, 55, 64].

The GA workload begins exploring the neighbors of Ⓧ, which incurs strided accesses on the neighbor array. These strided accesses are captured by SP, and the SP records the computed stride and the memory instruction that accesses the neighbor array. Figure 6(c) shows the SP contents for the memory instruction *P* that accesses the neighbor array. The SP sees the same stride value of 8 bytes during the exploration of Ⓧ's neighbors. After exploring all neighbors of Ⓧ, the GA workload begins to explore the neighbors of Ⓨ. On accessing the first neighbor of Ⓨ, the computed stride value does not match that recorded in the SP entry (8-bytes). This is shown in Figure 6(d) where the computed stride between the last accessed neighbor of Ⓧ and the first neighbor of Ⓨ

A. M. Kaushik, G. Pekhimenko, and H. Patel, "Gretch: A Hardware Prefetcher for Graph Analytics," ACM Transactions on Architecture and Code Optimization (TACO), pp. 1–25, Nov. 2020.

does not match the stride recorded in the SP. This is because the memory layout of neighbors of Ⓨ *are not contiguous* with that of neighbors of Ⓧ. In real-world graphs that have millions of nodes and billions of connections, neighbor arrays of different nodes are dynamically allocated, and as a result, the memory layout of neighbor arrays of adjacent nodes may not be contiguous to each other. As a result, SP observes a *deviation in computed stride* when the GA workload begins exploring a new node. After exploring the first neighbor of Ⓨ, the SP observes constant stride value of 8-bytes, and identifies the strided access pattern to neighbors of Ⓨ.

Gretch uses this deviation in the computed stride as a way to identify the first access of a node's neighbors, and activate its recording of memory access information. As a consequence, this hardware mechanism allows Gretch to detect array-indirect access patterns in the presence of out-of-order instruction scheduling. We implement this hardware mechanism by enabling communication between SP and Gretch as shown in Figure 5. SP communicates the PC of a high confidence memory instruction that previously exhibited strided accesses when it observed a deviation in the computed stride to Gretch.

ADT and RT ③. To identify DDAs, Gretch first records access information about L1-D cache misses such as the PC of the memory instruction that issued the L1-D cache miss, the memory address, and data returned by the cache miss in a hardware structure called the address-data table (ADT). Gretch then trains on the contents of the ADT, and records DDAs inferred by the training logic in a relationship table (RT). Gretch looks up the RT to generate prefetch requests.

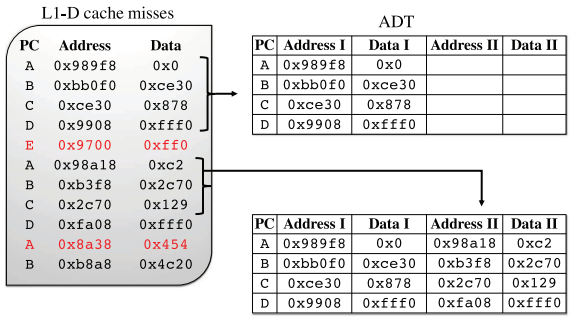
Distinct recording and training phases ④. Gretch separates the recording of access information in the ADT and the training phase to infer DDAs from the ADT contents. This results in two benefits. First, the ADT design is straightforward, and is not optimized for one particular DDA type. This is unlike hardware structures used in prior works that were *tightly coupled* with the logic to identify a particular DDA type [61, 66]. The second benefit is that multiple training logic for identifying different DDA types can be simultaneously applied on the ADT contents. An alternative is to combine multiple specific DDA prefetchers that identify different DDAs, which is unattractive due to its high hardware overhead.

Gretch operates on virtual addresses that result in misses in the L1-D cache, and prefetches data into the L1-D cache. Gretch generates virtual addresses for prefetch requests, and accesses the L1 TLB to convert the virtual addresses of prefetch requests to corresponding physical addresses. If a prefetch request generated by Gretch misses in the TLB, then the TLB generates the necessary memory requests to retrieve the translation followed by the data contents of the prefetch request. Note that Gretch does not optimize the address translation of prefetch requests, and any address translation misses due to prefetch requests triggers the appropriate mechanisms (software or hardware) to resolve the address translation.

5 GRETCH: DETAILED OPERATION

Gretch's operation begins when it receives communication from SP that a deviation in stride occurred. We limit SP to only communicate stride deviations for memory instructions that previously exhibited stride behavior with high confidence. On receiving this communication, Gretch performs the following two steps: ADT population, and training on the contents of the ADT to recognize DDAs. Note that when Gretch is populating the ADT, Gretch does not simultaneously train on the ADT contents. Similarly, when Gretch is training on the ADT contents, no new entries are added to the ADT. After completing these steps, Gretch waits for the next communication from SP. When this is received, Gretch clears the ADT and repeats these two steps. We provide details about these steps in the following subsections using the example in Figure 7.

A. M. Kaushik, G. Pekhimenko, and H. Patel, "Gretch: A Hardware Prefetcher for Graph Analytics," ACM Transactions on Architecture and Code Optimization (TACO), pp. 1–25, Nov. 2020.



(a) ADT population.

ADT pairs	DD inferred
<A,B>, <A,C>, <B,D>, <C,D>	None
<A,D>	Absolute data difference of A = 0xc2 Absolute address difference of D = 0x6100 Remainder of 0x6100/0xc2 = 0 ➔ Array-indirect access between A and D
<B,C>	Data accessed by B = Memory address of C ➔ Pointer-based access between B and C

Producer	Consumer	Type	Base address	Size	Predicted address	Conf
B	C	1	--	--		0
A	D	0	0x989f8	0x80		0

Offset size = Quotient of 0x6100/0xc2 = 0x80
Base address = 0xfa08 – 0xc2 * 0x80 = 0x989f8

(b) Training on ADT pairs.

Fig. 7. Gretch detailed operation example.

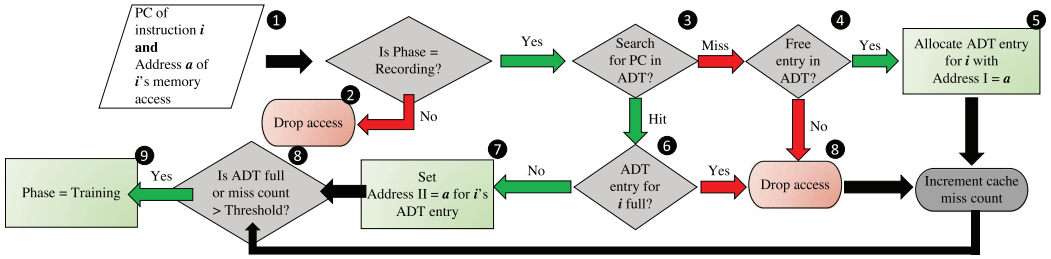


Fig. 8. ADT population mechanism.

5.1 ADT Population

The ADT records access information of L1-D cache misses. As shown in Figure 7(a), the ADT stores the PC, the memory address, and the contents of the data accessed by two L1-D cache misses of a memory instruction. In Figure 7(a), Address I/II and Data I/II refer to the first/second L1-D cache miss to the same PC. Gretch requires access information for two L1-D cache misses due to its general training logic to identify DDAs, which we describe in Section 5.2.

We make two key design choices in the mechanism used for populating the ADT. These choices are necessary for the training logic to correctly identify DDAs. First, entries in the ADT are inserted in the order of L1-D cache misses. Thus, for a DDA pair, the producer is recorded *before* the consumer in the ADT. In Section 5.2, we explain how this simplifies the training logic. Second, Gretch does not evict an ADT entry on an ADT capacity miss. This is because evicting an ADT entry on an ADT capacity miss disrupts the order in which the ADT records the information. Allowing for this disruption disallows the training logic from correctly identifying DDAs.

Mechanism. Figure 8 describes the ADT population mechanism. The ADT first receives the PC of a memory instruction that misses in the L1-D cache, and its memory address (①). Gretch accesses the ADT if Gretch is recording information in the ADT (②). The PC is used as an index into the ADT. Gretch *creates* a new ADT entry if no ADT entry associated with PC is found and the ADT is not full (③, ④, ⑤). Gretch *updates* an existing ADT entry on an ADT hit if the corresponding ADT entry has access information for only one L1-D cache miss (③, ⑥, ⑦). Gretch *does not update the ADT* on an ADT hit if two L1-D cache misses are recorded in the corresponding ADT entry (③, ⑥, ⑧). Gretch *does not evict an ADT entry* on an ADT capacity miss. Gretch stops the ADT population when the ADT is completely full or a threshold number of cache misses has been observed (1000 cache misses), and begins the training phase (⑧, ⑨).

A. M. Kaushik, G. Pekhimenko, and H. Patel, "Gretch: A Hardware Prefetcher for Graph Analytics," ACM Transactions on Architecture and Code Optimization (TACO), pp. 1–25, Nov. 2020.

Illustrative example. Figure 7(a) shows the ADT population mechanism on a 4-entry ADT for a sequence of L1-D cache misses. Initially, the ADT is empty. Memory request issued by PC `A` misses in the L1-D cache. Gretch looks for an existing ADT entry for PC `A`, and does not find an ADT entry for PC `A`. As a result, Gretch creates a new ADT entry for PC `A`, and populates this entry with the access information of PC `A`'s cache miss. Similarly, ADT entries are created for L1-D cache misses issued by PCs `B`, `C`, `D`. The L1-D cache miss from PC `E` (highlighted in red) encounters an ADT capacity miss; thus, it is not recorded in the ADT. The second time the L1-D cache misses on PCs `A`, `B`, `C`, `D`, the corresponding memory address and data are stored in Address II/Data II of their corresponding ADT entries. Once the second access fields are filled, the ADT does not record additional information for the same PC. For instance, the third L1-D cache miss on PC `A` to address `0x8a38` is not recorded as the corresponding ADT entry has access information for two L1-D cache misses.

5.2 Training on ADT Contents

The training step creates potential producer-consumer pairs from the contents of the ADT, and identifies DDAs from the potential producer-consumer pairs. Identified DDAs are inserted into the RT.

5.2.1 Creating ADT Pairs. Recall from Section 5.1 that the ADT is filled in the order of L1-D cache misses. Hence, a producer for a DDA must be recorded *before* the corresponding consumer in the ADT. Exploiting this ADT population order *reduces* the number of potential producer-consumer pairs to train on by only considering pairs of ADT entries such that one ADT entry (potential producer) precedes another ADT entry (potential consumer). For example, a 4-entry ADT has 16 possible ADT pair combinations. By exploiting the ADT population order, Gretch only needs to train on 6 ADT pairs.

5.2.2 Identifying DDAs. For each ADT pair, Gretch applies training logic to infer pointer-based or array-indirect DDAs relationship.

Pointer-based accesses. Gretch identifies pointer-based accesses by comparing the data accessed by a potential producer with the memory address of the corresponding potential consumer [61]. ADT pair with PCs `B` and `C` exhibit pointer-based access as shown in Figure 7(b). The data accessed by PC `B` is equal to the memory address accessed by PC `C`. The RT entry for this DDA is shown in Figure 7 where the producer is set to PC `B` and the consumer is set to PC `C`. The DDA type is a binary bit where bit value 1 denotes a pointer-based DDA.

Array-indirect accesses. Equation (1) (Section 3.2) captures the array-indirect DDA relationship. There are two unknown values in Equation (1), which are necessary to identify an array-indirect DDA: (1) A's base address, and (2) the offset size. Prior GA-specific accelerators explicitly communicated these unknown values to the accelerator [4, 12, 53]. However, prior work IMP [66] used the accessed data and memory address for a pair of memory accesses and assumed a fixed set of offset sizes to compute A's base address. This set restricted the offset sizes to common data types, and was done primarily to reduce IMP's hardware overhead [66].

We find that identifying array-indirect accesses for a set of offset sizes is restrictive especially for GA. This is because graph representations may encode the properties of nodes and edges in a custom structure resulting in custom offset sizes. For example, GraphBIG [55] uses custom structures to encode the properties of nodes. As a result, applying IMP [66] to GA workloads in GraphBIG does not infer the array-indirect DDAs.

Gretch's approach to identifying array-indirect accesses is different, and general when compared to IMP. Figure 7 shows the identification of array-indirect accesses for the ADT pair PC `A` and PC `D`. Gretch uses an unsigned integer division circuit that divides the absolute difference in

A. M. Kaushik, G. Pekhimenko, and H. Patel, "Gretch: A Hardware Prefetcher for Graph Analytics," ACM Transactions on Architecture and Code Optimization (TACO), pp. 1–25, Nov. 2020.

memory addresses of the potential consumer and absolute difference in accessed data of the potential producer. If the remainder is 0, then an array-indirect DDAs is inferred and recorded in the RT with the quotient as the offset size. *A*'s base address can then be computed using the access information in the ADT pair along with the computed offset size by rearranging Equation (1). This information is necessary to generate prefetch requests for array-indirect DDAs. The ADT pair with PCs `A` and `D` exhibit array-indirect DDAs, and is recorded in the RT as shown in Figure 7. The type is set to 0, which represents array-indirect DDA type.

5.3 Generating Prefetch Requests

Gretch first builds confidence in the RT entries populated by the training logic, and then generates prefetch requests for RT entries that satisfy a confidence threshold. The confidence of a RT entry is updated when a predicted address for the consumer, which is constructed using the information in the RT and data accessed by the producer, matches the memory address accessed by the consumer. The Conf field in a RT entry records its confidence as shown in Figure 7(b). On a cache access, Gretch compares the PC of the access with the producer PCs in the RT. On a RT hit, Gretch fills the consumer's predicted memory address field based on the DDA type. For a pointer-based DDA, Gretch predicts the consumer's memory address as the data accessed by the producer. For an array-indirect DDA, Gretch uses the base address and offset size information to construct the consumer's predicted address.

The confidence update mechanism in Gretch takes into account the impact of out-of-order instruction scheduling on data-dependent accesses. To this end, Gretch compares a predicted address in an RT entry to a window of subsequent memory addresses issued by the corresponding consumer. If the predicted address does not match any of the memory addresses issued by the corresponding consumer in this window, then the confidence of the RT entry is reduced. Otherwise, the confidence of the RT entry is increased on a match. Gretch computes a new predicted address of an RT entry when it updates the confidence of the RT entry.

Gretch generates prefetch requests for RT entries that have confidence greater than a threshold. For a strided producer, Gretch schedules prefetch requests based on the data returned by strided prefetch candidates issued by SP. Hence, Gretch's prefetch distance is equal to that of SP. Gretch also generates multi-way (one producer, multiple consumers) and multi-level (consumer of one DDAs is producer for another DDAs) prefetch requests [66].

5.4 Implementation Details

The training logic in Gretch uses a simple 64-bit wide comparison circuit to identify pointer-based accesses, and an unsigned integer division circuit to identify array-indirect accesses. In our simulation, we use a single unpipelined division circuit, which has a latency of 10 cycles and throughput of 1 operation every 10 cycles [30]. To minimize the hardware overhead, Gretch uses one division and comparison circuit for all ADT entries. Hence, Gretch trains on one ADT pair, and then trains on the next ADT pair after 10 cycles. From our empirical evaluation, we found that the performance benefits of this training approach is close to the ideal scenario where each ADT pair is trained in parallel, and there is no latency to identify DDAs. This is because DDAs feature predominantly in the L1-D cache misses of GA workloads, and once Gretch identifies a DDA relationship, it generates prefetch requests on producer accesses.

Area overhead. We evaluated different ADT and RT sizes and found that an 8-entry ADT and a 4-way 4-entry RT offered the best performance benefits across the GA workloads. We assume virtual memory address space of 48-bits and data words of 64-bits. Each ADT entry consists of three address fields (PC, Address I, and Address II in Figure 7) and two data word fields (Data I and Data

A. M. Kaushik, G. Pekhimenko, and H. Patel, "Gretch: A Hardware Prefetcher for Graph Analytics," ACM Transactions on Architecture and Code Optimization (TACO), pp. 1–25, Nov. 2020.

Table 1. GA Workloads and Graph Inputs

(a) Evaluated GA workloads

GA framework	Workloads
GraphBIG	Breadth-first search (BFS), Betweenness Centrality (BC), Depth-First Search (DFS), Graph Coloring (GC), k-Core decomposition (KC), Page-Rank (PR), Single Source Shortest Path (SSSP), Triangle Count (TC), Topological Morph (TM), Connected Components (CC), Degree Centrality (DC)
Ligra	BFS, CC, Radii Estimation (Radii), PR, BC, SSSP, BFS-based CC (BFSCC), Collaborative Filtering (CF), Maximal Independent Set (MIS), Delta stepping PR (PRD)
Graph-500 (G500)	BFS

(b) Graph inputs

Graph	Topology	$\|V\|$	$\|E\|$
Pokec [40]	Social network	1M	30M
LDBC [17]	Social network	0.1M	3M
USA-road [40]	Road network	24M	57M
Kronecker [54]	Synthetic power-law	4M	128M
Uniform [15]	Synthetic uniform	1M	33M

II in Figure 7). The hardware overhead of an 8-entry ADT is $8 \times (48 \times 3 + 64 \times 2) = 0.28$kB. Each RT entry consists of four address fields (Producer, Consumer, Base address, and Predicted address in Figure 7). The remaining fields in an RT entry take up 12 bits. The hardware overhead of a 16-entry RT (4-way 4-entry RT) is $16 \times (48 \times 4 + 12) = 0.4$kB. Comparing the combined storage overhead of Gretch (0.68 kB) with prior DDAs prefetchers such as IMP [66] and LDSP [61], IMP is 3% bigger than Gretch, and LDSP is 9.9× bigger than Gretch. We attribute Gretch's low hardware overhead to its simple ADT design due to distinct recording and training phases.

Energy overheads. We model the energy per access to Gretch's structures using CACTI [42], and assume 32 nm technology process. An access to the ADT and RT consume 0.8 and 0.5 pJ of energy, respectively, which are 2.3% and 1.4% of a 32-kB-sized L1-D cache access energy.

6 METHODOLOGY

We evaluate the performance of Gretch using the GA frameworks and workloads described in Table 1(a). We use a combination of real-world graphs derived from social networks and road networks, and synthetically generated graphs. Table 1(b) describes the configurations of the graph inputs. We prototype Gretch using the full-system gem5 micro-architectural simulator [16]. We use gem5 because it provides a detailed model of out-of-order cores, and we need this detailed model to develop a method to identify DDAs for out-of-order instruction scheduling. gem5's core and memory models have been subjected to several validation efforts that quantify the modeling errors in comparison to real hardware platforms [7, 18, 28]. One recent validation effort by Akram and Sawlha [7] showed that for dependent memory instruction execution on out-of-order cores, which is the main focus of this article, the difference in average error in instruction per cycle (IPC) values between gem5 models and real hardware platforms is less than 10%. Table 2 describes the ALPHA ISA-based multi-core system simulated in gem5. Prior works such as References [8, 46, 47, 50] have also used the ALPHA ISA support in gem5.

We collect performance metrics when the GA workload starts exploring the graph. The reported performance metrics do not take into account the time to construct or read the input graph. We run each workload for a billion instructions across all cores in our multi-core setup. We use McPAT [41] to obtain the runtime dynamic power of the simulated system with Gretch enabled. We use the following methodology to derive GA workload power consumption. We first use McPAT [41] to derive the runtime dynamic power of the simulated multi-core processor when executing GA workloads for the no prefetching configuration and with Gretch enabled. We then derive the runtime main-memory power consumption for each GA workload from the gem5 simulator, along with statistics on reads and writes to Gretch's hardware structures (ADT and RT). The runtime main-memory

A. M. Kaushik, G. Pekhimenko, and H. Patel, "Gretch: A Hardware Prefetcher for Graph Analytics," ACM Transactions on Architecture and Code Optimization (TACO), pp. 1–25, Nov. 2020.

Table 2. Simulation Framework

Parameter	Configuration
Core	Out-of-order, 4-wide issue, 128-entry ROB, ALPHA ISA
Cache	L1-I and L1-D cache: 32 kB, 2-way, 2-cycle tag access, 2-cycle data access latency, 64B cache line, 12 MSHRs per core, shared L2 cache, 8 MB, 8-way, 20-cycle tag access, 40-cycle data access, 32 MSHRs, 64-entry D-TLB
Multicore	8 cores, private L1 caches, shared L2 cache, point-to-point interconnect, snooping, MOESI cache coherence protocol
DRAM	4GB memory, 1600MHz, 2 ranks/channel, 8 banks/rank, tRC=48.75ns, tRCD=13.76ns, tRAS=35ns, tWTR=7.5ns, tRP=13.75ns, adaptive open-page policy, FR-FCFS scheduling policy, 12.8 GB/sec bandwidth
Prefetcher	128 entry prefetch queue, IMP: 4 entry IPDT, 16 entry PT, LDSP: 128-entry PPWT, 256-entry CT, Gretch: 8-entry ADT, 4-set 4-way RT, 10-cycle division latency

power consumption takes into account additional memory accesses due to Gretch (inaccurate prefetches and write-backs). Finally, we use the energy estimates of the ADT and RT presented in Section 5.4 to compute the total power consumption of GA workloads with Gretch enabled.

The prefetcher configurations are listed in Table 2. We compare Gretch against SP [21], IMP [66] that identifies array-indirect accesses, and the LDS prefetcher (LDSP) [61] that identifies pointer-based accesses. In IMP, the indirect pattern detector table (IPDT) computes base address candidates, and the prefetch table (PT) records the memory instructions exhibiting array-indirect accesses. In LDSP, the potential producer window (PPW) records access history of memory instructions, and the correlation table (CT) records pointer-based relationships. We evaluate Gretch with an 8-entry ADT and a 4-entry 4-way RT. All the evaluated prefetchers generate prefetch candidates with a prefetch depth of 4. For Gretch and LDSP, pointer-based prefetch candidates are generated with a prefetch width of 2. All the data-dependent prefetchers are equipped with a SP [21] of 64 entries that identifies strided memory access patterns. All the evaluated prefetchers prefetch into the private L1-D caches on cache misses and prefetch hits.

We are aware of several recent address-based prefetchers such as References [35, 36, 49, 59] that have been proposed to improve the identification of complex address patterns. These prefetchers perform better than the conventional SP as they can identify other patterns in the memory addresses such as repeating sequences of multiple strides, and correlating addresses in addition to regular strided access patterns. DD accesses in GA typically do not exhibit repeating sequences of strided accesses and address correlation patterns. This is because nodes in real-world graphs have varying degrees of connections, and nodes have different neighbor connections. As a result, these prefetchers are insufficient to capture the DD accesses in GA workloads, and their performance benefits are equivalent to SP as they can only identify and optimize strided neighbor accesses. A recent in-depth characterization of GA workloads on micro-architecture by Basak et al. [12] showed that the performance offered by address correlation prefetchers such as References [56, 63] was on par with that offered by SP. Hence, we compare Gretch against SP, and do not evaluate Gretch against these recent address-based prefetchers.

7 RESULTS

We evaluate Gretch using (1) execution time speedup (Section 7.1), (2) prefetch metrics such as prefetch accuracy and coverage (Section 7.2), (3) overall power consumption (Section 7.3), and (4) performance sensitivity to ADT size, and graph configurations (Section 7.4). Unless specified,

A. M. Kaushik, G. Pekhimenko, and H. Patel, "Gretch: A Hardware Prefetcher for Graph Analytics," ACM Transactions on Architecture and Code Optimization (TACO), pp. 1–25, Nov. 2020.

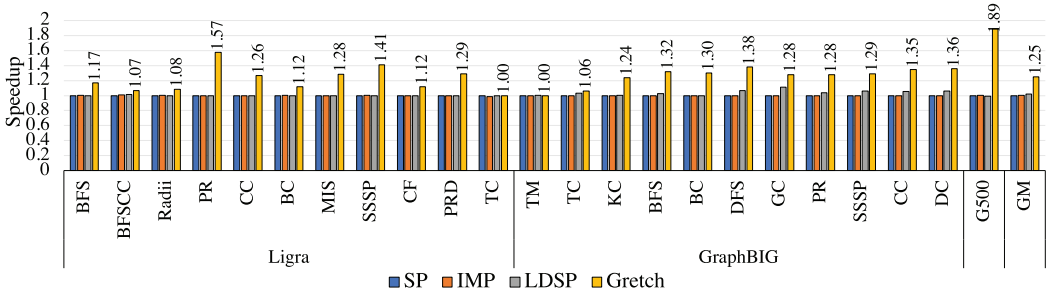


Fig. 9. Performance speedup across different GA workloads and frameworks.

GA workloads on Ligra operate on the Pokec social network graph, GA workloads on GraphBIG operate on the LDBC social network graph, and Graph-500 operates on the Kronecker graph. For each evaluation criterion, we discuss observations for each GA framework with respect to the memory access patterns they exhibit. We evaluate the effectiveness of Gretch and other DDA-specific prefetchers across *all* GA workloads and frameworks listed in Table 1. Due to space constraints, we present only the speedup results for all workloads, and present the remaining evaluation metrics for the BFS, CC, PR, and SSSP workloads.

7.1 Speedup

Figure 9 shows the performance speedup of Gretch and different DDA-specific prefetchers compared to the baseline that has SP.

General observations. (1) Gretch provides an average speedup of 25% (up to 89%) over SP and IMP, and 20% (up to 89%) over LDSP. This is because Gretch can identify different DDAs resulting in performance speedups across different GA frameworks. (2) Gretch does not improve performance over SP for TC and TM GA workloads. This is because these workloads do not operate on node properties resulting in no DDAs. Recall from Figure 2 that TC and TM workloads across Ligra and GraphBIG frameworks exhibit low or no DDA contribution to the off-chip memory accesses. (3) GA workloads such as PR and CC exhibit higher performance benefits with Gretch than other GA workloads. This is because these workloads are *active-all* workloads where all the nodes in the graph are visited and their properties are computed upon. Hence, the prefetch coverage (Section 7.2) of Gretch on these workloads are high. However, workloads such as BFS and BC explore a select number of nodes in the graph based on the choice of source node resulting in lower prefetch coverage as shown in Section 7.2. We next describe GA-framework-specific performance observations.

Ligra and Graph-500. GA workloads in Ligra and Graph-500 exhibit array-indirect DDAs as they operate on the CSR graph representation (Figure 2). Gretch shows 18% average performance speedup (up to 57%) over SP and IMP for Ligra, and 89% performance speedup over SP and IMP on Graph-500. LDSP does not provide performance benefits over SP as property accesses in Ligra do not exhibit pointer-based DDAs. As a result, LDSP's performance benefits are solely from the attached SP that identifies strided accesses. The array-indirect DDAs in Ligra and Graph-500 operate on commonly used data types, and can be identified by IMP [66]. However, our evaluation shows that IMP provides little to no performance speedup over SP for workloads on Ligra and the Graph-500 benchmark. In Section 7.1.1, we perform additional experiments and provide a detailed analysis of IMP's inability to identify array-indirect DDAs in the presence of out-of-order instruction scheduling.

A. M. Kaushik, G. Pekhimenko, and H. Patel, "Gretch: A Hardware Prefetcher for Graph Analytics," ACM Transactions on Architecture and Code Optimization (TACO), pp. 1–25, Nov. 2020.

GraphBIG. GA workloads in GraphBIG exhibit both array-indirect and pointer-based DDAs. Gretch identifies both these DDAs, and delivers 28% average performance speedup (up to 35%) over SP and IMP, and 20% average performance speedup (up to 30%) over LDSP. The array-indirect DDAs in GraphBIG are to custom sized data structures that IMP cannot identify. Hence, IMP does not improve over SP. However, Gretch's division circuit captures the array-indirect DDAs in GraphBIG, and generates appropriate prefetch requests. LDSP captures the pointer-based DDAs resulting in an average performance improvement of 4% (up to 11%) over SP. However, LDSP cannot capture array-indirect DDAs resulting in missed opportunities to further improve GA performance.

7.1.1 Analyzing Impact of Out-of-Order Instruction Scheduling on IMP. Although IMP is designed to identify array-indirect DDAs, we observed that it provides little to no performance speedup over SP. This is because IMP's mechanism to update confidence of an identified DDA is hinged on in-order instruction scheduling. In particular, we observed that IMP identifies the array-indirect DDA relationship in Ligra and G500.[2] However, it does not generate enough confidence in the identified array-indirect DDA, which results in missed opportunities for IMP to generate prefetch requests. This is because IMP's mechanism to adjust the confidence in the identified array-indirect DDA relationships is also tightly coupled with in-order instruction execution [66]. However, as described in Section 3.2, the interaction of DDAs accesses and out-of-order instruction scheduling renders identification techniques that work for in-order instruction scheduling ineffective. As a result, IMP's performance benefits are solely due to strided prefetch candidates issued by SP.

To confirm this observation regarding IMP, we perform two experiments. In the first experiment, we executed the Graph-500 workload on a multi-core simulation configuration with *in-order cores*. Graph-500 was used in the IMP evaluation [66], and is publicly available. The cache hierarchy and prefetcher configurations remain unchanged. We found that IMP and Gretch exhibited similar performance improvements (2× performance improvement over SP) with in-order cores. This is because in-order instruction scheduling does not allow independent producer accesses to access the cache hierarchy ahead of the corresponding consumer accesses. As a result, the prefetchers observe the array-indirect DDAs as a sequence of one producer access followed by the corresponding consumer access, which aligns with IMP's identification and confidence update mechanism.

In the second experiment, we increase the hardware table sizes in IMP, and change the confidence update mechanism in our IMP implementation to a window-based mechanism used in Gretch (Section 5.3). In this experiment, we executed the Graph-500 workload on the multi-core simulation configuration described in Table 2 with out-of-order cores. There are three main hardware tables in IMP: (1) indirect pattern detector table (IPDT), (2) prefetch table (PT), and (3) the base address table [66]. For an array-indirect DDAs $A[B[i]]$, (1) computes possible base addresses of A using predefined offset sizes of $B[i]$ and the data of $B[i]$, and (2) stores the identified array-indirect DDAs, and generates prefetch array-indirect prefetch candidates. Each entry in (1) consists of (3), which stores the computed-based addresses. We use the notation $\langle IPDT, PT, BA \rangle$ to denote the table sizes in IMP. For example, $\langle 4, 16, 4 \rangle$ describes an IMP configuration with IPDT size of 4 entries, PT size of 16 entries, and the base address table size of 4 entries.

Figure 10 shows the speedup of different IMP configurations over the baseline using the original confidence update mechanism described in Reference [66], and with the window-based confidence mechanism described in Section 5.3. We make two key observations. First, increasing the sizes of

[2]IMP identifies array-indirect DDA relationships when the strided access $B[i]$ is a prefetch hit, and the address translation of the subsequent $A[B[i]]$ access is available in the TLB (TLB hit).

A. M. Kaushik, G. Pekhimenko, and H. Patel, "Gretch: A Hardware Prefetcher for Graph Analytics," ACM Transactions on Architecture and Code Optimization (TACO), pp. 1–25, Nov. 2020.

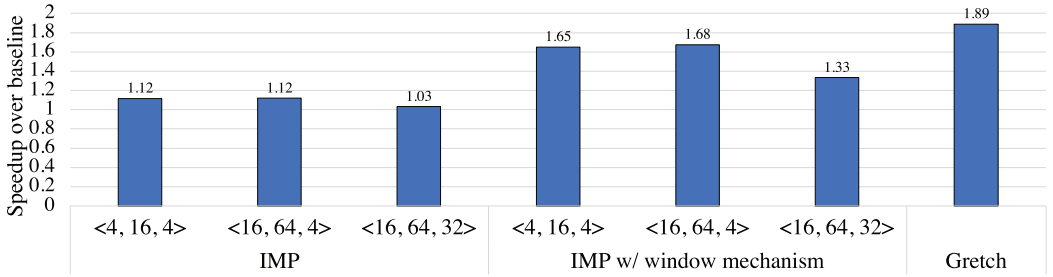


Fig. 10. Speedup of different IMP [66] configurations on Graph-500 benchmark.

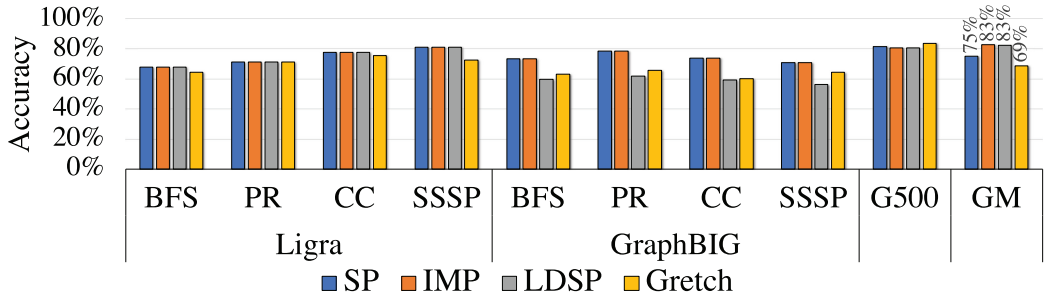


Fig. 11. Prefetch accuracy.

the hardware tables in the original IMP design does not offer the same performance improvement over the baseline compared to Gretch. For example, the configuration ⟨16, 64, 4⟩ quadruples the IPDT and PT sizes used in Reference [66]. We observe that this configuration offers the same speedup (12% over the baseline) compared to the ⟨4, 16, 4⟩ configuration, which we used in Figure 9. Second, changing the confidence update mechanism to a window-based mechanism in IMP improves IMP's performance over the baseline across different configurations. The ⟨4, 16, 4⟩ configuration in the modified IMP implementation offers 65% performance improvement over the baseline. This is because the window-based confidence mechanism allows IMP to build confidence in the identified array-indirect DDAs in the presence of out-of-order instruction scheduling, and generate prefetch candidates. However, the performance of the updated IMP implementation still falls short of that provided by Gretch (89%). Unlike Gretch, IMP uses any producer access to identify array-indirect DDAs accesses. As shown in Section 3.2 (Figure 4), such a technique has low success rate in identifying array-indirect DDAs accesses. As a result, IMP takes longer to identify the array-indirect accesses compared to Gretch, resulting in its lower performance benefits compared to Gretch.

7.2 Prefetch Efficiency

We measure prefetch efficiency using two common metrics: prefetch accuracy (Figure 11) and prefetch coverage (Figure 12). These metrics have been used in prior works to evaluate prefetch efficiency [49, 58, 63, 66]. From Figure 11, Gretch achieves an average prefetch accuracy of 69% (up to 81%). However, SP, IMP, and LDSP achieve an average prefetch accuracy of 75%, 75%, and 68%, respectively. We make two observations regarding Gretch's prefetch accuracy. First, active-all GA workloads such as PR and CC register high prefetch accuracy with Gretch (>70% for Ligra) as all nodes in the graph and their properties are explored on every iteration.

A. M. Kaushik, G. Pekhimenko, and H. Patel, "Gretch: A Hardware Prefetcher for Graph Analytics," ACM Transactions on Architecture and Code Optimization (TACO), pp. 1–25, Nov. 2020.

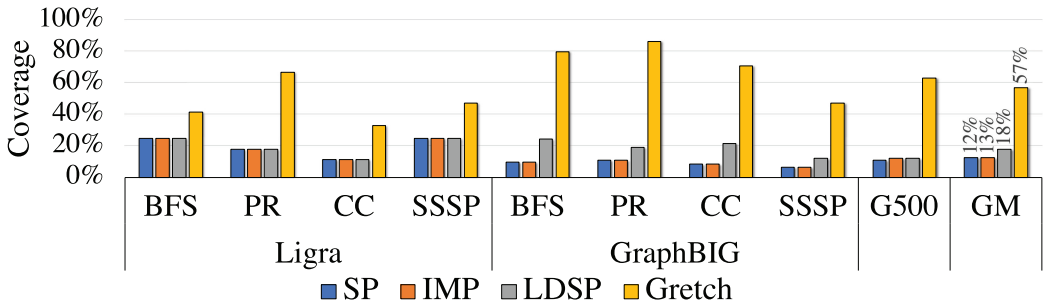


Fig. 12. Prefetch coverage.

Second, workload such as BFS on Ligra register lower prefetch accuracy with Gretch (65%). This BFS implementation uses the direction-optimized BFS approach [13] that reduces the number of neighbors visited for a node based on the graph exploration state. As a result, SP generates prefetch requests to neighbors that may not be visited, and Gretch generates unused prefetch requests to properties of these unexplored neighbors resulting in unused prefetches. The BFS implementation in Graph-500 does not implement this optimization, and hence, SP and Gretch's prefetch accuracy is high (81%).

Figure 12 shows the prefetch coverage of the evaluated prefetchers. Gretch achieves an average prefetch coverage of 56% (up to 86%). SP, IMP, and LDSP achieve an average prefetch coverage of 13%, 13%, and 18%, respectively. Across all workloads and frameworks, Gretch achieves a higher prefetch coverage compared to other prefetchers as it identifies different DDAs. The Ligra framework maintains several intermediate data structures to activate different run-time optimizations that improve GA workload performance. These accesses are not captured by SP and Gretch resulting in lower coverage. However, GA workloads executed on GraphBIG and the Graph-500 benchmark do not implement run-time optimizations. As a result, the DDAs identified and optimized by Gretch make up the bulk of the off-chip memory accesses resulting in higher coverage. IMP's prefetch coverage is close to SP as it cannot identify the DDAs in Ligra as described in Section 7.1.1, and the array-indirect DDAs in GraphBIG are to custom data structures. LDSP achieves higher prefetch coverage (19%) than SP (9%) for the GraphBIG workloads as it identifies the pointer-based DDAs. However, LDSP's coverage is lower than that of Gretch as it cannot identify the array-indirect DDAs in GraphBIG workloads. Recall that Gretch, IMP and LDSP generate virtual addresses of prefetch requests (Section 4). As a result, these prefetch requests are subjected to address translation prior to bringing in the data contents of the prefetch request. Gretch, IMP, and LDSP do not optimize for the memory requests involved in the address translation mechanism (page walk) resulting in lower prefetch coverage.

Figure 13 shows the increase in off-chip memory accesses due to prefetching for the different prefetchers compared to no prefetching. Gretch increases off-chip memory accesses by 13% (up to 30%) over no prefetching and 4% (up to 19%) over SP. This increase in off-chip memory accesses is due to the inaccurate DDA prefetch requests generated by Gretch as described earlier. We find this increase in off-chip memory accesses due to Gretch acceptable for the performance benefits it provides as described in Section 7.1.

7.3 Power Consumption

Figure 14 shows the per-core power consumption of GA workloads with Gretch compared to no prefetching using McPAT [41]. The power consumption encompasses core pipeline activity and L1-D cache activity. For Gretch, we compute the additional power consumption due to ADT

A. M. Kaushik, G. Pekhimenko, and H. Patel, "Gretch: A Hardware Prefetcher for Graph Analytics," ACM Transactions on Architecture and Code Optimization (TACO), pp. 1–25, Nov. 2020.

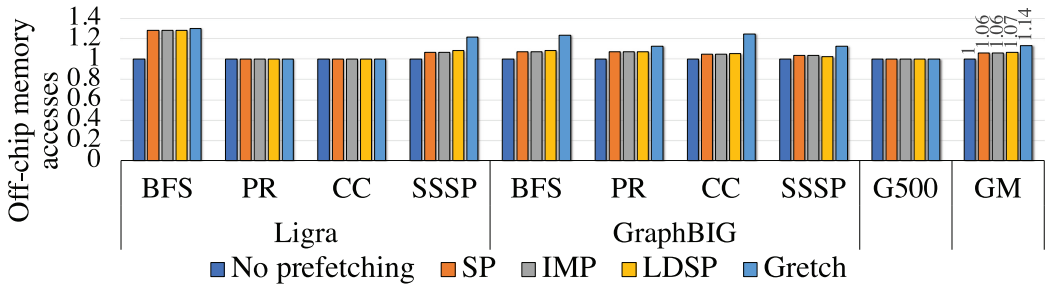


Fig. 13. Off-chip memory accesses.

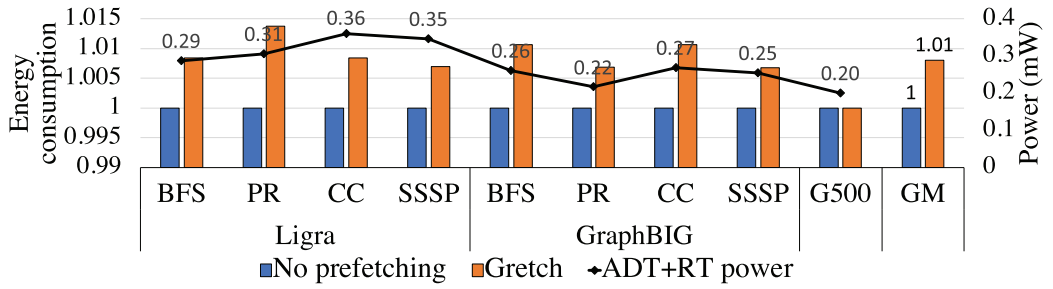


Fig. 14. Energy consumption.

population and RT management using the access energy estimated in Section 5.4. Gretch increases average power consumption by 1% on average. This low-power overhead of Gretch is attributed to the ADT population mechanism that is triggered on receiving communication from the SP regarding a stride deviation. Hence, the ADT is populated only on stride changes, and remains idle otherwise resulting in low power consumption. Figure 14 also shows the power consumption of the ADT and RT management. The RT management contributes to nearly 90% of the total power dissipation of Gretch's structures as it is referenced on every L1-D cache miss or prefetch hit.

7.4 Sensitivity Studies

ADT size. Choosing the right ADT size is dependent on (1) number of DDA relationships that constitute a node's property access and (2) the number of independent memory accesses that can be interleaved between a DDA due to available MLP. A large ADT can collect the required access information of a DDA in the presence of interleaved independent memory accesses. As a result, a large ADT will collect more access information, and the training logic can infer the necessary DDAs. However, a large ADT incurs longer recording and training latencies as the training logic creates more ADT pairs to train on. However, a small ADT collects less access information, and the training logic trains on fewer ADT pairs resulting in lower recording and training latencies. However, a small ADT may not be able to collect the necessary access information of a DDA in the presence of multiple interleaving independent memory accesses. Hence, the performance of GA workloads on different GA frameworks may have varying requirements on the ADT size.

Figure 15 shows the performance sensitivity of Gretch to different ADT sizes: 4, 8, and 16 entries. We normalize the performance to the 4-entry ADT. For GA workloads on GraphBIG, 4-entry ADT does not perform better than 8-entry ADT. This is because GA workloads on GraphBIG comprise of two levels of DDAs (array-indirect followed by pointer-based), which requires at least 3 ADT entries to capture the DDAs. As a result, a 4-entry ADT is sufficient to capture the necessary

A. M. Kaushik, G. Pekhimenko, and H. Patel, "Gretch: A Hardware Prefetcher for Graph Analytics," ACM Transactions on Architecture and Code Optimization (TACO), pp. 1–25, Nov. 2020.

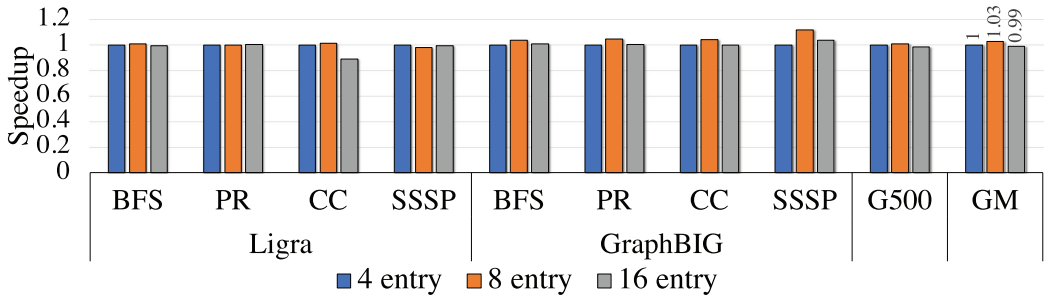


Fig. 15. Sensitivity to ADT size.

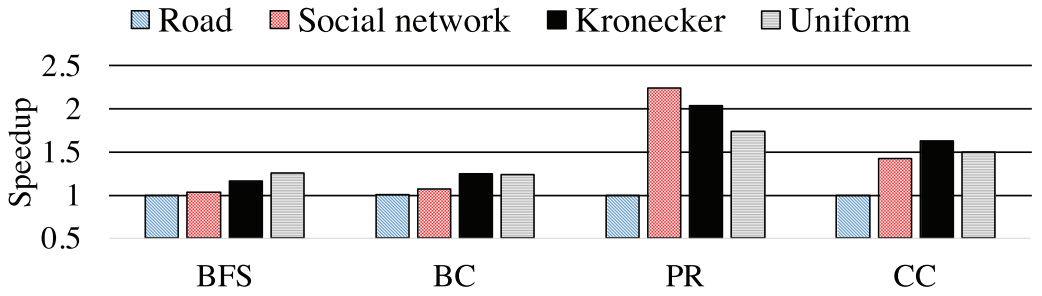


Fig. 16. Effect of graph topology.

access information to correctly infer the DDAs in GraphBIG workloads. For Ligra and Graph-500, 4-entry ADT offers similar performance benefits compared to 8-entry ADT. This is because these workloads feature one DDA type for node property accesses. As a result, a 4-entry ADT has enough entries to accommodate the DDAs and any interleaved independent memory accesses. 16-entry ADT captures all the DDAs across GA workloads as there are enough entries However, the 16-entry ADT incurs higher training latency overhead as more ADT pairs are constructed and trained upon resulting in 3% performance slowdown compared to the 8-entry ADT.

Graph topology. Graph topology defines the connectivity of nodes in a graph. Figure 16 shows the performance of Gretch on the graph topologies described in Table 1(b). For this sensitivity study, we use the Ligra framework as it provides a simple interface to load different graph inputs.[3]

We observe that for graphs following power law degree distribution (Kronecker and social network), Gretch shows better performance benefits over SP. For social network graphs (LDBC), Gretch provides an average speedup of 37% over SP, and a maximum of 2.23× speedup. For uniform graphs, Gretch provides an average speedup of 41% over SP, and a maximum of 73% over SP. However, for road networks, Gretch does not provide any performance benefits over SP. This is because the road network consists of nodes with low number of neighbors (average node degree is 2.4). Hence, exploring the node's neighbors does not build enough confidence in the SP to generate strided prefetch requests, and cannot detect stride deviations for the memory instruction responsible for exploring a node's neighbors. As a consequence, Gretch does not initiate ADT population as it does not receive any communication from SP resulting in no DDA prefetch requests.

Graph pre-processing. Graph pre-processing techniques change the graph layout of an input graph to improve the cache locality of node property accesses [11]. We apply a recent

[3]We cannot represent the above graphs in GraphBIG due to memory limitations on our simulated system. However, we use scaled down versions of the graphs that fit in memory for GraphBIG, and note similar observations.

A. M. Kaushik, G. Pekhimenko, and H. Patel, "Gretch: A Hardware Prefetcher for Graph Analytics," ACM Transactions on Architecture and Code Optimization (TACO), pp. 1–25, Nov. 2020.

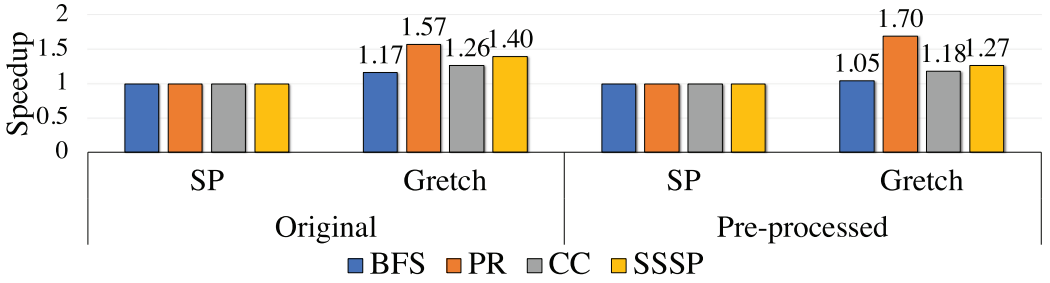


Fig. 17. Sensitivity to graph pre-processing.

pre-processing technique, HubSort [11], on the Pokec social network graph, and evaluate Gretch's performance on this pre-processed graph. HubSort improves the spatial locality primarily for large degree nodes in power law graphs [11]. Figure 17 shows Gretch's performance benefits normalized to the execution time with SP enabled. Benchmarks such as BFS and SSSP that explore the connections of high degree nodes benefit from pre-processing. These benchmarks also benefit from Gretch's prefetching as pre-processing does not completely eliminate DDAs. However, Gretch's performance benefits are lower on pre-processed graphs for these workloads (4% for BFS and 18% for SSSP) compared to the original graphs that are not subjected to any pre-processing (16% for BFS and 26% for SSSP). The PR and CC workloads on pre-processed graphs equally benefit from Gretch (69% for PR and 26% for SSSP) compared to those obtained on the original graph. This is because PR and CC operate on all nodes of the graph, and hence, there is an abundance of DDAs.

8 RELATED WORK

We discuss prior works in data prefetching techniques and GA-specific on-chip accelerators [4, 12, 53, 67]. While there is a large volume of prior research in prefetching techniques such as software prefetching [5, 34, 44, 52, 62], and hardware prefetching [4, 6, 10, 21, 22, 24, 31–33, 49, 56, 58, 59, 61, 63, 66], we restrict our discussion to prefetching techniques that improve DDAs [6, 22, 24, 61, 66].

DDAs prefetchers. Roth et al. [61] proposed a hardware prefetch mechanism for identifying pointer-based accesses in linked data structures (LDSP). Linked data structures consists of pointers that represent links between objects, and exploration of LDS uses these pointers. The design in Reference [61] recorded access histories, and identified pointer-based accesses by comparing the referenced addresses with the data recorded from prior memory instructions. Cooksey et al. [22] proposed the content-directed prefetcher (CDP) that identified pointers present in the data. CDP identified pointers by comparing the higher order bits of the data with the address that brought in the data. For array indirect accesses $A[B[i]]$, Yu et al. [66] proposed the indirect memory prefetcher (IMP). IMP used the value stored in $B[i]$ to compute different base addresses of A using sizes of commonly observed array data types. The data types considered by IMP were restricted to reduce the hardware overhead [66]. Peled et al. [58] proposed a compiler assisted hardware prefetcher design that used reinforcement learning to identify different access patterns, which includes DDAs. However, their approach required feedback from performance counters, and compiler hints to identify memory access patterns, and is expensive in terms of hardware overhead. Recently, Cavus et al. [20] proposed a hardware-software approach to identify and prefetch array-indirect and pointer-based data-dependent accesses. Their approach modified the application to add instructions to communicate data structure knowledge, and this knowledge is communicated to the prefetcher, which identified data-dependent accesses. On the contrary,

A. M. Kaushik, G. Pekhimenko, and H. Patel, "Gretch: A Hardware Prefetcher for Graph Analytics," ACM Transactions on Architecture and Code Optimization (TACO), pp. 1–25, Nov. 2020.

our approach does not modify the application, and identifies data-dependent accesses in GA workloads using a purely hardware approach.

GA-specific on-chip accelerators. Ainsworth and Jones [4] proposed a GA-specific hardware prefetcher that operated on CSR graph representation. This prefetcher required application changes to convey information about the address ranges of the data structures used in the graph representation to accurately identify and optimize the array-indirect DDAs. Zhang et al. [67] recently proposed a programmable GA-specific accelerator, Minnow, that leveraged the information in GA worklist data structures that store nodes ready to be explored to generate prefetch candidates. Mukkara et al. [53] proposed an on-chip accelerator, HATS, that changed the schedule of nodes explored based on the cache occupancy of the graph. The schedule prioritizes exploration of nodes that have more neighbors currently in the cache hierarchy. Basak et al. [12] proposed a GA-specific prefetcher, DROPLET, that was located along with the memory controller, and issued DDA prefetch requests. Minnow, HATS and DROPLET were designed for CSR representations, and required communication from the framework regarding information about the base address and offset size to generate DDA prefetch requests [12, 53, 67]. Gretch, however, works for different graph representations including CSR, and does not require any communication from the GA framework to identify and generate prefetch requests.

We note that there are several hardware accelerators for GA [3, 9, 29, 57]. Gretch is orthogonal to these works, and can be used in these accelerators as DDAs still feature even for these accelerators [29].

9 CONCLUSION

We present Gretch, a hardware prefetcher for GA that identifies different DDAs. There are two key design novelties of Gretch. First, Gretch identifies different DDAs using a unified set of hardware structures. This allows Gretch to deliver performance benefits across different graph representations. Second, Gretch uses the interaction of memory accesses issued by GA workloads and a conventional stride prefetcher to accurately identify DDAs in the presence of out-of-order instruction scheduling. As a result, Gretch does not require any hardware-software communication from the framework to guide its operation. Our evaluation shows that Gretch provides an average performance improvement of 37% over no prefetching, 25% over SP, and 22% over the best DDA-specific prefetcher across different GA workloads and frameworks with only 1% increase in power consumption compared to no prefetching.

REFERENCES

[1] Neo4j [n.d.]. Neo4j graph database. Retrieved from http://neo4j.com/.

[2] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan. 2015. CRONO: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'15)*. IEEE, 44–55.

[3] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. ACM, 105–117.

[4] Sam Ainsworth and Timothy M. Jones. 2016. Graph prefetching using data structure knowledge. In *Proceedings of the International Conference on Supercomputing (ICS'16)*. ACM, 1–11.

[5] S. Ainsworth and T. M. Jones. 2017. Software prefetching for indirect memory accesses. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO'17)*. IEEE/ACM, 305–217.

[6] Sam Ainsworth and Timothy M. Jones. 2018. An event-triggered programmable prefetcher for irregular workloads. InProceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18). ACM, 578–592.

[7] Ayaz Akram and Lina Sawalha. 2019. Validation of the gem5 simulator for x86 architectures. In *Proceedings of the Conference on IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS'19)*. IEEE, 53–58.

A. M. Kaushik, G. Pekhimenko, and H. Patel, "Gretch: A Hardware Prefetcher for Graph Analytics," ACM Transactions on Architecture and Code Optimization (TACO), pp. 1–25, Nov. 2020.

[8] L. M. AlBarakat, P. V. Gratz, and D. A. Jimenez. 2018. MTB-fetch: Multithreading aware hardware prefetching for chip multiprocessors. *IEEE Comput. Architect. Lett.* (2018), 175–178.

[9] M. J. Anderson, N. Sundaram, N. Satish, M. M. A. Patwary, T. L. Willke, and P. Dubey. 2016. GraphPad: Optimized graph primitives for parallel and distributed platforms. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'16)*. IEEE, 313–322.

[10] M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad. 2018. Domino temporal data prefetcher. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*. IEEE, 131–142.

[11] V. Balaji and B. Lucia. 2018. When is graph reordering an optimization? Studying the effect of lightweight graph reordering across applications and input graphs. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'18)*. IEEE, 203–214.

[12] Abanti Basak, Shuangchen Li, Xing Hu, Sang Min Oh, Xinfeng Xie, Li Zhao, Xiaowei Jiang, and Yuan Xie. 2019. Analysis and optimization of the memory hierarchy for graph processing workloads. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'19)*. IEEE, 373–386.

[13] Scott Beamer, Krste Asanovic, and David Patterson. 2012. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*. IEEE, 1–10.

[14] Scott Beamer, Krste Asanovic, and David Patterson. 2015. Locality exists in graph processing: Workload characterization on an ivy bridge server. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'15)*. IEEE, 56–65.

[15] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. Retrieved from https://arXiv:1508.03619.

[16] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH Comput. Architect. News* 39, 2 (2011), 1–7.

[17] Peter Boncz. 2013. LDBC: Benchmarks for graph and RDF data management. In *Proceedings of the 17th International Database Engineering and Applications Symposium*. ACM, 1–2.

[18] Anastasiia Butko, Rafael Garibotti, Luciano Ost, and Gilles Sassatelli. 2012. Accuracy evaluation of gem5 simulator system. In *Proceedings of the International Workshop on Reconfigurable and Communication-centric Systems-on-chip (ReCoSoC'12)*. IEEE, 1–7.

[19] Mustafa Canim and Yuan-Chi Chang. 2013. System G data store: Big, rich graph data analytics in the cloud. In *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E'13)*. IEEE, 328–337.

[20] Mustafa Cavus, Resit Sendag, and Joshua J. Yi. 2020. Informed prefetching for indirect memory accesses. *ACM Trans. Architect. Code Optimiz.* (2020), 1–29.

[21] Tien-Fu Chen and Jean-Loup Baer. 1995. Effective hardware-based data prefetching for high-performance processors. *IEEE Trans. Comput.* 44, 5 (1995), 609–623.

[22] Robert Cooksey, Stephan Jourdan, and Dirk Grunwald. 2002. A stateless, content-directed data prefetching mechanism. *ACM SIGPLAN Notices* 37, 10, 279–290.

[23] James Davidson, Benjamin Liebald, Junning Liu, Palash Nandy, Taylor Van Vleet, Ullas Gargi, Sujoy Gupta, Yu He, Mike Lambert, Blake Livingston, et al. 2010. The YouTube video recommendation system. In *Proceedings of the 4th ACM Conference on Recommender Systems*. 293–296.

[24] Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt. 2009. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *Proceedings of the IEEE 15th International Symposium on High Performance Computer Architecture (HPCA'09)*. IEEE, 7–17.

[25] D. Ediger, R. McColl, J. Riedy, and D. A. Bader. 2012. STINGER: High-performance data structure for streaming graphs. In *Proceedings of the IEEE Conference on High Performance Extreme Computing*. IEEE, 1–5.

[26] Assaf Eisenman, Lucy Cherkasova, Guilherme Magalhaes, Qiong Cai, and Sachin Katti. 2016. Parallel graph processing on modern multi-core servers: New findings and remaining challenges. In *Proceedings of the IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'16)*. IEEE, 49–58.

[27] Facebook. 2013. Introducing Graph Search Beta. Retrieved from https://newsroom.fb.com/news/2013/01/introducing-graph-search-beta/.

[28] Anthony Gutierrez, Joseph Pusdesris, Ronald G. Dreslinski, Trevor Mudge, Chander Sudanthi, Christopher D. Emmons, Mitchell Hayenga, and Nigel Paver. 2014. Sources of error in full-system simulation. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS'14)*. IEEE, 13–22.

[29] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. IEEE, 1–13.

A. M. Kaushik, G. Pekhimenko, and H. Patel, "Gretch: A Hardware Prefetcher for Graph Analytics," ACM Transactions on Architecture and Code Optimization (TACO), pp. 1–25, Nov. 2020.

[30] Intel. 2016. Intel 64 and IA-32 architectures optimization reference manual (Section 12.1. 1), 2014. Retrieved from http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[31] Akanksha Jain and Calvin Lin. 2013. Linearizing irregular memory accesses for improved correlated prefetching. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'13)*. 247–259.

[32] Victor Jiménez, Roberto Gioiosa, Francisco J. Cazorla, Alper Buyuktosunoglu, Pradip Bose, and Francis P. O'Connell. 2012. Making data prefetch smarter: Adaptive prefetching on POWER7. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. ACM, 137–146.

[33] Norman P. Jouppi. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *ACM SIGARCH Comput. Architect. News* 18, 2SI (1990), 364–373.

[34] Magnus Karlsson, Fredrik Dahlgren, and Per Stenstrom. 2000. A prefetching technique for irregular accesses to linked data structures. In *Proceedings of the 6th International Symposium on High-Performance Computer Architecture (HPCA'00)*. IEEE, 206–217.

[35] Jinchun Kim, Seth H. Pugsley, Paul V. Gratz, A. L. Narasimha Reddy, Chris Wilkerson, and Zeshan Chishti. 2016. Path confidence based lookahead prefetching. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. IEEE, 1–12.

[36] Jinchun Kim, Elvira Teran, Paul V. Gratz, Daniel A. Jiménez, Seth H. Pugsley, and Chris Wilkerson. 2017. Kill the program counter: Reconstructing program behavior in the processor cache hierarchy. *ACM SIGPLAN Notices* 52, 4 (2017), 737–749.

[37] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. 2007. Optimistic parallelism requires abstractions. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*. ACM, 211–222.

[38] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*. ACM, 591–600.

[39] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. 2016. Parallel graph analytics. *Commun. ACM* 59, 5 (2016), 78–87.

[40] Jure Leskovec and Rok Sosič. 2016. SNAP: A General-Purpose Network Analysis and Graph-Mining Library. *ACM Trans. Intell. Syst. Technol.* 8, 1, Article 1 (2016), 20 pages. https://doi.org/10.1145/2898361

[41] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*. ACM, 469–480.

[42] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. 2011. CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'11)*. IEEE, 694–701.

[43] Greg Linden, Brent Smith, and Jeremy York. 2003. Amazon. com recommendations: Item-to-item collaborative filtering. *IEEE Internet Comput.* 7, 1 (2003), 76–80.

[44] Chi-Keung Luk. 2001. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings 28th Annual International Symposium on Computer Architecture (ISCA'01)*. IEEE, 40–51.

[45] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. 2007. Challenges in parallel graph processing. *Parallel Process. Lett.* 17, 01 (2007), 5–20.

[46] J. Luo, H. Cheng, I. Lin, and D. Chang. 2019. TAP: Reducing the energy of asymmetric hybrid last-level cache via thrashing aware placement and migration. *IEEE Trans. Comput.* (2019), 1704–1719.

[47] P. M. Yaghini, G. Michelogiannakis, and P. V. Gratz. 2019. SpecLock: Speculative lock forwarding. In *Proceedings of the International Conference on Computer Design (ICCD'19)*. IEEE, 273–282.

[48] Vaibhav Mehta, Constantinos Bartzis, Haifeng Zhu, Edmund Clarke, and Jeannette Wing. 2006. Ranking attack graphs. In *Proceedings of the International Workshop on Recent Advances in Intrusion Detection*. Springer, 127–144.

[49] Pierre Michaud. 2016. Best-offset hardware prefetching. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*. IEEE, 469–480.

[50] G. Michelogiannakis and J. Shalf. 2017. Last level collective hardware prefetching for data-parallel applications. In *Proceedings of the International Conference on High Performance Computing (HiPC'17)*. IEEE, 72–83.

[51] Rada Mihalcea and Dragomir Radev. 2011. *Graph-based Natural Language Processing and Information Retrieval*. Cambridge University Press.

[52] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. 1992. Design and evaluation of a compiler algorithm for prefetching. *ACM Sigplan Notices* 27, 9, 62–73.

[53] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. 2018. Exploiting locality in graph analytics through hardware-accelerated traversal scheduling. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'18)*. IEEE, 1–14.

A. M. Kaushik, G. Pekhimenko, and H. Patel, "Gretch: A Hardware Prefetcher for Graph Analytics," ACM Transactions on Architecture and Code Optimization (TACO), pp. 1–25, Nov. 2020.

[54] Richard C. Murphy, Kyle B. Wheeler, Brian W. Barrett, and James A. Ang. 2010. Introducing the graph 500. *Cray Users Group (CUG)*.

[55] Lifeng Nai, Yinglong Xia, Ilie G. Tanase, Hyesoon Kim, and Ching-Yung Lin. 2015. GraphBIG: Understanding graph computing in the context of industrial solutions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*. IEEE, 1–12.

[56] Kyle J. Nesbit and James E. Smith. 2004. Data cache prefetching using a global history buffer. In *Proceedings of the10th International Symposium on High Performance Computer Architecture (HPCA'04)*. IEEE, 96–96.

[57] Muhammet Mustafa Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven Burns, and Ozcan Ozturk. 2016. Energy efficient architecture for graph analytics accelerators. *ACM SIGARCH Comput. Architect. News* 44, 3 (2016), 166–177.

[58] Leeor Peled, Shie Mannor, Uri Weiser, and Yoav Etsion. 2015. Semantic locality and context-based prefetching using reinforcement learning. In *Proceedings of the ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. IEEE, 285–297.

[59] Seth H. Pugsley, Zeshan Chishti, Chris Wilkerson, Peng-fei Chuang, Robert L. Scott, Aamer Jaleel, Shih-Lien Lu, Kingsum Chow, and Rajeev Balasubramonian. 2014. Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers. In *Proceedings of the IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14)*. IEEE, 626–637.

[60] S. Ravi. 2016. Graph-powered Machine Learning at Google.

[61] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. 1998. Dependence based prefetching for linked data structures. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)*. 115–126.

[62] A. Roth and G. S. Sohi. 1999. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA'99)*. ACM, 111–121.

[63] Manjunath Shevgoor, Sahil Koladiya, Rajeev Balasubramonian, Chris Wilkerson, Seth H. Pugsley, and Zeshan Chishti. 2015. Efficiently prefetching complex address patterns. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'15)*. IEEE, 141–152.

[64] Julian Shun and Guy E. Blelloch. 2013. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 135–146.

[65] Stuart Staniford-Chen, Steven Cheung, Richard Crawford, Mark Dilger, Jeremy Frank, James Hoagland, Karl Levitt, Christopher Wee, Raymond Yip, and Dan Zerkle. 1996. GrIDS-a graph based intrusion detection system for large networks. In *Proceedings of the 19th National Information Systems Security Conference*, Vol. 1. 361–370.

[66] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, and Srinivas Devadas. 2015. IMP: Indirect memory prefetcher. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO'15)*. ACM, 178–190.

[67] Dan Zhang, Xiaoyu Ma, Michael Thomson, and Derek Chiou. 2018. Minnow: Lightweight offload engines for worklist management and worklist-directed prefetching. *ACM SIGPLAN Notices* 53, 2 (2018), 593–607.

Received January 2020; revised November 2020; accepted November 2020