SIP: Boosting Up Graph Computing by Separating the Irregular Property Data

Jiacheng Ni
School of Microelectronics
Beihang University
Beijing, China
nival@buaa.edu.cn

Xiaochen Guo
Department of Electrical & Computer Engineering
Lehigh University
Bethlehem, Pennsylvania, USA
xig515@lehigh.edu

Yuanqing Cheng
School of Microelectronics
Beihang University
Beijing, China
yuanqing@ieee.org

ABSTRACT

Graph analytics is an important class of applications and is one of the cornerstone of big-data workloads. Unfortunately, due to poor data locality in most graph applications, conventional general-purpose computer architectures are unable to perform the best of their processing abilities. The main source of poor locality comes from accessing vertex properties. Upper-level caches cannot hold data blocks long enough due to their limited capacity and the long reuse distance of vertex properties. Moreover, accesses to properties can evict other useful data with good locality, which causes more conflicting misses. In this work, a small cache is added exclusively for the properties to solve this problem. We further enhance this structure with prefetchers to increase the hit rate of properties and improve performance of system. Experimental results show that compared to two state-of-the-art prefetcher and accelerator for graph computing, our proposed architecture achieves 1.13×-2.54× and 1.04×-1.27× performance improvements. In the meanwhile, the energy consumptions can be saved by 6.41%-13.43% and 34.67%-43.92% respectively.

CCS CONCEPTS

• **Computer systems organization → Special purpose systems**.

KEYWORDS

graph computing; domain-specific architecture; cache hierarchy

ACM Reference Format:
Jiacheng Ni, Xiaochen Guo, and Yuanqing Cheng. 2020. SIP: Boosting Up Graph Computing by Separating the Irregular Property Data. In *Proceedings of the Great Lakes Symposium on VLSI 2020 (GLSVLSI '20), September 7–9, 2020, Virtual Event, China.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3386263.3406905

1 INTRODUCTION

High performance big-graph processing techniques are desperately required in various real-world applications, and it is expected that graph processing will play a more important role in big-data analytics. However, processing big graphs typically involves complicated data structure and generates irregular memory access patterns. Many prior works focused on algorithm optimizations [4] and software framework developments [10] to achieve better graph computing performance. Recently, researchers have noticed that conventional general-purpose microprocessors are inefficient when running big-graph processing workloads [1]. The primary problem can be attributed to the irregular data accesses in graph computing that lead to poor spatial and temporal locality. As a result, the cache hit rate is low, resulting in more main memory accesses, long execution time and high energy consumption.

To solve this problem, some researchers tried to expose the potential data locality in graph applications [11], while others chose to use domain-specific prefetchers to bring data into caches in advance [2, 3]. Although accessing vertex properties in graphs shows poor locality compared to other types of data such as structure and weights [3], the access pattern of vertex properties is regular. For example, when using CSR format (please refer to 2.2 for details) to store the graph, the graph traversal firstly accesses the neighbors of one vertex, and then visits the properties of these vertices. So it is easy to predict which property data will be accessed in the near future, and facilitates the design of domain-specific accelerators [2]. In addition, [3] proved that the cache line storing graph property data has a longer reuse distance than what a typical L2-cache can serve. This observation enlightens us that directly fetching and accessing properties from the LLC into L1-cache rather than going through L2-cache can obtain better performance.

The conventional cache hierarchy is ineffective for the irregular property accesses. Even worse, these accesses may evict useful data (e.g., structure data, weights) with good locality. Therefore, we add a cache named P-cache, which is dedicated for property data, to eliminate this negative effect. P-cache interacts with LLC directly without accessing L2-cache and respond the property requests sent by processor directly. As property data are loaded from and stored to P-cache rather than D-cache and L2-cache, other data with good locality, resided in the conventional upper-level caches will not be affected. The added P-cache has smaller size than D-cache to obtain faster hit latency. However, the hit rates for properties become worse as they go into a smaller cache. To solve this problem, we further proposed two collaborative prefetchers to facilitate D-cache and P-cache respectively. The hit rates of both of structure data and property data can be increased.

We name the proposed structure as SIP architecture, because the key idea is to "Separate the Irregular Properties" from other

GLSVLSI '20, September 7–9, 2020, Virtual Event, China

ACM ISBN 978-1-4503-7944-1/20/09…$15.00
https://doi.org/10.1145/3386263.3406905

data, and equip property data and structure data with prefetchers respectively considering their features. The organization of this paper is as follows. Section 2 presents the necessary background on graph computing and the motivation of our work. Section 3 details the proposed SIP architecture. Experimental results are shown in Section 4. Section 5 introduces the related works and Section 6 concludes this paper.

2 BACKGROUND & MOTIVATION

In this section, we firstly introduce graph features, CSR format and the conception of active vertex, and then present the motivation of this work.

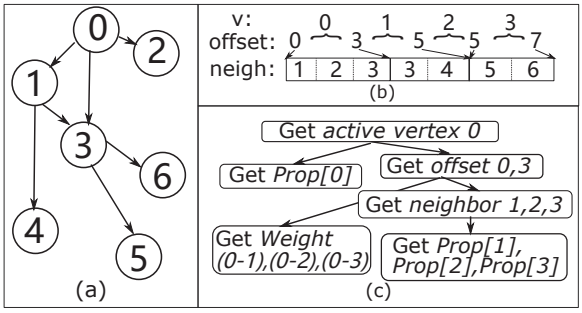

Figure 1: (a) A sample graph structure. (b) CSR format for the sample graph. (c) The way to access different types of data.

2.1 Graph features

Fig. 1(a) shows one example graph. Structure and property are two key graph features. Vertices and edges which represent entities and the relations among them, compose the graph structure. Vertex IDs can be represented by integer numbers. There are various values in graphs: ① Vertices usually have some values, which stand for various meaning in different applications, such as name, age or the importance of a person in a social network. ② Edges can have weights as well. Weights commonly have better locality compared to the vertex properties since edges of a vertex are usually stored sequentially. ③ Miscellaneous data (e.g., constants specified in some applications). We adopt the same treatment as [3] to take ① as property data in the rest of the paper. ② and ③ are considered as intermediate data, and are not the optimization objectives in this work.

2.2 CSR format

There are several storage formats for graphs, including adjacency matrix, adjacency list, COO (coordinate format) and CSR (compressed sparse row). Among them, CSR is flexible, compact and can be easily partitioned and organized. So it is widely used in many graph frameworks [11]. We adopt CSR format to illustrate our proposed SIP architecture in this work.

CSR utilizes the offset list and the neighbor list to represent the graph structure as described in Fig. 1(b). The length of the offset list is always one more than the number of vertices. Every adjacent two elements (belonging to one vertex *src*) in the offset list denote the *front* and *rear* offset respectively. Every element between these two offsets in the neighbor list is the *dst* vertex of an edge, i.e, one directed edge is composed of a pair of *src* and *dst* vertices. The length of the neighbor list is equal to the number of edges. The way to access the sample graph with CSR format is shown in Fig. 1(c). Once we get the vertex ID, we can access the properties of this vertex by "Prop[ID]".

2.3 Active vertex and frontier

An iteration is a loop to process all the active vertices. "Active" means whether one vertex will be processed in the next iteration during the graph processing. Graph applications can be divided into two categories: non-all-active and all-active [11]. In non-all-active applications, like Breadth-First Search (BFS) and Single Source Shortest Path (SSSP), only a portion of vertices are active in one iteration. If one property value is changed in current iteration, the corresponding vertex becomes active. When there are no active vertices, the application terminates. For all-active applications like Page Rank (PR), all vertices are active and processed in every iteration. The termination condition depends on the specific demands of various applications (e.g., the maximum iteration count).

The collection of active vertices is called frontier [15]. For all-active applications, since all vertices are active, recording the range of vertex IDs is enough. For non-all-active applications, there are two common ways to indicate which vertices are active: using active list [2, 18] to store the active vertex IDs, or using bit-vector [11]. Every bit in bit-vector tells whether a vertex is active. Which one is better to use depends on whether the graph is relatively sparse or dense [15]. We use the active list for non-all-active applications to illustrate the SIP architecture.

2.4 The motivation of this work

In this paper, we focus on the main cause of cache inefficiency in graph computing: *the property data accesses are irregular and intensive*. This phenomenon reflects in two aspects: the property data itself and the influence from property data on other data.

The first aspect is the poor locality of the irregular properties. As mentioned in Section 2.2, the regularity of access mode for graph can be leveraged to design prefetcher to solve it. The second aspect is that, many properties are only used once or a few times and become useless shortly. However, they are still fetched from lower to upper level caches, which probably evicts other useful data. Many previous works including VO-HATS [11] and DROPLET [3] only take account of the first problem. To solve the second problem as well, we propose SIP architecture, which is inspired by and improved based on the design pattern of VO-HATS and DROPLET. If we can make properties accessed in a separate cache without storing other data, we can improve performance by avoiding the second problem. Moreover, according to [3], property data have longer reuse distances than that L2-cache can serve. If we directly fetch property data from the LLC to this special cache, and send them to CPU without going through L2-cache, the property accesses can be accelerated effectively.

3 SIP: SEPARATING THE IRREGULAR PROPERTIES ARCHITECTURE

In this section, we present the detail of SIP architecture, including the overall structure, the data prefetching mechanism and the software support for initialization of SIP.

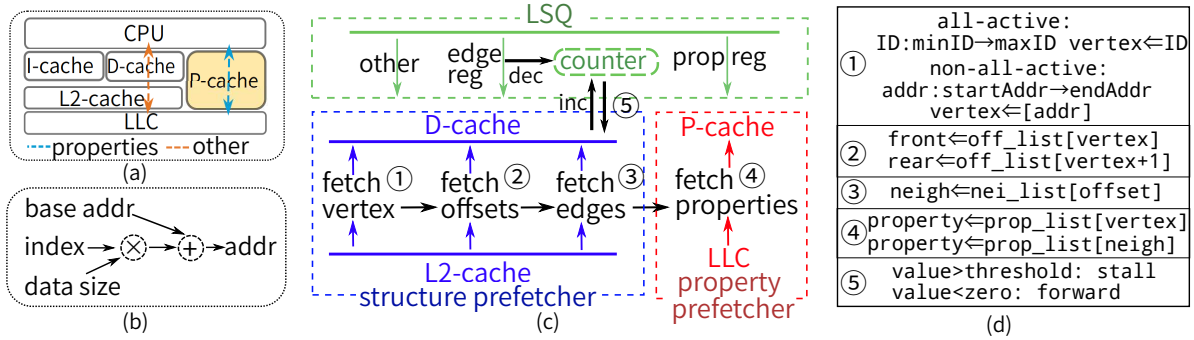

Figure 2: (a) Overview of SIP. Properties go through P-cache while other data go through the conventional cache hierarchy. (b) Address generations of prefetching targets (offsets, neighbors and properties) in SIP. (c) Diagram of functionalities and locations of structure and property prefetchers in SIP. (d) Detail operations in modules of (c). "→" means traversing from the left to the right and "⇐" means assignment.

3.1 Overview of SIP

The SIP architecture is shown in Fig 2(a). P-cache is added between the LLC and CPU for property data. There are two collaborative prefetchers named structure prefetcher and property prefetcher to increase the hit rate of caches. Structure prefetcher is located in D-cache to prefetch structure data from L2-cache, and property prefetcher is located in P-cache to prefetch property data from LLC.

Loading properties from or storing properties to the P-cache requires additional method to tell the difference between properties and other data. An intuitive way is to extend current ISA with new load and store instructions to identify the property data. However, this way causes software modification. Another way used in [3] is adding an extra bit in the TLB and page table by modifying the "malloc" routine in the system library. In this work, we propose a different way: by comparing addresses of incoming requests sent by CPU and the address range of properties to identify different data types. These addresses are virtual addresses used in most L1-cache today. To keep the address range, we need two registers located at the end of load/store queue, to store the start address and the end address of properties. Depending on the checking results, the request is sent into P-cache or D-cache. Although one new instruction is needed to configure the registers, the software of graph computing only needs to be modified in the initialization part instead of the main body, making it easy to fit in any programming model or framework.

In the proposed method, there is no data duplications between the P-cache and regular D-cache and L2-cache because P-cache only loads and stores properties, which never pass through D-cache and L2-cache. For the same reason, other data never go into the P-cache. Therefore, the cache coherency protocol does not need any modifications to guarantee the data coherence in the multi-core system.

The P-cache is designed to be very small and fully associative. Firstly, a small size cache has fast access latency and small area overhead. Secondly, since there are prefetchers (introduced in the Section 3.2) to avoid cache miss, large P-cache size is unnecessary. Thirdly, fully associativity can avoid data eviction by the conflict misses. Besides, the replacement policy of P-cache should be FIFO, as the the property data is prefetching in order and long distance data reuse is not taken into account in P-cache.

3.2 Mechanism of prefetchers

The architecture of structure prefetcher and property prefetcher is shown in Fig. 2(c). These two prefetchers have a similar design with VO-HATS since the access mode of graph is regular. A brief introduction of the operations in each part of SIP is shown in Fig. 2(d). The working procedure of the structure prefetcher contains three stages: ① get the active vertex from the active list, ② get the front and rear offset of this active vertex, and ③ with these two offsets, get the neighbor vertices (edges) of the active vertex. After the structure prefetcher processes the active vertex and its neighbors, the IDs of the active vertex and its neighbor vertices are used to calculate the addresses of their properties. These addresses are sent to the property prefetcher, which uses these addresses to prefetch the properties data into P-cache (④ in Fig. 2(c)) to prepare them for CPU requests in advance. Meanwhile, the structure data have been put into D-cache and become ready to be used. Small FIFOs are used as the interface between adjacent stages to buffer data.

The way to generate the memory addresses of data for prefetching is indicated in Fig. 2(b). Since the addresses used in SIP are virtual addresses, the target address can be calculated by base address and address offset. The address offset can be calculated by index offset multiply by the data size. Since the data size is always the power of two, we can use a shifter instead of a multiplier to obtain the result with an adder [17]. The base address and data size used are specified during the initialization, achieved by the register configuration instruction mentioned in Section 3.1.

3.3 Prefetching distance

Timeliness is important for prefetchers. If prefetching is too early, the data may be evicted before being accessed. If prefetching is too late, when the requests reach the cache, the data may still not be ready. Both cause useless prefetching. Different from regular prefetchers that are triggered by specified conditions like cache misses, prefetchers in SIP run self-paced after the initialization. To guarantee this automatic prefetching at the appropriate time, we keep track of the number of edge accesses from both SIP and processor. Between the load/store queue and the structure prefetcher, a counter is added to do this job (⑤ in Fig. 2(c)). In the third stage of the structure prefetcher, i.e., fetching edges, whenever an edge prefetching request is sent, the counter is increased by one. Each

Table 1: ADDED REGISTERS IN SIP

structure prefetcher	
size	data size
active	whether the application is all-active or not
active-1/2	all-active: the minimal/maximal vertex ID non: the start/end address of the active list
off-list	the start address of the offset list
nei-list	the start address of the neighbor list
t-v	the current processed vertex
t-front	the front offset of current active vertex
t-rear	the rear offset of current active vertex
t-offset	the current offset, varies from t-front to t-rear
t-neigh	the current processed neighbor
property prefetcher	
prop-list	the start address of the property list
load/store queue	
nei-1/2	the start/end address of the neighbor list
prop-1/2	the start/end address of the property list
threshold	the counter threshold of stall prefetching

time an edge reading request from the processor is obtained in D-cache, the counter is decreased by one. To tell the difference from the edge requests and other requests, the same technique as the property identification can be used here. Two more registers are added at the end of load/store queue, which are used to save the range of the neighbor list.

To prevent early prefetching, when the counter value reaches a threshold that has been set in the initialization, the third stage of the structure prefetcher is stalled, which in turn makes other stages stalled because of the empty or full states of the FIFOs, until the the counter value decreases due to new requests issued from the processor. To make the prefetching of the vertex and offsets not too early, the adjacent FIFOs in the structure prefetcher should be small enough considering the size of D-cache. The depth of these three FIFOs (one for active vertex, the other two for its two offsets) are set as 5 in our design. The selection of the threshold should not exceed the associativity of P-cache as the previous prefetched data may be evicted before reaching CPU.

To prevent late prefetching, when the counter value is below zero, the structure prefetcher discards the active vertex, its offsets and its neighbors processed currently, and directly processes the next active vertex in frontier. The number of unprocessed neighbors is added to the counter value, until the counter value is greater than zero again. This method also can make prefetchers running ahead of the processor at the boot time, no needs to demand the prefetcher to boot ahead of processor like VO-HATS.

3.4 Initialization of SIP

The only added instruction in SIP is used for the register initialization. A different opcode is used to distinguish between the special register configuration and a normal register control. Two operand fields are needed in this special instruction: one specifies the index of the added register in the structure prefetcher and load/store queue, the other is the register value. The instruction is only used in the initialization phase, so SIP can be used in many applications, programming models and frameworks easily without affecting the main body of the software (if active list is used, the addresses of the active list need to be configured before the start of each iteration as they are changed after one iteration). The configuration registers, and some temporary registers whose values are changed inside each iteration (with prefix "t") are listed in Table 1.

Table 2: GRAPHSET SUMMARY

Graphset	Vertex	Edge	Degree
g500-s18e16 [13]	174,147	7,600,696	43.6
g500-s19e16 [13]	335,318	15,459,350	46.1
amazon0302 [7]	262,111	1,234,877	4.7
web-google [13]	916,428	5,105,039	5.6
soc-livejournal1 [7]	4,847,571	68,993,773	14.2
soc-pokec [7]	1,632,803	30,622,564	18.6

Table 3: ARCHITECTURAL SETTINGS IN MULTI2SIM

CPU	4-core x86 3GHz OoO 32-bit
I/D-cache	16KB 4-cycle 8-way LRU private
L2-cache	128KB 8-cycle 8-way LRU private
LLC	4MB 32-cycle 16-way LRU shared
Memory	latency: 40ns bandwidth: 28GB/s
SIP P-cache	1KB 1-cycle fully-assoc FIFO private

4 EXPERIMENTAL RESULTS

4.1 Experiment setup

We used Multi2Sim [16] for architectural simulations. McPAT [8], CACTI [12] and Design Compiler were used to analyze latency, energy consumption and area overhead of components in SIP architecture. The experiments are evaluated under 45nm technology node. We evaluated three algorithms (BFS SSSP PR) on two synthetic and four real-world graphsets, as shown in Table 2. In BFS and SSSP, we used the vertex with minimal ID as the source to stretch out its connected component, and for PR only one iteration is evaluated. The configuration of the simulated system in Multi2Sim is shown in Table 3. Three graph processing architectures are considered in simulations: DROPLET [3] as the baseline, VO-HATS [11] and our proposed SIP. The comparisons of these three architectures are listed as follows:

- DROPLET: It uses special malloc function to mark the structure data. Structure prefetcher is located in L2-cache, only triggered by the marked data. Property prefetcher is located in memory controller.
- VO-HATS: It lies between L2-cache and processor, fetches the structure data from lower-level cache and directly supply them to processor and fetches property data into L2-cache. VO-HATS needs to be configured before running. Although supplying data to processor with FIFO instead of loading them from cache can save time, user needs to explicitly invoke $fetch\_edge$. Software modification is required and the design of processor/pipeline may need modifications as well. Besides, late prefetching in VO-HATS may stall CPU and degrade performance.
- SIP: It uses P-cache for the separated property data. Structure prefetcher is located in D-cache and property prefetcher is located in P-cache. Only initialization part needs to be modified on the side of software. Counter is added to guarantee prefetching timeliness.

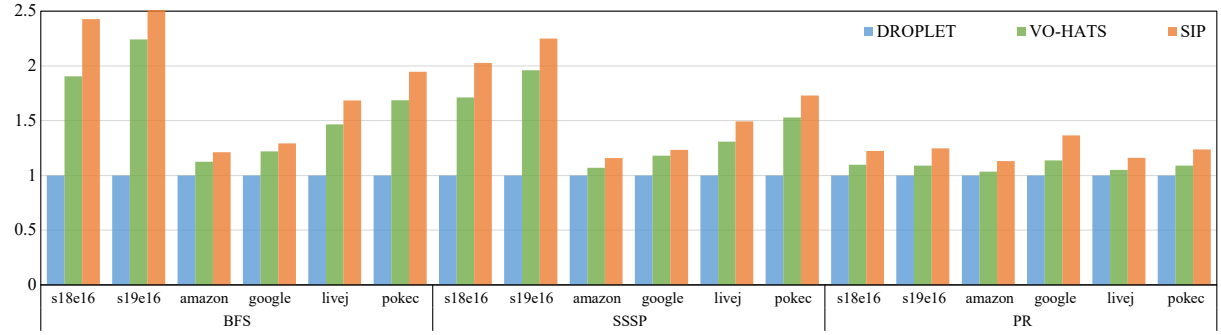

Figure 3: Speedup comparisons of VO-HATS and SIP over the baseline DROPLET.

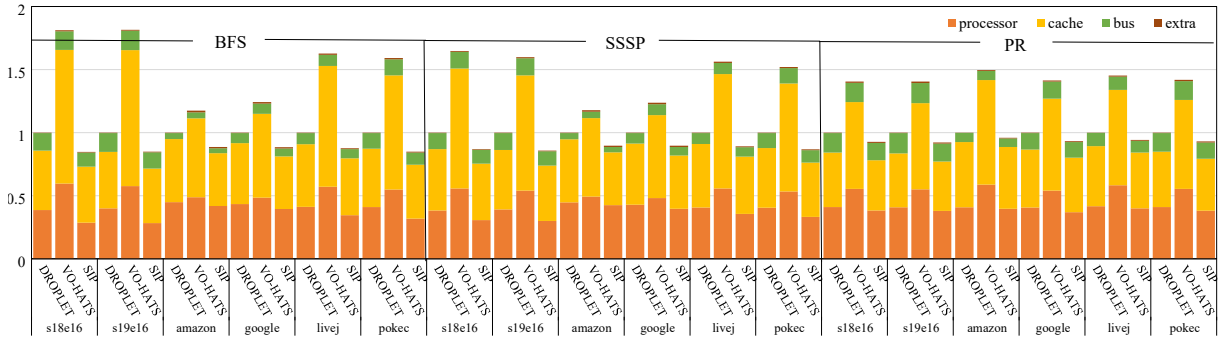

Figure 4: Energy consumption breakdown of VO-HATS and SIP normalized to that of the baseline DROPLET. The extra in the figure represents the energy consumption of added hardware in each architecture.

In order to support the extended instruction in simulations, we used ".byte" pseudo assembly instruction with inline assembly to add new instructions in the executable program to get rid of the modification of the compiler. Additionally, Multi2Sim is modified to identify the new instruction.

4.2 Performance evaluations

Fig. 3 shows the performance comparisons between the DROPLET, VO-HATS and our propose SIP. As shown in the figure, SIP can improve performance over the baseline by 1.21×-2.54× for BFS, 1.16×-2.25× for SSSP and 1.13×-1.36× for PR, and 1.85×, 1.65× and 1.23× on average respectively. The speedup over the baseline is significant, as the baseline DROPLET may mispredict the prefetching target, while SIP can prefetch the correct data by the graph access mode with enough information. The figure shows the improvements of SSSP are less than those of BFS. This is because in SSSP, the accesses of weights cannot get improvements from P-cache or prefetchers. The performance improvements of PR are the smallest among three kinds of applications. Unlike the other two, PR is all-active, which means the active vertices are traversed increasingly in the order of vertex IDs, showing better spatial locality. Besides, there are no accesses to the active list, limiting the performance improvements.

Compared to VO-HATS, SIP improves performance by 1.06×-1.27× for BFS, 1.04×-1.18× for SSSP and 1.09×-1.20× for PR, and 1.15×, 1.13×, 1.13× on average respectively. The improvements of VO-HATS and SIP have similar trends since the prefetchers in VO-HATS and SIP have similar structure. However, SIP has better performance due to the separated property data cache, even VO-HATS can directly supply the structure data to the processor.

4.3 Energy consumption comparisons

We extracted execution statistics generated by Multi2Sim, and imported them into McPAT to get the energy consumption of the system. For P-caches, we obtained the energy consumption by CACTI. The energy consumption of prefetchers in SIP was obtained by synthesizing the RTL code of the prefetcher using Design Complier with 45nm FreePDK [14].

As shown in Fig. 4, the average energy consumption of SIP is 86.57% (BFS), 93.59% (SSSP) and 88.03% (PR) of the baseline case. The figure also shows that SIP consumes less processor and bus energy than the baseline due to shorter execution time, but the energy consumption of cache has little optimization. This is because data prefetching and cache accesses in SIP are more extensive than the DROPLET to gain the data in time.

Compared to VO-HATS, SIP consumes less energy as well. the energy consumption of SIP can be reduced to 56.08%, 65.33% and 60.41% of that consumed by VO-HATS respectively. The figure shows that VO-HATS has much more energy consumptions than the baseline DROPLET and SIP, which comes from the more processor and cache energy consumption. This is because the special operation *fetch_edge* in VO-HATS cannot be automatically generated in compilation. To use it, its predefined function has to be explicitly invoked, involving that additional instructions (e.g., "call/ret" and "push/pop") are executed with more memory accesses (e.g., stack push and pop operations), causing higher CPU energy and cache energy. Besides, from the figure, we can find that the energy consumed by the extra hardware varies from 0.3% to 1.3% among these three architectures.

4.4 Area overhead analysis

A quad-core system without any optimization configured as in Table 3 occupies 165.875 mm^2 obtained from McPAT. The extra area of the baseline DROPLET is around 65,424 μm^2. The area overhead of VO-HATS and the prefetchers in SIP are around 28,816.88 μm^2 and 24,830.18 μm^2 reported by Design Compiler. Besides, the P-cache in SIP is estimated about 28,000.64μm^2. So the area overhead of SIP is 52,830.82μm^2. In summary, the area overheads of DROPLET, VO-HATS and our proposed SIP are 0.0394%, 0.0174% and 0.0318% respectively, which are negligible.

5 RELATED WORK

Apart from the previous mentioned DROPLET and VO-HATS, there are many other graph computing accelerating methods as well. PIM (Processing-in-Memory) is a popular technique to solve the "Memory Wall" problem and can be adopted to improve graph processing. Most of proposed PIM-based graph architectures involve adopting either emerging memory technologies [6] or 3D stacking [1]. PIM architecture can reduce the traffic to main memory by processing data before they are sent to CPU. The major challenge in PIM is that the fabrication process for emerging memory is still immature and costly, and adding logic into memory increases design complexity and decreases memory density. Compared to PIM-based designs, our proposed architecture does not modify the memory architecture.

There are many other novel designs to optimize graph computing. Graphicionado [5] is a specialized pipeline-design processor for graph computing to achieve the best hardware efficiency. MMAP [9] aimed at improving disk-based graph applications by changing the memory mapping at the level of operating system. It adopted a simple memory mapping scheme to gain significant performance improvement. Similar to these works, our proposed SIP also focus on the memory accessing bottleneck in graph computing but from the prefetcher perspective.

6 CONCLUSION & FUTURE WORK

Due to the irregular accesses of property data in graphs, the running of graph analytics has low performance on the general-purpose processor. These irregular accesses evict other useful data and degrade cache performance. Based on the observation from the previous work, we proposed the SIP architecture, which leverages separating property data scheme enhanced with structure and property prefetchers. Experimental results show that compared to two state-of-the-art prefetchers/accelerators DROPLET and VO-HATS, our proposed architecture achieves 1.13×-2.54× and 1.04×-1.27× performance improvements respectively. In the meanwhile, the energy consumptions can be saved by 6.41%-13.43% and 34.67%-43.92% respectively. The area overhead of SIP is estimated about 0.0318%, which is negligible.

Considering the diversity of graph applications, a fixed counter threshold for SIP may not obtain the optimal performance. In the future work, we will explore the dynamically changing threshold adjustment scheme depending on the state of P-cache to obtain more significant performance improvements.

ACKNOWLEDGMENTS

This work is partially supported by the Beijing Natural Science Foundation under Grant 4192035, and the National Science Foundation at Lehigh University under Grant CCF-1750826. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Beijing Natural Science Foundation and the National Science Foundation.

REFERENCES

[1] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2016. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the International Symposium on Computer Architecture*. 105–117.

[2] Sam Ainsworth and Timothy M. Jones. 2016. Graph prefetching using data structure knowledge. In *Proceedings of the International Conference on Supercomputing*. 1–10.

[3] 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 International Symposium on High Performance Computer Architecture*. 373–386.

[4] Scott Beamer, Krste Asanović, and David Patterson. 2017. Reducing pagerank communication via propagation blocking. In *Proceedings of the International Parallel and Distributed Processing Symposium*. 820–831.

[5] 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 International Symposium on Microarchitecture*. 1–13.

[6] Lei Han, Zhaoyan Shen, Zili Shao, H. Howie Huang, and Tao Li. 2017. A novel ReRAM-based processing-in-memory architecture for graph computing. In *Proceedings of the Non-Volatile Memory Systems and Applications Symposium*. 1–6.

[7] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[8] 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 International Symposium on Microarchitecture*. 469–480.

[9] Zhiyuan Lin, Minsuk Kahng, Kaeser Md. Sabrin, Duen Horng Polo Chau, Ho Lee, and U Kang. 2014. Mmap: Fast billion-scale graph computation on a pc via memory mapping. In *Proceedings of the International Conference on Big Data*. 159–164.

[10] Grzegorz Malewicz, Matthew H. Austern, Aart J.C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 135–146.

[11] 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 International Symposium on Microarchitecture*. 1–14.

[12] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. 2009. CACTI 6.0: A tool to model large caches. *HP laboratories* (2009), 1–24.

[13] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. http://networkrepository.com

[14] James E. Stine, Ivan Castellanos, Michael Wood, Jeff Henson, Fred Love, W. Rhett Davis, Paul D. Franzon, Michael Bucher, Sunil Basavarajaiah, Julie Oh, and Ravi Jenkal. 2007. FreePDK: An open-source variation-aware design kit. In *Proceedings of the International Conference on Microelectronic Systems Education*. 173–174.

[15] Jiawen Sun, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. 2017. GraphGrind: addressing load imbalance of graph partitioning. In *Proceedings of the International Conference on Supercomputing*. 1–11.

[16] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. 2012. Multi2Sim: a simulation framework for CPU-GPU computing. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 335–344.

[17] Xiangyao Yu, Christopher J Hughes, Nadathur Satish, and Srinivas Devadas. 2015. IMP: Indirect memory prefetcher. In *Proceedings of the International Symposium on Microarchitecture*. 178–190.

[18] Minxuan Zhou, Mohsen Imani, Saransh Gupta, and Tajana Rosing. 2018. GAS: A heterogeneous memory architecture for graph processing. In *Proceedings of the International Symposium on Low Power Electronics and Design*. 1–6.