Practically Tackling Memory Bottlenecks of Graph-Processing Workloads

Alexandre Valentin Jamet
alexandre.jamet@bsc.es
Barcelona Supercomputing Center (BSC)
Universitat Politècnica de Catalunya (UPC)

Georgios Vavouliotis
georgios.vavouliotis2@huawei.com
Huawei Zurich Research Center

Daniel A. Jiménez
djimenez@acm.org
Texas A&M University

Lluc Alvarez
lluc.alvarez@bsc.es
Barcelona Supercomputing Center (BSC)
Universitat Politècnica de Catalunya (UPC)

Marc Casas
marc.casas@bsc.es
Barcelona Supercomputing Center (BSC)
Universitat Politècnica de Catalunya (UPC)

***Abstract*—Graph-processing workloads have become widespread due to their relevance on a wide range of application domains such as network analysis, path-planning, bioinformatics, and machine learning. Graph-processing workloads have massive data footprints that exceed cache storage capacity and exhibit highly irregular memory access patterns due to data-dependent graph traversals. This irregular behaviour causes graph-processing workloads to exhibit poor data locality, undermining their performance.**

This paper makes two fundamental observations on the memory access patterns of graph-processing workloads: First, conventional cache hierarchies become mostly useless when dealing with graph-processing workloads, since 78.6% of the accesses that miss in the L1 Data Cache (L1D) result in misses in the L2 Cache (L2C) and in the Last Level Cache (LLC), requiring a DRAM access. Second, it is possible to predict whether a memory access will be served by DRAM or not in the context of graph-processing workloads by observing strides between accesses triggered by instructions with the same Program Counter (PC). Our key insight is that bypassing the L2C and the LLC for highly irregular accesses significantly reduces latency cost while also reducing pressure on the lower levels of the cache hierarchy.

Based on these observations, this paper proposes the Large Predictor (LP), a low-cost micro-architectural predictor capable of distinguishing between regular and irregular memory accesses. We propose to serve accesses tagged as regular by LP via the standard memory hierarchy, while irregular access are served via the Side Data Cache (SDC). The SDC is a private per-core set-associative cache placed alongside the L1D specifically aimed at reducing the latency cost of highly irregular accesses while avoiding polluting the rest of the cache hierarchy with data that exhibits poor locality. SDC coupled with LP yields geometric mean speed-ups of 20.3% and 20.2% on single- and multi-core scenarios, respectively, over an architecture featuring a conventional cache hierarchy across a set of contemporary graph-processing workloads. In addition, SDC combined with LP outperforms the Transpose-based Cache Replacement (T-OPT), the state-of-the-art cache replacement policy for graph-processing applications, by 10.9% and 13.8% on single-core and multi-core contexts, respectively. Regarding the hardware budget, SDC coupled with LP requires 10KB of storage per core.

***Index Terms*—graph processing, cache management, off-chip prediction, micro-architecture**

I. Introduction

In recent years, graph-processing has become an important class of workloads with applications in a rapidly growing and diverse number of fields (*e.g.*, network analysis [16], bioinformatics [22], path-planning [12], and machine learning [13]). Graph-processing workloads typically use very large input sets, often in multi-gigabyte scale, and data-dependent graph traversal methods [33] that exhibit highly irregular memory access patterns. Processing such massive and irregularly accessed data prevents graph-processing applications from exhibiting good locality, imposing great difficulty on traditional cache hierarchies to efficiently serve memory requests, leading to frequent main memory accesses that incur high latency overheads and compromise application performance. Indeed, recent work [50] demonstrates that, due to the irregular memory access patterns of data-depend graph traversals, state-of-the-art graph-processing workloads spend up to 80 % of the total execution time waiting for the DRAM.

Prior work has quantified the overheads of graph-processing workloads [31] and has proposed several approaches to ameliorate their costs. These approaches mainly fall into the following categories: (i) domain-specialized cache management for graph-processing applications [8], [39]; (ii) indirect memory prefetching [6], [9], [48], which aims at improving the throughput of memory-bound graph-processing workloads by fetching data blocks before they are explicitly requested by the application; (iii) graph-processing accelerators [15], [21], [31], [34]–[36]; and (iv) pre-processing schemes that improve the locality of graph-processing workloads [7], [14], [45]. Our proposal is in the first category, which applies fine-grained micro-architectural cache management optimizations, avoiding costly pre-processing of the graph data without requiring any changes to the architecture nor to any software layer.

Two key observations drive our proposal: First, conventional cache hierarchies become mostly useless when dealing with graph-processing workloads, since 78.6% of the accesses that miss in the L1 Data Cache (L1D) result in misses in the

L2 Cache (L2C) and in the Last Level Cache (LLC), thus requiring a DRAM access. This is caused by the poor locality of part of the data set of graph-processing applications that is accessed with highly irregular memory access patterns. The second observation is that strides between accesses triggered by instructions featuring the same Program Counter (PC) are a good program feature to predict whether a certain memory access will be served by DRAM or not. Our key insight is that bypassing the L2C and the LLC for highly irregular accesses significantly reduces their latency cost and, at the same time, it lowers the pressure on the lower levels of the cache hierarchy. This, in turn, minimizes cache pollution and increases the effective capacity of the L2C and the LLC for the subset of data that exhibits good locality and that is accessed by regular accesses patterns.

To exploit the above explained key insights and overcome the limitations of current cache hierarchies, this work proposes the Large Predictor (LP), a novel and low-cost micro-architectural predictor capable of dynamically identifying regular and irregular access patterns. To do so, LP employs a small prediction table, indexed with the PC, that provides a historic knowledge of the strides of the memory accesses. LP leverages this information to classify a memory access as regular or irregular; those classified as regular are routed to the normal cache hierarchy while the others go through the Side Data Cache (SDC), a new per-core private auxiliary cache. The SDC is a small set-associative cache placed alongside the L1D with the specific purpose of reducing the latency of highly irregular accesses while avoiding polluting the rest of the cache hierarchy with data that exhibits poor locality.

This paper makes the following contributions:

• We corroborate that memory requests of graph-processing workloads that miss in the L1D also miss in the L2C and the LLC with high probability. The key insight of this study is that bypassing the L2C and LLC for irregular accesses that do not exhibit locality has potential to provide significant benefits.

• We propose and design the Side Data Cache (SDC), a small on-chip cache placed alongside the L1D, and the Large Predictor (LP), a novel micro-architectural predictor that dynamically classifies the memory requests into regular and irregular. The accesses identified by LP as regular and irregular are routed to L1D and to the SDC, respectively. Our proposal improves the performance of graph-processing applications by removing the useless L2C and LLC look-ups for the irregular accesses while also reducing cache pollution and improving the locality in the rest of the cache hierarchy.

• We demonstrate that, across a set of contemporary and diverse graph-processing workloads, our proposal outperforms conventional cache hierarchies as well as the Transpose-based Cache Replacement (T-OPT) [8], the state-of-the-art cache management scheme for graph-processing workloads, achieving respective geometric mean speedups of 20.3% and 10.9% in single-core scenarios, and of 20.2 % and 13.8% on a multi-core setup. Furthermore, T-OPT (and its practical but less performant implementation, P-OPT) requires modifying the original application, whereas our proposal does not. Regarding the hardware budget, SDC coupled with LP requires 10KB of storage per core.

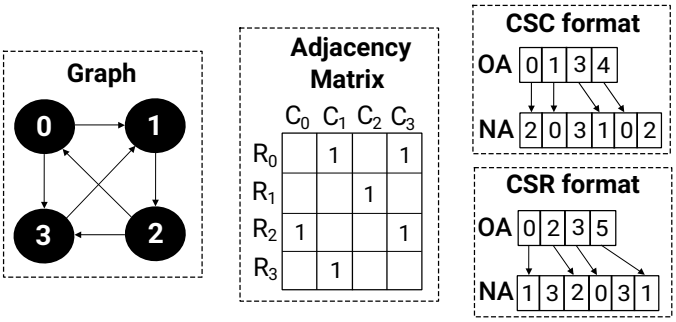


Fig. 1. Graph representations using the CSR/CSC format.

II. Background and Motivation

A. Graph-Processing Workloads

Graph-processing is becoming a fundamental tool in a wide range of areas including bionformatics [22], social network analysis [16], and web analytics [12]. Graph-processing workloads typically use sparse data formats like the *Compressed Sparse Row/Column (CSR/CSC)* [20], or even more sophisticated proposals [29], to manage large amounts of graph data. The CSR/CSC format is used to encode the graph's adjacency matrix using several data structures. The first one contains the row or the column indices of the adjacency matrix for the cases of CSR and CSC, respectively, and it is typically called the *Neighbors Array (NA)*. The second data structure indexes the beginning of each adjacency matrix row for the case of CSR (column for CSC), and it is denoted as the *Offset Array (OA)*. Figure 1 represents both the OA and the NA arrays corresponding to the left-hand side graph for both the CSR and CSC formats. Finally, there are additional data structures that contain numerical data corresponding to graphs vertices that we call *Property Arrays*. In the context of graph-processing, the CSR format encodes outgoing neighbors while CSC contains incoming ones.

Manipulating these sparse data structures often produces irregular memory access patterns. For example, when computing the Sparse Matrix-Vector (SpMV) multiplication $y = Ax$, accesses to vector x are indexed by the column indices of matrix A, which are non-contiguous and constitute an irregular access stream. Graph-processing workloads also display highly irregular memory access patterns driven by operations like graph traversals that require visiting all the nodes of graph V, that is, scanning adjacency matrix rows following the graph connectivity. For a certain vertex v_i, its neighbors correspond to the non-zero elements of adjacency matrix row i and define the next rows of the adjacency matrix to be accessed, producing an irregular memory access stream driven by the graph connectivity.

Algorithm 1 shows one of the most important graph-processing workloads, *Page Rank* [12]. Algorithm 1 shows the two main steps of *Page Rank* per each main loop iteration. First, for each vertex v_u, the algorithm updates the content of

Algorithm 1 Computation of Page Rank scores

Require: $G(V, OA, NA)$ {OA and NA respectively denote the offset array and the neighbors array in the CSC format.}
Require: δ {Dampling factor.}
Require: ϵ {Convergence threshold.}
Require: $iterations$ {Maximum # of iterations to process.}
Ensure: $scores[:]$ {Page Rank scores for all vertices}

```
1:  scores[:] ← 1/#V
2:  for iter ← [0; iterations − 1] do
3:      error ← 0
4:      for all vertex v_u ∈ V do
5:          outgoing_contrib[u] ← scores[u] / d+(u)
6:      end for
7:      for all vertex v_u ∈ V do
8:          sum ← 0
9:          for i ← [OA[u], OA[u + 1] − 1] do
10:             sum ← sum + outgoing_contrib[NA[i]]
11:         end for
12:         old_score ← scores[u]
13:         scores[u] ← (1−δ)/#V + δ · sum
14:         error ← error + |scores[u] − old_score|
15:     end for
16:     if error < ϵ then
17:         Convergence has been reached.
18:     end if
19: end for
```

the $outgoing\_contrib$ array using the scores obtained in the previous iteration divided by the number of outgoing neighbors of v_u, $d^+(u)$. Then, the algorithm computes the *Page Rank* score of each vertex v_u as the sum of the $outgoing\_contrib$ and a constant value that depends on the index count, $\#V$, and the dampling factor δ. This process is repeated until either convergence or the maximum number of iterations is reached.

The code region that produces irregular accesses is composed of a loop that ranges from line 7 to 15. The purpose of this loop is to traverse the entire graph by iterating over the incoming neighbors of a given vertex v_u, which are located between positions $OA[u]$ and $OA[u+1]-1$ of the NA array. The algorithm uses these indices to access the $outgoing\_contrib$ property array, thus, the access pattern to $outgoing\_contrib$ is driven by the matrix connectivity.

B. Characterizing the Memory Hierarchy Behavior of Graph-Processing Workloads

Irregular memory access patterns driven by graph connectivity have a strong impact on the memory hierarchy behavior of graph-processing workloads. Figure 2 shows the Misses-Per-Kilo-Instruction (MPKI) rates experienced by graph-processing workloads belonging to the GAP benchmark suite [10]. We show MPKI rates concerning three cache hierarchy levels (L1D, L2C, and LLC). Section IV describes in detail our experimental setup. Figure 2 shows that graph-processing workloads suffer from a large number of misses on

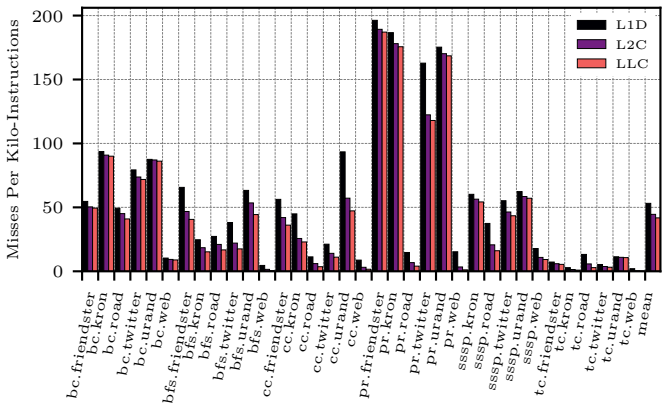


Fig. 2. Misses-Per-Kilo-Instruction (MPKI) across the different levels of the cache hierarchy triggered by graph-processing workloads.

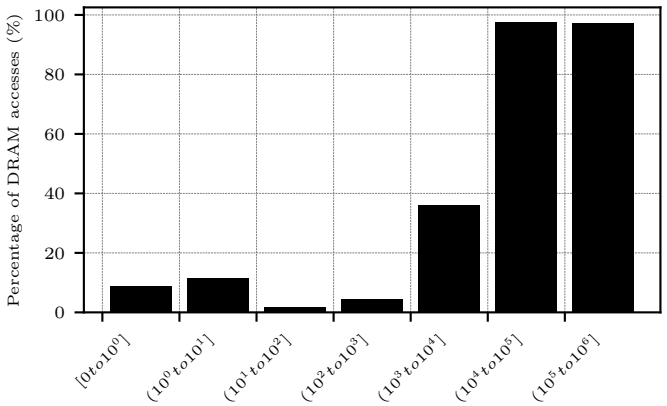


Fig. 3. Probability of accessing DRAM for accesses exhibiting different strides. Data corresponds to the *cc.friendster* workload.

all the levels of the cache hierarchy. The average MPKI rates of graph workloads for the L1D, the L2C, and the LLC are, respectively, 53.2, 44.5, and 41.8.

> ***Finding 1.*** *Graph-processing workloads exhibit very high MPKI rates in all cache levels.*

In addition, Figure 2 shows that L2C and LLC MPKI rates are just slightly smaller than L1D MPKI, which indicates that just a minor portion of L1D misses are served either by the L2C and LLC. Our analysis indicates that 78.6% of the access that miss in the L1D also miss in the lower cache levels, so they require a DRAM access.

> ***Finding 2.*** *A very large portion (78.6%) of the accesses that trigger L1D misses also miss in the lower-levels of the cache hierarchy and require a DRAM access.*

Figure 3 shows a characterization of the memory accesses triggered by graph-processing workloads in terms of strides between accesses issued by instructions with the same PC. The x-axis displays different intervals of strides. The y-axis shows the percentage of memory accesses that are served at DRAM per each interval. We obtain data on Figure 3 by considering the *Connected Components (CC)* application [10] of the GAP suite applied to the Friendster graph [47]. Section IV describes

in detail our experimental setup. Figure 3 indicates that memory accesses featuring small strides have a much lower probability of accessing DRAM than access displaying large strides. For example, just 11.6% of memory accesses whose strides fall between 2 and 10 (*i. e.* interval $(10^0$ to $10^1]$ of Figure 3) access DRAM, while this percentage grows to 97.6% for accesses with strides between 10^5+1 and 10^6. We analyze all GAP benchmarks suite and observe similar behaviour.

Finding 3. *Memory accesses featuring large strides have a very high probability of accessing DRAM.*

These three findings imply that memory accesses of graph-processing workloads can be divided into two categories: i) accesses displaying a large stride with respect to the last access triggered by an instruction with the same PC, that have a very high probability of missing on all cache levels and accessing DRAM; and ii) accesses featuring a small stride, that are in general served by the cache hierarchy. It is thus natural to propose hardware mechanisms able to differentiate between these two categories and avoid cache averse accesses to pollute cache content. These mechanisms significantly reduce the latency for cache-averse and cache-friendly accesses, respectively, by eliminating unnecessary cache look-ups for the former and preventing cache lines containing data for the latter from being evicted by insertions of cache-averse data.

III. Managing Memory Accesses of Graph-Processing Workloads

To address irregular memory access patterns in graph-processing workloads, we introduce two hardware innovations: the *Side Data Cache (SDC)*, a compact cache positioned alongside each core's L1D cache, and the Large Predictor (LP), a cost-effective microarchitectural predictor. The LP classifies memory accesses as cache-friendly or cache-averse and routes cache-averse accesses to the SDC. This approach reduces latency by avoiding costly cache hierarchy look-ups and improves cache locality by preventing cache pollution.

A. Side Data Cache (SDC)

The *Side Data Cache (SDC*) is a small on-chip cache placed alongside the L1D cache of every core. The goal of the SDC is to provide an alternative and fast data path for cache averse memory accesses. As Section II indicates, 78.6% of the accesses that miss in the L1D also miss in the lower levels of the cache hierarchy and end up accessing DRAM, and most of these are caused by cache averse accesses that exhibit large strides. To prevent performance inefficiencies when handling cache averse accesses, these are routed to the SDC instead of to the traditional cache hirarchy. If the look-up in the SDC hits, the requested cache block returned to the CPU with low latency, given that the SDC is smaller than a conventional L1D. In case the look-up in the SDC misses, a coherence message is sent to the cache directory to ensure coherence (as described later in detail in Section III-C) and, if the data is not present in the rest of the cache hierarchy, the requested block is fetched from DRAM and directly inserted in the SDC, bypassing the L2C and the LLC. This provides a fast path to DRAM, avoiding costly accesses to the lower-level caches (L2C, LLC). In addition, cache pollution is reduced in all the levels of the cache hierarchy (L1D, L2C, LLC), which get cleared from irregular access patterns interfering with regularly accessed data and, thus, improve the cache management for the regularly accessed blocks.

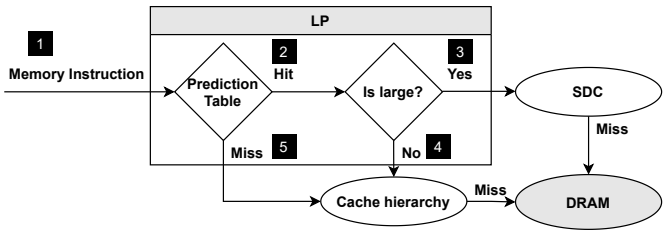


Fig. 4. Operation of the LP on a prediction event.

In order to capitalize on the benefits of the SDC, some prediction logic able to categorize memory accesses as cache friendly or cache averse is required.

B. Large Predictor (LP)

To identify memory accesses that should be directed to the SDC we propose the *Large Predictor (LP)*, a microarchitectural prediction mechanism capable of dynamically distinguish between regular and irregular access patterns. Before triggering a memory access, the core consults LP to decide whether the access must be directed to the SDC or to the normal cache hierarchy (L1D, L2C, LLC). LP builds its predictions on the historic knowledge of strides between memory accesses triggered by instructions with the same PC. LP classifies memory accesses as either regular or irregular. Section II indicates that accesses featuring large strides have a high probability of being served at DRAM. This behavior is a distinctive characteristic that allows LP to predict whether a certain memory access benefits from accessing the cache hierarchy or the SDC.

LP practically classifies memory accesses between cache averse and cache friendly by leveraging a PC-indexed prediction table. Each prediction table entry consists of i) a tag of the PC used to access memory (`entry.tag`); ii) the block address of the previous access performed by this PC (`entry.addr`); iii) a field that stores an accumulation of the previous strides between cache blocks accessed by instructions corresponding to this predictor entry (`entry.s_acc`); and iv) a valid bit (`entry.valid`).

1) LP Operation: Figure 4 provides a step-by-step illustration of the prediction process. Upon executing a memory instruction and prior to sending the memory request to the memory sub-system **1**, the core consults the LP providing it a tuple $(PC, v@)$, where PC is the instruction's PC and $v@$ is the block address of the fetched data. The PC is hashed into a tag and set index that are both used to look-up the prediction table. The set index is computed as $PC \mod \#sets$ and the tag as $PC \gg log_2(\#sets)$. On a prediction table hit **2**, the stride field `entry.s_acc` of the matching entry is compared to a global threshold τ_{glob}. If the value of the `entry.s_acc`

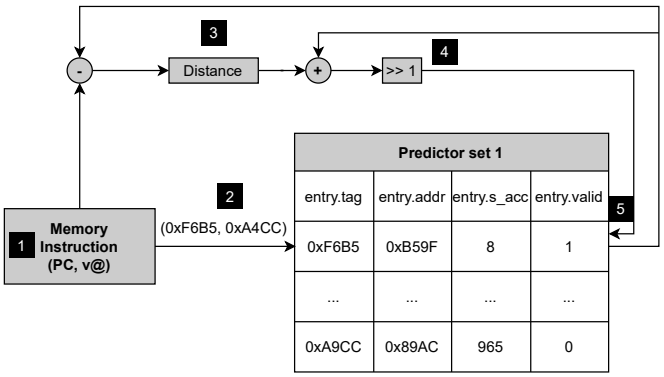


Fig. 5. Organization of the LP predictor and a functional example of its update sequence.

field satisfies the condition $entry.s\_acc \geq \tau_{glob}$, the access is served by the SDC 3. Conversely, if the `entry.s_acc` field is lower than the global threshold, the access is routed to the cache hierarchy (L1D, L2C and LLC) 4. On a prediction table miss 5, the memory access is served by the cache hierarchy.

2) LP Update: Figure 5 provides a step-by-step illustration of an operational example of a LP update. Upon the trigger of a memory access 1, the core consults the LP providing it with a tuple containing the instruction PC and the block address: $(PC, v@)$. The PC is hashed into a tag and a set index that are used to look-up the prediction table 2. On a hit in the prediction table, the block address ($entry.addr$) stored in the prediction table entry is read, and the stride s between the block address ($entry.addr$) and the current block address $v@$ is computed as $s = |v@ - entry.addr|$ 3. The value stored in the `entry.s_acc` field of the prediction table entry is accumulated to this stride s, and a right bit shift is performed on the accumulated value 4.

3) LP Replacement: Upon a prediction table miss, a victim is selected using the LRU replacement policy (*i.e.*, an entry is either selected as the LRU entry of a given set or when the `entry.valid` equals 0). The content of the victim entry is then initialized such that the tag is set to $PC \gg log_2(\#sets)$, the `entry.addr` field is initialized to the block address $v@$ attached to the instruction, the `entry.s_acc` field is initialized to 0 , and the `entry.valid` field is set to 1.

C. Coherence

To ensure coherence between the SDC and the cache hierarchy, we introduce the SDCDir, an extension of the cache directory. SDCDir entries hold the cache block's tag, coherence status, and sharer core data, as depicted in Figure 6.

The SDCDir allows for coherence without major protocol changes. Requests from SDCs and L2s simultaneously access both the cache directory and the SDCDir. In invalidation-based protocols like MESI [37] and MOESI [43], read requests check the cache directory and SDCDir together. If no valid copy is found in any cache or SDC, the request goes to DRAM. Otherwise, the request is served by the cache or SDC with the valid copy. Write requests also perform parallel directory and SDCDir lookups, invalidating the cache block in remote cores

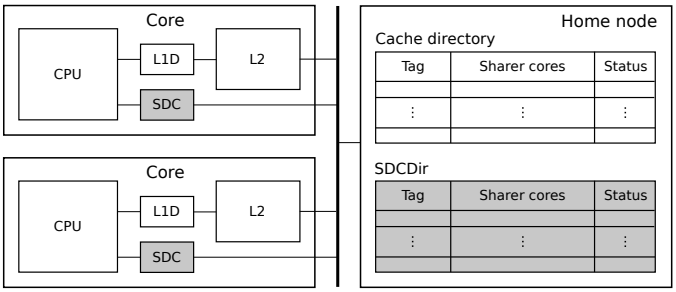


Fig. 6. Hardware support for SDC coherence.

and writing it back to DRAM if dirty. This ensures only one valid copy of a cache block in either the cache hierarchy (L1, L2, LLC) or the SDCs, except for clean blocks.

The SDCDir maintains precise information of the data stored in the SDCs. When a SDC request misses in the SDCDir, the request is served by the cache hierarchy or the DRAM and a new SDCDir entry is created for the cache block. Subsequent requests to the same cache block from any SDC update the status bits and the bit vector of sharer cores of the SDCDir entry as defined by the coherence protocol. Upon replacements in the SDCDir, the entry selected as victim is invalidated and all the copies of the corresponding cache block in all the SDCs are also invalidated, writing the block back to DRAM if needed.

D. Putting It All Together

This section explains the operation when the LP and the SDC are combined (`SDC+LP`) to accelerate the memory accesses of irregular workloads.

As shown on Figure 4, upon a memory access, LP is consulted by the core and answers with a Boolean prediction, saying whether this memory access must be directed to the SDC or to the standard cache hierarchy (L1D, L2C, LLC). If the access is directed to the SDC and the look-up hits, the data block is returned to the core benefiting the very low latency of the SDC. Otherwise, if the access misses in the SDC, a lightweight coherence message is sent to the cache directory to ensure correctness and the access is served from either a remote cache or the DRAM. In case of accessing the DRAM, the cache block is directly inserted into the SDC, bypassing the L2C and the LLC and thus reducing the latency of the memory operation while avoiding polluting the rest of the cache hierarchy. Conversely, if the LP predicts that the L1D must be accessed, the cache block is requested to the L1D and the operation of the cache hierarchy proceeds as in a regular system without SDCs.

This combination allows feeding the SDC with accesses to irregularly accessed data-structures that, if directed to the L1D, would most likely miss all the way through the cache hierarchy and access the DRAM, leading to poor performance due to the latency costs of useless cache look-ups. In addition, directing irregular memory accesses to the SDC avoids polluting the rest of the cache hierarchy. As a result, the L1D, the L2C, and the LLC experience better locality and can make better cache management decisions for cache friendly memory accesses,

Component	Description
Branch Predictor	hashed perceptron
CPU	2.166 GHz, 4-wide out-of-order processor 6-stage pipeline, 224-entries re-order buffer
L1 ITLB	64-entry, 4-way, 1-cycle latency, 8-entry MSHR
L1 DTLB	64-entry, 4-way, 1-cycle latency, 8-entry MSHR
L2 TLB	1536-entry, 12-way, 8-cycle latency, 16-entry MSHR
L1-I Cache	32 KiB, 8-way, 4-cycle latency, 10-entry MSHR
L1-D Cache	32 KiB, 8-way, 4-cycle latency, 10-entry MSHR LRU replacement, next line prefetcher
SDC	8 KiB, 2-way, 1-cycle latency, 10-entry MSHR LRU replacement, next line prefetcher
LP Predictor	552 B, 32-entries, 8-way, LRU replacement, τ_{glob}=8
L2 Cache	1 MiB, 16-way, 10-cycle latency, 16-entry MSHR LRU replacement, SPP prefetcher [28]
LLC	1.375 MiB per core, 11-way, 56-cycle latency, 64-entry MSHR LRU replacement
SDCDir	128 entries per core, 8-way, 1-cycle latency, LRU replacement
DRAM	16 GiB per core, DDR4 SDRAM data-rate: 2.933 GT/s, I/O bus frequency: 1466.5 MHz $t_{RP} = t_{RCD} = t_{CAS} = 24$ cycles

TABLE I
SYSTEM CONFIGURATION

without suffering the negative effects of handling cache averse memory accesses. In addition, capitalising on these benefits does not require modifications in any software layer and only introduces minimal storage overheads.

E. Context Switches

Upon context switches, one must consider what should happen of the information stored in the LP and the SDC. Similarly to the L1D, the SDC is a *Virtually Indexed Physically Tagged* (VIPT) cache structure, as such it does not need to be flushed upon context switches as different processes will refer to disjoint regions of the physical memory.

IV. METHODOLOGY

A. Simulation Infrastructure

We evaluate our proposal with ChampSim [4], a detailed simulator that models a 4-wide out-of-order CPU along with its cache hierarchy, prefetching mechanisms, and memory subsystem. Table I provides our system configuration based on the recent server Intel Cascade Lake micro-architecture [1].

B. Graph-Processing Applications

We use six graph-processing applications from the GAP benchmark suite [10]. Breadth-First Search (BFS) is a fundamental graph traversal algorithm. Page Rank (PR) iteratively updates per-vertex ranks until convergence. Connected Components (CC) applies the Shiloach-Vishkin [41] algorithm to compute the largest connected components of the graph. Betweenness Centrality (BC) uses the Brandes algorithm [11] to approximate the per-vertex centrality scores. Triangle Count (TC) counts the number of triangles in the graph. Finally, Single-source Shortest Paths (SSSP) uses δ-stepping [30] to return the distance of all vertices of a graph to a given source vertex. Table II shows the main characteristics of these six applications, including the size of property array elements, and input parameters like the execution style (push or pull), or the use of frontiers.

	BC [10]	**BFS** [10]	**CC** [10]	**PR** [10]	**TC** [10]	**SSSP** [10]
irregData ElemSz	8 B + 4 B	4 B	4 B	4 B	4 B	4 B
Execution style	Push-Mostly	Push & Pull	Push-Mostly	Pull-Only	Push-Only	Push-Only
Use Frontier	Yes	Yes	No	No	No	Yes

TABLE II
GRAPH KERNELS

	Web [10]	**Road** [10]	**Twitter** [10]	**Kron** [10]	**Urand** [10]	**Friendster** [47]
# Vertices (in M)	50.6	23.9	61.6	134.2	134.2	65.6
# Edges (in M)	1,949.4	58.3	1,468.4	2,111.6	2,147.4	3,612.1

TABLE III
INPUT GRAPHS

For each kernel we consider 6 different input graphs from areas like social networks or path-planning. Table III lists them. These graphs feature different sizes and distributions of node degrees (power-law, normal, etc.). Different degree-distributions produce different memory access patterns. For instance, when node degrees are distributed following a power-law function, there are a few highly connected graph nodes that yield more data reuse than vertices with a few connections.

C. Single-Core Workloads

Our set of single-thread graph-processing workloads is composed of the 36 combinations of kernels and inputs listed in Tables II and III, respectively. We use the SimPoint methodology [38] to identify intervals representative of each workload. Each SimPoint is 1 billion instructions long and characterizes a different phase of these workloads. Each SimPoint is executed for 200 million instructions to warm-up the memory hierarchy and other microarchitectural structures, and it is executed for an additional set of 200 million instructions to obtain experimental results.

D. Multi-Core Workloads

We randomly generate 50 distinct 4-thread workloads composed of mixes of all available 36 single-thread workloads. Our performance results on multi-thread workloads report the weighted speed-up normalized to the baseline. This metric is commonly used to evaluate multi-threaded scenarios [25], [40] since it avoids performance overestimation due to high-IPC threads. The metric is computed as follows: for each thread running on the simulated system, we compute its IPC in a shared environment (IPC_{shared}) and its IPC in isolation on the same system (IPC_{single}). We then compute the weighted IPC of the mix as the weighted sum of $IPC_{shared}/IPC_{single}$ for all the benchmarks in the mix, and we normalize this weighted IPC with the weighted IPC of the baseline design.

E. Alternative Approaches

We compare our proposal, `SDC+LP`, against five previously proposed approaches aimed at boosting the performance of graph-processing and irregular workloads: i) The `Transpose-based Cache Replacement (T-OPT)` [8], the state-of-the-art cache replacement policy for graph data based on the graph adjacency matrix. `T-OPT` (and its practical but less performant implementation, `P-OPT`) requires modifying the original application; ii) the `Distill Cache` [39] technique that retains only the used words within a cache line and evicts the unused ones to optimize the use of

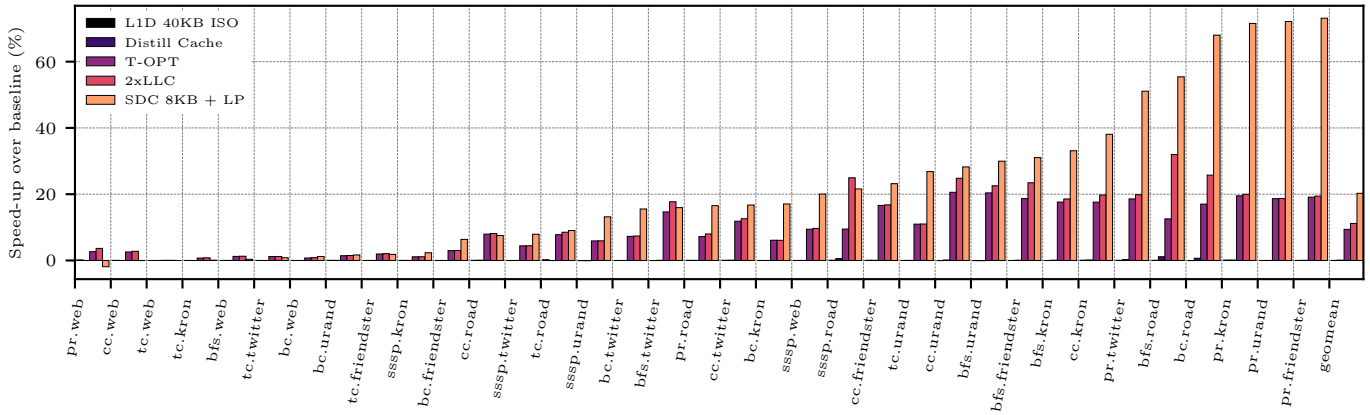


Fig. 7. Performance improvement of SDC+LP and other considered scenarios with respect to the Baseline architecture.

cache storage capacity; iii) a scenario where we enhance the L1D with 8KB of storage capacity by increasing its number of ways from 8 to 10. These additional 8KB correspond to the storage budget of the SDC per core. We call this approach L1D 40KB ISO; iv) a scenario where we double the size of the LLC by increasing the number of sets from 2048 to 4096 that we call 2xLLC; and v) an expert-based classification scheme that identifies via judicious source code and performance data analysis which data structures produce memory access patterns that are cache averse and should thus be routed to the SDC. We perform a characterization of the access patterns seen by each individual data structure for all considered workloads. We call this scenario Expert Programmer. Our evaluation also considers a Baseline configuration that corresponds to a system with a standard cache hierarchy. All system configuration parameters are specified in Table I.

V. Evaluation

This section evaluates SDC+LP. Section V-A compares SDC+LP against other hardware proposals in the single-core context, as well as an explanation of the performance delivered by SDC+LP. Section V-B presents an exhaustive design space exploration to justify SDC+LP design choices. Section V-C compares the performance achieved by SDC+LP against the Expert Programmer approach. Section V-D evaluates SDC+LP in the multi-core context. Section V-E describes the hardware cost of SDC+LP.

A. Single-Core Evaluation

This section compares SDC+LP against four hardware approaches: T-OPT, Distill Cache, L1D 40KB ISO, and 2xLLC. Section IV describes these four hardware approaches. Figure 7 shows in the y-axis the performance improvement that SDC+LP and the other hardware approaches achieve with respect to the Baseline architecture with a standard cache hierarchy. The x-axis displays the 36 graph processing workloads we consider. Both L1D 40KB ISO and the Distill Cache designs fail to provide significant performance improvements, as they only achieve geometric mean speedups of 0.0% and 0.1%, respectively. By contrast, T-OPT can provide a significant geometric mean speedup of 9.4% by making cache replacement decisions based on graph adjacency matrices. The 2xLLC is also able to significantly improve performance, achieving a geometric mean speedup of 11.2% over the baseline and slightly outperforming T-OPT. Finally, LP+SDC doubles the performance improvements of both T-OPT and 2xLLC by providing a geometric mean speedup of 20.3% over the baseline.

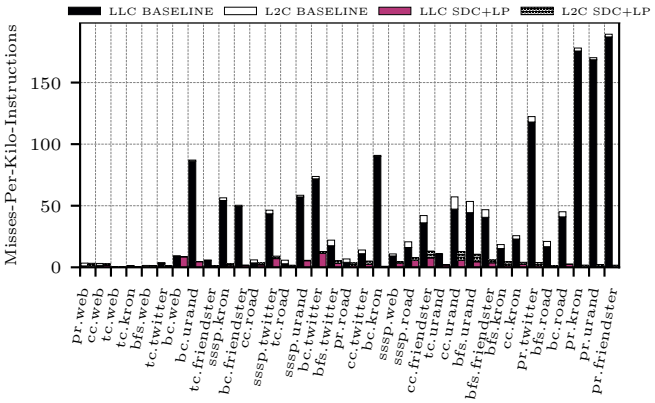


Fig. 8. MPKI of L2C and LLC relatively to theirs counterparts in the baseline.

We conduct a performance analysis considering MPKI rates of both Baseline and SDC+LP to explain the large performance improvements of SDC+LP. Figure 8 displays L2C and LLC MPKI rates considering both the Baseline and the SDC+LP approaches for each workload. Two stacked bars per workload are depicted: the left-hand side bar represents MPKI rates of Baseline, and the right-hand side bar represents MPKI rates of SDC+LP. For each bar, the lower stack refers to the LLC MPKI, and the total height refers to the L2C MPKI. Therefore, the difference between the the lower stack and the total bar height comes from accesses that miss in the L2C and hit in the LLC. Similarly, Figure 9 shows MPKI rates in the first-level caches of Baseline and SDC+LP. The leftmost bar of each workload shows the L1D MPKI of Baseline, while the rightmost bar of each workload presents the accumulated MPKI of the L1D and the SDC of SDC+LP. These two figures present the workloads sorted in ascending

order in terms of the speed-up provided by the SDC+LP design as shown on Figure 7.

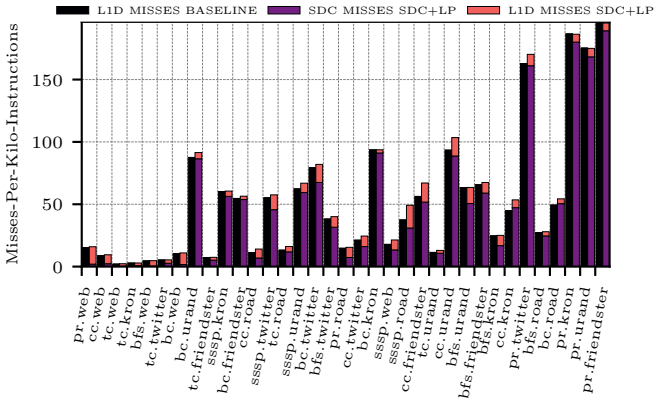


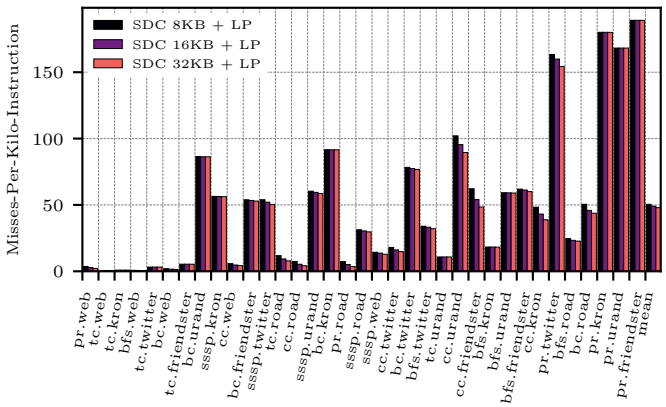
Fig. 9. Comparison of hits and misses in the L1D and the SDC using the baseline design and the SDC+LP design.

Figure 8 indicates that SDC+LP significantly reduces the pressure on both L2C and LLC with respect to Baseline. Indeed, average MPKI rates drop from 44.5 and 41.8 (Baseline) to 4.4 and 2.8 (SDC+LP) for L2C and LLC, respectively. Figure 9 shows that SDC handles the vast majority of the misses experienced by the L1D in the Baseline configuration. The Average MPKI rate of L1D decreases from 53.2 (Baseline) to 7.4 (SDC+LP) while the SDC experiences an average MPKI rate of 48.3. These measurements illustrate how the LP predictor successfully identifies cache averse accesses and redirects them to the SDC, avoiding pollution of the contents of L1D, L2C and LLC caches, and eliminates useless cache lookups for cache averse accesses. These two effects decrease the latencies of cache averse accesses, since they do not waste time in useless cache look-ups, and cache friendly accesses, since cache lines that serve them do not suffer from evictions due to insertions of cache averse data. These two effects combined explain the large benefits of SDC+LP.

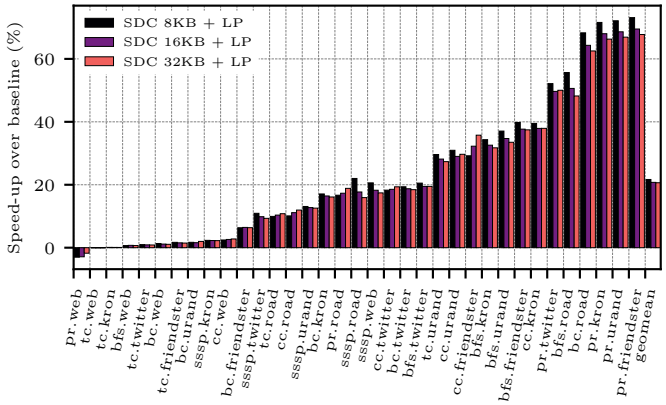
B. Design Space Exploration

1) SDC Size: We evaluate different SDC sizes (8KB, 16KB, and 32KB) and their performance impact. The SDC sizes are associated with varying associativities and latencies: the 8KB SDC is 2-way set associative with a 1-cycle latency, while the 16KB and 32KB SDCs are 4-way and 8-way set associative with 3-cycle and 4-cycle latencies, respectively. LP parameters from Table I are used. Figure 10a shows that increasing the SDC size results in only marginal reductions in SDC MPKI, with rates of 50.5, 49.1, and 48.0 for 8KB, 16KB, and 32KB SDCs, respectively. Figure 10b reveals that while the 32KB and 16KB SDCs have fewer misses than the 8KB SDC, they offer slightly smaller performance benefits due to their longer latencies. In contrast, the 8KB SDC offers fast cache look-ups, with only a slight increase in MPKI.

2) LP Organization: We evaluate the performance impact of the two most important LP configuration parameters: (i) number of entries of the LP prediction table; and (ii) associativity of the table.



(a) SDC MPKI ratios considering 8KB, 16KB, and 32KB of storage.



(b) Performance improvement of SDC+LP for 8KB, 16KB and 32KB SDCs.

Fig. 10. Exploration on the size of the SDC.

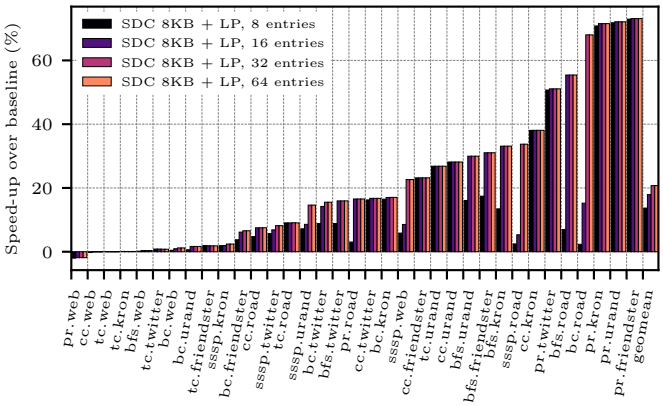


Fig. 11. Speed-up of SDC+LP considering a fully-associative LP with different entry counts.

Figure 11 shows SDC+LP's performance improvements against Baseline with fully-associative LP prediction tables of varying sizes (8, 16, 32, and 64 entries), resulting in gains of 13.7 %, 17.9 %, 20.7 %, and 20.7 %, respectively. Figure 12 explores different LP prediction table associativities (from direct-mapped to fully-associative) using a 32-entry table, achieving performance boosts of 17.0 %, 20.3 %, 20.7 %,

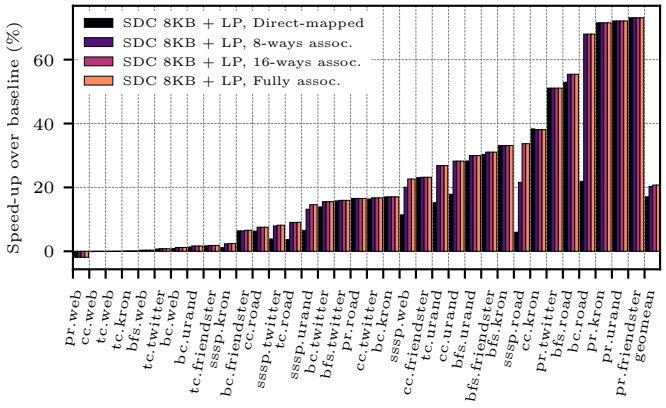


Fig. 12. Speed-up of SDC+LP considering different LP associativities.

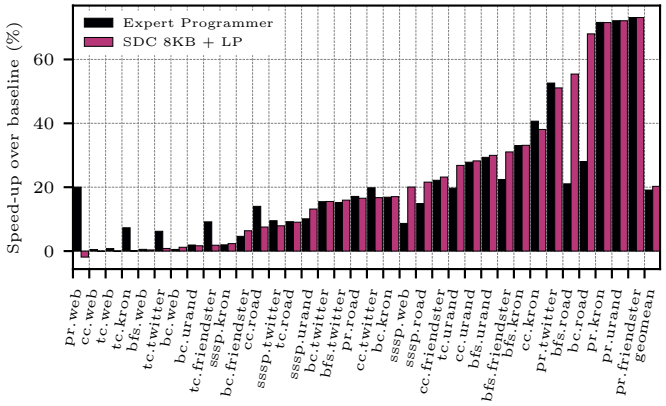


Fig. 13. Speed-up over Baseline of SDC+LP and the Expert Programmer approach, which manually categorizes cache averse and cache friendly accesses.

and 20.7 %. Notably, the 8-way design approaches optimal results.

3) Global Threshold: We assess the impact of the LP global threshold, τ_{glob}, over a range of values from 0 to 256. A τ_{glob} of 0 routes all memory accesses to the SDC, while a large value like 256 directs all accesses to the L1D, akin to the Baseline scenario. Our analysis encompasses the GAP benchmarks and the broader SPEC 2006 [2] and SPEC 2017 [3] benchmark suites to ensure that SDC+LP doesn't adversely affect general-purpose workloads. We determine that setting τ_{glob} to 8 yields significant performance gains for graph-processing workloads like GAP benchmarks (20.3 % improvement over Baseline) while maintaining the performance of general-purpose workloads like SPEC (0.5 % improvement over Baseline).

C. Comparison with An Expert Programmer

We compare SDC+LP with Expert Programmer, an approach that relies on a judicious analysis of performance data to identify data structures that display either cache averse or cache friendly memory access patterns. The SDC+LP configuration parameters are described in Table I. Expert Programmer uses an 8KB SDC configured as Table I describes. Figure 13 shows how Expert Programmer achieves a 19.1% performance improvement over Baseline, while SDC+LP achieves 20.3%. These results confirm that the LP predictor is able to distinguish between cache averse and cache friendly memory accesses and achieve very similar performance as an expert-driven approach where cache averse accesses are identified via analyzing performance data. The performance improvement of SDC+LP over Baseline is very close to the one achieved by Expert Programmer for a large portion of the 36 considered workloads. However, SDC+LP outperforms Expert Programmer on scenarios where graph connectivity is very heterogeneous and accesses to graph data are sometimes cache averse and sometimes cache friendly, like *bc.road*. Alternatively, Expert Programmer outperforms SDC+LP in scenarios where setting the τ_{glob} to 8 is not adequate, like *pr.web*.

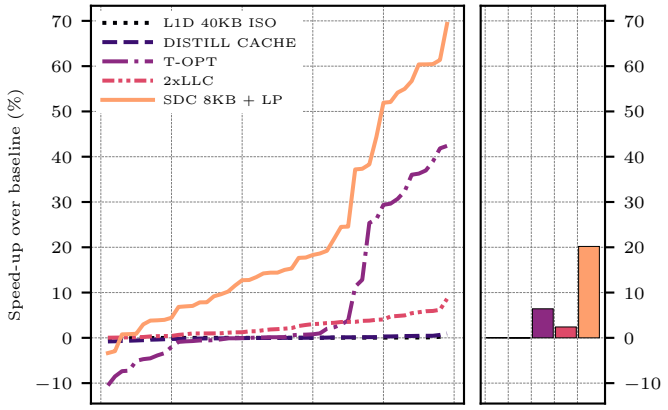


Fig. 14. Performance improvement of SDC+LP and other considered scenarios with respect to the Baseline architecture in a multi-core scenario.

D. Multi-Core Evaluation

This section evaluates in the multi-core context the performance of SDC+LP and four additional hardware approaches: T-OPT, Distill Cache, L1D 40KB ISO, and 2xLLC. Each core contains its own private SDC+LP. Figure 14 shows the performance of all evaluated designs across a set of the 50 multi-thread graph-processing workloads described in Section IV-D. The y-axis shows the performance improvement over Baseline, while the x-axis shows the 50 considered workloads sorted in terms of the improvement achieved by SDC+LP over Baseline. Figure 14 also displays the geometric mean performance improvement of all considered scenarios in terms of bars. Section IV-D describes how we compute performance in multi-core scenarios.

Figure 14 reveals that the L1D 40KB ISO and Distill Cache approaches achieve very similar performance to Baseline. Their geometric mean speed-ups are 0.02% and -0.04%, respectively. T-OPT achieves remarkable performance gains, particularly for one third of the workloads. T-OPT reaches 6.4% geometric mean improvement over the baseline. The 2xLLC scenario provides improvements above 5% for 10 workloads and achieves a 2.4% geometric mean speed-up

	Entries	Bits per entry	Total KB
SDC	128	512 data + 42 tag + 1 valid + 1 dirty	8.69
LP	32	65 tag + 58 address + 14 stride + 1 valid	0.54
SDCDir	128	42 tag + 6 state + 1 sharer per core	0.77

TABLE IV
HARDWARE BUDGET PER CORE

over the `Baseline`. `SDC+LP` outperforms all considered scenarios as it provides a 20.2% improvement over `Baseline`. `SDC+LP` achieves very remarkable performance improvements above 20% for one third of the multi-thread workloads, and reaches a maximum speed-up of 69.3%.

E. Hardware Cost and Power Considerations

Table IV details the hardware budget per core of each element of the `SDC+LP` proposal, assuming an architecture with 48-bit physical addresses. The total hardware budget of `SDC+LP` is 10KB per core. The most expensive structure in terms of area is the SDC, with a total hardware budget of 8.69 KB per core. The other two structures, LP and SDCDir, have very small area requirements of only 0.54 KB and 0.77 KB per core, respectively.

Furthermore, we employ CACTI [44] to evaluate the typical access time of LP. Utilizing the 22nm technology, we observe that LP's typical access time is estimated to be as low as 0.24ns. In comparison, our experimental setup (*cf.*, Table I) implies that a CPU cycle spans over 0.46ns (*i.e.*, the CPU clock frequency being 2.166GHz). Consequently, accessing LP comfortably fits within a single CPU cycle.

The remarkably brief timing requirements of LP present the opportunity for seamless integration into the memory pipeline, causing minimal disruption. We propose accessing LP during the same cycle as the *Address Generation Unit* (AGU), immediately after generating the memory address (a process involving a simple addition and shift) and preceding the parallel access to the TLB, L1D, or SDC. Notably, LP's leakage power is below 10mW, and its read/write accesses consume only 0.010 nJ/0.015 nJ, further underscoring its efficiency and suitability for integration.

A power consumption analysis of the SDCdir indicates that this structure consumes 0.014 nJ and 0.019 nJ during read and write operations, respectively. Similarly, the SDC consumes 0.026 nJ and 0.034 nJ per read and write access, respectively.

VI. RELATED WORK

In the recent years, numerous designs have been proposed to accelerate the irregular accesses of graph-processing applications. In Section V we compared our proposal to the most relevant published works available to date. This section comments on additional works on graph-processing workloads.

Replacement Policies. Recent literature proposes various complex cache replacement policies [23], [26], [40], [46], which enhance general-purpose computing applications but struggle with graph-processing workloads, as shown in recent research [24]. To address cache management challenges, researchers have explored cache-bypassing and reuse prediction techniques. Cache bypassing selectively prevents certain memory blocks from entering caches to avoid evicting valuable data, while reuse prediction prioritizes cache block eviction based on future reuse predictions, resulting in substantial performance gains.

Hardware Prefetching. Stream and strided cache prefetchers struggle with indirect memory access patterns in graph-processing workloads [8], [9]. Yu et al. [48] propose IMP, Ainsworth et al. [6] introduce an application-level prefetcher, and Basak et al. [9] present DROPLET, which considers reuse distances for different graph types. These hardware prefetchers can saturate memory bandwidth. In contrast, our approach optimizes graph memory access latency through efficient path routing and has the potential for enhanced performance when combined with existing prefetching techniques, which we leave for future work.

Memory System Optimizations for Graph-Processing. Recent research emphasizes memory hierarchy optimization for graph-processing workloads. Ozdal et al. [36] and Gonzalez et al. [19] employ scratchpads, including a large eDRAM scratchpad for larger graph data. Previous studies [5], [17], [32], [42], [49] reduce graph memory latency through near-memory computing. Our approach complements these efforts by enhancing memory management within the cache hierarchy.

The Distill Cache [39] optimizes L2C cache utilization by storing used words from evicted cache lines, reducing cache pollution. Our design categorizes memory accesses as regular or irregular to address irregular access patterns.

Selective Cache [18] bypasses the L1D cache for memory accesses lacking locality, while our approach leverages graph-processing memory patterns to predict when DRAM can serve accesses, optimizing the cache hierarchy for regular graph processing while avoiding unnecessary look-ups.

Victim Cache [27] reduces conflict misses by storing eviction victims but relies on spatial locality for insertion. In contrast, our approach bypasses the cache hierarchy without relying on locality assumptions when inefficiency is expected.

Pre-Processing Algorithms. Several pre-processing algorithms have been proposed [7], [14], [45] to improve the locality of graph-processing workloads. Although effective, these pre-processing works are orders of magnitude more expensive compared to the runtime of a single traversal [31]. Our work aims at reducing the latency cost of single traversals by routing each memory access to the most appropriate memory path.

Graph-Processing Accelerators. Several prior efforts have developed graph-processing accelerators [15], [21], [34]–[36] to address memory bottlenecks, introducing specialized logic and memory optimizations for improved performance. However, these approaches depend on costly pre-processing methods, as discussed above. More recently, Mukkara et al. [31] introduce a hardware-accelerated traversal scheduler that replaces pre-processing with an efficient online locality-aware scheduler. Our work is distinct from graph-processing acceleration.

VII. Conclusion

In this paper, we reveal that conventional cache hierarchies are underutilized by graph-processing workloads, with 78.6 % of L1D misses cascading into L2C and LLC, resulting in costly DRAM accesses. We also observe that DRAM-served memory accesses often exhibit long strides compared to the last access with the same program counter (PC). In response, we propose `SDC+LP`, a cost-effective solution that eliminates unnecessary L2C and LLC look-ups for cache-averse memory accesses.

`SDC+LP` consists of two components: the Side Data Cache (SDC), a per-core auxiliary set-associative cache placed alongside L1D, and the Large Predictor (LP), a low-cost microarchitectural predictor that distinguishes regular from irregular memory accesses based on historic stride knowledge. Combining SDC with LP results in a 20.3 % geometric mean speed-up over the baseline and an additional 10.9 % geometric mean speed-up compared to T-OPT, a state-of-the-art cache management scheme for graph-processing workloads, in the single-core context. In multi-core scenarios, our proposal achieves a 20.2 % speed-up over the baseline and an additional 13.8 % compared to T-OPT.

Our study emphasizes that hardware approaches capable of distinguishing between regular and irregular cache-averse memory references can significantly enhance the performance of workloads characterized by large memory footprints and data-driven memory access patterns.

Acknowledgments

The authors are grateful to the anonymous reviewers for their valuable comments and constructive feedback that significantly improved the quality of the paper. This work has been partially supported by the European HiPEAC Network of Excellence, by the Spanish Ministry of Science and Innovation MCIN/AEI/10.13039/501100011033 (contracts PID2019-107255GB-C21 and PID2019-105660RB-C22) and by the Generalitat de Catalunya (contract 2021-SGR-00763). This work is supported by the National Science Foundation through grant CCF-1912617 and generous gifts from Intel. Marc Casas has been partially supported by the Grant RYC-2017-23269 funded by MCIN/AEI/10.13039/501100011033 and by ESF Investing in your future. Els autors agraeixen el suport del Departament de Recerca i Universitats de la Generalitat de Catalunya al Grup de Recerca "Performance understanding, analysis, and simulation/emulation of novel architectures" (Codi: 2021 SGR 00865). This research has received funding from the European High Performance Computing Joint Undertaking (JU) under Framework Partnership Agreement No 800928 (European Processor Initiative) and Specific Grant Agreement No 101036168 (EPI SGA2). The JU receives support from the European Union's Horizon 2020 research and innovation programme and from Croatia, France, Germany, Greece, Italy, Netherlands, Portugal, Spain, Sweden, and Switzerland. The EPI-SGA2 project, PCI2022-132935 is also co-funded by MCIN/AEI /10.13039/501100011033 and by the UE NextGenerationEU/PRTR.

References

[1] Cascade lake - microarchitectures - intel - WikiChip. [Online]. Available: https://en.wikichip.org/wiki/intel/microarchitectures/cascade_lake#Memory_Hierarchy

[2] SPEC CPU® 2006. [Online]. Available: https://www.spec.org/cpu2006/

[3] SPEC CPU® 2017. [Online]. Available: https://www.spec.org/cpu2017/

[4] "ChampSim," https://crc2.ece.tamu.edu/, 2021, [Online].

[5] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015, pp. 105–117.

[6] S. Ainsworth and T. M. Jones, "Graph Prefetching Using Data Structure Knowledge," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2925426.2926254

[7] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura, "Rabbit Order: Just-in-Time Parallel Reordering for Fast Graph Analysis," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 22–31.

[8] V. Balaji, N. Crago, A. Jaleel, and B. Lucia, "P-OPT: Practical Optimal Cache Replacement for Graph Analytics," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 668–681.

[9] A. Basak, S. Li, X. Hu, S. M. Oh, X. Xie, L. Zhao, X. Jiang, and Y. Xie, "Analysis and optimization of the memory hierarchy for graph processing workloads," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 373–386.

[10] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[11] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of mathematical sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[12] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer networks and ISDN systems*, vol. 30, no. 1-7, pp. 107–117, 1998.

[13] T. D. Bui, S. Ravi, and V. Ramavajjala, "Neural graph learning: Training neural networks using graphs," in *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, ser. WSDM '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 64–71. [Online]. Available: https://doi.org/10.1145/3159652.3159731

[14] E. Cuthill and J. McKee, "Reducing the Bandwidth of Sparse Symmetric Matrices," in *Proceedings of the 1969 24th National Conference*, ser. ACM '69. New York, NY, USA: Association for Computing Machinery, 1969, p. 157?172. [Online]. Available: https://doi.org/10.1145/800195.805928

[15] G. Dai, Y. Chi, Y. Wang, and H. Yang, "Fpgp: Graph processing framework on fpga a case study of breadth-first search," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 105?110. [Online]. Available: https://doi.org/10.1145/2847263.2847339

[16] O. Evelien and R. Rousseau, "Social network analysis: A powerful strategy, also for the information sciences," *Journal of Information Science*, vol. 28, no. 6, p. 441–453, December 2002.

[17] M. Gao, G. Ayers, and C. Kozyrakis, "Practical near-data processing for in-memory analytics frameworks," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, 2015, pp. 113–124.

[18] A. González, C. Aliagas, and M. Valero, "A data cache with multiple caching strategies tuned to different types of locality," in *ACM International Conference on Supercomputing 25th Anniversary Volume*, 1995, pp. 217–226.

[19] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph Processing in a Distributed Dataflow Framework," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. USA: USENIX Association, 2014, p. 599?613.

[20] J. L. Greathouse and M. Daga, "Efficient sparse matrix-vector multiplication on gpus using the csr storage format," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 769–780.

[21] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.

[22] C. T. Have and L. J. Jensen, "Are graph databases ready for bioinformatics?" *Bioinformatics*, vol. 29, no. 24, pp. 3107–3108, 10 2013. [Online]. Available: https://doi.org/10.1093/bioinformatics/btt549

[23] A. Jaleel, K. B. Theobald, S. C. Steely, and J. Emer, "High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 60–71. [Online]. Available: https://doi.org/10.1145/1815961.1815971

[24] A. V. Jamet, L. Alvarez, D. A. Jiménez, and M. Casas, "Characterizing the impact of last-level cache replacement policies on big-data workloads," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2020, pp. 134–144. [Online]. Available: https://upcommons.upc.edu/bitstream/handle/2117/343622/IISWC20-paper.pdf?sequence=1

[25] D. A. Jiménez and E. Teran, "Multiperspective reuse prediction," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2017, pp. 436–448.

[26] D. A. Jiménez and E. Teran, "Multiperspective reuse prediction," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2017, pp. 436–448.

[27] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *ACM SIGARCH Computer Architecture News*, vol. 18, no. 2SI, pp. 364–373, 1990.

[28] J. Kim, S. H. Pugsley, P. V. Gratz, A. N. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[29] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, "A unified sparse matrix data format for modern processors with wide SIMD units," *CoRR*, vol. abs/1307.6209, 2013. [Online]. Available: http://arxiv.org/abs/1307.6209

[30] U. Meyer and P. Sanders, "δ-stepping: a parallelizable shortest path algorithm," *Journal of Algorithms*, vol. 49, no. 1, pp. 114–152, 2003.

[31] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, "Exploiting locality in graph analytics through hardware-accelerated traversal scheduling," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 1–14.

[32] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 457–468.

[33] K. Nilakant, V. Dalibard, A. Roy, and E. Yoneki, "Prefedge: Ssd prefetcher for large-scale graph traversal," in *Proceedings of International Conference on Systems and Storage*, ser. SYSTOR 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 1?12. [Online]. Available: https://doi.org/10.1145/2611354.2611365

[34] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martínez, and C. Guestrin, "Graphgen: An fpga framework for vertex-centric graph computation," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2014, pp. 25–28.

[35] T. Oguntebi and K. Olukotun, "Graphops: A dataflow library for graph analytics acceleration," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 111?117. [Online]. Available: https://doi.org/10.1145/2847263.2847337

[36] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy Efficient Architecture for Graph Analytics Accelerators," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. IEEE Press, 2016, p. 166–177. [Online]. Available: https://doi.org/10.1109/ISCA.2016.24

[37] M. S. Papamarcos and J. H. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," in *11th annual International Symposium on Computer Architecture (ISCA)*. New York, NY, USA: Association for Computing Machinery, 1984, p. 348–354. [Online]. Available: https://doi.org/10.1145/800015.808204

[38] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 1, pp. 318–319, 2003.

[39] M. K. Qureshi, M. A. Suleman, and Y. N. Patt, "Line Distillation: Increasing Cache Capacity by Filtering Unused Words in Cache Lines," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, pp. 250–259.

[40] Z. Shi, X. Huang, A. Jain, and C. Lin, "Applying deep learning to the cache replacement problem," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 413–425.

[41] Y. Shiloach and U. Vishkin, "An o (log n) parallel connectivity algorithm," Computer Science Department, Technion, Tech. Rep., 1980.

[42] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "GraphR: Accelerating Graph Processing Using ReRAM," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 531–543.

[43] P. Sweazey and A. J. Smith, "A class of compatible cache consistency protocols and their support by the ieee futurebus," in *Proceedings of the 13th Annual International Symposium on Computer Architecture*, ser. ISCA '86. Washington, DC, USA: IEEE Computer Society Press, 1986, p. 414–423.

[44] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "Cacti 5.1," Technical Report HPL-2008-20, HP Labs, Tech. Rep., 2008.

[45] H. Wei, J. X. Yu, C. Lu, and X. Lin, "Speedup graph processing by graph ordering," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1813?1828. [Online]. Available: https://doi.org/10.1145/2882903.2915220

[46] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely Jr, and J. Emer, "Ship: Signature-based hit predictor for high performance caching," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 430–441.

[47] J. Yang and J. Leskovec, "Community-affiliation graph model for overlapping network community detection," in *2012 IEEE 12th international conference on data mining*. IEEE, 2012, pp. 1170–1175.

[48] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "IMP: Indirect Memory Prefetcher," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: Association for Computing Machinery, 2015, p. 178–190. [Online]. Available: https://doi.org/10.1145/2830772.2830807

[49] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, "GraphP: Reducing Communication for PIM-Based Graph Processing with Efficient Data Partition," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 544–557.

[50] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia, "Making caches work for graph analytics," in *2017 IEEE International Conference on Big Data (Big Data)*, 2017, pp. 293–302.