In Proceedings of the 26th International Symposium on High-Performance Computer Architecture (HPCA'20)

Domain-Specialized Cache Management for Graph Analytics

Priyank Faldu
The University of Edinburgh
priyank.faldu@ed.ac.uk

Jeff Diamond
Oracle Labs
jeff.diamond@oracle.com

Boris Grot
The University of Edinburgh
boris.grot@ed.ac.uk

***Abstract*—Graph analytics power a range of applications in areas as diverse as finance, networking and business logistics. A common property of graphs used in the domain of graph analytics is a *power-law distribution* of vertex connectivity, wherein a small number of vertices are responsible for a high fraction of all connections in the graph. These richly-connected, *hot*, vertices inherently exhibit high reuse. However, this work finds that state-of-the-art hardware cache management schemes struggle in capitalizing on their reuse due to highly irregular access patterns of graph analytics.**

In response, we propose GRASP, domain-specialized cache management at the last-level cache for graph analytics. GRASP augments existing cache policies to maximize reuse of hot vertices by protecting them against cache thrashing, while maintaining sufficient flexibility to capture the reuse of other vertices as needed. GRASP keeps hardware cost negligible by leveraging lightweight software support to pinpoint hot vertices, thus eliding the need for storage-intensive prediction mechanisms employed by state-of-the-art cache management schemes. On a set of diverse graph-analytic applications with large high-skew graph datasets, GRASP outperforms prior domain-agnostic schemes on all datapoints, yielding an average speed-up of 4.2% (max 9.4%) over the best-performing prior scheme. GRASP remains robust on low-/no-skew datasets, whereas prior schemes consistently cause a slowdown.

***Keywords*-graph analytics; graph reordering; skew; last-level cache; cache management; domain-specialized;**

I. Introduction

Graph analytics is an exciting and rapidly growing field with applications spanning diverse areas such as optimizing routes, uncovering latent relationships, pinpointing influencers in social graphs, and many more. Graph analytics commonly process large graph datasets whose main memory footprint span tens to hundreds of gigabytes [4, 54]. When processing such large graphs, graph-analytic applications exhibit a lack of cache locality, leading to frequent misses in on-chip caches, compromising application performance [2, 3, 19, 20, 30, 31, 36].

A distinguishing property of graph datasets common in many graph-analytic applications is that the vertex degrees follow a skewed *power-law distribution*, in which a small fraction of vertices have many connections while the majority of vertices have relatively few connections [2, 3, 19, 45, 65, 66]. Graphs characterized by such a distribution are known as *natural* or *scale-free* graphs and are prevalent in a variety of domains, including social networks, computer networks, financial networks, semantic networks, and airline networks.

We observe that the skew in the degree distribution means that a small set of vertices with a large fraction of connections is responsible for a major share of off-chip memory accesses. The fact that these richly-connected vertices, *hot vertices*, comprise a small fraction of the overall footprint while exhibiting high reuse makes them prime candidates for caching. Meanwhile, the rest of the vertices, *cold vertices*, comprise a large fraction of the overall footprint while exhibiting low or no reuse.

We find that the existing caches struggle in exploiting the high reuse inherent in hot vertices due to the following two reasons: First, graph-analytic applications are notorious for exhibiting irregular access patterns that cause severe *cache thrashing* when processing large graphs. Accesses to a large number of cold vertices are responsible for thrashing, often forcing hot vertices out of the cache. Second, hot vertices are sparsely distributed throughout the memory space, exhibiting a lack of spatial locality. When hot vertices share the same cache block with cold vertices, valuable cache space is underutilized. Existing software techniques [2, 3, 19] solve the latter problem by reordering vertices in the memory space, such that hot vertices share cache blocks with other hot vertices. However, the former problem of protecting hot vertices from premature evictions remains an open challenge, even for the state-of-the-art thrash-resistant hardware cache management schemes.

Almost all prior works on hardware cache management (i.e., cache replacement) that target cache thrashing are *domain-agnostic*. These hardware schemes aim to perform two tasks: (1) identify cache blocks that are likely to exhibit high reuse, and (2) protect high reuse cache blocks from cache thrashing. To accomplish the first task, these schemes deploy either probabilistic or prediction-based hardware mechanisms [5, 10, 13, 26, 28, 29, 41, 49, 51, 52, 53, 57, 58, 59, 60]. However, our work finds that graph-dependent irregular access patterns prevent these schemes from correctly learning which cache blocks to preserve, rendering them deficient for the broad domain of graph analytics. Meanwhile, to accomplish the second task, recent work proposes *pinning* of high-reuse cache blocks in LLC to ensure that these blocks are not evicted [7]. However, we find that pinning-based schemes are overly rigid and result in sub-optimal utilization of cache capacity.

To overcome the limitations of existing hardware cache management schemes, we propose *GRASP – GRAph-SPecialized* Last-Level Cache (LLC) management. To the best of our knowledge, this is the first work to introduce domain-specialized cache management for the domain of graph analytics. GRASP augments existing cache insertion and hit-promotion policies to provide preferential treatment to cache blocks containing hot vertices to shield them from thrashing. To cater to the irregular access patterns, GRASP policies are designed to be flexible to cache other blocks exhibiting reuse. Unlike pinning, GRASP maximizes cache efficiency based on observed access patterns.

GRASP relies on lightweight software support to accurately pinpoint hot vertices amidst irregular access patterns, in contrast to state-of-the-art schemes that rely on storage-intensive hardware mechanisms. By leveraging existing vertex reordering techniques, GRASP enables a lightweight software-hardware interface comprising of only a few configurable registers, which are programmed by software using its knowledge of the graph data structures.

GRASP requires minimal changes to the existing micro-architecture as GRASP only augments existing cache policies and its interface is lightweight. GRASP does not require additional metadata in the LLC or storage-intensive prediction tables. Thus, GRASP can easily be integrated into commodity server processors, enabling domain-specific acceleration for graph analytics at minimal hardware cost.

To summarize, our contributions are as follows:

- We qualitatively and quantitatively show that a wide range of state-of-the-art domain-agnostic cache management schemes, despite their sophisticated prediction mechanisms, are inefficient for the domain of graph analytics.
- We introduce *GRASP*, graph-specialized LLC management for graph analytics in processing natural graphs. GRASP augments existing cache policies to protect hot vertices against cache thrashing while also maintaining flexibility to capture reuse in other cache blocks. GRASP leverages a lightweight software interface to pinpoint hot vertices amidst irregular accesses, which eliminates the need for prediction metedata storage at the LLC, keeping the existing cache structure largely unchanged.
- Our evaluation on several multi-threaded graph-analytic applications operating on large, high-skew datasets shows that GRASP outperforms state-of-the-art domain-agnostic schemes on all datapoints, yielding an average speed-up of 4.2% (max 9.4%) over the best-performing prior scheme. GRASP is also robust on low-/no-skew datasets whereas prior schemes consistently cause a slowdown.

II. Motivation

A. Skew in Natural Graphs

A distinguishing property of natural graphs is the *skew* in their degree distribution [2, 3, 19, 45, 65, 66]. The skew follows a power-law, with the vast majority of the vertices having relatively few edges and a small fraction of vertices featuring a large number of edges. Such skewed distribution is prevalent in many domains and found, for instance, in nodes in large-scale communication networks (e.g., the internet), web pages in the web graph, and individuals in social graphs.

Table I
Rows #2 and #4 show the percentage of vertices having degree equal or greater than the average (i.e., hot vertices), with respect to in-edges and out-edges, respectively; the higher the skew, the lower the percentage. Rows #3 and #5 show the percentage of in-edges and out-edges connected to the hot vertices, respectively; the higher the skew, the higher the percentage.

	Dataset	*lj*	*pl*	*tw*	*kr*	*sd*
In Edges	Hot Vertices (%)	25	16	12	9	11
	Edge Coverage (%)	81	83	84	93	88
Out Edges	Hot Vertices (%)	26	13	10	9	13
	Edge Coverage (%)	82	88	83	93	88

Table I quantifies the skew for the datasets evaluated in this work (more details of the datasets in Table V). For example, in the Twitter [54] dataset (labeled *tw*), 12% of total vertices are classified as hot vertices in terms of their in-degree (10% for out-degree) distribution. These hot vertices are connected to 84% of all in-edges (83% of all out-edges) in the graph. Similarly, in other datasets, 9-26% of vertices are classified as hot vertices, which are connected to 81-93% of all edges. In the following sections, we explain how this skew can be leveraged to improve cache efficiency.

B. Graph Processing Basics

The majority of shared-memory graph frameworks are based on a vertex-centric model, in which an application computes some information for each vertex based on the properties of its neighbouring vertices [32, 35, 42, 43, 46, 55]. Applications may perform pull- or push-based computations. In pull-based computations, a vertex pulls updates from its in-neighbors. In push-based computations, a vertex pushes updates to its out-neighbors. This process may be iterative, and all or only a subset of vertices may participate in a given iteration.

The *Compressed Sparse Row (CSR)* format is commonly used to represent graphs in a storage-efficient manner. CSR uses a pair of arrays, *Vertex* and *Edge*, to encode the graph. CSR encodes in-edges for pull-based computations and out-edges for push-based computations. In this discussion, we focus on pull-based computations and note that the observations hold for push-based computation. For every vertex, the Vertex Array maintains an index that points to its first in-edge in the Edge Array. The Edge Array stores all in-edges, grouped by destination vertex ID. For each in-edge, the Edge Array entry stores the associated source vertex ID.

The graph applications use an additional *Property* Array(s) to hold partial or final results for every vertex. For example, the *Pagerank* application maintains two ranks for every

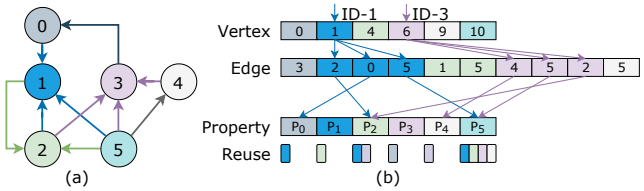


Figure 1. (a) An example graph. (b) CSR format encoding in-edges. Elements of the same colors in all arrays, correspond to the same destination vertex. Number of bars (labeled *Reuse*) below each element of the Property Array shows the number of times an element is accessed in one full iteration, where the color of a bar indicates the vertex making an access.

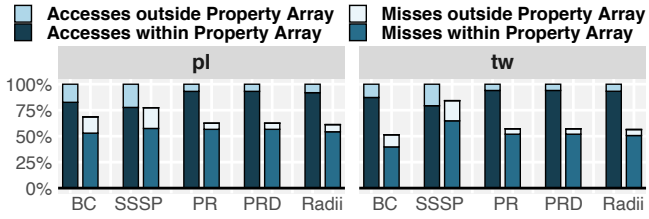


Figure 2. Classification of LLC accesses and misses (normalized to total accesses) for the *pl* and *tw* datasets for the applications from Table III.

vertex; one computed from the previous iteration and one being computed in the current iteration. Implementation may use either two separate arrays (each storing one rank per vertex) or may use one array (storing two ranks per vertex). Fig. 1(a) and 1(b) shows a simple graph and its CSR representation for pull-based computations, along with one Property Array.

C. Cache Behavior in Graph Analytics

At the most fundamental level, a graph application computes a property for a vertex based on the properties of its neighbours. To find the neighbouring vertices, an application traverses the portion of the Edge Array corresponding to a given vertex, and then accesses elements of the Property Array corresponding to these neighbouring vertices. Fig. 1(b) highlights the elements accessed during the computations for vertex ID-1 and ID-3.

As the figure shows, each element in the Vertex and the Edge Array is accessed exactly once during an iteration, exhibiting no temporal locality at LLC. These arrays may exhibit high spatial locality, which is filtered by the L1-D cache, leading to a streaming access pattern in the LLC.

In contrast, the Property Array does exhibit temporal reuse. However, reuse is not consistent for all elements. Specifically, reuse is proportional to the number of out-edges for pull-based algorithms. Thus, the elements corresponding to high out-degree vertices exhibit high reuse. Fig. 1(b) shows the reuse for high out-degree (i.e., hot) vertices P_2 and P_5 of the Property Array assuming pull-based computations; other elements do not exhibit reuse. The same observation applies to high in-degree vertices in push-based algorithms.

Finally, Fig. 2 quantifies the LLC behavior of the graph applications listed in Table III on the *pl* and *tw* datasets as representative examples (Refer to Sec. IV for methodology details). The figure differentiates all LLC accesses and misses as falling either within or outside the Property Array. Unsurprisingly, the Property Array accounts for 78-94% of all LLC accesses. However, despite the high reuse, the Property Array is also responsible for a large fraction of LLC misses, the reasons for which are explained next.

D. Challenges in Caching the Property Array

As discussed in the previous section, elements in the Property Array corresponding to the hot vertices exhibit high reuse. Unfortunately, on-chip caches struggle in capitalizing on the high reuse for the following two reasons:

① ***Lack of spatial locality:*** the hot vertices are sparsely distributed throughout the memory space of the Property Array. Moreover, the size of each element in the Property Array is much smaller than the size of a cache block. Thus, inevitably, hot vertices share space in a cache block with cold vertices. This leads to cache underutilization as a considerable fraction of a cache block capacity is occupied by the cold vertices.

② ***Thrashing in the LLC:*** the access pattern to the Property Array is highly irregular, being heavily dependent on both graph structure and application. Between a pair of accesses to a given hot vertex in the Property Array, a number of other, unrelated, cache blocks may be accessed, leading to thrashing. Any block allocated by these unrelated accesses will trigger evictions at the LLC, potentially displacing blocks holding hot vertices.

Overcoming the former problem requires improving cache block utilization by focusing on intra-block reuse, which is effectively addressed by existing software techniques [2, 3, 19]. The latter problem requires retaining high-reuse blocks in the LLC by focusing on reuse across cache blocks; as explained below, this remains an open challenge not addressed by prior works. We note that the two problems are orthogonal in nature; solving one problem does not solve the other one.

We next discuss most relevant state-of-the-art techniques in both software and hardware and their shortcomings in addressing cache thrashing for graph analytics.

E. Prior Software Schemes

Prior works have proposed leveraging application-visible properties, such as vertex connectivity or vertex degree, to improve cache locality. This is accomplished by reordering vertices in memory such that vertices that will be frequently accessed together are kept close by. These techniques are particularly attractive because they require no modifications to the graph algorithms [2, 3, 15, 19, 21, 30, 47, 48, 69, 70, 71]. To be effective, these techniques face two constraints during reordering. First, they must keep the reordering

cost to a minimum to improve end-to-end application performance. Second, they should minimize disruption to the underlying graph structure, specifically for graphs that exhibit community structure. Prior works have noted that vertex order for many real-world graph datasets closely follow underlying community structure, meaning vertices from the same community are ordered close by in memory, exhibiting good spatio-temporal locality that should be preserved [2, 3, 19].

While finding an optimal vertex ordering is NP-hard, techniques like Gorder attempt to approximate such ordering by comprehensively analyzing the graph structure based on a likely traversal order [30]. However, recent works show that while such techniques are effective in reducing LLC misses, they incur a staggering reordering cost that is often multiple orders of magnitude higher than the application runtime, thus rendering them impractical [2].

Consequently, the same works argue for lightweight skew-aware techniques that provide application speed-up even after accounting for their reordering cost. To keep the reordering cost low, these techniques reorder vertices solely based on vertex degree, with the goal of improving spatial locality by ensuring that hot vertices share cache blocks only with other hot vertices. To achieve this effect, skew-aware techniques (e.g., HubSort [19] and DBG [2]) rely on some variant of degree-based sorting to segregate hot vertices form the cold ones. While effective in improving spatial locality (as explained in Sec. II-D), these techniques still suffer from cache thrashing stemming from the fact that the footprint of only hot vertices also exceeds the available cache capacity [2].

F. Prior Hardware Schemes

Prior hardware cache management schemes targeting cache thrashing can be broadly classified into three categories:

① ***History-agnostic lightweight schemes*** use simple heuristics to manage cache [41, 51, 52, 57, 58, 60]. RRIP [52] is the state-of-the-art technique in this category that relies on a probabilistic approach to classify a cache block as low- or high-reuse at the time of inserting a new block in the cache. As these techniques do not record the past reuse behavior of cache blocks, they are limited in accurately identifying high-reuse blocks.

② ***History-based predictive schemes*** such as the state-of-the-art Hawkeye [26] and many others [5, 10, 13, 28, 29, 49, 53] learn past reuse behavior of cache blocks by employing sophisticated storage-intensive prediction mechanisms. A large body of recent works focus on history-based schemes as they generally provide higher performance than the lightweight schemes for a wide range of applications. However, for graph analytics, we find that graph-dependent irregular access patterns prevent these history-based schemes from correctly learning which cache blocks to preserve. For example, most history-based schemes rely on a PC-based (*Program Counter*) reuse correlation[1] to learn which set of PC addresses access high-reuse cache blocks to prioritize these blocks for caching over others. Meanwhile, we observe that the reuse for elements of the Property Array, which are the prime target for LLC caching in graph analytics (Sec II-C), does not correlate with the PC because the same PC accesses hot and cold vertices alike.

③ ***Pinning-based schemes*** such as XMem [7] dedicate partial or full cache capacity by *pinning* high-reuse blocks to cache. Hardware ensures that the pinned blocks cannot be evicted by other cache blocks and thus are protected from cache thrashing. Such an approach is only feasible when the high-reuse working set fits in the available cache capacity. Unfortunately, for large graph datasets, even with high skew, it is unlikely that all hot vertices will fit in the LLC; recall from Table I that hot vertices account for up to 26% of the total vertices. Moreover, some of the colder vertices might also exhibit short-term temporal reuse, particularly in graphs with community structure.

These observations call for a new LLC management scheme that employ (1) a reliable mechanism to identify hot vertices amidst irregular access patterns and (2) flexible cache policies that maximize reuse among hot vertices by protecting them in the cache without denying colder vertices the ability to be cache resident if they exhibit reuse.

III. GRASP: Caching in on the Skew

This work introduces GRASP, graph-specialized LLC management for graph analytics processing natural graphs. GRASP augments existing cache management schemes with simple additions to their insertion and hit-promotion policies that provide preferential treatment to cache blocks containing hot vertices to protect them from thrashing. GRASP policies are sufficiently flexible to capture reuse of other blocks as needed.

GRASP's domain-specialized design is influenced by the following two challenges faced by existing hardware cache management schemes. First, hardware alone cannot enforce spatial locality, which is dictated by vertex placement in the memory space and is under software control. Second, domain-agnostic hardware cache management schemes struggle in pinpointing hot vertices under cache thrashing due to irregular access patterns endemic of graph analytics.

To overcome both challenges, GRASP relies on existing skew-aware reordering software techniques to induce spatial locality by segregating hot vertices in a contiguous memory region [2, 19]. While these techniques offer different trade-offs in terms of reordering cost and their ability to preserve graph structure, they all work by isolating hot vertices from the cold ones. Fig. 3(a) shows a logical view of the placement of hot vertices in the Property Array after reordering by such

[1] All seven schemes [8, 9, 12, 14, 16, 17, 18] presented at the Cache Replacement Championship'17 [11] rely on PC-based reuse correlation.

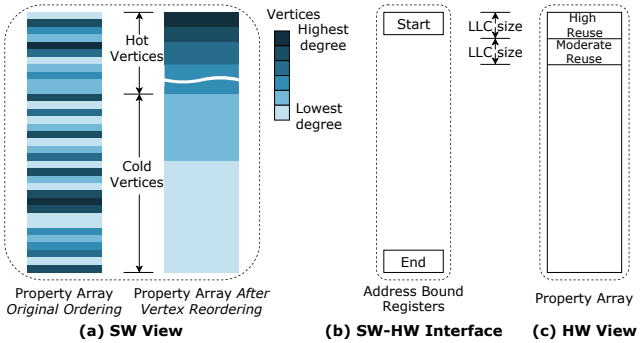


Figure 3. GRASP overview. (a) Software applies vertex reordering, which segregates hot vertices at the beginning of the array. (b) GRASP interface exposes an ABR pair per Property Array to be configured with the bounds of the array. (c) GRASP identifies regions exhibiting different reuse based on an LLC size.

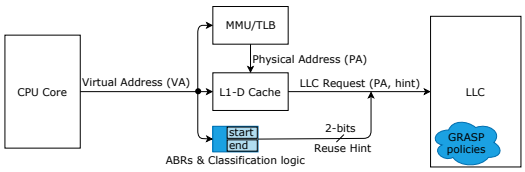


Figure 4. Block diagram of GRASP and other hardware components with which it interacts. GRASP components are shown in color. For brevity, the figure shows only one CPU core.

a technique. GRASP subsequently leverages the contiguity among hot vertices in the memory space to (1) pinpoint them via a lightweight interface and (2) protect them from thrashing. GRASP design consists of three hardware components as follows.

Ⓐ *Software-hardware interface:* GRASP interface is minimal, consisting of a few configurable registers that software populates with the bounds of the Property Array during the initialization of an application (see Fig. 3(b)). Once populated, GRASP does not rely on any further intervention from software.

Ⓑ *Classification logic:* GRASP logically partitions the Property Array into different regions based on expected reuse. (See Fig. 3(c)). GRASP implements simple comparison-based logic, which, at runtime, classifies whether a cache request belongs to one of these regions.

Ⓒ *Specialized cache policies:* GRASP specializes cache policies for each region to ensure hot vertices are protected from thrashing while retaining flexibility in caching other blocks. The classification logic informs the choice of which policy to apply to a given cache block.

Fig. 4 shows how GRASP interacts with other hardware components in the system. In the following sections, we describe each of GRASP's components in detail.

A. Software-Hardware Interface

GRASP's interface consists of one pair of *Address Bound Registers (ABR)* per Property Array; recall from Sec. II-B that an application may maintain more than one Property Array, each of which requires a dedicated ABR pair. ABRs are part of an application context and are exposed to the software. At application start-up, the graph framework populates each ABR pair with the start and end virtual address of the entire Property Array (Fig. 3(b)). Setting these registers activates the custom cache management for graph analytics. When the ABRs are not set by the software (i.e., the default case for other applications), specialized cache management is essentially disabled.

The use of virtual addresses keeps the GRASP interface independent of the existing TLB design, allowing GRASP to perform address classification (described next) in parallel with the usual virtual-to-physical address translation carried out by TLB (see Fig. 4). Prior works have used similar approaches to pass data-structure bounds to aid micro-architecture mechanisms [7, 20, 27, 39].

B. Classification Logic

This component of GRASP is responsible for reliably identifying cache blocks containing hot vertices in hardware by leveraging the bounds of the Property Array(s) available in the ABRs as explained in the following sections:

Identifying Hot Vertices: In theory, all hot vertices should be cached. In practice, it is unlikely that all hot vertices will fit in the LLC for large datasets. In such a case, providing preferential treatment to all hot vertices is *not* beneficial as they can thrash each other in the LLC. To avoid this problem, GRASP prioritizes cache blocks containing only a subset of hot vertices, comprised of only the hottest vertices based on available LLC capacity. Conveniently, the hottest vertices are located at the beginning of the Property Array in a contiguous region thanks to the application of skew-aware reordering as seen in Fig. 3(a).

Pinpointing the High Reuse Region: GRASP labels two LLC-sized sub-regions within the Property Array: The LLC-sized memory region at the start of the Property Array is labeled as *High Reuse Region*; another LLC-sized memory region starting immediately after the High Reuse Region is labeled as the *Moderate Reuse Region* (Fig. 3(c)). Finally, if an application specifies more than one Property Array, GRASP divides LLC-size by the number of Property Arrays before labeling the regions.

Classifying LLC Accesses: At runtime, GRASP classifies a memory address making an LLC access as *High-Reuse* if the address belongs to the High Reuse Region of any Property Array; GRASP determines this by comparing the address with the bounds of the High Reuse Region of each Property Array. Similarly, an address is classified as *Moderate-Reuse* if the address belongs to the Moderate Reuse Region. All other LLC accesses are classified as *Low-Reuse*. For non-graph applications, the ABRs are not initialized and all accesses are classified as *Default*, effectively disabling domain-specialized

cache management. GRASP encodes the classification result (High-Reuse, Moderate-Reuse, Low-Reuse or Default) as a 2-bit *Reuse Hint*, and forwards it to the LLC along with each cache request, as shown in Fig. 4, to guide specialized insertion and hit-promotion policies as described next.

C. Specialized Cache Policies

This component of GRASP implements specialized cache policies that protect the cache blocks associated with High-Reuse LLC accesses against thrashing. One naive way of doing so is to pin the High-Reuse cache blocks in the LLC. However, pinning would sacrifice any opportunity in exploiting temporal reuse that may be exposed by other cache blocks (e.g., Moderate-Reuse cache blocks).

To overcome this challenge, GRASP adopts a flexible approach by augmenting an existing cache replacement policy with a specialized insertion policy for LLC misses and a hit-promotion policy for LLC hits. GRASP's specialized policies provide preferential treatment to High-Reuse blocks while maintaining flexibility in exploiting temporal reuse in other cache blocks, as discussed next.

Insertion Policy: Accesses tagged as High-Reuse, comprising the set of the hottest vertices belonging to the High Reuse Region, are inserted in the cache at the MRU position to protect them from thrashing. Accesses tagged as Moderate-Reuse, likely exhibiting lower reuse when compared to the High-Reuse region, are inserted near the LRU position. Such insertion policy allows Moderate-Reuse cache blocks an opportunity to experience a hit without causing thrashing. Finally, accesses tagged as Low-Reuse, comprising the rest of the graph dataset, including the long tail of the Property Array containing cold vertices, are inserted at the LRU position, thus making them immediate candidates for replacement while still providing them with an opportunity to experience a hit and be promoted using the specialized policy described next.

Hit-Promotion Policy: Cache blocks associated with High-Reuse LLC accesses are immediately promoted to the MRU position on a hit to protect them from thrashing. LLC hits to blocks classified as Moderate-Reuse or Low-Reuse make for an interesting case. On the one hand, the likelihood of these blocks having further reuse is quite limited, which means they should not be promoted directly to the MRU position. On the other hand, by experiencing at least one hit, these blocks have demonstrated temporal locality, which cannot be completely ignored. GRASP takes a middle ground for such blocks by gradually promoting them towards MRU position on every hit.

Eviction Policy: GRASP's eviction policy does not utilize hint to differentiate blocks at replacement time; hence, it is unmodified from the baseline scheme. This is a key factor that keeps the cache management flexible for GRASP. By not prioritizing candidates for eviction, GRASP ensures that blocks classified as High-Reuse but not referenced for a long time can yield cache space to other blocks that do exhibit reuse. Because the unchanged eviction policy does not need to differentiate between blocks with High-Reuse and other hints, cache blocks do *not* need to explicitly store the Reuse Hint as additional LLC metadata.

Table II
POLICY COLUMNS SHOW HOW GRASP UPDATES PER-BLOCK 3-BIT RRPV COUNTER OF RRIP (BASE SCHEME) FOR A GIVEN REUSE HINT. HIGHER RRPV VALUE INDICATES HIGHER EVICTION PRIORITY.

Reuse Hint	Insertion Policy	Hit Policy
High-Reuse	RRPV = 0	RRPV = 0
Moderate-Reuse	RRPV = 6	if RRPV $>$ 0: RRPV - -
Low-Reuse	RRPV = 7	
Default	RRPV = 6 or 7	RRPV = 0

Table II shows the specialized cache policies for all Reuse Hints under GRASP. While the table, and our evaluation, assumes RRIP [52] as the base replacement scheme, we note that GRASP is not fundamentally dependent on RRIP and can be implemented over many other schemes including, but not limited to, LRU, Pseudo-LRU and DIP [60].

D. Benefits of GRASP over Prior Schemes

The state-of-the-art history-based schemes [5, 10, 13, 26, 28, 29, 49, 53] require intrusive modifications to the cache structure in form of embedded metadata in cache blocks and/or dedicated predictor tables. These schemes also require propagating a PC signature through the core pipeline all the way to the LLC, which so far has hindered their commercial adoption.

In comparison, GRASP is implemented within the same hardware structure required by the base lightweight scheme (e.g., RRIP). GRASP propagates only a 2-bit Reuse Hint to the LLC on each cache access to guide cache policy decisions. By relying on lightweight software support, GRASP reliably pinpoints hot vertices in hardware without requiring costly prediction tables and/or additional per-cache-block metadata.

When compared to pinning-based schemes, GRASP policies protect hot vertices from thrashing while remaining flexible to capture reuse of other blocks as needed. Combining robust cache policies with minimal hardware modifications makes GRASP feasible for commercial adoption while also providing higher LLC efficiency.

IV. METHODOLOGY

A. Graph Processing Framework

For the evaluation, we use *Ligra* [43], a widely used graph processing framework that supports both pull- and push-based computations, including switching from pull to push (and vice versa) at the start of every iteration. We combine the five diverse applications listed in Table III with the five high-skew graph datasets listed in Table V, resulting in 25 benchmarks. To test the robustness of GRASP to adversarial workloads, we use two additional datasets with low-/no-skew.

Table III
GRAPH-ANALYTIC APPLICATIONS.

Application	Brief description
Betweenness Centrality (BC)	finds the most central vertices in a graph by using a BFS kernel to count the number of shortest paths passing through each vertex from a given root vertex.
Single Source Shortest Path (SSSP)	computes shortest distance for vertices in a weighted graph from a given root vertex using the Bellman-Ford algorithm.
Pagerank (PR)	is an iterative algorithm that calculates ranks of vertices based on the number and quality of incoming edges to them [68].
PageRank-Delta (PRD)	is a faster variant of PageRank in which vertices are active in an iteration only if they have accumulated enough change in their PageRank score.
Radii Estimation (Radii)	estimates the radius of each vertex by performing multiple parallel BFS's from a small sample of vertices [56].

Table IV
EFFECT OF OUR OPTIMIZATION ON THE ORIGINAL LIGRA IMPLEMENTATION FOR DIFFERENT APPLICATIONS. PR APPLIES PULL-BASED COMPUTATIONS WHEREAS SSSP APPLIES PUSH-BASED COMPUTATIONS THROUGHOUT THE EXECUTION; THE REST OF THE APPLICATIONS SWITCH BETWEEN PULL OR PUSH BASED ON A NUMBER OF ACTIVE VERTICES IN A GIVEN ITERATION.

Application	Merging opportunity?	Speed-up
BC	No	-
SSSP	Yes	3-8%
PR	Yes	40-52%
PRD	Yes	14-49%
Radii	No	-

We obtained the source code for the graph applications from Ligra [43] and applied a simple data-structure optimization to improve locality in the baseline implementation as follows. As explained in Sec. II-C, graph applications exhibit irregular accesses for the Property Array, with applications potentially maintaining more than one such array. When multiple Property arrays are used, elements corresponding to a given vertex may need to be sourced from all of the arrays. We merge these arrays to induce spatial locality, which reduces number of misses, and in turn, improves performance on all datasets for PR, PRD and SSSP (see Table IV). We use the optimized implementation of these three applications as a stronger baseline for our evaluation. The optimized applications are available at https://github.com/faldupriyank/grasp. We do note that GRASP does not mandate merging arrays as GRASP design can accommodate multiple arrays. Nevertheless, merging does reduce the number of arrays needed to be tracked.

For PRD, two versions of the algorithm are provided with Ligra: push-based and pull-push. In the baseline implementation, the push-based version is faster. However, after merging the Property Arrays, the pull-push variant performs better, and is what we use for the evaluation.

B. Methodology for Software Evaluation

Server Configuration: Native experiments (Sec. V-C) are performed on a dual-socket server with two Broadwell based *Intel Xeon CPU E5-2630* [25], each with 10 cores clocked at 2.2GHz and a 25MB shared LLC. Hyper-threading is enabled, exposing 40 hardware execution contexts across both CPUs. The machine has 128GB of DRAM provided by eight DIMMs clocked at 2133MHz. All experiments were run using 40 threads, and we pinned the software threads to avoid performance variations due to OS scheduling. To further reduce sources of performance variation, we also disable the *turbo boost* DVFS features. Finally, we enabled *Transparent Huge Pages* to reduce TLB misses.

Table V
PROPERTIES OF THE GRAPH DATASETS. TOP FIVE DATASETS ARE USED IN THE MAIN EVALUATION WHEREAS THE BOTTOM TWO DATASETS ARE USED AS ADVERSARIAL DATASETS.

Dataset	Vertex Count	Edge Count	Avg. Degree
LiveJournal (*lj*) [38]	5M	68M	14
PLD (*pl*) [4]	43M	623M	15
Twitter (*tw*) [54]	62M	1,468M	24
Kron (*kr*) [32]	67M	1,323M	20
SD1-ARC (*sd*) [4]	95M	1,937M	20
Friendster (*fr*) [23]	64M	2,147M	33
Uniform (*uni*) [62]	50M	1,000M	20

Evaluation of software reordering techniques are carried out on the server mentioned above. We report the performance speed-up over the entire application runtime (including reordering cost) but exclude the graph loading time from the disk. For iterative applications, PR and PRD, we run them until convergence and consider the runtime over all iterations. For root-dependent traversal applications, SSSP and BC, we run them from eight different root vertices for each input dataset and consider the runtime over all eight traversals. Finally, we run six executions for each application-dataset pair and report the geometric mean of the five trials, excluding the timing of the first trial to allow the caches to warm up. We note that runtime is relatively stable across executions; for each reported datapoint, coefficient of variation is below 2%.

Reordering Techniques: We evaluate the following reordering techniques and use the source code for skew-aware techniques from https://github.com/faldupriyank/dbg and for Gorder from https://github.com/datourat/Gorder.

Sort reorders vertices in the memory space by sorting them in the descending order of their degree.

HubSort [19] segregates hot vertices in a contiguous region by assigning them a continuous range of vertex IDs in their descending order of degree. In doing so, Hub Sorting essentially *sorts all hot vertices*, while largely preserving structure for the cold vertices.

DBG [2], unlike Sort and HubSort, does not rely on sorting to segregate hot vertices. Instead, DBG coarsely partitions all vertices into a small number of groups based on their degree. Similar to Sort and HubSort, DBG is effective at improving spatial locality; however, unlike the other two techniques, DBG is able to largely preserve the existing graph structure.

Table VI
PARAMETERS OF THE SIMULATED SYSTEM FOR EVALUATION OF THE HARDWARE SCHEMES.

Core	OoO @ 2.66GHz, 4-wide front-end
L1-I/D Cache	4/8-ways 32KB, 4 cycles access latency stride-based prefetchers with 16 streams
L2 Cache	Unified, 8-ways 256KB, 6 cycles access latency
L3 Cache	16-ways 16MB NUCA (2MB slice per core), 10 cycles bank access latency
NOC	Ring network with 2 cycles per hop
Memory	50ns latency, 2 on-chip memory controllers

Gorder [30] is evaluated as a representative of complex techniques. As Gorder is only available in a single-thread implementation, while reporting the net runtime of Gorder for a given dataset, we optimistically divide the reordering time by 40 (maximum number of threads supported on the server) to provide a fair comparison with skew-aware techniques whose reordering implementation is fully parallelized.

C. Methodology for Hardware Evaluation

Simulation Infrastructure: We use the *Sniper* [37] simulator modeling 8 OoO cores. Table VI lists the parameters of the simulated system. The applications are evaluated in a multi-threaded mode with 8-threads.

We find that the graph applications spend significant fraction (86% on average in our evaluations) of time in push-based iterations for SSSP or pull-based iterations for all other evaluated applications. Thus, we simulate the *Region of Interest (ROI)* covering only push- or pull-based iterations (whichever one dominates) for the respective applications. Because simulating all iterations of a graph-analytic application in a detailed microarchitectural simulator is prohibitive, time-wise, we instead simulate one iteration that has the highest number of active vertices. To validate the soundness of our methodology, we also simulated one more randomly chosen iteration for each application-dataset pair with at least 20% of vertices active and observed trends similar to the ones reported in the paper.

Hardware Cache Management Schemes: We evaluate GRASP and compare it with the state-of-the-art thrash-resistant cache management schemes described below.

RRIP [52] is the state-of-the-art lightweight scheme that does not depend on history-based learning. RRIP is the most appropriate comparison point given that GRASP builds upon RRIP as the base scheme (Sec. III-C). We implement RRIP (specifically, *DRRIP*) based on the source code from the cache replacement championship (CRC1) [50] for RRIP, and use a 3-bit counter per cache block. We use RRIP as a high-performance baseline and report speed-up for all hardware schemes over the RRIP baseline (except for the studies in Sec V-D that use LRU baseline).

Signature-based Hit Predictor (SHiP) [49] is the state-of-the-art insertion policy which builds on RRIP [52]. Due to the shortcomings of PC-based reuse correlation for graph applications as explained in Sec. II-F, we evaluate a SHiP-MEM variant that correlates a block's reuse to the block's memory region. We evaluate 16KB memory regions as in the original proposal. The predictor table is provisioned with an *unlimited number of entries* to assess the maximum potential of the scheme. Each table entry consists of a 3-bit saturating counter that tracks the re-reference behavior of cache blocks of the memory region associated with that entry.

Hawkeye [26] is the state-of-the-art cache management scheme and winner of the cache replacement championship (CRC2) [11]. Hawkeye trains its predictor table by applying Belady's MIN algorithm on past LLC accesses to infer block's cache friendliness. We use the source code from the CRC2 for Hawkeye that improves upon the prefetcher-agnostic design of Hawkeye from [26]. We appropriately scale the number of sampling sets and predictor table entries for a 16MB cache.

Leeway [10] is a history-based cache management scheme that applies dead block predictions based on a metric called Live Distance, which conservatively captures the reuse interval of a cache block. We use the most recent version of the source code for Leeway from https://github.com/faldupriyank/leeway. We appropriately scale the number of sampling sets and predictor table entries for a 16MB cache.

XMem [7] is a pinning-based scheme, originally proposed for algorithms that benefit from *cache tiling*. A cache block, once pinned, cannot be evicted until explicitly unpinned by the software, usually done when the processing of a tile is complete. In the original proposal, XMem reserves 75% of LLC capacity to pin tile data whereas the remaining capacity is managed by the base replacement scheme for the rest of the data. In this work, we explore four configurations of XMem, labeled PIN-X, where X refers to the percentage (25%, 50%, 75% or 100%) of LLC capacity reserved for pinning. We adopt XMem design for graph analytics and identify the cache blocks from the high reuse region that benefit from pinning using the GRASP interface. Finally, XMem requires an additional 1-bit for every cache block to identify whether a cache block is pinned, along with an additional mechanism to track how much of the capacity is used by the pinned cache blocks at any given time.

GRASP is the proposed domain-specialized cache management scheme for graph analytics. We instrument the applications to communicate the address bounds of the Property Arrays to the simulated GRASP hardware. For each evaluated application, we needed to instrument at most two arrays. Finally, GRASP uses RRIP as the base cache scheme with a 3-bit saturating counter and does *not* add any further storage to per-block metadata.

V. EVALUATION

We first evaluate hardware cache management schemes on top of a skew-aware software reordering technique (Sec. V-A & V-B). Due to long simulation time, evaluating

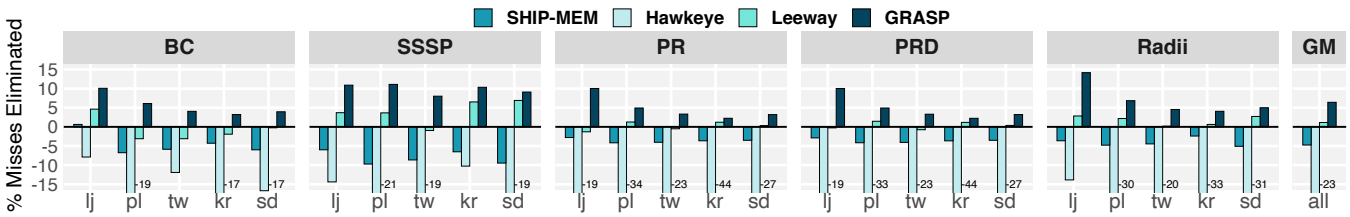


Figure 5. LLC miss reduction for GRASP and state-of-the-art history-based cache management schemes over the RRIP baseline.

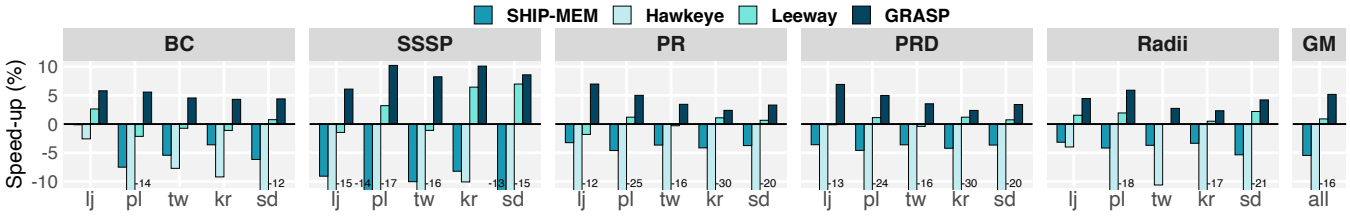


Figure 6. Speed-up for GRASP and state-of-the-art history-based cache management schemes over the RRIP baseline.

all hardware schemes on top of all four software reordering techniques would be prohibitive. Thus, without loss of generality, we evaluate hardware schemes on top of DBG, which consistently outperforms other reordering techniques (Sec. V-C). In Sec. V-C, we evaluate GRASP with other reordering techniques to show GRASP's generality.

A. History-based Predictive Schemes

In this section, we compare GRASP with the state-of-the-art hardware schemes, SHiP-MEM [49], Hawkeye [26] and Leeway [10]. Prior cache management proposals typically used LRU as the baseline, which is known to be inefficient against thrashing access patterns. Thus, we use RRIP, the state-of-the-art history-agnostic caching scheme as a strong baseline. Finally, we use DBG as the software baseline; thus, all speed-ups reported in this section are *over and above DBG*.

Miss reduction: Fig. 5 shows the miss reduction over the RRIP baseline. GRASP consistently reduces misses on all datapoints, eliminating 6.4% of LLC misses on average and up to 14.2% in the best case (on *lj* dataset for the Radii application). The domain-specialized design allows GRASP to accurately identify the high-reuse working set (i.e., hot vertices), which GRASP is able to retain in the cache through its specialized policies, effectively exploiting the temporal reuse.

Among prior techniques, Leeway is the only technique that reduces misses, albeit marginal, with an average miss reduction of 1.1% over the RRIP baseline. The other two techniques are not effective for graph applications, with SHiP-MEM and Hawkeye *increasing* misses across datapoints, with an average miss reduction of -4.8% and -22.7%, respectively, over the baseline. This is a new result as prior works show that Hawkeye and SHiP-MEM outperform RRIP on a wide range of applications [26, 49]. The result indicates that the learning mechanisms of the state-of-the-art domain-agnostic schemes are deficient in retaining the high-reuse working set for graph applications, which ends up hurting application performance as discussed next.

Application speed-up: Fig. 6 shows the speed-up for hardware schemes over the RRIP baseline. Overall, performance correlates well with the change in LLC misses; GRASP consistently provides a speed-up across datapoints with an average speed-up of 5.2% and up to 10.2% in the best case (on *pl* dataset for the SSSP application) over the baseline. When compared to the same baseline, SHiP-MEM and Hawkeye consistently cause slowdown with an average speed-up of -5.5% and -16.2%, respectively whereas Leeway yields a marginal speed-up of 0.9%. Finally, when compared to prior works directly, GRASP yields 4.2%, 5.2%, 11.2% and 25.5% average speed-up over Leeway, RRIP, SHiP-MEM and Hawkeye, respectively, while not causing slowdown on any datapoints.

We also evaluated prior schemes without applying any vertex reordering. Leeway, SHiP-MEM and Hawkeye yield an average speed-up of -0.8%, -5.7% and -14.8%, respectively, over RRIP on the datasets with no reordering. However, detailed study is omitted due to space constraints.

Dissecting Performance of SHiP-MEM: SHiP-MEM is a history-based scheme that predicts reuse of a cache block based on the fine-grained memory region, to which it belongs. Thus, SHiP-MEM relies on a homogeneous cache behavior for all blocks belonging to the same memory region. In theory, DBG (or another skew-aware technique) should allow SHiP-MEM to identify memory regions containing hottest of vertices (corresponding to High Reuse Region from Fig. 3(c)). In practice, irregular access patterns to these regions and thrashing by cache blocks from other regions impede learning. Thus, despite leveraging software and utilizing a sophisticated storage-intensive prediction mechanism in hardware, SHiP-

MEM underperforms domain-specialized GRASP.

Dissecting Performance of Hawkeye: Hawkeye is the state-of-the-art history-based scheme that uses PC-based reuse correlation to predict whether a cache block has a cache-friendly or cache-averse behavior based on past LLC accesses. Thus, Hawkeye fundamentally relies on homogeneous cache behavior for all blocks accessed by the same PC address. When Hawkeye is employed for graph analytics, Hawkeye struggles to learn the behavior of cache blocks in the Property Array as hot vertices exhibit cache-friendly behavior while cold vertices exhibit cache-averse behavior, yet all vertices are accessed by the same PC address. To make matters worse, if a block incurs a hit and Hawkeye predicts the PC making the access as cache-averse, the cache block is prioritized for eviction instead of promoting the block to MRU as is done in the baseline. Thus, Hawkeye performs even worse than the baseline for all combinations of graph applications and datasets. While not evaluated, other PC-based schemes (e.g., [49, 53]) that rely on a PC-based correlation would also struggle on graph applications for the same reason.

Dissecting Performance of Leeway: Leeway, like Hawkeye, also relies on a PC-based reuse correlation, and thus is not expected to provide significant speed-ups for graph-analytics. However, Leeway successfully avoids the slowdown on 10 of the 25 datapoints and significantly limits the slowdown on the rest of the datapoints (max slowdown of 2.1% vs 13.6% for SHiP-MEM and 30.2% for Hawkeye). The reasons why Leeway perfroms better than prior PC-based schemes can be attributed to (1) the conservative nature of the Live Distance metric, which Leeway uses to determine if a cache block is dead, and (2) adaptive reuse-aware policies that control the rate of predictions based on the observed access patterns. Because of these two factors, performance of Leeway remains close the the base replacement scheme in the presence of variability in cache blocks reuse behavior.

Dissecting Performance of GRASP: Performance of GRASP over its base scheme, RRIP, can be attributed to three features: software hints, insertion policy and hit-promotion policy. Fig. 7 shows the performance impact due to each of these features. RRIP inserts every new cache block at one of the two positions (as specified in the Default Reuse Hint of Table II); a cache block is inserted at the LRU position with a high probability or near the LRU position with a low probability. RRIP+Hints is identical to RRIP except for how a new cache block is assigned these positions. RRIP+Hints uses software hints (similar to GRASP) to guide the insertion. A cache block with High-Reuse hint is inserted near the LRU position and all other blocks are inserted at the LRU position. GRASP (Insertion-Only) refers to the scheme that applies insertion policy of GRASP as specified in Table II but the hit-promotion policy is unchanged from RRIP. Finally, GRASP (Hit-Promotion) refers to the scheme that applies hit-promotion policy of GRASP along with its insertion policy,

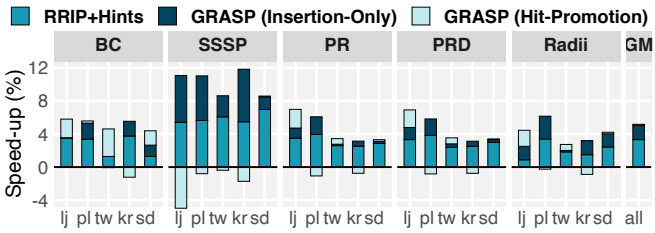


Figure 7. Impact of GRASP features on performance.

which is essentially the full GRASP design. Note that each successive scheme adds a new feature on top of the features incorporated by the previous ones. For example, GRASP (Insertion-Only) features a new insertion policy in addition to the software hints.

As the figure shows, RRIP+Hints yields an average speed-up of 3.3% over probabilistic RRIP, confirming the utility of software hints. GRASP (Insertion-Only) further increases performance by yielding an average speed-up of 5.0%. GRASP (Insertion-Only) provides additional protection to the High-Reuse cache blocks in comparison to RRIP+Hints by inserting High-Reuse cache blocks directly at the MRU position. Finally, GRASP (Hit-Promotion) yields an average speed-up of 5.2%. Difference between GRASP (Hit-Promotion) and GRASP (Insertion-Only) is marginal as the hit-promotion policy of GRASP has negative effect on slightly less than half the datapoints. The results are inline with the observations from prior work that showed that the value-addition of hit-promotion policies over insertion policies is low in presence of cache thrashing [22].

Summary: Hardware cache management is an established difficult problem, which is reflected in the small average speed-ups (usually 1%-5%) achieved by state-of-the-art cache management schemes over the prior best schemes [5, 26, 28, 29, 49, 52, 53]. Our work shows that graph applications present a particularly challenging workload for these schemes, in many cases leading to significant performance slowdowns. In this light, GRASP is quite successful in improving performance of graph applications by yielding an average speed-up of 5.2% (max 10.2%) while not causing slow-down on any datapoint. Moreover, unlike state-of-the-art schemes, GRASP achieves this without requiring storage-intensive metadata.

B. Pinning-based Schemes

In this section, we show the benefit of flexible GRASP policies over pinning-based rigid approaches. We first present the results on the high-skew datasets and then on the low-/no-skew datasets to test their resilience in adversarial scenarios.

High-skew datasets: Fig. 8 shows speed-ups for four XMem configuration (PIN-25, PIN-50, PIN-75 and PIN-100) and GRASP over the RRIP baseline on high-skew datasets. GRASP outperforms all XMem configurations on 24 of 25

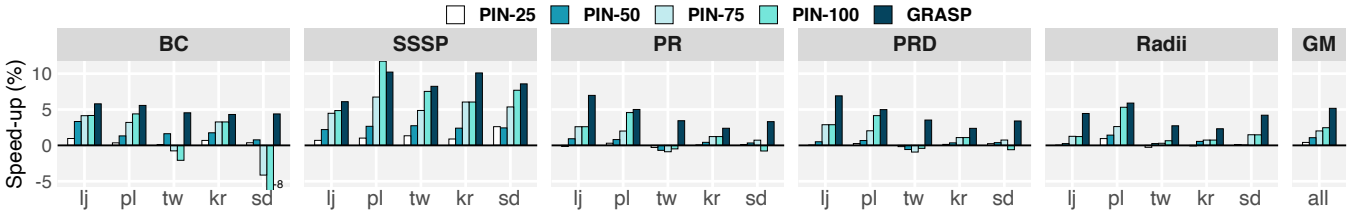


Figure 8. Speed-up for GRASP and pinning-based schemes over the RRIP baseline on high-skew datasets.

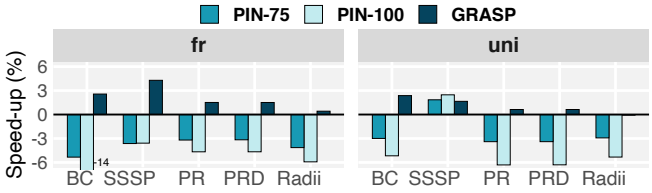


Figure 9. Speed-up over the RRIP baseline on *fr*, a low-skew dataset and *uni*, a no-skew dataset.

datapoints with an average speed-up of 5.2%. In comparison, PIN-25, PIN-50, PIN-75 and PIN-100 yield 0.4%, 1.1%, 2.0% and 2.5%, respectively.

PIN-100 outperforms the other three XMem configurations as for those configurations, a significant fraction of the capacity can still be occupied by cold vertices, which causes thrashing in the unreserved capacity. Nevertheless, PIN-100 causes slowdown on many datapoints (e.g., for BC, PR and PRD applications on *tw* and *sd* datasets). Moreover, PIN-100 cannot capitalize on reuse from Moderate Reuse Region as pinned vertices cannot be evicted even when they stopped exhibiting reuse. Thus, PIN-100 provides only a marginal speed-up on many datapoints (e.g., Radii application on *lj*, *tw* and *kr* datasets).

PIN-75 and PIN-100, two best-performing XMem configurations, while yield only marginal speed-ups, still outperform the state-of-the-art history-based schemes, SHiP-MEM, Leeway and Hawkeye (Figs. 6 & 8), which confirms that utilizing software knowledge for cache management is a promising direction over storage-intensive domain-agnostic design for the challenging access patterns of graph analytics.

Low-/No-skew datasets: Next, we evaluate the robustness of GRASP and pinning-based schemes (PIN-75 and PIN-100) for adversarial datasets with low-/no-skew. Naturally, these schemes are not expected to provide a significant speed-up in the absence of high skew; however, a robust scheme would reduce/avoid the slowdown. Fig. 9 shows the speed-up for a low-skew dataset *fr* and a no-skew dataset *uni* for these schemes over the RRIP baseline.

GRASP provides a net speed-up on 9 out of 10 datapoints even for low-/no-skew datasets. On the low-skew dataset *fr*, GRASP yields a speed-up between 0.4% and 4.3% whereas on the no-skew dataset *uni*, GRASP yields a speed-up between -0.1% and 2.4%. In contrast, PIN-75 and PIN-100 cause slowdown on almost all datapoints.

In the absence of high skew, cache blocks belonging to the High Reuse Region do not dominate the overall LLC accesses. Thus, pinning these blocks throughout the execution is counter-productive for PIN-75 and PIN-100. In contrast, GRASP adopts a flexible approach, wherein the high priority cache blocks from High Reuse Region can make way for other blocks that observe some reuse, as needed. Thus, GRASP successfully limits slowdown, and even provides reasonable speed-up on some datapoints, for such highly adversarial datasets.

Finally, combining results on all 7 datasets (5 datasets from Fig. 8 and 2 from Fig. 9), GRASP yields an average speed-up of 4.1%. In comparison, PIN-75 and PIN-100 provide a marginal speed-up of only 0.5% and 0.1%, respectively. PIN-75 and PIN-100 cause slowdown of up to 5.3% and 14.2% whereas max slowdown for GRASP is only 0.1%.

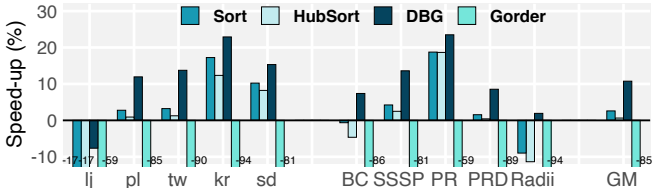
C. Reordering Techniques and GRASP

Thus far, we evaluated GRASP on graph applications processing datasets that are reordered using DBG. In this section, we compare performance of vertex reordering techniques, followed by an evaluation of GRASP on top of these techniques, demonstrating GRASP's generality.

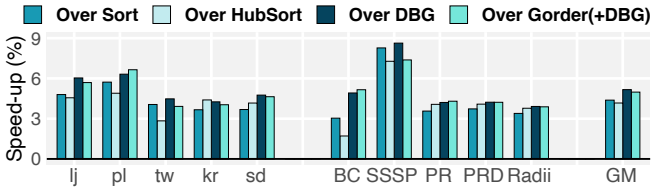
Effectiveness of Reordering Techniques: In this section, we first show that skew-aware techniques can improve performance of graph applications even as a standalone software optimization, thus justifying their existence. We evaluate three skew-aware techniques – Sort, HubSort [19] and DBG [2] – and a complex vertex reordering technique – Gorder [30]. We perform these studies on a real machine with 40 hardware threads as mentioned in Sec. IV-B.

Fig. 10(a) shows the speed-up for these existing software techniques after accounting for their reordering cost over the baseline with no reordering. Among skew-aware techniques, all techniques are effective on largest of the datasets (e.g., *kr* and *sd*) and long iterative applications (e.g., PR). As these techniques rely on a low cost approach for reordering, the reordering cost is amortized quickly when the application runtime is high, making these solutions practically attractive. Averaged across all application and dataset pairs, skew-aware techniques yield a net speed-up of 2.6% for Sort, 0.6% for HubSort and 10.8% for DBG.

Unsurprisingly, Gorder causes significant slowdown on all datapoints due to its large reordering cost, yielding an average



(a) Net speed-up for existing software reordering techniques after accounting for their reordering cost on a real machine.



(b) Application speed-up of GRASP over the RRIP baseline on top of different reordering techniques.

Figure 10. Reordering Techniques + GRASP: the left group shows speed-up for a dataset across all applications while the right group shows speed-up for an application across all datasets.

speed-up of -85.4%. Thus, Gorder is less practical when compared to simple yet effective skew-aware techniques, corroborating prior work [2].

Generality of GRASP: As software vertex reordering techniques offer different trade-offs in preserving graph structure and reducing reordering cost, it is important for GRASP to not be coupled to any one software technique. In this section, we evaluate GRASP with different reordering techniques, both skew-aware and complex ones. While skew-aware techniques are readily compatible with GRASP, Gorder requires a simple tweak as follows.

After applying Gorder on an original dataset, we apply DBG to further reorder vertices, which results in a vertex order that retains most of the Gorder ordering while also segregating hot vertices in a contiguous region, making Gorder compatible with GRASP.

Fig. 10(b) shows the speed-up for GRASP over RRIP on top of the same reordering technique as the baseline. As with DBG, GRASP consistently provides a speed-up across datasets and applications on top of other reordering techniques as well. On average, GRASP yields a speed-up of 4.4%, 4.2%, 5.2% and 5.0% on top of Sort, HubSort, DBG and Gorder, respectively. The result confirms that GRASP complements a broad class of existing software reordering techniques.

D. GRASP vs Optimal Replacement (OPT)

In this section, we compare GRASP with Belady's optimal replacement policy (OPT) [72]. As OPT requires the perfect knowledge of the future, we generate the traces of LLC accesses (up to 2 billion for each trace) for the applications processing graph datasets reordered using DBG on the simulation baseline configuration specified in Sec. IV-C. We apply OPT on each trace for five different LLC sizes – 1MB, 4MB, 8MB, 16MB and 32MB – to obtain the minimum number of misses for a given cache size and report the percentage of misses eliminated over *LRU* on the same LLC size.

Table VII
PERCENTAGE OF MISSES ELIMINATED OVER LRU FOR DIFFERENT LLC SIZE.

Scheme	1MB	4MB	8MB	16MB	32MB
RRIP	15.9%	16.4%	15.7%	15.2%	16.2%
GRASP	15.4%	17.0%	18.1%	19.7%	21.2%
OPT	27.5%	32.2%	33.3%	34.3%	34.5%

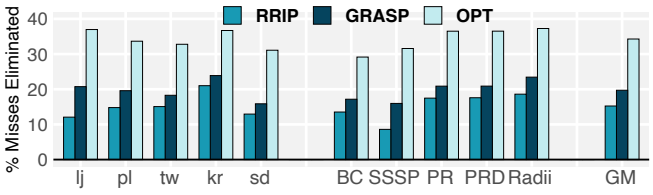


Figure 11. Percentage of misses eliminated over LRU.

Miss reduction on 16MB LLC: Fig. 11 shows the results for OPT along with RRIP and GRASP for 16MB LLC size. OPT eliminates 34.3% of total misses over LRU. In comparison, GRASP eliminates 19.7% of misses (vs 15.2% for RRIP). Overall, GRASP is 57.5% effective in eliminating misses when compared to OPT, an offline technique with perfect knowledge of the future. While GRASP is the most effective among the online techniques, the results also show that the remaining opportunity (difference between OPT and GRASP) is still significant, which warrants further research in this direction.

Sensitivity of GRASP to LLC size: Table VII shows the average percentage of misses eliminated by RRIP, GRASP and OPT for different LLC sizes over LRU. With the increase in LLC size, GRASP becomes more effective at eliminating misses over LRU (average miss reduction of 15.4% for 1MB vs 21.2% for 32MB). This is expected, as the larger LLC size allows GRASP to provide preferential treatment to more hot vertices. In general, yet larger LLC sizes are expected to benefit even more from GRASP until the LLC size becomes large enough to accommodate all hot vertices.

VI. RELATED WORK

Shared-memory graph frameworks: A significant amount of research has focused on designing high performance shared-memory frameworks for graph applications. Majority of these frameworks are vertex-centric [32, 35, 42, 43, 46, 55] and use CSR or its variants to encode a graph, making GRASP readily compatible with these frameworks. More generally, GRASP requires classification of only the Property Array(s), making it independent of the specific data structure used to represent the graph, which further increases compatibility across the spectrum of frameworks. Thus, we expect GRASP to reduce misses across frameworks, though absolute speed-ups will

likely vary.

Distributed-memory graph frameworks: these frameworks can also benefit from GRASP. For example, PGX [33] and PowerGraph [45] proposed duplicating high degree vertices in the graph partitions to reduce high communication overhead across computing nodes. These optimizations are largely orthogonal to GRASP cache management. As such, GRASP can be applied to distributed graph processing by caching high-degree vertices within each nodes's LLC to improve node-level cache behavior.

Streaming graph frameworks: In this work, we have assumed that graphs are static. In practice, graphs may evolve over time and a stream of graph updates (i.e., addition or removal of vertices or edges) are interleaved with graph-analytic queries (e.g., computing pagerank of vertices or computing shortest path from different root vertices). For such deployment settings, a CSR-based structure is infeasible. Instead, researchers have proposed various data structures for graph encoding that can accommodate fast graph updates and allow space-efficient versioning [1, 34, 44]. Meanwhile, each graph query is performed on a consistent view (i.e., static snapshot) of a graph. For example, Aspen [1], a recent graph-streaming framework, uses Ligra (a static graph-processing framework) in the back-end to run graph-analytic queries. Thus, the observations made in this paper regarding cache thrashing due to the irregular access patterns of the Property Array, as well as skew-aware reordering and GRASP being complementary in combating cache thrashing, are also relevant for dynamic graphs.

For static graphs, vertex reordering cost is amortized over multiple graph traversals for a single graph query (as shown in Fig. 10(a)). However, for dynamic graphs, reordering cost can be further amortized over multiple graph queries. Intuitively, addition or deletion of some vertices or edges in a large graph would not lead to a drastic change in the degree distribution, and thus unlikely to change which vertices are classified hot in a short time window. Therefore, skew-aware reordering can be applied at periodic intervals to improve cache behavior after a series of updates has been made to a graph, amortizing reordering cost over multiple graph queries.

Software-hints through profiling: Prior works have proposed ISA changes by embedding load/store instructions with reuse hints to improve cache replacement decisions [40, 61, 63, 64, 67]. These works perform program analysis via an additional compiler pass and/or runtime profiling to identify data that are unlikely to be referenced again. Compiler uses a custom memory instruction tagged with a low-reuse hint to access such data so that the cache hardware can prioritize the associated cache block for eviction. Such mechanisms, however, are largely effective for only those programs that are dominated by loops with regular access patterns. For example, Pacman [40] fails when a loop index variable and the reuse distance for the element accessed in a given loop iteration does not exhibit a linear correlation (e.g., indirect memory accesses, dominant type of memory accesses for graph analytics). In contrast, GRASP leverages vertex placement in memory after skew-aware reordering pass to correctly learn high-reuse vertices, that too without requiring custom instructions or a priori program analysis.

Hardware prefetchers: Modern processors typically employ prefetchers that target stride-based access patterns and thus are not amenable to graph analytics. Researchers have proposed custom prefetchers at L1-D that specifically target indirect memory access patterns of graph analytics [20, 36]. Nevertheless, prefetching can only *hide* memory access latency. Unlike cache replacement, prefetching cannot reduce memory bandwidth pressure or DRAM energy expenditure. Indeed, prior work observes that even a 100% accurate prefetcher for graph analytics is bottlenecked by memory bandwidth [36]. In contrast, GRASP reduces bandwidth pressure by reducing LLC misses, and thus is complementary to prefetching.

Traversal scheduling: Mukkara et al. proposed HATS [6], a hardware-accelerator implementing locality-aware scheduling to exploit cache locality for graphs exhibiting community structure. While effective, it requires intrusive hardware changes, including a specialized hardware unit with each core and an ISA change on the host core. In contrast, GRASP requires a minimal hardware interface and trivial changes to the cache policy while largely utilizing the existing cache hardware.

Graph slicing: Researchers have proposed slicing, a software optimization that slices the graph in LLC-size partitions and processes one partition at a time to reduce irregular memory accesses. Specifically, Graphicionado [24] uses slicing to fit a working set into a large 64MB on-chip scratchpad while Zhang et al. use *CSR Segmenting* to break the vertices into segments that fit in LLC [19].

While generally effective, slicing has two important limitations. First, it requires invasive framework changes to form the slices (which may include replicating vertices to avoid out-of-slice accesses) and manage them at runtime. Secondly, for a given cache size, the number of slices increases with the size of the graph, resulting in greater processing overheads in creating and maintaining partitions for larger graphs.

In comparison, GRASP is a hardware scheme that intelligently leverages lightweight software support. GRASP requires minimal changes in a graph framework and does not require any changes in the graph algorithms. Having said that, GRASP is complementary to slicing; by preserving the critical working set in the cache (i.e., hot vertices), GRASP could be used to improve the performance of Graphicionado to reduce the number of slices by making intra-slice cache misses infrequent.

VII. Conclusion

This work explores how hardware cache management should be designed to tackle cache thrashing at LLC for graph-analytic applications. We show that state-of-the-art cache management schemes are deficient in presence of cache thrashing stemming from irregular access patterns of graph applications processing large graphs. In response, we introduce GRASP – specialized cache management for LLC for graph analytics on power-law graphs. GRASP's specialized cache policies exploit the high reuse inherent in hot vertices while retaining the flexibility to capture reuse in other cache blocks. GRASP leverages existing software reordering optimizations to enable a lightweight interface that allows hardware to pinpoint hot vertices amidst irregular access patterns. In doing so, GRASP avoids the need for a storage-intensive prediction mechanism or additional metadata storage in the LLC. GRASP requires minimal hardware support, making it attractive for integration into a commodity server processor to enable acceleration for the domain of graph analytics. Finally, GRASP delivers consistent performance gains on high-skew datasets, while preventing slowdowns on low-skew datasets.

Acknowledgment

This work was supported in part by a research grant from Oracle Labs. We thank Amna Shahab, Antonios Katsarakis, Arpit Joshi, Artemiy Margaritov, Avadh Patel, Dmitrii Ustiugov, Rakesh Kumar, Vasilis Gavrielatos, Vijay Nagarajan, and the anonymous reviewers for their valuable feedback on an earlier draft of this work.

References

[1] L. Dhulipala, G. E. Blelloch, and J. Shun. "Low-latency Graph Streaming Using Compressed Purely-functional Trees". In: *International Conference on Programming Language Design and Implementation (PLDI)*. 2019.

[2] P. Faldu, J. Diamond, and B. Grot. "A Closer Look at Lightweight Graph Reordering". In: *International Symposium on Workload Characterization (IISWC)*. 2019.

[3] V. Balaji and B. Lucia. "When is Graph Reordering an Optimization? Studying the Effect of Lightweight Graph Reordering Across Applications and Input Graphs". In: *International Symposium on Workload Characterization (IISWC)*. 2018.

[4] *Hyperlink Graphs*. http://webdatacommons.org/hyperlinkgraph. Web Data Commons, 2018.

[5] A. Jain and C. Lin. "Rethinking Belady's Algorithm to Accommodate Prefetching". In: *International Symposium on Computer Architecture (ISCA)*. 2018.

[6] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez. "Exploiting Locality in Graph Analytics through Hardware-Accelerated Traversal Scheduling". In: *International Symposium on Microarchitecture (MICRO)*. 2018.

[7] N. Vijaykumar, A. Jain, D. Majumdar, K. Hsieh, G. Pekhimenko, E. Ebrahimi, N. Hajinazar, P. B. Gibbons, and O. Mutlu. "A Case for Richer Cross-Layer Abstractions: Bridging the Semantic Gap with Expressive Memory". In: *International Symposium on Computer Architecture (ISCA)*. 2018.

[8] J. Díaz, P. Ibáñez, T. Monreal, V. Viñals, and J. Llabería. "ReD: A Policy Based on Reuse Detection for a Demanding Block Selection in Last-Level Caches". In: *International Workshop on Cache Replacement Championship (CRC), co-located with ISCA*. 2017.

[9] P. Faldu and B. Grot. "Reuse-Aware Management for Last-Level Caches". In: *International Workshop on Cache Replacement Championship (CRC), co-located with ISCA*. 2017.

[10] P. Faldu and B. Grot. "Leeway: Addressing Variability in Dead-Block Prediction for Last-Level Caches". In: *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2017.

[11] P. V. Gratz, J. Kim, A. R. Alameldeen, C. Wilkerson, S. Pugsley, A. Jaleel, B. Falsafi, M. Sutherland, and J. Picorel. *The 2nd Cache Replacement Championship (CRC), co-located with ISCA*. http://crc2.ece.tamu.edu. 2017.

[12] A. Jain and C. Lin. "Hawkeye Cache Replacement: Leveraging Belady's Algorithm for Improved Cache Replacement". In: *International Workshop on Cache Replacement Championship (CRC), co-located with ISCA*. 2017.

[13] D. A. Jiménez and E. Teran. "Multiperspective Reuse Prediction". In: *International Symposium on Microarchitecture (MICRO)*. 2017.

[14] D. A. Jiménez. "Multiperspective Reuse Prediction". In: *International Workshop on Cache Replacement Championship (CRC), co-located with ISCA*. 2017.

[15] K. Lakhotia, S. Singapura, R. Kannan, and V. Prasanna. "ReCALL: Reordered Cache Aware Locality Based Graph Processing". In: *International Conference on High Performance Computing (HiPC)*. 2017.

[16] A. Vakil-Ghahani, S. Mahdizadeh-Shahri, M. Lotfi-Namin, M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad. "Cache Replacement Policy Based on Expected Hit Count". In: *International Workshop on Cache Replacement Championship (CRC), co-located with ISCA*. 2017.

[17] J. Wang, L. Zhang, R. Panda, and L. John. "Less is More: Leveraging Belady's Algorithm with Demand-based Learning". In: *International Workshop on Cache Replacement Championship (CRC), co-located with ISCA*. 2017.

[18] V. Young, C. Chou, A. Jaleel, and M. K. Qureshi. "SHiP++: Enhancing Signature-Based Hit Predictor for Improved Cache Performance". In: *International Workshop on Cache Replacement Championship (CRC), co-located with ISCA*. 2017.

[19] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia. "Making caches work for graph analytics". In: *International Conference on Big Data (Big Data)*. 2017.

[20] S. Ainsworth and T. M. Jones. "Graph Prefetching Using Data Structure Knowledge". In: *International Conference on Supercomputing (ICS)*. 2016.

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

[22] P. Faldu and B. Grot. "LLC Dead Block Prediction Considered Not Useful". In: *International Workshop on Duplicating, Deconstructing and Debunking (WDDD), co-located with ISCA*. 2016.

[23] *Friendster network dataset – KONECT*. http://konect.uni-koblenz.de/networks/friendster. The Koblenz Network Collection, 2016.

[24] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi. "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics". In: *International Symposium on Microarchitecture (MICRO)*. 2016.

[25] *Intel Xeon Processor E5-2630 v4*. https://ark.intel.com/products/92981/Intel-Xeon-Processor-E5-2630-v4-25M-Cache-2_20-GHz. Intel Corporation, 2016.

[26] A. Jain and C. Lin. "Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement". In: *International Symposium on Computer Architecture (ISCA)*. 2016.

[27] A. Mukkara, N. Beckmann, and D. Sanchez. "Whirlpool: Improving Dynamic Cache Management with Static Data Classification". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2016.

[28] E. Teran, Y. Tian, Z. Wang, and D. A. Jiménez. "Minimal disturbance placement and promotion". In: *International Symposium on High Performance Computer Architecture*. 2016.

[29] E. Teran, Z. Wang, and D. A. Jiménez. "Perceptron Learning for Reuse Prediction". In: *International Symposium on Microarchitecture (MICRO)*. 2016.

[30] H. Wei, J. X. Yu, C. Lu, and X. Lin. “Speedup Graph Processing by Graph Ordering”. In: *International Conference on Management of Data (SIGMOD)*. 2016.

[31] S. Beamer, K. Asanovic, and D. Patterson. “Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server”. In: *International Symposium on Workload Characterization (IISWC)*. 2015.

[32] S. Beamer, K. Asanovic, and D. A. Patterson. “The GAP Benchmark Suite”. In: *CoRR* (2015). http://arxiv.org/abs/1508.03619.

[33] S. Hong, S. Depner, T. Manhardt, J. V. D. Lugt, M. Verstraaten, and H. Chafi. “PGX.D: a fast distributed graph processing engine”. In: *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 2015.

[34] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer. “LLAMA: Efficient graph analytics using Large Multiversioned Arrays”. In: *International Conference on Data Engineering (ICDE)*. 2015.

[35] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey. “GraphMat: High Performance Graph Analytics Made Productive”. In: *The VLDB Endowment* (2015).

[36] X. Yu, C. J. Hughes, N. Satish, and S. Devadas. “IMP: Indirect Memory Prefetcher”. In: *International Symposium on Microarchitecture (MICRO)*. 2015.

[37] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout. “An Evaluation of High-Level Mechanistic Core Models”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* (2014).

[38] J. Leskovec and A. Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data. 2014.

[39] X. Tong and A. Moshovos. “BarTLB: Barren page resistant TLB for managed runtime languages”. In: *International Conference on Computer Design (ICCD)*. 2014.

[40] J. Brock, X. Gu, B. Bao, and C. Ding. “Pacman: Program-assisted Cache Management”. In: *International Symposium on Memory Management (ISMM)*. 2013.

[41] D. A. Jiménez. “Insertion and promotion for tree-based PseudoLRU last-level caches”. In: *International Symposium on Microarchitecture (MICRO)*. 2013.

[42] D. Nguyen, A. Lenharth, and K. Pingali. “A Lightweight Infrastructure for Graph Analytics”. In: *International Symposium on Operating Systems Principles (SOSP)*. 2013.

[43] J. Shun and G. E. Blelloch. “Ligra: A Lightweight Graph Processing Framework for Shared Memory”. In: *International Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 2013.

[44] D. Ediger, R. McColl, J. Riedy, and D. A. Bader. “Stinger: High performance data structure for streaming graphs”. In: *International Conference on High Performance Extreme Computing (HPEC)*. 2012.

[45] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. “PowerGraph: Distributed Graph-parallel Computation on Natural Graphs”. In: *International Conference on Operating Systems Design and Implementation (OSDI)*. 2012.

[46] A. Kyrola, G. Blelloch, and C. Guestrin. “GraphChi: Large-scale Graph Computation on Just a PC”. In: *International Conference on Operating Systems Design and Implementation (OSDI)*. 2012.

[47] I. Stanton and G. Kliot. “Streaming Graph Partitioning for Large Distributed Graphs”. In: *International Conference on Knowledge Discovery and Data Mining (KDD)*. 2012.

[48] U. Kang and C. Faloutsos. “Beyond ‘Caveman Communities’: Hubs and Spokes for Graph Compression and Mining”. In: *International Conference on Data Mining (ICDM)*. 2011.

[49] 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: *International Symposium on Microarchitecture (MICRO)*. 2011.

[50] A. R. Alameldeen, A. Jaleel, M. K. Qureshi, and J. Emer. *JILP Workshop on Cache Replacement Championship (CRC)*. http://www.jilp.org/jwac-1. 2010.

[51] H. Gao and C. Wilkerson. “A dueling segmented LRU replacement algorithm with adaptive bypassing”. In: *In JILP Workshop on Computer Architecture Competitions: Cache Replacement Championship (CRC)*. 2010.

[52] A. Jaleel, K. B. Theobald, S. C. Steely Jr., and J. Emer. “High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP)”. In: *International Symposium on Computer Architecture (ISCA)*. 2010.

[53] S. M. Khan, Y. Tian, and D. A. Jiménez. “Sampling Dead Block Prediction for Last-Level Caches”. In: *International Symposium on Microarchitecture (MICRO)*. 2010.

[54] H. Kwak, C. Lee, H. Park, and S. Moon. “What is Twitter, a social network or a news media?” In: *International Conference on World Wide Web (WWW)*. 2010.

[55] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. “GraphLab: A New Framework For Parallel Machine Learning”. In: *The Conference on Uncertainty in Artificial Intelligence (UAI)*. 2010.

[56] C. Magnien, M. Latapy, and M. Habib. “Fast Computation of Empirically Tight Bounds for the Diameter of Massive Graphs”. In: *Journal of Experimental Algorithmics* (2009).

[57] Y. Xie and G. H. Loh. “PIPP: Promotion/Insertion Pseudo-partitioning of Multi-core Shared Caches”. In: *International Symposium on Computer Architecture (ISCA)*. 2009.

[58] A. Jaleel, W. Hasenplaugh, M. K. Qureshi, J. Sebot, S. Steely Jr., and J. Emer. “Adaptive Insertion Policies for Managing Shared Caches”. In: *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2008.

[59] M. Kharbutli and Y. Solihin. “Counter-Based Cache Replacement and Bypassing Algorithms”. In: *IEEE Transactions on Computers* (2008).

[60] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. “Adaptive Insertion Policies for High Performance Caching”. In: *International Symposium on Computer Architecture (ISCA)*. 2007.

[61] K. Beyls and E. H. D’Hollander. “Generating Cache Hints for Improved Program Efficiency”. In: *Journal of Systems Architecture* (2005).

[62] D. Chakrabarti, Y. Zhan, and C. Faloutsos. “R-MAT: A recursive model for graph mining”. In: *SIAM International Conference on Data Mining (SDM)*. 2004.

[63] Z. Wang, K. S. McKinley, A. L. Rosenberg, and C. C. Weems. “Using the compiler to improve cache replacement decisions”. In: *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2002.

[64] P. Jain, S. Devadas, D. Engels, and L. Rudolph. “Software-assisted cache replacement mechanisms for embedded systems”. In: *International Conference on Computer Aided Design (ICCD)*. 2001.

[65] A.-L. Barabási and R. Albert. “Emergence of Scaling in Random Networks”. In: *Science* (1999).

[66] M. Faloutsos, P. Faloutsos, and C. Faloutsos. “On Power-law Relationships of the Internet Topology”. In: *The Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*. 1999.

[67] A. R. Lebeck, D. R. Raymond, C.-L. Yang, and M. Thottethodi. “Annotated Memory References: A Mechanism for Informed Cache Management”. In: *International Euro-Par Conference on Parallel Processing (Euro-Par)*. 1999.

[68] L. Page, S. Brin, R. Motwani, and T. Winograd. *The PageRank Citation Ranking: Bringing Order to the Web*. Technical Report 1999-66. http://ilpubs.stanford.edu:8090/422. Stanford InfoLab, 1999.

[69] G. Karypis and V. Kumar. “Multilevel k-way Partitioning Scheme for Irregular Graphs”. In: *Journal of Parallel and Distributed Computing* (1998).

[70] J. Banerjee, W. Kim, S. Kim, and J. F. Garza. “Clustering a DAG for CAD databases”. In: *IEEE Transactions on Software Engineering* (1988).

[71] E. Cuthill and J. McKee. “Reducing the Bandwidth of Sparse Symmetric Matrices”. In: *The National Conference*. 1969.

[72] L. A. Belady. “A Study of Replacement Algorithms for a Virtual-storage Computer”. In: *IBM Systems Journal* (1966).