Improving Locality of Irregular Updates with Hardware Assisted Propagation Blocking

Vignesh Balaji*
NVIDIA
vbalaji@nvidia.com

Brandon Lucia
Carnegie Mellon University
blucia@andrew.cmu.edu

***Abstract*—Many application domains perform irregular memory updates. Irregular accesses lead to inefficient use of conventional cache hierarchies. To make better use of the cache, we focus on Propagation Blocking (PB), a software-based cache locality optimization initially designed for graph processing applications. We make two contributions in this work. First, we show that PB generalizes beyond graph processing applications to any application with unordered parallelism and irregular memory updates. Second, we identify the inefficiencies of a PB execution on conventional multicore processors and propose architecture support to further improve the performance gains from PB. Our proposed architecture, COBRA, optimizes the PB execution of a range of applications with irregular memory updates, offering speedups of up to 3.78x compared to PB (1.74x on average).**

***Keywords*-caches; locality optimization; graph analytics; sparse linear algebra; irregular workloads**

I. Introduction

Graph processing applications are an important category of computing workloads with valuable applications in social network analysis, COVID19 therapeutics discovery, path-planning, and graph learning [1], [21], [60]. In the past, large graphs were primarily processed on distributed, datacenter-scale systems [23], [38], [48]. Recently, increasing main memory capacities and core counts enable processing graphs with billions of edges using just a single machine [19], [22], [34], [56]. Prior work argues that when a graph fits in a single machine, distributed graph processing is less efficient [41].

High performance, single machine graph processing is challenging. The characteristic irregular memory access pattern of graph processing workloads leads to poor cache locality and an execution time dominated by DRAM latency [8], [11], [53], [59], [63]. Therefore, cache locality optimizations have the potential to provide significant performance gains for graph processing applications. Propagation Blocking [13] is one such software-based locality optimization that is particularly lucrative because of its low (runtime and programmability) overheads. The key insight of Propagation Blocking is that the order of updates to graph application data does not affect correctness and, therefore, the irregular updates can be reordered to improve cache locality. Propagation Blocking has been shown to be effective at improving the locality of a wide range of graph processing workloads [14], [24], [32].

The effectiveness of Propagation Blocking (PB) has prompted prior work [31], [43], [50], [52] to develop architecture support for improving its efficiency, further increasing PB's gains. However, all these PB optimizations rely on the application's irregular updates being commutative.While commutative updates are typical in graph processing workloads, many applications perform irregular updates which do not satisfy the commutativity property, limiting the scope of existing hardware optimizations for PB. Therefore, the focus of our work is to develop architecture support for PB that does not require an application's irregular updates to be commutative. Not requiring update commutativity allows us to extend the benefits of the PB optimization to a broader range of applications beyond graph processing workloads.

To develop architecture support for the general case of PB, we focused on identifying the inefficiencies of a PB execution on conventional multicore processors. PB reorders an application's irregular updates to improve cache locality. To reorder updates, PB breaks an application execution into two phases – *Binning* and *Accumulate*. In the *Binning* phase, PB streams through data structures with regular access patterns (e.g. the graph) but does not directly apply the irregular updates which span a large range of memory locations (the primary contributor to poor cache locality). Instead, PB buffers the irregular updates (key, value pairs) in *bins* where each bin only stores updates corresponding to a smaller subset of memory locations. Next, in the *Accumulate* phase, PB processes bins, applying all of a bin's updates to the irregularly accessed data before moving to the next bin. The *Accumulate* phase achieves good cache locality because a bin's updates apply to a small range of memory locations that fit in cache. However, *Binning* is a tax paid by PB for improved locality during *Accumulate*. Specifically, the *Binning* phase imposes two overheads in all PB executions. First, to buffer irregular updates into bins, the *Binning* phase requires executing many additional instructions (including branches) that impose a high control overhead. Second, the *Binning* phase forces all PB executions to make a sub-optimal choice on the number of bins. The *Accumulate*

*Work done when author was at Carnegie Mellon University

phase achieves the best cache locality when there are a large number of bins because each bin stores updates to a very small range of memory locations that can fit in the on-chip cache closest to the processor. However, *Binning* performance is prohibitively expensive with a large number of bins, leading all PB executions to compromise with fewer bins than the ideal case. We show that architecture support can eliminate both the overheads of *Binning*, allowing PB executions to avoid the software overheads associated with binning updates and selecting the ideal number of bins for maximum cache locality.

We propose COBRA: a set of modifications to the ISA and the memory hierarchy of a multicore processor to optimize PB's *Binning* phase. Instead of executing additional instructions to bin updates as in software-based PB, COBRA introduces simple fixed-function logic in each level's cache controllers to efficiently bin updates. COBRA introduces a single (CISC-like) instruction to offload the binning computation to fixed function units in each cache level. COBRA's architecture extensions eliminate the instruction overheads of *Binning* and improve *Binning* performance when there are a large number of bins (the ideal operating point for *Accumulate*). Furthermore, the COBRA architecture does not assume commutativity of irregular updates and, hence, serves as a more general PB optimization that applies to irregular-update applications beyond just graph analytics. We evaluate COBRA across a range of graph processing, pre-processing, integer sorting, and sparse linear algebra kernels showing a mean speedup of 1.74x over an optimized software PB implementation and 3.16x over the baseline. In summary, we make the following contributions in this paper:

- We show that commutativity of irregular updates is not a necessary condition for PB, allowing PB to apply more broadly beyond graph analytics (Section III-B).
- We characterize the inefficiencies of software PB on commodity multicores (Section III-C).
- We present the insight behind the COBRA architecture (Section IV) and describe its implementation (Section V).
- We show COBRA's effectiveness across a broad range of kernels and characterize why COBRA works (Section VII).
- We compare COBRA (and PB) against state-of-the-art hardware PB optimization (PHI [43]) and software cache locality optimization (CSR-Segmenting [63]).

II. Background: Irregular Updates

The goal of this work is to improve the locality and performance of applications that perform irregular memory updates. To provide background, we identify the common sources of irregular updates in applications and characterize the poor cache locality resulting from irregular updates.

Sources of Irregularity: A common source of irregular memory accesses is the input data representation. Graph analytics and sparse linear algebra applications often analyze extremely sparse inputs (a typical adjacency matrix is $>99\%$ sparse [18]). Therefore, compressed formats are essential for storing the input graph/matrix in memory efficiently. The popular *Compressed Sparse Row* (CSR) format offers an additional benefit of quickly identifying a vertex's neighbors. As shown in Figure 1, CSR uses two arrays to represent outgoing edges (sorted by edge source IDs). The Neighbors Array (NA) contiguously stores each vertex's neighbors and the Offsets Array (OA) stores the starting offset of each vertex's neighborhood in the NA. While the CSR (and its transpose CSC) allow quick access to vertex neighbors, they can lead to irregular memory accesses. The contents of the CSR (i.e. NA in Figure 1) are arbitrarily ordered, defined by the sparsity pattern and the vertex ordering. Therefore, applications traversing the CSR and accessing a second data structure based on the indices in the NA perform irregular memory accesses. Besides data representations, other input properties such as the distribution of keys for counting sort [16] can also lead to irregular memory accesses.

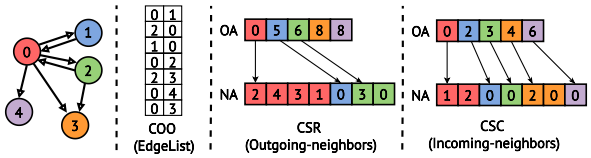


Figure 1: **Popular Compressed Representations**

Poor Locality of Irregular Updates: Applications with irregular memory updates suffer from poor cache locality. Conventional cache hierarchies are designed to optimize for spatial and temporal locality, both of which are absent in irregular access patterns. Using hardware performance counters, we characterized the Last Level Cache (LLC) miss rates of applications performing irregular memory updates (methodology in Section VI). Figure 2 shows that a broad range of applications spanning graph analytics, graph pre-processing, integer sorting, and sparse linear algebra all suffer from poor LLC locality because of irregular updates. Additionally, prior work [7], [9], [11], [63] has showed that the high LLC miss rates cause graph analytics kernels to spend up to 80% of their execution time stalled on DRAM. Therefore, cache locality optimizations are critical for improving performance of these irregular workloads.

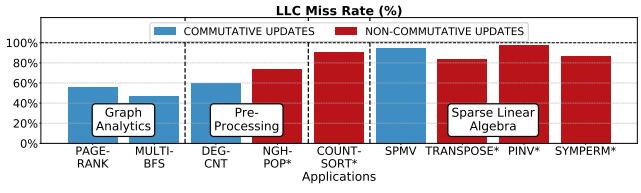


Figure 2: **Locality of irregular updates:** *Applications with irregular updates experience a high LLC miss rate.*

III. Propagation Blocking

Propagation Blocking (PB) has been shown to be an effective optimization for graph processing workloads [13], [32]. We show that PB applies more broadly to any application exhibiting unordered parallelism and, therefore, PB generalizes to applications beyond graph analytics. Software PB incurs fundamental overheads that prevent achieving optimal performance. This section also identifies the opportunity to improve PB's benefits with architecture support.

A. Propagation Blocking High level Overview

Propagation Blocking (PB) was originally developed to optimize `Pagerank` [13]. Figure 3 shows an unoptimized `Pagerank` execution operating on a CSR input graph. The `Pagerank` execution streams in edges and auxiliary data (`auxData`), and updates the `vtxData` array at dst_k using the value $auxData[src_k]$. The stream of indices (dst_k) in the CSR are unordered and span the full range of the graph's vertex IDs (Figure 1). Therefore, an unoptimized `Pagerank` execution suffers from poor locality because the irregular update's working set exceeds the cache capacity.

PB improves cache locality of `Pagerank` by breaking the execution into two phases: *Binning* and *Accumulate*. During *Binning*, the core streams in edges and auxiliary data but the core does not directly update `vtxData` elements. Instead, the core writes the pair of index location and update value $(dst_k, auxData[src_k])$ to one of several *bins* created by PB. A bin is a data structure that *sequentially* stores each update belonging to a particular range of data elements. Each bin stores updates for a disjoint range of elements and the union of all the bin-ranges equals the total number of elements (i.e. number of vertices in the graph). Once all updates have been written to bins, PB starts the *Accumulate* phase. During *Accumulate*, the core sequentially accesses each tuple in a bin before moving to the next bin (Figure 3). Since each bin stores updates for a small index range, only a part of the `vtxData` array is accessed at any point in time which reduces the range of irregular writes, allowing the bin's updates to fit in cache. In this work, we focus on parallel PB which simply creates per-thread duplicates of all bin structures in Figure 3, eliminating the need for synchronization during *Binning*.

B. Applicability of Propagation Blocking

The effectiveness of PB across many graph workloads [13], [14], [32] has prompted hardware optimizations for PB [31], [43], [50]. However, these hardware PB optimizations rely on the application performing *commutative updates*. Commutative updates allow coalescing multiple updates destined to the same index, reducing PB's main memory traffic without affecting application correctness. However, we find that commutativity is *not necessary* to benefit from PB.

We observe that PB applies to non-commutative kernels as well. Algorithm 1 shows a part of the kernel for building

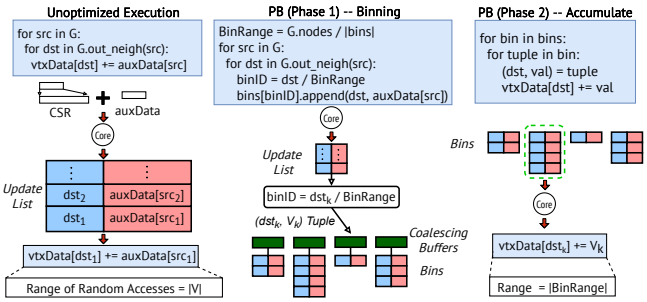


Figure 3: **High level overview of PB:** *PB reduces the range of irregular updates. Note that the Update List exists only at a logical level and is never physically materialized.*

Algorithm 1 Kernel to populate neighbors (Edgelist-to-CSR)

```
1: offsets ← PrefixSum(degrees)                ▷ OA in Figure 1
2: par_for e in EL do
3:     neighs[offsets[e.src]] ← e.dst           ▷ NA
4:     AtomicAdd(offsets[e.src], 1)
```

a CSR from an Edgelist [1]. The kernel (henceforth referred to as `Neighbor-Populate`) uses a copy of the Offsets Array (OA) to populate the contents of the Neighbors Array (NA) in Figure 1. The updates to the `offsets` array in `Neighbor-Populate` (Algorithm 1; line 4) are not commutative because the order of updates to the `offsets` array determines the contents of the NA. Coalescing updates to the `offsets` array would break correctness by skipping elements of the `neighs` array (NA). Algorithm 2 shows how PB optimizes the kernel. The *Binning* phase streams in edges and assigns each edge to a bin. After *Binning* reorganizes edges into bins, *Accumulate* processes the edges in each bin, updating `offsets` and `neighs` with high cache locality. The PB optimization is applicable to the non-commutative `Neighbor-Populate` kernel because

[1]Building a graph data structure from an Edgelist is one of three kernels used by Graph500 [45] to benchmark graph processing supercomputers.

Algorithm 2 PB version of Algorithm 1

```
1: offsets ← PrefixSum(degrees)
2: par_for e in EL do                          ▷ Binning Phase
3:     tid ← GetThreadID()
4:     binID ← (e.src/BinRange)
5:     bins[tid][binID] ← (e.src, e.dst)
6: par_for binID in NumBins do                 ▷ Accumulate Phase
7:     for tid in NumThreads do
8:         for tuple in bins[tid][binID] do
9:             offsetVal ← offsets[tuple.src]
10:            neighs[offsetVal] ← tuple.dst
11:            Add(offsets[tuple.src], 1)
```

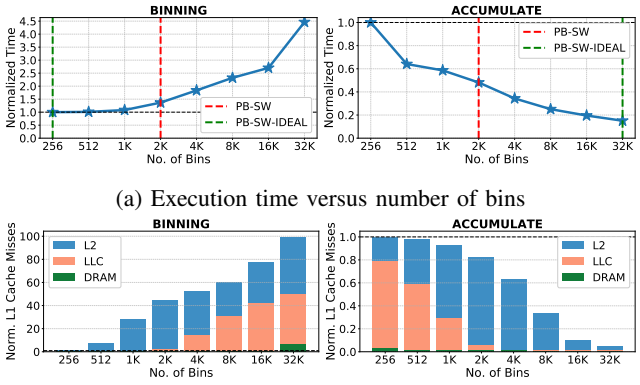
a vertex's neighbors can be listed in any order; the non-commutative updates permit *unordered parallelism* [27], [30]. This example shows that the applicability of PB goes beyond commutative updates: the PB optimization applies to applications with **irregular updates and unordered parallelism**. All the applications listed in Figure 2 can benefit from PB however not all applications perform commutative updates and, therefore, not all applications benefit from existing hardware PB optimizations. The focus of our work is to develop a general hardware PB optimization that does not rely on update commutativity, enabling acceleration of a broader range of applications.

C. Limitations of Propagation Blocking

All PB executions on conventional processors are primarily limited in two ways: (i) PB must compromise with a sub-optimal number of bins and (ii) binning updates requires executing many extra instructions, eroding PB's gains.

Compromising on the number of bins: The locality of the *Accumulate* phase (Figure 3) is highly sensitive to the number of bins because the range of updates belonging to a bin (i.e. range of irregular updates) is inversely related to the number of bins ($BinRange = \frac{|UniqueIndices|}{|Bins|}$). The *Binning* phase is also sensitive to the number of bins. To amortize the cost of writing to bins, the *Binning* phase uses cacheline-sized *coalescing buffers* for each bin that accumulate updates to bins and enable coarse granularity writes to bins. Figure 4a shows the performance of the *Binning* and *Accumulate* phases as the number of bins vary, for the `Neighbor-Populate` kernel. Figure 4b shows normalized L1 load misses (broken into L2, LLC, and DRAM accesses) collected using hardware performance counters (Section VI). Optimal *Accumulate* performance is achieved when there are a large number of bins because the range of locations modified by a bin's updates is reduced to the point that they can fit within the L1 cache. However, selecting a large number of bins is not feasible because the *Binning* phase achieves the worst performance as all the bins' *coalescing buffers* do not fit in the L1 and L2 caches (Figure 4b). Competing bin requirements by the *Binning* and *Accumulate* phases force all PB executions to make a *compromise* and select a medium number of bins (red dotted line in Figure 4a), leading to sub-optimal performance in both phases. The architecture mechanism that we design in Section IV shows how to break PB's dependence on the number of bins and get high performance in both phases *without a compromise*.

An ideal PB mechanism would use the best number of bins for each phase – a small number of bins for *Binning* and a large number of bins for *Accumulate* (green dotted lines in Figure 4a). Figure 5 shows PB's performance gains compared to this idealized version of PB (PB-SW-IDEAL). While this ideal PB variant is unrealizable, the data highlight the ample headroom for improvement in PB.



(a) Execution time versus number of bins

(b) Cache locality versus number of bins

Figure 4: **Sensitivity of PB to the number of bins:** *The Binning phase achieves better locality with a small number of bins whereas the Accumulate phase prefers a large number of bins.*

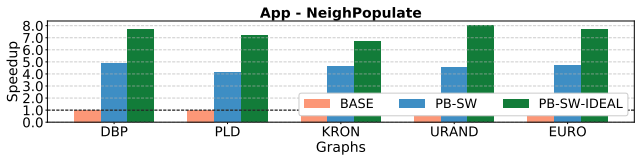


Figure 5: **Ideal performance with Propagation Blocking**: *Allowing each phase to operate with the best number of bins shows the headroom for performance improvement in PB.*

Control overheads of Software PB: PB implemented in software requires executing extra instructions for binning which degrades instruction level parallelism (ILP) by occupying core resources (e.g., reservation stations, load-store queue, reorder buffer). We found that PB executes up to 4x more instructions compared to the baseline execution of `Neighbor-Populate`. The two inefficiencies of PB – compromising on the number of bins and instruction overhead for binning updates – present an opportunity to improve the performance of PB. In the next section, we discuss our architecture support for *Binning* (the dominant phase of PB as shown in Table I) that eliminates both the inefficiencies of PB.

APPS →	*PB Phases* ↓	DC	NP	PR	RD	IS	SPMV	PINV	TR	SP
Medium No. of Bins	**Init**	9.91%	5.68%	18.43%	23.75%	5.72%	0.29%	8.6%	14.63%	6.76%
	Binning	73%	54.18%	47%	51.78%	44.47%	77.09%	56.11%	48.16%	14.26%
	Accumulate	17.09%	40.14%	34.57%	24.47%	49.81%	22.61%	35.28%	37.21%	78.98%
Large No. of Bins	**Init**	7.72%	6.01%	13.17%	18.22%	6.95%	0.22%	8.88%	11.96%	6.71%
	Binning	86.15%	78.57%	65.52%	71.60%	77.94%	87.46%	64.42%	67.82%	17.1%
	Accumulate	6.01%	15.42%	21.32%	10.18%	15.12%	12.32%	26.7%	20.22%	76.19%

Table I: **PB execution breakup:** *Binning dominates a PB execution both when using a medium no. of bins (which offers the best overall PB performance) and when using a large no. of bins (which offers the best Accumulate performance).*

IV. Optimizing PB with COBRA

The core contribution of this paper is a new system called COBRA[2] that specializes the cache hierarchy to eliminate the two inefficiencies of PB executions. COBRA's architecture extensions are specifically targeted at improving *Binning* performance when there are a large number of bins. Since *Accumulate* is naturally efficient with a large number of bins, the improved *Binning* performance with COBRA allows achieving performance close to ideal PB (Figure 5).

Inefficiencies of the Binning phase: PB maintains bins in main memory that each accumulate irregular updates to disjoint sub-ranges during the *Binning* phase. Later, bins are sequentially processed and the updates of each bin are applied with high cache locality during the *Accumulate* phase. The *Binning* phase uses per-bin, cacheline-sized coalescing buffers (henceforth referred to as *C-Buffers*) to amortize the cost of writing update tuples to in-memory bins. The size of index and update values determines the number of tuples in a *C-Buffer*. When a *C-Buffer* fills up, the core bulk-transfers all of the *C-Buffer*'s tuples into its corresponding bin in memory and clears the *C-Buffer* to start collecting tuples again.

The *Binning* phase has two main sources of inefficiency. First, *Binning* suffers poor cache performance when there are a large number of bins. Figure 6 (left) shows why *Binning* performs poorly with many bins in a typical 3-level cache hierarchy. With a large number of bins, all the per-bin *C-Buffers* do not fit in a small cache (e.g., L1), increasing the average latency of inserting tuples into *C-Buffers*. Compounding the problem, increased cache demands by other program data can displace *C-Buffers* to lower levels of cache (e.g., LLC), further increasing access latency. The second main inefficiency in *Binning* is that *C-Buffers* are managed entirely in software. Extra instructions need to be executed to write to *C-Buffers*, detect when a *C-Buffer* fills, and bulk-transfer the *C-Buffer*'s tuples to in-memory bins.

An Architecture for Binning: COBRA optimizes PB by enabling the selection of a large number of bins that offers optimal *Accumulate* performance (Figure 4a) and changing the operation of the memory hierarchy to make *Binning* more efficient with many bins. The key insight of COBRA is to **decouple** *Binning* performance from the number of bins in memory. Instead of using a single set of software-managed *C-Buffers* that can spread across the cache hierarchy, COBRA introduces a *hierarchy of hardware-managed C-Buffers*. Each level of the cache hierarchy has its own set of *C-Buffers* with the number of *C-Buffers* in a level bounded by the capacity of that level. Therefore, the L1 cache has the fewest *C-Buffers* and the Last Level Cache (LLC) has the most *C-Buffers*. In contrast to software-PB where a single bin range maps update tuples to bins (Algorithm 2; Line 4), in COBRA each cache level has a unique bin range used to map tuples into one of the level's *C-Buffers*. For example, in Figure 6, the bin range used for mapping tuples into L1 *C-Buffers* is $\texttt{L1BinRange}(R_{L1}) = \frac{|UniqIndices|}{|Y_1|}$ while the bin range for the LLC is $\texttt{LLCBinRange}(R_{LLC}) = \frac{|UniqIndices|}{|Y_3|}$.

[2]Since PB is an instance of radix partitioning [13], [54], we named our system COBRA (Cache Optimized Binning for RAdix partitioning)

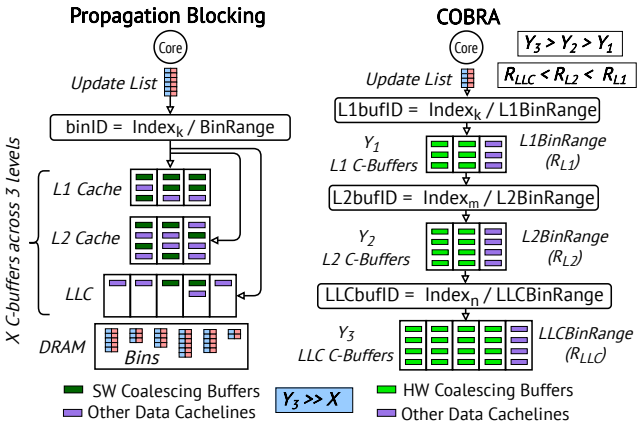


Figure 6: **Comparing Binning phases of PB and COBRA**: *COBRA maintains a hierarchy of HW-managed C-Buffers to provide the illusion of a small number of bins for Binning while actually using a large number of bins for Accumulate. We do not show bins in DRAM for COBRA (COBRA uses Y_3 bins in DRAM). The per-level bin ranges (R_{L1}, R_{L2}, R_{LLC}) in COBRA are defined by the input range and cache sizes.*

In COBRA, a core interacts only with the L1 *C-Buffers*, writing tuples into one of the Y_1 *C-Buffers* using the `L1BinRange` ($L1BufferID = \frac{Index}{R_{L1}}$). When an L1 *C-Buffer* fills up, COBRA does not transfer its contents directly to an in-memory bin (as in software PB). Instead, COBRA *evicts* the L1 *C-Buffer* by unpacking its tuples and sending each tuple to its *C-Buffer* in the L2 cache. Unlike a traditional cache eviction where the evicted line is sent to the next cache level *as a whole*, during a *C-Buffer* eviction each tuple in the filled *C-Buffer* in level L_i may need to be written to a different *C-Buffer* in the next cache level (L_{i+1}). COBRA writes each tuple evicted from the filled L1 *C-Buffer* into one of Y_2 *C-Buffers* in the L2 cache identified by the `L2BinRange` ($L2BufferID = \frac{EvictedIndex}{R_{L2}}$). Similarly, when an L2 *C-Buffer* fills up, COBRA evicts it from L2 and sends each of its tuples to one of the Y_3 *C-Buffers* present in the LLC. In COBRA, the number of bins in memory equals the number of LLC *C-Buffers*. Therefore, when finally a LLC *C-Buffer* fills up, COBRA transfers all the tuples in the filled LLC *C-Buffer* to the corresponding bin in main memory (as in software PB). During the *Binning* phase in a COBRA execution, all tuples generated by the core are inserted into the L1 *C-Buffers*, eventually evicted into L2 *C-Buffers* followed by the LLC *C-Buffers*, before finally being written to the bins in memory.

The hierarchical buffering mechanism in COBRA causes each *C-Buffer* eviction to *scatter* tuples across the *C-Buffers* of the next cache level. A small number of eviction

buffers between cache levels suffice to hide the latency of scattering tuples and remove *C-Buffer* eviction latency off the critical path (Section V-D). Consequently, for the example in Figure 6, during the *Binning* phase the core observes a latency of inserting tuples into a small number of bins (Y_1 *C-Buffers* at the L1) while actually operating on a large number of bins in memory (equal to the Y_3 *C-Buffers* at the LLC).

Besides hierarchical buffering, the COBRA architecture offers a second major efficiency boost to *Binning*. COBRA relies on simple fixed function logic in each cache level's controllers to handle *C-Buffer* management operations (such as detecting when a *C-Buffer* fills, and unpacking tuples from a filled L_i *C-Buffer* and inserting each tuple to an appropriate L_{i+1} *C-Buffer*). COBRA reserves space within each cache level to pin *C-Buffers* for the entirety of *Binning* (Figure 6), allowing simple logic to determine the unique location of a *C-Buffer* within a cache level. Offloading *C-Buffer* management to hardware allows COBRA to eliminate the extra instruction overhead of *Binning* in software-based PB.

V. Architecture Support for COBRA

COBRA's architecture support for optimizing *Binning* in PB include extensions to cache controllers for managing *C-Buffers* and buffering to hide *C-Buffer* eviction latencies.

A. Caches Designed for Binning

COBRA modifies the cache hierarchy in two ways. First, COBRA uses widely available way-based cache partitioning [3] to reserve space for *C-Buffers* within each cache level, ensuring that other program data never displace *C-Buffers*. Second, COBRA keeps a hierarchy of *C-Buffers*. Each level in the cache hierarchy has its own set of *C-Buffers* bounded by the level's capacity. COBRA also uses a unique bin range for each cache level to map update tuples into one of the level's *C-Buffers*. Practically, a cache level's bin range must also be a power of two, which makes binning a tuple cheap (Line 4 of Algorithm 2 can use a bitshift).

A new instruction – `bininit` – configures the number of ways to reserve for *C-Buffers* at each cache level. The `bininit` instruction takes four operands: (1) a cache level identifier (e.g., L1, L2, or LLC in most systems), (2) number of ways to reserve for *C-Buffers*, (3) number of unique indices in the data namespace (e.g., the number of vertices in a graph), and (4) tuple size in bytes. A program executes `bininit` once for each cache level. `bininit` first reserves the specified number of cache ways and then computes the smallest power-of-two bin range for which *C-Buffers* fit in the reserved cache ways. This per-level bin range is stored in a special register, used later during *Binning*. Due to the power-of-two requirement on bin ranges, the *C-Buffers* may not use all the reserved ways. So the `bininit` instruction saves the number of ways actually used by *C-Buffers* to allow other data to reclaim unused ways.

The number of ways to reserve at a cache level depends on the cache pressure from non-*C-Buffer* accesses. For our simulated architecture, we reserve all but one way in each level of cache except the L2 cache. Due to the presence of an L2 stream prefetcher, we reserve a single way for L2 *C-Buffers* to retain cache capacity for prefetched data. Later, we show that COBRA's performance is not very sensitive to the number of ways reserved for *C-Buffers* (Figure 13b).

B. An ISA Extension for Binning

COBRA introduces a new instruction (`binupdate`) to improve the efficiency of the *Binning* phase. The `binupdate` instruction replaces **all** *Binning* related operations performed in a baseline software PB execution. The `binupdate` instruction takes two operands - an *index* and a *value* to be used to update the data stored at the index. For example, the *Binning* phase in Algorithm 2 (lines 3–5) would be replaced by a single instruction {`binupdate e.src e.dst`}. Dedicated hardware in the cache controller identifies the right *C-Buffer* for the input tuple and inserts the tuple (`e.src`, `e.dst`) into that *C-Buffer*.

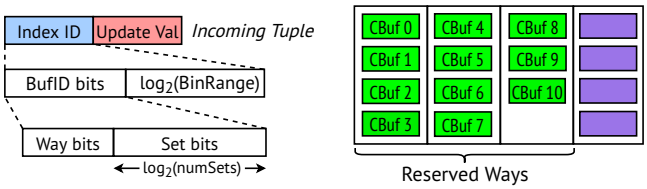


Figure 7: ***C-Buffer* organization within each cache level:** *Each cache level has a unique bin range that is used to map an incoming tuple into one of the C-Buffers pinned in cache.*

To use COBRA, a program first executes the `bininit` instruction for each cache level and then starts binning data using `binupdate` instructions. A `binupdate` instruction uses the index part of its input tuple to find a target L1 *C-Buffer* (Figure 7). Since the bin range is a power-of-two, the lower $log_2(bin\_range)$ bits of the index represent an intra-bin-range offset and the remaining bits identify the buffer ID. Reserving ways allows each *C-Buffer* to have a unique location within the cache and, hence, the buffer ID further breaks down into sets bits and way bits, fully determining the location of the *C-Buffer* within the cache.

C. Inserting tuples into C-Buffers

COBRA collects update tuples in cacheline-sized *C-Buffers*. When a *C-Buffer* fills up with update tuples, the cache evicts the *C-Buffer* line, scattering its tuples into the *C-Buffers* of the next level of the memory hierarchy. COBRA adds fixed function logic at the cache controller to support inserting an update tuple into a *C-Buffer*. Normally, a cache uses the address to identify the bytes to be accessed within a cache line. A `binupdate` instruction accesses a cache line differently, because the *C-Buffer* that the line contains

is not byte addressable. To insert a tuple into a *C-Buffer*, COBRA must determine the tuple's offset within the *C-Buffer* line. COBRA maintains offset counters for each *C-Buffer* to explicitly track the offset of the next tuple within each *C-Buffer* line, essentially providing *append-only* access to *C-Buffer* lines. To insert a tuple into a *C-Buffer*, the controller first reads the offset counter, inserts the tuple at the right offset within the *C-Buffer* line, and increments the counter to point to the location for the next incoming tuple. When a *C-Buffer* cache line fills, the counter wraps around to zero.

To store per-*C-Buffer* offset counters, COBRA repurposes existing metadata bits associated with the cache lines containing *C-Buffers*. Repurposing these bits is safe because a *C-Buffer* line exists outside the shared-memory address space (i.e. *C-Buffers* only reside in the cache hierarchy and are not present in memory). COBRA can also repurpose the coherence state bits because *C-Buffers* are core-private (software PB already duplicates all bins and *C-Buffers* across threads – Algorithm 2). For example, a typical tuple size of 8B (4B for index and value) requires tracking 8 tuples in a typical 64B cache line. For tracking the 8 tuples within L1 and L2 cache lines, COBRA can repurpose 1bit from PLRU replacement, 1bit from dirty status bit, and 2bits from the MESI coherence status bits for a 3-bit offset counter.

D. Handling C-Buffer evictions

As the `binupdate` instruction fills L1 *C-Buffers* with tuples, eventually a L1 *C-Buffer* fills up and COBRA must evict its tuples to *C-Buffers* in the next cache level (L2). When a *C-Buffer* fills up, COBRA is responsible for inserting each tuple in the full buffer into the appropriate *C-Buffer* in the next cache level. After an eviction, the *C-Buffer* is empty and can service future incoming tuples.

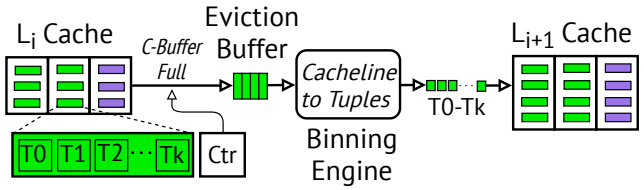


Figure 8: **Handling evictions when a *C-Buffer* fills up:** *Eviction buffers hide the latency of evicting tuples.*

In a naive implementation, the filled *C-Buffer* line is evicted and directly sent to a dedicated hardware unit within the cache controller called the *binning engine*. The binning engine sequentially extracts tuples from an evicted *C-Buffer* cache line and serially issues each tuple to the *C-Buffers* in the next level of cache. The process of inserting a tuple into the next cache level's *C-Buffer* is exactly the same as a `binupdate` instruction inserting tuples into an L1 *C-Buffer* (Figure 7). COBRA inserts each evicted tuple into a *C-Buffer* in the next cache level using that level's unique bin range. Figure 8 shows how COBRA evicts a full *C-Buffer*. The cache controller determines when to evict a *C-Buffer* cache line by monitoring the line's offset counter, which is incremented on each tuple insertion. When the counter wraps around, the *C-Buffer* line is at capacity and the controller evicts the line.

In the above naive implementation, COBRA incurs the full eviction latency to sequentially issue all tuples from a filled buffer to the *C-Buffers* in the next level of cache. To hide the latency of *C-Buffer* evictions, COBRA uses a set of first-in/first-out (FIFO) *eviction buffers* between cache levels. When a *C-Buffer* fills, COBRA simply inserts the cache line containing all of its tuples into the eviction buffer. Later, the binning engine pulls the cache line from the eviction buffer, extracts tuples from the line, and inserts the tuples into next level's *C-Buffers*. Accounting for the eviction rate from each level of cache and the cycle time to serially insert all the tuples to the next level of cache, Little's Law [28] suggests that a 14-entry eviction buffer between L1 and L2 and single entry eviction buffer between L2 and LLC hides all eviction latency. However, Little's law assumes *steady state* eviction rate and does not account for bursts in evictions, which may underestimate the eviction buffer size. In a later section, we refine the Little's Law estimate using a DES model that accounts for bursts and the model shows that a 32-entry eviction buffer between L1 and L2 fully hides *C-Buffer* eviction latency (Figure 13a) .

E. Additional implementation details

COBRA manages the LLC differently from L1 and L2 because the LLC is shared among all the cores and evictions of LLC *C-Buffers* interact with bins in main memory.

C-Buffers organization in NUCA caches: As Section III explains, PB creates a per-thread duplicate of bins and *C-Buffers* and COBRA exploits the duplication by making the *C-Buffer* in each level private to a core. For core-private caches, most cache space is reserved for the *C-Buffers*. In a shared, NUCA LLC that is physically distributed across cores, COBRA evenly divides NUCA cache banks among cores. Each core's LLC *C-Buffers* use the set of NUCA banks assigned to that core only. Our simulated architecture (Section VI) models a typical three-level hierarchy with core-private the L1 and L2 caches and the LLC is a NUCA cache with a bank associated with each core. For this cache hierarchy, the number of LLC *C-Buffers* for each core is bounded by the capacity of the core-local NUCA bank.

Evicting from LLC: Eviction of a full LLC *C-Buffer* moves buffered updates to a bin in memory, which is unlike a *C-Buffer* eviction to the next cache level (Section V-D). The process of evicting a *C-Buffer* from the LLC depends on how COBRA represents bins in memory. COBRA assumes threads' bin data are stored sequentially in memory, as in Figure 9: for each thread, a bin's tuples are stored contiguously. The sequential bin organization requires pre-computing the number of tuples in each bin (for each thread). Fortunately, to avoid dynamic memory allocation overheads,

a baseline PB execution already precomputes the number of tuples per bin and encodes it in the $BinOffset$ array (Init phase in Table I). With the above organization, each core can store the base pointer for its thread's bin data structures and use per-bin offsets to access each bin. When an LLC *C-Buffer* fills up, COBRA writes the buffered tuples to the bin data structure at the location pointed by $BinOffset[binID]$. After an eviction from LLC, the bin offset is incremented by the number of tuples in the *C-Buffer*.

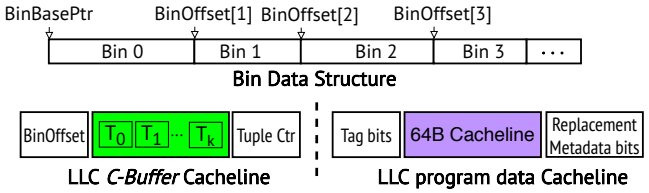


Figure 9: **Organization of per-thread bins in memory:** *BinOffsets are stored in the tag bits of cache lines containing LLC C-Buffers.*

The bin offset pointers do not require any additional storage. Since the LLC *C-Buffers* are chosen to fit in a core's local NUCA bank, the tag bits for the *C-Buffer* lines are unnecessary and can be repurposed to store bin offsets ($BinOffset[binID]$) as shown in Figure 9. Before *Binning*, COBRA initializes the starting offsets for each LLC *C-Buffer* (in every NUCA bank) in the *C-Buffer*'s tag entry using a new ISA instruction that takes a buffer ID and starting bin offset as operand. At a LLC *C-Buffer* eviction, the contents of the LLC *C-Buffer* line are written to the memory address computed using the bin offset in line's tag entry ($BinBasePtr + BinOffset[binID]$). To avoid the need for address translation, we assume system support for ensuring matching virtual and physical addresses for important data structures (e.g. the bin data structure) as proposed in prior work [26], [43]. After writing the contents of the full LLC *C-Buffer* to an in-memory bin, the bin offset value in the tag is incremented by the number of tuples in the *C-Buffer* using fixed function logic and the *C-Buffer*'s offset counters are reset to start receiving tuples again.

Flushing the Cache After Binning: In COBRA, all tuples are inserted by a `binupdate` instruction into L1 *C-Buffers*, later evicted to L2 and LLC *C-Buffers*, and eventually written to a bin in main memory. When *Binning* ends, some tuples may still be resident in a *C-Buffer* in cache. Before starting the next phase of PB (*Accumulate*), COBRA must ensure that all remaining tuples in cache end up in in-memory bins. We add a `binflush` ISA instruction that signals the end of *Binning* and causes each cache level's controller to serially walk all *C-Buffer* cache lines, forcing an eviction if the line is non-empty. In each core, `binflush` starts with L1, proceeds to L2 and, finishes by writing the residual tuples in the local NUCA bank to memory. The eviction process initiated by `binflush` proceeds as described in Sections V-D and V-E, with the difference that the eviction buffers, binning engine, and bin offsets update logic must handle partially filled *C-Buffers*. The number of tuples remaining in a *C-Buffer* are identified with the help of per-*C-Buffer* offset counters (Figures 8,9). The `binflush` instruction is also invoked in case a page containing per-thread bin data structures is swapped out of memory. Such premature invocations of `binflush` can be avoided by locking critical pages in memory (e.g., using `mlock()` in Linux).

Handling virtualization: COBRA extends a commodity multicore processor to accelerate PB, requiring its extensions to support virtualization for OS preemption and context switching. We rely on per-process, way-partitioning (as in Intel CAT [3]) to reserve space for per-level *C-Buffers* across the cache hierarchy. Using static cache partitioning for the COBRA process ensures that each level's *C-Buffers* are pinned for the duration of the *Binning* phase of PB. However, if the COBRA process is preempted during *Binning*, then other processes scheduled to run intermediately can evict *C-Buffer* cache lines. Evictions triggered by other processes may lead to transfer of partially-filled *C-Buffer* cache lines which reduces the efficiency of data transfer. Fortunately, COBRA's architecture extensions significantly optimize *Binning* phase latency, allowing *Binning* to complete with the minimum number of OS preemptions (Figure 13c).

Need for Static Cache Partitioning: The COBRA architecture described to this point assumes static cache partitioning at each cache level to reserve space for *C-Buffers* and ensure that *C-Buffer* accesses never miss. COBRA can also work in architectures lacking support for static cache partitioning. However, without static cache partitioning, locality of *C-Buffer* cache lines is defined by the underlying cache replacement policy and pressure from other data accesses during the *Binning* phase. Fortunately, all other data accesses besides *C-Buffers* are streaming accesses (e.g., CSR and `auxData` in Figure 3) and do not impose cache pressure on *C-Buffer* cache lines. Evaluations on our cache simulator revealed that the baseline replacement policy (PLRU in L1/L2 and DRRIP in LLC) can provide a *C-Buffer* miss rate of $<1\%$ in the absence of static cache partitioning.

Hardware overheads of COBRA: COBRA repurposes cache line metadata whenever possible (Figure 9) and the only storage overhead incurred by COBRA are the eviction buffers used to hide *C-Buffer* eviction latency. However, the small eviction buffers between cache levels amount for less than 7% of an L1 cache area [44]. Finally, the binning engine and fixed function logic to update per-*C-Buffer* counters incur low complexity because they perform simple integer arithmetic.

VI. Experimental Setup

Real System: Section II, III & VII-D experiments were run on an Intel Xeon processor (14 cores, 35MB LLC, 32GB

DRAM) with hyperthreading and "turbo boost" disabled. We used LIKWID [58] to collect performance counters.

Simulator: We use Sniper [15] to evaluate COBRA. Table II shows the architecture parameters we simulated, with cache timing parameters collected from CACTI [44]. We made several modifications to Sniper. For PB, we added support for non-temporal stores which are required for efficient *Binning* [13], [54]. For COBRA, we model the interaction of the `binupdate` instruction with the Out-of-Order engine. We ensure that a `binupdate` only retires when it reaches the head of the ROB because a `binupdate` writes the data caches (i.e., like a store). Stores require two ports to issue (address generation and data), but the `binupdate` does not need the address generation port because the L1 *C-Buffers* are directly addressed based on operand value. We also use a custom Pin-based [39] cache simulator for a subset of our evaluations (Section VII-C). The cache simulator models the hierarchy in Table II and the LLC statistics from our simulator are within 5% of Sniper's values.

Cores	16 OoO-cores, 2.66GHz, 4-wide issue, 128-entry ROB, 48-entry Load Queue, 32-entry Store Queue
L1(D/I)	32KB, 8-way set associative, Bit-PLRU policy, Load-to-use = 3 cycles
L2	256KB, 8-way set associative, Bit-PLRU policy, Load-to-use = 8 cycles
LLC	2MB/core, 16-ways, DRRIP [29], Load-to-use = 21 cycles (local NUCA bank)
NoC	4x4 mesh, 2 cycles hop-latency, 64 bits/cycle link B/W, MESI coherence
DRAM	80ns access latency

Table II: **Simulation parameters**

Workloads: We evaluate COBRA across workloads spanning multiple domains. `Degree-Counting` and `Neighbor-Populate` are the dominant kernels in Edgelist-to-CSR conversion (an important graph preprocessing step that has been shown to be as expensive as the downstream graph analytics kernel [6], [10], [40], [51]). `Pagerank` is a popular kernel representative of graph applications where all vertices are processed every iteration. Our Edgelist-to-CSR conversion and `Pagerank` implementations are from the GAP [12] benchmark. `Radii` from the Ligra [55] benchmark estimates a graph's diameter by performing multi-source BFS and is representative of graph applications which only process a subset of the vertices every iteration. In addition to graph (pre-)processing workloads, we also evaluate `Integer Sorting`. We use `__gnu_parallel::sort()` as our baseline sort implementation because we found it to be up to 14% faster (2.7% on average) than NAS benchmark's [5] integer sort implementation. The PB and COBRA versions optimize a parallel counting sort implementation [16]. We also evaluate COBRA across four sparse linear algebra kernels – `SpMV` from HPCG [20], and parallelized versions of `PINV`, `Transpose`, and `SymPerm` from SuiteSparse [17]. `PINV` computes the inverse mapping for a given permutation of a matrix rows/columns. `Transpose` constructs the sparse representation of a matrix's transpose. `SymPerm` permutes the upper triangular portion of a matrix and is a subroutine of Cholesky factorization. We simulate a single iteration of `Pagerank` because of its constant runtime across iterations and we use *iteration sampling* [43] to simulate every second pull iteration for `Radii`.

The workloads have tuple sizes of 4B (`Degree-Counting` and `Integer Sort`), 8B (`Neighbor-Populate`, `Pagerank`), and 16B for the rest. PB (and COBRA) work for applications performing irregular update and streaming reads which required slight modification of `Pagerank`, `Radii`, and `SpMV` (specifically making the PB versions process the transpose representation of the input graph/matrix). For the PB runs, we use the original source code which we received from the authors [13] and we simulated multiple bin ranges for PB, selecting the best bin range for each workload and input pair.

Inputs: We evaluate the graph (pre-)processing workloads across a diverse set of input graphs (covering power-law, uniform normal, and bounded degree distributions). For `Integer Sort`, we sort 256 million randomly generated keys with varying maximum key values. We use matrices representative of simulation and optimization problems for the sparse linear algebra kernels. We do not simulate `Radii` on `EURO` because `Radii` never executes a pull iteration.

GRAPHS	DBP [33]	PLD [37]	KRON [12]	URND [12]	EURO [18]
# Vertices	18.26M	42.89M	33.55M	33.55M	50.91M
# Edges	136.54M	623.06M	133.52M	134.22M	108.11M
MATRICES	**HBUBL [18]**	**HTRACE [18]**	**KMER [18]**	**DELAUNAY [18]**	
# Rows/Columns	21.12M	16M	67.72	16.78M	
# NNZs	63.58	48M	138.78M	100.66M	

Table III: **Input Graphs and Matrices**

VII. Evaluation

We provide a detailed quantitative explanation for COBRA's speedups and assess COBRA's composability with recently proposed commutativity optimizations for PB [43].

A. Speedups with COBRA

The main result of this evaluation is that COBRA consistently improves the performance of PB. COBRA improves PB performance in two ways – by eliminating the need to compromise with a sub-optimal number of bins and by eliminating the instruction overheads associated with *Binning*. To isolate the contributions from each optimization, Figure 10 compares speedups from a baseline software-based PB (PB-SW), PB-SW-Ideal (an idealized PB execution combining *Binning* with a small number of bins and *Accumulate* with a large number of bins), and COBRA. PB is an effective software locality optimization offering a mean speedup of 1.81x over the baseline. Eliminating the compromise on the number of bins (PB-SW-IDEAL) provides an additional mean speedup of 1.2x over PB. COBRA combines the benefits of using the optimal number of bins for *Accumulate* with the efficiency improvements from offloading *C-Buffer*

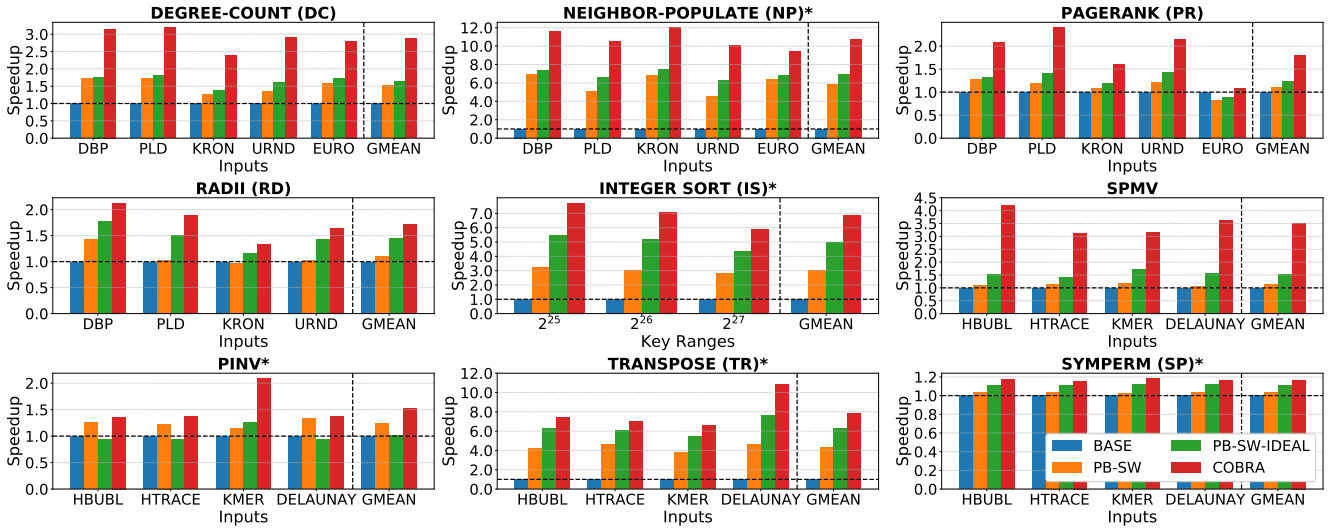


Figure 10: **Speedups with COBRA:** *COBRA provides significant performance gains over PB-SW (and PB-SW-IDEAL) across a broad set of applications. (* indicates that the application performs non-commutative updates)*

management to hardware, allowing COBRA to gain an additional mean speedups of 1.45x over PB-SW-IDEAL. In summary, COBRA provides a mean speedups of 1.74x over PB and 3.16x over the baseline. These COBRA speedup numbers include the cost of initializing LLC *C-Buffers* tags with starting bin offset values and flushing every level's *C-Buffers* after *Binning* (Section V-E). The numbers in Figure 10 also account for the initialization cost of computing per-thread bin sizes in both COBRA and PB. The results for `PINV` and `SymPerm` need additional explanations. `PINV` was the only application where increasing the number of bins did not improve *Accumulate* performance (due to parallelism artifacts overshadowing locality benefits). Consequently, PB-SW-IDEAL underperforms PB-SW for `PINV` and COBRA offers limited benefits over PB-SW. We ran a version of COBRA using a medium number of LLC *C-Buffers* (which offers the best *Accumulate* performance for `PINV`) and observed that COBRA's mean performance improvement increased to 2.4x over the baseline and 1.94x over SW-PB. `SymPerm` achieves limited benefit from COBRA because it only processes coordinates in the upper triangular portion of the matrix, limiting the headroom for spatial and temporal locality optimization.

Looking at the speedup for each phase of PB in Figure 11 helps explain COBRA's performance benefit. A COBRA execution optimizes *both* phases of PB. Compared to software-PB which much compromise with a sub-optimal number of bins, COBRA optimizes the *Accumulate* phase by using a large number of bins (allowing irregular updates to operate from faster caches). The *Binning* phase sees even more speedup, ranging from 2.2–32x owing to elimination of extra instructions and offloading the *C-Buffer* management to dedicated hardware in the cache controllers. The next section

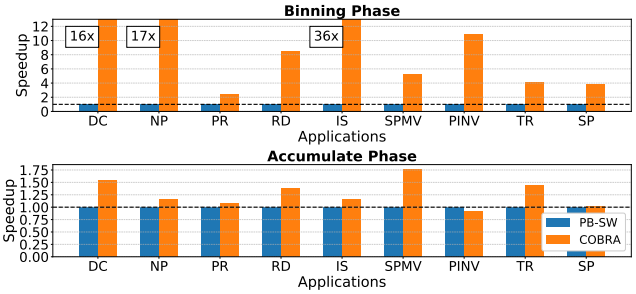


Figure 11: **COBRA speedup across both phases of PB**: *COBRA uses a large number of bins naturally optimizing Accumulate and uses architecture support to optimize Binning.*

further characterizes COBRA's improvements to *Binning*.

*B. Characterizing COBRA's **Binning** speedups*

In this section, we explore the reasons for COBRA's *Binning* speedups and show that COBRA's *Binning* performance is robust to different architecture and system parameters.

Improvements from eliminating instructions: COBRA's `binupdate` instruction replaces all *Binning* instructions executed in software by PB. Figure 12 (top) shows COBRA's 2-5.5× reduction in total instructions executed (averaged across inputs) compared to software PB. COBRA also reduces PB's control overheads during *Binning*. PB manages *C-Buffers* in software: after every tuple insertion the core must check if a *C-Buffer* is full. COBRA instead manages *C-Buffers* using dedicated hardware in the cache controller, reducing the rate of branch mispredictions, as Figure 12 (bottom) shows. COBRA eliminates all branch misses associated with managing *C-Buffers* in software, often achieving near-zero

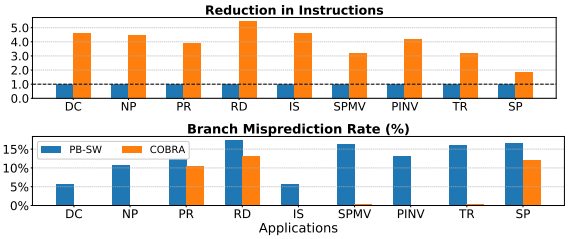
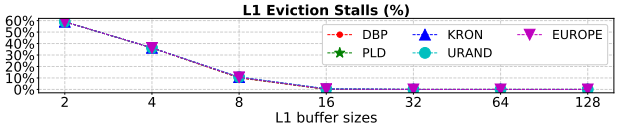
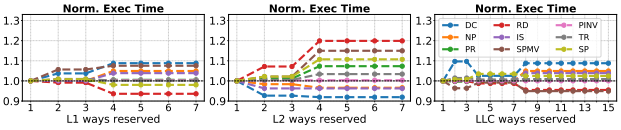


Figure 12: **Efficiency gains from eliminating instruction overhead of binning:** *The* `binupdate` *instruction in COBRA enables an OoO core to exploit more ILP.*

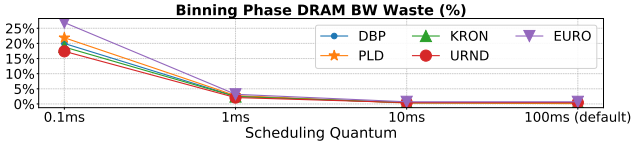
branch misprediction rates as the baseline versions[3]. By reducing the instruction and control overheads of *Binning*, COBRA enables an Out-of-Order processor to better exploit Instruction Level Parallelism (ILP) and we observe that the Instructions-per-Cycle (IPC) of the *Binning* phase improves from 0.71 in PB-SW to 1.55 in COBRA.



(a) Sensitivity to L1 eviction buffer size



(b) Sensitivity to ways reserved for *C-Buffers*



(c) Sensitivity to scheduling quantum

Figure 13: **Sensitivity of *Binning* performance in COBRA**

Sensitivity to eviction buffer sizes: COBRA uses eviction buffers to push *C-Buffer* eviction latencies off the critical path. We built a Discrete Event Simulation (DES) model of COBRA to estimate the eviction buffer sizes required to handle input-specific eviction bursts. The DES model consumes a trace of update tuples and reports the fraction of time stalled on a full eviction buffer. Figure 13a reports the fraction of `Neighbor-Populate` execution stalled for different sizes of eviction buffer between L1 and L2. The data show that a 32-entry L1 eviction buffer hides eviction latency for all inputs. Little's Law estimates that a single-entry buffer between L2 and LLC suffices because L2 evictions are infrequent. Overprovisioning the buffer to 8 entries incurs a modest cost and should suffice to handle rare bursts of L2 evictions [28].

Sensitivity to ways reserved for *C-Buffers*: We measured the sensitivity of COBRA's *Binning* performance for different workloads as the number of ways reserved for *C-Buffers* is varied (Figure 13b). The result shows robustness of COBRA's performance (variation$\leq$10%) to the L1 and LLC cache ways reserved for *C-Buffers* because all non-*C-Buffer* accesses during *Binning* are streaming, requiring limited cache capacity. COBRA's performance is more sensitive to the L2 cache ways reserved for *C-Buffers* because of the presence of a L2 stream prefetcher which gainfully uses the additional cache capacity to prefetch streaming data. Therefore, our default COBRA configuration reserves a maximum of all but one way in the L1 and LLC and a single way in the L2 for *C-Buffers*.

Sensitivity to context switches: COBRA uses static cache partitioning to pin the *C-Buffers* in caches during the *Binning* phase. However, on a context switch, other processes may evict (possibly partially-filled) *C-Buffer* cache lines. Evicting partially-filled *C-Buffer* cache lines at the LLC leads to DRAM bandwidth waste because DRAM is always accessed at the cache line granularity (64B). To measure the worst-case bandwidth waste, we built a cache simulator that models eviction of *all* the LLC *C-Buffers* on every context switch. Figure 13c shows the reduction in bandwidth waste as we vary the OS scheduling quantum for the `Neighbor-Populate` application. The worst-case bandwidth waste is less than 5% even when the scheduling quantum is 1/100th the default value used in linux [2]. COBRA's architecture extensions provides significant speedups to the *Binning* phase (8.3x on average in Figure 11) which reduces the number of context switches (and the associated bandwidth waste).

C. Specialization for Commutative Updates

The COBRA design described up to this point is application-agnostic and is primarily a latency optimization (choosing an optimal number of bins for *Accumulate* and using architecture support to accelerate *Binning*). Applications with commutative updates can additionally reduce main memory traffic by coalescing updates destined to the same index. For commutative applications, PHI [43] proposed adding simple reduction units (ALU) at private caches and an atomic reduction unit at the shared LLC to allow coalescing updates within each cache level. COBRA can also be specialized with similar reduction units to reduce main memory traffic for commutative applications. Instead of simply appending at the end of a *C-Buffer*, COBRA could scan all the tuples present in a *C-Buffer*, checking to see if a tuple with the update's index already exists. If a tuple with that index exists, the new update could be coalesced with the existing tuple using a local reduction unit. Otherwise,

[3](`Pagerank` and `Radii` still incur branch misses because checking neighborhood boundaries in power-law graphs leads to unpredictable branches. `SymPerm` incurs branch misses when searching for upper triangular coordinates).

COBRA appends the new update's tuple at the end of the *C-Buffer* as usual. To reduce the changes required to COBRA, we propose a commutativity-specialized version of COBRA (called COBRA-COMM) that performs update coalescing only at the LLC (where the coalescing opportunity is the largest). Using an atomic LLC reduction unit, as in PHI, allows COBRA-COMM to share LLC *C-Buffers* among cores, increasing the total number of LLC *C-Buffers*.

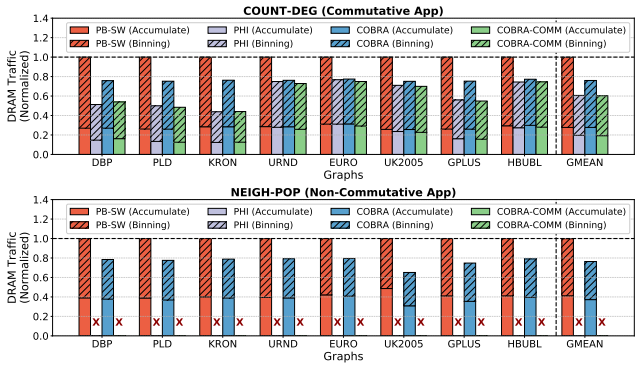
We compare PB (PB-SW), COBRA, COBRA-COMM and PHI [4] using our custom cache simulator. Figure 14 shows the reduction in DRAM traffic and L1 cache misses (across *Binning* and *Accumulate* phases) under PB-SW, PHI, COBRA, and COBRA-COMM for the `Count-Degrees` and `Neighbor-Populate` applications. The result reveals three interesting trends. First, PHI and COBRA-COMM are inapplicable for the non-commutative applications because they would violate correctness (Section III-B). For non-commutative applications (`Neighbor-Populate`, `Integer Sort`, `Transpose`, `PINV`, `SymPerm`), COBRA is the only viable hardware PB optimization. Second, for the commutative `Count-Degrees` application, PHI is able to provide greater DRAM traffic reduction than COBRA by hierarchically coalescing updates at each cache level (Figure 14a). Across all graphs, COBRA-COMM achieves the same traffic reductions as PHI in spite of only coalescing updates in LLC *C-Buffers* because even PHI coalesces a majority of the updates only at the LLC (97% on average). Traffic reductions in PHI (and COBRA-COMM) are tied to the highly skewed graphs and graphs with lower temporal reuse (*URND, EURO, UK2005, HBUBL*) see limited benefits from PHI over COBRA. Third, COBRA consistently reduce L1 caches misses compared to PHI (Figure 14b). COBRA minimizes L1 misses by choosing the optimal number of bins for the *Accumulate* phase whereas PHI's L1 miss reductions are a product of coalescing updates and reducing the number of tuples to be read from bins. For graphs with low coalescing opportunity (*URND, EURO, UK2005, HBUBL*), the *Accumulate* phase in PHI suffers from choosing a sub-optimal number of bins (as in PB-SW) and provides limited L1 miss reduction over PB-SW.

In summary, COBRA is a more general PB optimization because it applies to both commutative and non-commutative applications. With simple modifications, COBRA can be specialized for commutative updates (COBRA-COMM) to combine the benefits of update coalescing (achieving similar traffic reductions as PHI) and choosing the optimal number of bins for *Accumulate* (improving L1 locality over PHI).

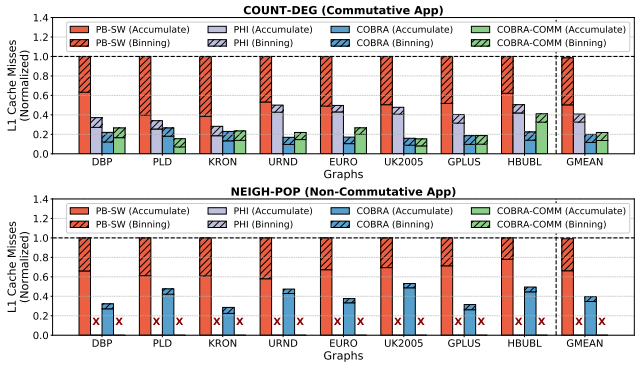
D. Comparison to Graph Tiling

In this work, we developed architecture support for software Propagation Blocking (PB). Another popular software

[4]We implement PHI's optimizations (hierarchical buffering/coalescing and selecting update batching) as described in the paper [43] and model an idealized version of PHI that incurs zero overheads for managing PB data.



(a) DRAM Traffic Reduction



(b) L1 Cache Miss Reduction

Figure 14: **Comparisons against PHI:** *PHI and COBRA-COMM only work for commutative updates*

locality optimization is Graph Tiling [51], [57], [62], [63], [66] where the input graph is divided into "sub-graphs" to reduce the range of irregular accesses. To understand the difference between the PB and Tiling, we compare PB to CSR-Segmenting [63], [64] (state-of-the-art 1D tiling). Figure 15 shows the runtime reduction from CSR-Segmenting (Tiling) and PB for the `Pagerank` application run until convergence. The shaded portion in the bars represents the initialization overheads of Tiling (constructing per-tile CSRs) and PB (allocating memory for bins). Ignoring overheads, PB provides a mean speedup of 1.35x compared to 1.27x from Tiling. However, PB incurs significantly lower initialization overhead compared to Tiling. Therefore, we selected PB as the basis for COBRA because PB is able to provide speedups even after accounting for overheads.

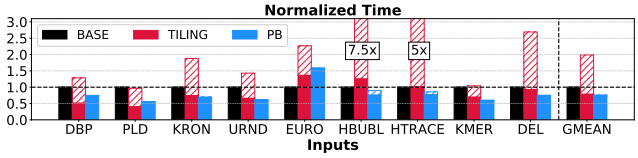


Figure 15: **Comparing PB to Tiling:** *PB is competitive to Tiling and incurs significantly lower overheads.*

VIII. Related Work

PB Optimizations for commutative updates: The Milk [32] compiler simplifies applying PB to applications. COBRA could be a target for the Milk compiler, extending COBRA's benefits to new applications. GraFBoost [31] exploits commutativity in PB for out-of-core graph analytics while COBRA targets in-memory analytics. GraphPulse [50] extends PB principles to an accelerator targeting asynchronous graph applications. In contrast to these works, COBRA is a general PB optimization that supports both commutative *and* non-commutative updates.

General PB optimizations: PCP [36] and GPOP [35] introduce new graph representations to reduce *Binning*'s memory traffic but they incur higher preprocessing overheads than PB. Ozdal et al. [46] propose algorithmic changes to PB to reduce memory footprint. COBRA could support variations of PB using extensions similar to Section VII-C. Prior works [54], [65] highlighted the sensitivity of radix partitioning to the number of bins and demonstrated huge performance cliffs. COBRA eliminates the need to tune the number of bins by maintaining a hierarchy of *C-Buffers*.

Architectures for graph processing: Minnow [61] added architecture support for efficient worklist management [49]. HATS [42] introduced hardware for online vertex scheduling to improve locality. OMEGA [4] tailored scratchpad-based memories to optimize graph analytics on power-law inputs. Graph processing accelerators [25], [47] optimize common framework operations. Similar to these works, COBRA provides architecture support for *Binning* to optimize PB.

IX. Conclusion

We presented COBRA, a general hardware optimization for Propagation Blocking (PB). With a limited set of architecture extensions, COBRA is able to eliminate the inefficiencies of PB executions on conventional processors. COBRA achieves speedups of up to 3.78x over PB (1.74x on average). Finally, by not relying on update commutativity, COBRA is able to target a broader range of workloads beyond graph analytics (including sparse linear algebra and integer sorting).

Acknowledgment

Thanks to the anonymous reviewers of HPCA22, MICRO21, ASPLOS21, ISCA21, and MICRO20 for their feedback on different versions of the paper. This work was supported in part by NSF grant XPS-1629196.

References

[1] "Graph-powered Machine Learning at Google," https://ai.googleblog.com/2016/10/graph-powered-machine-learning-at-google.html, accessed: 2019-01-23.

[2] "Linux scheduling," https://man7.org/linux/man-pages/man2/sched_rr_get_interval.2.html, accessed: 2021-04-16.

[3] "Intel CAT," https://github.com/intel/intel-cmt-cat, 2020, [Online; accessed 17-April-2020].

[4] A. Addisie, H. Kassa, O. Matthews, and V. Bertacco, "Heterogeneous memory subsystem for natural graph analytics," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, 2018, pp. 134–145.

[5] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber *et al.*, "The nas parallel benchmarks summary and preliminary results," in *Supercomputing'91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. IEEE, 1991, pp. 158–165.

[6] V. Balaji, "Input, representation, and access pattern guided cache locality optimizations for graph analytics," Ph.D. dissertation, Carnegie Mellon University, 2021.

[7] 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)*, 2021, pp. 668–681.

[8] V. Balaji and B. Lucia, "When is graph reordering an optimization? studying the effect of lightweight graph reordering across applications and input graphs," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, 2018, pp. 203–214.

[9] V. Balaji and B. Lucia, "Combining data duplication and graph reordering to accelerate parallel graph processing," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 133–144. [Online]. Available: https://doi.org/10.1145/3307681.3326609

[10] V. Balaji and B. Lucia, "Optimizing graph processing and preprocessing with hardware assisted propagation blocking," *CoRR*, vol. abs/2011.08451, 2020. [Online]. Available: https://arxiv.org/abs/2011.08451

[11] 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.

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

[13] S. Beamer, K. Asanović, and D. Patterson, "Reducing pagerank communication via propagation blocking," in *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 2017, pp. 820–831.

[14] D. Buono, F. Petrini, F. Checconi, X. Liu, X. Que, C. Long, and T.-C. Tuan, "Optimizing sparse matrix-vector multiplication for large-scale data analytics," in *Proceedings of the 2016 International Conference on Supercomputing*. ACM, 2016, p. 37.

[15] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: Association for Computing Machinery, 2011. [Online]. Available: https://doi.org/10.1145/2063384.2063454

[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[17] T. A. Davis, "Algorithm 1000: Suitesparse: Graphblas: Graph algorithms in the language of sparse linear algebra," *ACM Transactions on Mathematical Software (TOMS)*, vol. 45, no. 4, pp. 1–25, 2019.

[18] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.

[19] L. Dhulipala, G. E. Blelloch, and J. Shun, "Theoretically efficient parallel graph algorithms can be fast and scalable," *ACM Transactions on Parallel Computing (TOPC)*, vol. 8, no. 1, pp. 1–70, 2021.

[20] J. Dongarra, M. A. Heroux, and P. Luszczek, "Hpcg benchmark: a new metric for ranking high performance computing systems," *Knoxville, Tennessee*, pp. 1–11, 2015.

[21] D. Easley and J. Kleinberg, *Networks, crowds, and markets: Reasoning about a highly connected world*. Cambridge University Press, 2010.

[22] G. Gill, R. Dathathri, L. Hoang, R. Peri, and K. Pingali, "Single machine graph analytics on massive datasets using intel optane dc persistent memory," *Proceedings of the VLDB Endowment*, vol. 13, no. 8, pp. 1304–1318, 2020.

[23] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs." in *OSDI*, vol. 12, no. 1, 2012, p. 2.

[24] Z. Gu, J. Moreira, D. Edelsohn, and A. Azad, "Bandwidth optimized parallel algorithms for sparse matrix-matrix multiplication using propagation blocking," in *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, 2020, pp. 293–303.

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

[26] S. Haria, M. D. Hill, and M. M. Swift, "Devirtualizing memory in heterogeneous systems," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 637–650.

[27] M. A. Hassaan, M. Burtscher, and K. Pingali, "Ordered vs. unordered: a comparison of parallelism and work-efficiency in irregular algorithms," *Acm Sigplan Notices*, vol. 46, no. 8, pp. 3–12, 2011.

[28] M. D. Hill, "Three other models of computer system performance," *arXiv preprint arXiv:1901.02926*, 2019.

[29] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 60–71, 2010.

[30] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez, "Unlocking ordered parallelism with the swarm architecture," *IEEE Micro*, vol. 36, no. 3, pp. 105–117, 2016.

[31] S.-W. Jun, A. Wright, S. Zhang, S. Xu, and Arvind, "Grafboost: Using accelerated flash storage for external graph analytics," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18. IEEE Press, 2018, p. 411–424. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00042

[32] V. Kiriansky, Y. Zhang, and S. Amarasinghe, "Optimizing indirect memory references with milk," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, ser. PACT '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 299–312. [Online]. Available: https://doi.org/10.1145/2967938.2967948

[33] J. Kunegis, "Konect: the koblenz network collection," in *Proceedings of the 22nd International Conference on World Wide Web*. ACM, 2013, pp. 1343–1350.

[34] A. Kyrola, G. E. Blelloch, C. Guestrin *et al.*, "Graphchi: Large-scale graph computation on just a pc." in *OSDI*, vol. 12, 2012, pp. 31–46.

[35] K. Lakhotia, R. Kannan, S. Pati, and V. Prasanna, "Gpop: A scalable cache- and memory-efficient framework for graph processing over parts," *ACM Trans. Parallel Comput.*, vol. 7, no. 1, Mar. 2020. [Online]. Available: https://doi.org/10.1145/3380942

[36] K. Lakhotia, R. Kannan, and V. Prasanna, "Accelerating pagerank using partition-centric processing," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, 2018, pp. 427–440.

[37] O. Lehmberg, R. Meusel, and C. Bizer, "Graph structure in the web: aggregated by pay-level domain," in *Proceedings of the 2014 ACM conference on Web science*, 2014, pp. 119–128.

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

[39] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065034

[40] J. Malicevic, B. Lepers, and W. Zwaenepoel, "Everything you always wanted to know about multicore graph processing but were afraid to ask," in *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, 2017, pp. 631–643.

[41] F. McSherry, M. Isard, and D. G. Murray, "Scalability! but at what cost?" in *HotOS*, 2015.

[42] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, "Exploiting Locality in Graph Analytics through Hardware-Accelerated Traversal Scheduling," in *Proceedings of the 51st annual IEEE/ACM international symposium on Microarchitecture (MICRO-51)*, October 2018.

[43] A. Mukkara, N. Beckmann, and D. Sanchez, "Phi: Architectural support for synchronization-and bandwidth-efficient commutative scatter updates," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 1009–1022.

[44] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP Laboratories*, pp. 22–31, 2009.

[45] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray Users Group (CUG)*, 2010.

[46] M. M. Ozdal, "Improving efficiency of parallel vertex-centric algorithms for irregular graphs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 10, pp. 2265–2282, 2019.

[47] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 166–177.

[48] R. Pearce, M. Gokhale, and N. M. Amato, "Faster parallel traversal of scale free graphs at extreme scale with vertex delegates," in *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*. IEEE, 2014, pp. 549–559.

[49] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo *et al.*, "The tao of parallelism in algorithms," in *ACM Sigplan Notices*, vol. 46, no. 6. ACM, 2011, pp. 12–25.

[50] S. Rahman, N. Abu-Ghazaleh, and R. Gupta, "Graphpulse: An event-driven hardware accelerator for asynchronous graph processing," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 908–921.

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

[52] F. Sadi, J. Sweeney, S. McMillan, T. M. Low, J. C. Hoe, L. Pileggi, and F. Franchetti, "Pagerank acceleration for large graphs with scalable hardware and two-step spmv," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–7.

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

[54] F. M. Schuhknecht, P. Khanchandani, and J. Dittrich, "On the surprising difficulty of simple things: the case of radix partitioning," *Proceedings of the VLDB Endowment*, vol. 8, no. 9, pp. 934–937, 2015.

[55] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *ACM Sigplan Notices*, vol. 48, no. 8. ACM, 2013, pp. 135–146.

[56] J. Shun, L. Dhulipala, and G. E. Blelloch, "Smaller and faster: Parallel processing of compressed graphs with ligra+," in *Data Compression Conference (DCC), 2015*. IEEE, 2015, pp. 403–412.

[57] J. Sun, H. Vandierendonck, and D. S. Nikolopoulos, "Graphgrind: addressing load imbalance of graph partitioning," in *Proceedings of the International Conference on Supercomputing*. ACM, 2017, p. 16.

[58] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*. IEEE, 2010, pp. 207–216.

[59] 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*. ACM, 2016, pp. 1813–1828.

[60] X. Zeng, X. Song, T. Ma, X. Pan, Y. Zhou, Y. Hou, Z. Zhang, K. Li, G. Karypis, and F. Cheng, "Repurpose open data to discover therapeutics for covid-19 using deep learning," *Journal of proteome research*, 2020.

[61] D. Zhang, X. Ma, M. Thomson, and D. Chiou, "Minnow: Lightweight offload engines for worklist management and worklist-directed prefetching," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2018, pp. 593–607.

[62] K. Zhang, R. Chen, and H. Chen, "Numa-aware graph-structured analytics," *SIGPLAN Not.*, vol. 50, no. 8, pp. 183–193, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2858788.2688507

[63] 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)*, Dec 2017, pp. 293–302.

[64] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, "Graphit: a high-performance graph dsl," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, p. 121, 2018.

[65] Z. Zhang, H. Deshmukh, and J. M. Patel, "Data partitioning for in-memory systems: Myths, challenges, and opportunities." in *CIDR*, 2019.

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