

P-OPT: Practical Optimal Cache Replacement for Graph Analytics

Vignesh Balaji
Carnegie Mellon University
vigneshb@andrew.cmu.edu

Neal Crago
NVIDIA
ncrago@nvidia.com

Aamer Jaleel
NVIDIA
ajaleel@nvidia.com

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

Abstract—Graph analytics is an important workload that achieves suboptimal performance due to poor cache locality. State-of-the-art cache replacement policies fail to capture the highly dynamic and input-specific reuse patterns of graph applications. The transpose of a graph succinctly represents the next references of all vertices in a graph execution; enabling an efficient emulation of Belady’s MIN replacement policy. In this work, we propose P-OPT, an architecture solution that uses a specialized compressed representation of a transpose’s next reference information to enable a practical implementation of Belady’s MIN replacement policy. Our evaluations across multiple applications and inputs reveal that P-OPT improves cache locality for graph applications providing an average performance improvement of 33% (56% maximum) over LRU replacement.

I. INTRODUCTION

Graph processing is an important class of computations with valuable applications in network analysis, path-planning, machine learning, and COVID-19 therapeutics discovery [1], [12], [18], [55]. In the past, large graphs were primarily processed on distributed systems [23], [35], [46]. Today, increased main memory capacity and core counts allow processing graphs with billions of edges more efficiently on a single machine rather than a distributed system [38].

The large size of input graphs make high performance single-machine graph processing a challenge. Processing a typical input graph leads to a working set size much larger than the available on-chip cache capacity, leading to many DRAM accesses. The latency of these DRAM accesses often dominate execution time because memory accesses in graph processing are irregular and depend on graph structure: prior work estimates that graph kernels spend up to 80% of total time simply waiting for DRAM [6], [7], [57]. Techniques to improve cache locality and eliminate DRAM accesses are a major opportunity for improving end-to-end graph application performance.

A cache’s replacement policy is a key determinant of locality. Decades of work have produced high-performance replacement policies for various workloads. However, we find that state-of-the-art replacement policies are ineffective for graph processing. Graph data reuse is dynamically variable and graph-structure-dependent, two properties not captured well by existing replacement policies. Belady’s MIN replacement policy is an ideal policy that perfectly captures dynamic, graph-structure-dependent reuse, but it is impractical because it relies on knowledge of future accesses.

The main insight of our work is that a practical cache replacement policy can approach Belady’s MIN replacement policy without oracular future knowledge by *directly referring to the graph’s structure for replacement decisions*.

Replacement using the adjacency matrix captures structure-dependent access patterns. A graph kernel traverses a graph’s adjacency matrix with an outer loop over vertices in one dimension (e.g., column) and an inner loop over a vertex’s neighbors in the other dimension (e.g., rows). Assuming an outer-loop over columns, processing a vertex v scans down v ’s adjacency matrix column, to find neighboring vertices (e.g. u) in each non-zero row. The adjacency matrix encodes the next use of u . Scanning across u ’s row, the column of the next non-zero element in the row corresponds to the outer-loop vertex during the traversal of which, the execution will next access u as a neighbor. An optimal cache replacement decision is to evict data for the vertex next accessed on the iteration furthest in the future.

This work develops *Transpose-based Cache Replacement* (T-OPT), a high-performance replacement policy for graph data that directly uses a graph’s adjacency matrix to make near-optimal replacement decisions. Graph kernels use a compressed, sparse data structure — Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC) — that allows efficient traversal of one dimension (e.g., columns) but not both. Finding a vertex’s next use requires scanning the matrix in the opposite dimension from the traversal. T-OPT leverages the fact that most graph processing frameworks already store a graph *and* its transpose in a sparse format, allowing traversal in either dimension [11], [50]. T-OPT directly refers to the transpose to replace the line next used furthest in the future. The key challenge is to make transpose accesses for cache replacement efficient enough to improve end-to-end performance.

Our main contribution is P-OPT: an architecture for T-OPT that allows efficient access to a graph’s adjacency matrix and its transpose, enabling near-optimal cache replacement for graph data. P-OPT uses *epoch quantization* to compress the next reference information in the graph’s transpose. P-OPT uses simple cache partitioning and NUCA bank mapping techniques to efficiently store a specialized data structure (called the Rereference Matrix) containing a summary of graph transpose information. P-OPT’s mechanisms allow low-cost access to a vertex’s next reference at replacement time.

We evaluate P-OPT and show consistent performance improvement across a range of input graphs and applications.

P-OPT reduces Last Level Cache (LLC) misses by 35% on average leading to an average performance improvement of 33% relative to the LRU replacement policy. Compared to the more representative DRRIP [30], [52] baseline, P-OPT reduces LLC misses by 24% on average (52% maximum) providing an average performance improvement of 22% (53% maximum).

In summary, we make the following contributions :

- We show that state-of-the-art cache replacement policies are ineffective for graph processing. (Section II).
- We show that guiding replacement based on graph structure in T-OPT allows emulating Belady’s MIN policy *without* oracular knowledge of all future accesses (Section III).
- We describe P-OPT’s quantized Rereference Matrix data structure (Section IV) and architectural mechanisms (Section V) that allow low-cost access to the graph transpose for optimal replacement.
- We evaluate P-OPT across a range of graphs and applications comparing to many state-of-the-art systems, showing P-OPT’s consistent performance benefits (Section VII).

II. BACKGROUND: GRAPH PROCESSING AND CACHE REPLACEMENT

The goal of this work is to develop a practical implementation of Belady’s MIN replacement policy for graph processing applications. This section overviews the challenges of single-machine, multi-core graph processing.

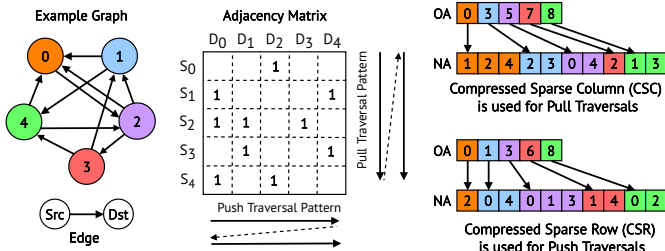


Fig. 1: Graph Traversal Patterns and Representations

A. Overview of Graph Processing

Graph processing frameworks ubiquitously use the same data structures and sub-computations [49]. A graph is abstractly represented as an *Adjacency Matrix*, which encodes directional edges between *source* and *destination* vertices as non-zero entries. Figure 1 shows an example adjacency matrix.

Data Structures: Analytics frameworks store graphs in compressed sparse formats because real-world graphs are often very sparse (>99% sparse [15]). The *Compressed Sparse Row/Column* (CSR/CSC) format is storage efficient and can quickly identify a vertex’s neighbors [14], [39]. Figure 1 shows the CSR and CSC for the example graph. CSR encodes *outgoing* destination neighbors for each source vertex. CSC encodes *incoming* source neighbors for each destination vertex. Both CSR and CSC use two arrays to represent the adjacency matrix. The Offsets Array (OA) stores the starting offset of a vertex’s neighbors in the Neighbor Array (NA). The NA

contiguously stores each vertex’s neighbors. To access the neighbor of vertex i in constant time, an application accesses the i^{th} and $(i+1)^{th}$ entries in OA to find the range of indices in NA containing vertex i ’s neighbors. Most frameworks [9], [47], [50], [51], [58] store CSR *and* CSC because common algorithms [44] and optimizations [11], [50] require fast access to both outgoing and incoming neighbors.

Graph Traversal Pattern: A common idiom in graph processing is to iteratively visit each edge and update per-vertex data corresponding to the edge’s source and destination vertices. Figure 1 shows that an application may traverse each source vertex’s *outgoing* neighbors (using CSR) or each destination vertex’s *incoming* neighbors (using CSC). Traversing outgoing neighbors (scanning adjacency matrix rows) is referred to as a *push* execution. Traversing incoming neighbors (scanning adjacency matrix columns) is a *pull* execution [11].

Algorithm 1 Pull execution of a graph kernel

```

1: for dst in G do
2:   for src in G.incoming_neighs(dst) do
3:     dstData[dst] += srcData[src]

```

Algorithm 1 shows a pull execution of a graph processing kernel which is similar to a Sparse Matrix Dense Vector product (SpM-DV). This kernel illustrates the key performance challenge of graph processing. Lines 1 and 2 traverse the graph’s CSC, first indexing into the OA using a destination vertex ID (dst) and then scanning incoming neighbors (source vertices) in the NA. Line 3 is a per-edge computation that indexes into graph application data ($dstData$ and $srcData$ arrays) using dst and src vertex IDs respectively. The CSC and $dstData$ accesses are streaming. However, the arbitrary order of the CSC’s Neighbor Array leads to an irregular, graph-dependent pattern of accesses to $srcData$. Real-world graphs are very large (GBs – TBs), and irregular accesses to large graphs have poor cache locality. Prior work showed that irregular DRAM access latency makes 60-80% of graph processing time [57]. Therefore, cache locality optimizations for graph processing can provide significant performance gains.

B. Limitations of existing replacement policies

Prior work produced high-performance replacement policies [28], [30], [32], [53] applicable to a range of workloads, but the characteristics of graph processing render state-of-the-art policies ineffective. We implement three state-of-the-art policies, comparing their cache miss rates for graph workloads against a baseline Least Recently Used (LRU) policy. DRRIP [30] offers scan-resistance and thrash-resistance. SHiP [53] uses signatures to predict re-references to application data. We implement two SHiP variants [53] – SHiP-PC and SHiP-Mem – that track replacement by PC and memory address respectively. We also compare to Hawkeye [28], the winning policy in the 2019 cache replacement championship [2]. Hawkeye retroactively applies Belady’s MIN replacement policy to a history of accesses to predict future re-references based on whether past accesses would have hit in cache.

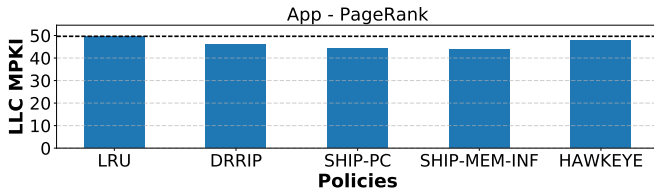


Fig. 2: **LLC Misses-Per-Kilo-Instructions (MPKI) across state-of-the-art policies:** Lower is better.

Figure 2 shows Last Level Cache (LLC) miss statistics for different policies for the `PageRank` application on a set of large graphs (Section VI details our setup). The data show that state-of-the-art policies do not substantially reduce misses compared to LRU. We observed that all policies have LLC miss rates in the range of 60% to 70%. The state-of-the-art policies fare poorly because graph processing applications do not meet their assumptions. Simple policies (LRU and DRRIP) do not learn graph-structure-dependent irregular access patterns. SHiP-PC and Hawkeye use the PC to predict re-reference, assuming all accesses by an instruction have the same reuse properties. As Algorithm 1 illustrates, graph applications violate this assumption because the same `srcData` access (line 3) will have different locality for high-connectivity vertices compared to the low-connectivity vertices. SHiP-Mem predicts re-reference using memory addresses, assuming that all accesses to a range of addresses will have the same reuse properties. Even with infinite storage to track individual cache lines, our idealized SHiP-Mem implementation provides little improvement over LRU, highlighting that graph workloads do not have static reuse properties. This data shows the ineffectiveness of state-of-the-art DRRIP, SHiP, and Hawkeye policies for graph processing, corroborating findings from prior work [20]. Therefore, we develop a graph-specific replacement policy to eliminate costly DRAM accesses and improve the performance of graph applications.

III. TRANSPOSE-BASED OPTIMAL CACHE REPLACEMENT

State-of-the-art replacement policies perform poorly for graph applications because they do not capture dynamically varied, graph-structure-dependent reuse patterns. Belady’s MIN replacement policy (which we call OPT) evicts the line accessed furthest in future. However, OPT is impractical because it requires oracular knowledge of future memory accesses. Our main insight is that for graph applications, the graph’s transpose stores sufficient information to practically emulate OPT behavior.

A. Transpose Encodes Future Reference Information

We first discuss a simple OPT implementation that (impractically) requires future knowledge, applied to the pull-based graph kernel in Algorithm 1. As shown in Figure 3 (left), a pull-based traversal sequentially visits each destination vertex’s incoming source neighbors (encoded in the CSC). The pull execution generates streaming accesses to the CSC (OA and NA) and `dstData`, while memory accesses to `srcData` depend on the contents of NA (Figure 1). To make

replacement decisions, this OPT implementation must scan the contents of NA to find the destination vertex for which the pull execution will next reference a particular source vertex element in `srcData`¹. In the example (Figure 3; left), after the first access to `srcData[S1]` while processing the incoming neighbors of vertex D_0 , OPT sequentially scans the NA to find that processing vertex D_4 will re-reference `srcData[S1]`. In the worst case, the entire NA may be scanned to find the next reference (if any) of a `srcData` array element, resulting in an extreme computational complexity of $O(|Edges|)$ for each replacement event.

Our main insight is that a graph’s transpose encodes the future re-reference information for each vertex allowing similar replacement as the OPT policy while incurring significantly lower computational complexity. Our insight is based on two observations about pull execution. First, a pull execution sequentially visits each vertex and processes all of its incoming neighbors (i.e., processing incoming neighbors of D_0 before moving on to incoming neighbors of D_1). Second, a pull execution processes the CSC for fast access to incoming neighbors (adjacency matrix columns) and the *transpose* of the graph (a CSR) allows quick access to outgoing neighbors (i.e. adjacency matrix rows). A cache can easily determine the next reference to `srcData[S1]` when it is first accessed as an incoming neighbor of vertex D_0 . By accessing the CSR, we can quickly learn that vertex S_1 has two outgoing neighbors – vertex D_0 and D_4 – and, hence, `srcData[S1]` will only be accessed next while processing the incoming neighbors of vertex D_4 . With the help of the transpose in the efficiently traversable CSR format, the complexity of finding the next future reference of a `srcData` element is reduced to $O(|OutDegree|)$, i.e., scanning the outgoing neighbors of a vertex². While this example describes a pull execution model, conversely, a push execution model using a CSR can also use its transpose (CSC) for estimation of next references to irregular `dstData` accesses.

B. Transpose-based Optimal Replacement Performance

We show how knowledge of next reference information estimated using a graph’s transpose is used to emulate OPT. Figure 3 (center panel) shows a 2-way set-associative cache in which each cache way can store only a single element of `srcData`. In replacement scenario (A), the cache has just incurred the cold misses for `srcData[S1]` and `srcData[S2]` and now needs to insert `srcData[S4]`. The cache must decide: will the execution access `srcData[S1]` or `srcData[S2]` further in the future? By scanning the outgoing neighbors of vertex S_1 (i.e. S_1 ’s row in the adjacency matrix), we can determine that S_1 will be accessed next when processing the neighbors of vertex D_4 . Similarly, the transpose informs us that S_2 will be accessed next when processing incoming neighbors of vertex D_1 . Therefore, to emulate OPT

¹By virtue of visiting each element only once, streaming data (like `dstData`) have a fixed re-reference distance of infinity.

²Real-world graphs typically have an average degree of 4-32 which is orders of magnitude lower than the number of graph edges (order of 100M-1B).

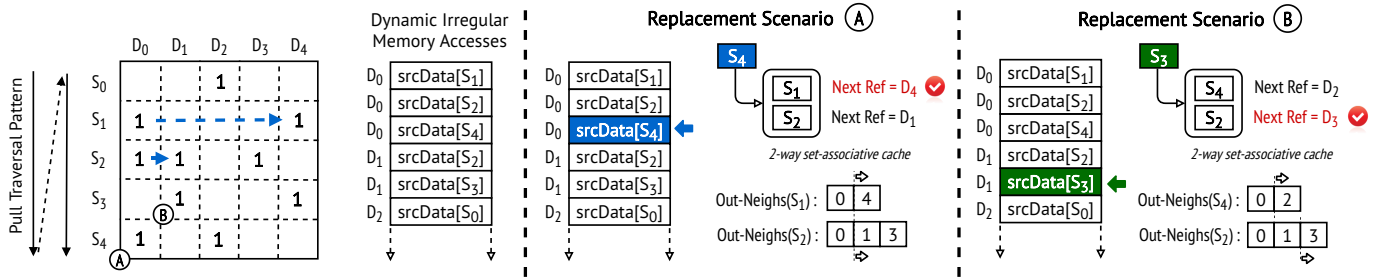


Fig. 3: **Using a graph’s transpose to emulate OPT:** For the sake of simplicity, we assume that only irregular accesses (`srcData` for a pull execution) enter the 2-way cache. In a pull execution using CSC, fast access to outgoing-neighbors (i.e. rows of adjacency matrix) encoded in the transpose (CSR) enables efficient emulation of OPT.

we must evict `srcData[S1]` because its next reuse is further into the future than `srcData[S2]`. Replacement scenario (B) (Figure 3; right panel) considers the execution two accesses later when the pull execution is processing the incoming neighbors of vertex D_1 and needs to cache `srcData[S3]`. The transpose informs that the next re-reference of vertex S_2 is further into the future and OPT evicts `srcData[S2]`.

line involves finding the next reference of each vertex in the line (and reporting the minimum of these values).

Increased Memory Accesses: Computing the next reference of each line requires accessing the transpose of cache-resident vertices involved in replacement. Since the vertices resident in cache can be arbitrary, the neighbor lookups using the Offset Array (OA) and Neighbor Array (NA) of the transpose (Figure 1) incur additional irregular memory accesses that increase cache contention with graph application data.

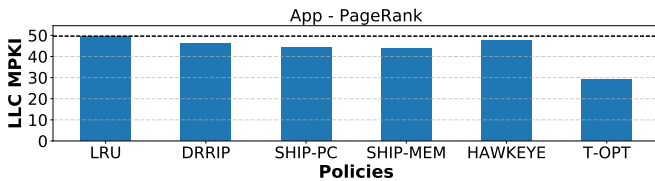


Fig. 4: **Transpose-based Optimal replacement (T-OPT) reduces misses by 1.67x on average compared to LRU.**

We studied the effectiveness of the transpose-based OPT (which we refer to as “T-OPT”) on graph applications by measuring the reduction LLC misses compared to the replacement policies introduced previously. Figure 4 shows that T-OPT reduces LLC MPKI for the PageRank workload. T-OPT significantly reduces LLC MPKI compared to LRU and other policies, achieving a 41% miss rate for PageRank (compared to a 60-70% miss rate for other policies). The main reason for the improvement is that, unlike other replacement policies, T-OPT does not use a heuristic to *guess* the re-reference pattern. Instead, T-OPT uses *precise* information of future reuse encoded in the graph’s transpose to make better replacement decisions.

IV. P-OPT: PRACTICAL OPTIMAL REPLACEMENT

The main contribution of this work is P-OPT, a transpose-based cache replacement policy and architecture implementation that brings virtually all of the benefits of transpose-based OPT (T-OPT) *without* its overheads. P-OPT uses a specialized data structure (called the Rereference Matrix) for fast access of re-reference information available in a graph’s transpose without incurring T-OPT’s overheads.

C. Limitations of Transpose-based Optimal Replacement

The benefits of T-OPT shown in Figure 4 are idealized, ignoring the costs of accessing transpose data to make replacement decisions. A key challenge posed by T-OPT is that naively accessing the transpose imposes an untenable run time overhead and cache footprint.

Increased Run Time: Finding the next reference of a vertex incurs a complexity of $O(|d|)$ where $|d|$ is the out-degree of the vertex. The cost of finding the next reference compounds when the granularity of graph data allows multiple vertices to fit in a cache line. Therefore, finding the next reference of a

A. Reducing the Overheads of T-OPT

Quantizing Re-reference Information: P-OPT reduces the cost of making a replacement decision by *quantizing* the graph’s transpose. By virtue of using the transpose, the range of *next references* for a vertex in T-OPT spans the entire vertex ID space (typically a 32-bit value). We observe that using only a few (e.g. 8) significant bits of the vertex ID space is sufficient to approximate T-OPT. By quantizing next references into fewer, uniform-sized *epochs*, P-OPT reduces the size of next reference information. Figure 5 (left panel) shows how the next references in our example pull execution have been quantized to three epochs (with each epoch spanning two vertices). Quantization reduces the range of next references for each vertex (spanning Epoch-0 to 2), unlike T-OPT where the next reference spans the entire range of vertices in the graph (D_0 to D_4).

Rereference Matrix: A Rereference Matrix is a quantized encoding of a graph’s transpose with dimensions $\{\text{numCacheLines} \times \text{numEpochs}\}$. `numCacheLines` is the number of lines spanned by the irregularly accessed graph data (i.e. `srcData` for the pull execution in Algorithm 1). `numEpochs` is determined by the number of bits required to store quantized next references. Figure 5 shows the Rereference Matrix for the running example. The number of cache lines in

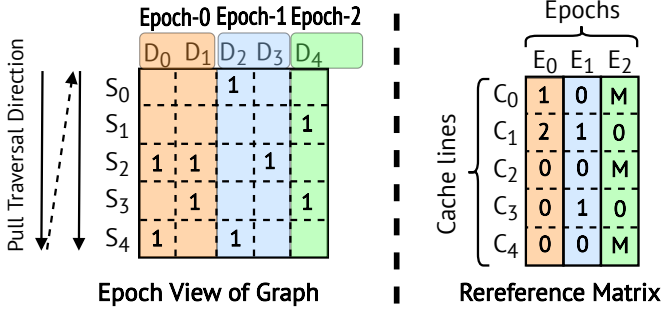


Fig. 5: **Reducing T-OPT overheads using the Rereference Matrix:** Quantizing next references into cachelines and a small number of epochs reduces the cost of accessing next references.

the Rereference Matrix is equal to the number of vertices as a cache line stores a single `srcData` element in Figure 3. Each Rereference Matrix entry stores the *distance* to the epoch of a cache line’s next reuse, which is the difference between the epoch of its next reuse and the current epoch. For example, at Epoch 0, the `srcData[S0]` cache line (C_0) will be accessed in the next epoch; its entry is 1. At epoch 1, `srcData[S0]` is accessed so the C_0 entry is 0, indicating an access in the current epoch. At epoch 2, `srcData[S0]` has no future re-reference and C_0 ’s entry is set to a sentinel value (e.g. maximum value (M) indicating next reference at infinity).

Using the Rereference Matrix, P-OPT approximates T-OPT while avoiding T-OPT’s overheads in two key ways. First, P-OPT stores a next reference per cache line, not per vertex. Instead of traversing the neighbors of *each* vertex in a cache line (as in T-OPT), P-OPT need only look up a single next reference for the cache line in $O(1)$. Second, P-OPT reduces cache contention because only a single epoch (i.e. column) of the Rereference Matrix needs to be resident in the cache at a time. When the execution transitions to a new epoch, P-OPT caches the next epoch of the Rereference Matrix, which contains updated next references for all lines. For a graph of 32 million vertices, 64B cache lines, and 4B per `srcData` element, 8-bit quantization yields a Rereference Matrix column size of 2MB (2M lines * 1B), consuming only a small part of a typical server CPU’s LLC.

B. Tolerating Quantization Loss

Quantizing next references in the Rereference Matrix is *lossy*. Figure 5 shows that the Rereference Matrix encodes the distance to the epoch of a cache line’s next reference. Only inter-epoch reference information is tracked and an execution cannot identify a cache line’s final reference within an epoch, which can lead to incorrect replacement decisions. After a cache line’s final access in an epoch, the zero entry in the Rereference Matrix indicates that the cache line will still be accessed in that epoch, but it will not be. To be more accurate, the next reuse of the cache line should be updated after the final access in an epoch.

P-OPT uses a modified Rereference Matrix entry structure that encodes inter-epoch *and* intra-epoch information. Figure 6 shows a Rereference Matrix entry with 8-bit quantization. Each

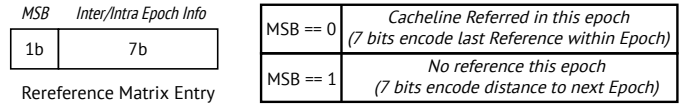


Fig. 6: **Modified Rereference Matrix design to avoid quantization loss:** Tracking intra-epoch information allows P-OPT to better approximate T-OPT.

Rereference Matrix entry’s most significant bit records whether the cache line will be accessed in the epoch. If the cache line is not accessed in the epoch, the MSB is set to one and the remaining lower bits encode the distance (in epochs) to the cache line’s next reference. If the cache line is accessed within the epoch, the MSB is set to zero and the remaining lower bits encode when the final access to the cache line will happen in the epoch. To encode final access, P-OPT divides the vertices spanned in a epoch into equal-sized partitions called “sub-epochs”. The number of sub-epochs in an epoch is equal to the maximum value representable with the remaining lower bits of a Rereference Matrix entry (127 in this example). When the MSB value is zero, the Rereference Matrix entry encodes a cache line’s final access sub-epoch, referring to the partition of vertices within the epoch during which a cache line’s final access occurs.

Algorithm 2 Finding the next reference via Rereference Matrix

```

1: procedure FINDNEXTREF(clineID, currDstID)
2:   epochID ← currDstID/epochSize
3:   currEntry ← RerefMatrix[clineID][epochID]
4:   nextEntry ← RerefMatrix[clineID][epochID + 1]
5:   if currEntry[7] == 1 then
6:     return currEntry[6:0]
7:   else
8:     lastSubEpoch ← currEntry[6:0]
9:     epochStart ← epochID * epochSize
10:    epochOffset ← currDstID - epochStart
11:    currSubEpoch ← epochOffset/subEpochSize
12:    if currSubEpoch ≤ lastSubEpoch then
13:      return 0
14:    else
15:      if nextEntry[7] == 1 then
16:        return 1 + nextEntry[6:0]
17:      else
18:        return 1

```

Pre-computing P-OPT’s modified Rereference Matrix is a low-cost preprocessing step that runs before execution (Section VII-D). During an execution, the modified Rereference Matrix requires some additional computation to find a cache line’s next reference. Algorithm 2 shows the computation to find next references with 8-bit quantization. To find the next reference of cache line (`clineID`) in Epoch `epochID` (which is defined by `currDstID` for pull executions), P-OPT checks the MSB of the cache line’s Rereference Matrix entry for the current epoch (`currEntry`) (Line 5). If the MSB of `currEntry` is set, then the cache line will not be accessed in the current epoch and the lower 7 bits of `currEntry` encode the epoch of the cache line’s next reference (Line 6). However, if the MSB is unset, then the cache line is accessed in the current epoch. The lower 7-bits of `currEntry` track the final

sub-epoch during which the execution accesses the cache line. Using the vertex ID currently being processed (`currDstID` in a pull execution), the computation checks if the execution is beyond the final reference to the cache line in the epoch (Lines 8-12). If execution is yet to reach the sub-epoch of the final reference to the cache line, the computation returns with a rereference distance of 0 (i.e., the cache line will be re-used during the current epoch). However, if execution has passed the sub-epoch of the last reference to the cache line, then the Rereference Matrix entry of the cache line for the *next* epoch (`nextEntry`) encodes the epoch of the cache line’s next reference (Line 4). If the MSB of `nextEntry` is unset, then the cache line is accessed in the next epoch (Line 18) (i.e., a rereference distance of 1). If the MSB of `nextEntry` is set, then `nextEntry`’s low order bits encode the distance to the epoch of the cache line’s next reference (Line 16).

The new Rereference Matrix has two key distinctions. First, finding a cache line’s next reference may require accessing the current *and* next epoch information. This double lookup requires fast access to *two* columns of the Rereference Matrix at each point in time. Second, P-OPT hijacks the MSB of an entry to distinguish between inter-epoch (distance to next epoch) and intra-epoch (final access sub-epoch) tracking which comes at the cost of halving the range of next reference epochs tracked.

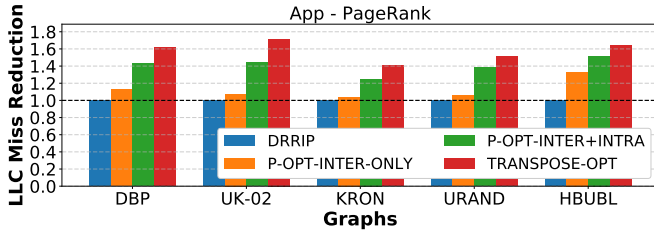


Fig. 7: Tracking inter- and intra-epoch information in the Rereference Matrix allows P-OPT to better approximate T-OPT: The P-OPT designs reserve a portion of the LLC to store Rereference Matrix column(s) whereas T-OPT is an ideal design that incurs no overhead for tracking next references.

We implemented two versions of P-OPT using the different Rereference Matrix designs in our cache simulator and compared their effectiveness to DRRIP and T-OPT. The P-OPT version that uses the first Rereference Matrix design is P-OPT-INTER-ONLY (Figure 5). The P-OPT version that uses the modified Rereference Matrix design (Figure 6) to track both intra- and inter-epoch reuse information is P-OPT-INTER+INTRA. Figure 7 shows the reduction in LLC misses on PageRank achieved by the different policies relative to DRRIP. Both the P-OPT versions achieve miss reduction over DRRIP highlighting that reserving a small portion of the LLC to drive better replacement is a worthwhile tradeoff. Furthermore, P-OPT-INTER+INTRA is able to achieve LLC miss reduction close to the idealized T-OPT that incurs zero overheads to access the graph transpose. We adopt P-OPT-INTER+INTRA as the default P-OPT design for the rest of the paper, due its effectiveness as a close approximation of T-OPT.

V. P-OPT ARCHITECTURE

P-OPT is an architecture that uses Rereference Matrix data stored within a small portion of the LLC to perform better cache replacement. This section first presents a simplified single-core, Uniform Cache Access (UCA) architecture implementation of P-OPT, supporting a single, irregularly-accessed data structure. Later, we show how P-OPT fits in a multi-core, NUCA architecture and supports multiple irregular access streams.

A. Storing Next References in LLC

P-OPT stores the current and next epoch columns of the Rereference Matrix within the LLC to ensure fast access of next reference information during cache replacement. P-OPT uses way-based cache partitioning [27] to reserve the minimum number of LLC ways that are sufficient to store the current and next epoch columns of the Rereference Matrix. Using the default 8-bit quantization, enough ways need to be reserved as to be able to store $2 * numLines * 1B$ where $numLines$ is the number of cache lines spanned by the irregularly-accessed data ($numLines = \frac{numVertices}{elemsPerLine}$). Figure 8 shows some LLC ways reserved for current (orange) and next (blue) epoch columns of the Rereference Matrix. P-OPT organizes the Rereference Matrix columns in LLC for easy access of next reference data. Within a reserved way, consecutive cache-line-sized blocks of a Rereference Matrix column are assigned to consecutive sets. After filling all the sets in one way, P-OPT fills consecutive sets of the next reserved way. P-OPT stores cache lines of the next epoch column of the Rereference Matrix right after the current epoch column (Figure 8). Therefore, P-OPT maintains two hardware registers for each epoch – way-base and set-base – to track the starting positions of the two Rereference Matrix columns within reserved ways of the LLC.

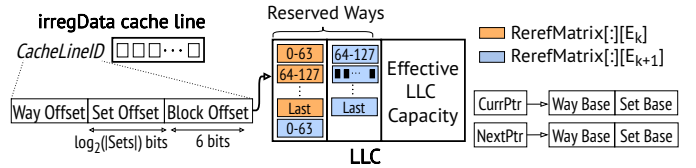


Fig. 8: Organization of Rereference Matrix columns in the LLC: P-OPT pins Rereference Matrix columns in the LLC.

The Rereference Matrix data organization within the LLC allows P-OPT to easily map irregularly accessed data (henceforth referred to as *irregData*) to their corresponding Rereference Matrix entries. The *irregData* array spans multiple cache lines consecutively numbered with an ID from 0 to $numLines - 1$. P-OPT uses the *irregData* cache line ID to find the unique location of the Rereference Matrix entry within the LLC. With P-OPT’s default 8-bit quantization, a typical cache line of 64B contains 64 entries of a Rereference Matrix column. The low 6 bits ($\log_2(64)$) of the cache line ID provides an offset within a cache line of Rereference Matrix data. The next $\log_2(numSets)$ bits of the cache line ID provides a set offset and the remaining cache line ID bits provide a way offset. The final set and way location of a Rereference Matrix entry for an *irregData* cache line is determined by adding

the set and way offsets to the `set-base` and `way-base` registers of the required epoch³.

B. Identifying Irregular Data

P-OPT needs to access the Rereference Matrix data only for `irregData` lines (since all other accesses are streaming in Algorithm 1). P-OPT maintains two registers – `irreg_base` and `irreg_bound` – to track the address range of a graph kernel’s `irregData` (Figure 9). During cache replacement, P-OPT compares the address in the tag portion of each way in the eviction set against `irreg_base` and `irreg_bound` registers to determine if the way contains an `irregData` cache line. The `irreg_base` and `irreg_bound` registers must track physical addresses as P-OPT reasonably assumes that LLC is a Physically-Indexed Physically-Tagged (PIPT) cache. P-OPT sidesteps the complexity of address translation by requiring that the entire `irregData` array fits in a single 1GB Huge Page [45]. By ensuring that all `irregData` elements map to a single (huge) page, P-OPT guarantees that the range of physical addresses associated with `irregData` array lie strictly within the range of physical addresses represented by `irreg_base` and `irreg_bound`. Software configures the two registers once at the start of execution using memory-mapped registers. Allocating `irregData` using a 1GB Huge Page uses widely-available system support [45] and allows processing sufficiently large graphs with up to 256 million vertices (assuming 4B per `irregData` element). To support larger graphs, P-OPT could incorporate prior proposals [26], [41] that provide system support to ensure identity mapping between physical and virtual addresses for important data structures (such as `irregData`).

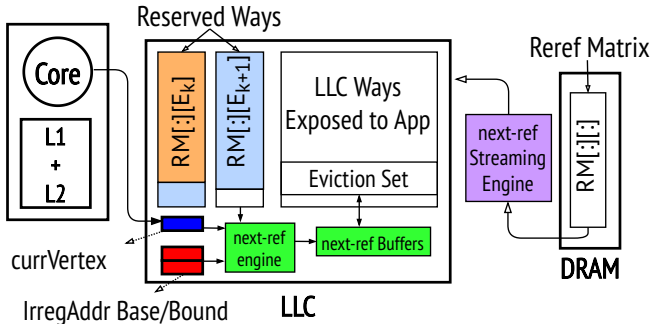


Fig. 9: **Architecture extensions required for P-OPT:** Components added to a baseline architecture are shown in color.

C. Finding a Replacement Candidate

P-OPT maintains a small number of buffers (called `next-ref` buffers) at the LLC to keep track of the next references of each way in the eviction set (Figure 9). A `next-ref` buffer tracks an 8-bit next reference entry for each (non-reserved) way in the LLC. At the start of a cache replacement, P-OPT assigns a free `next-ref` buffer

³For non power-of-two number of sets: $WayOffset = \frac{(cachelineID \gg 6)}{numSets}$ and $SetOffset = (cachelineID \gg 6) \% numSets$.

to the eviction set. To find a replacement candidate, P-OPT uses a Finite State Machine (called the `next-ref` engine) to compute the next reference of each non-reserved way in the eviction set and the `next-ref` engine stores next references in the corresponding entry of the `next-ref` buffer. The `next-ref` engine skips computing next references for the ways reserved for Rereference Matrix columns because P-OPT never evicts Rereference Matrix data. Among non-reserved ways, the `next-ref` engine uses the `irreg_base` and `irreg_bound` registers to first search for a way that does not contain `irregData` (i.e. contains streaming data). The `next-ref` engine reports the first way in the eviction set containing streaming data as the replacement candidate. If all ways in the eviction set contain `irregData`, then the `next-ref` engine runs P-OPT’s next reference computation (Algorithm 2) for each way. The next reference computation of an `irregData` cache line requires the cache line ID of the `irregData` and the vertex ID currently being processed in the outer loop of a graph application (e.g. `dstID` for pull executions). The cache line ID of the `irregData` line is determined by the `next-ref` engine using simple address arithmetic ($cacheLineID = \frac{(addr - irreg_base)}{64}$). The current destination being processed by a pull execution is tracked in a `currVertex` register located at the LLC (Figure 9). The `currVertex` register is updated by a new `update_index` instruction which allows software to pass graph application information (i.e. current vertex) to the LLC. The constants used in finding next reference of a cache line (epoch and sub-epoch size) are stored in special memory mapped registers co-located at the LLC and are configured once before the execution. (For 8-bit quantization, $EpochSize = \lceil numVertices / 256 \rceil$ and $SubEpochSize = \lceil EpochSize / 127 \rceil$). With all the necessary information (cache line ID, `currDstID`, constants), the `next-ref` engine computes next references by accessing Rereference Matrix entries for each `irregData` line in the eviction set; storing the computed next references in the `next-ref` buffer. The `next-ref` engine then searches the `next-ref` buffer to find the way with the largest (i.e., furthest in future) next reference value, settling a tie using a baseline replacement policy (P-OPT uses DRRIP). The `next-ref` engine starts its computations immediately after an LLC miss, overlapping the replacement candidate search with the fetch from DRAM. A P-OPT implementation could pipeline computing a next reference from a way’s Rereference Matrix entry with fetching the Rereference Matrix entry for the next way. DRAM latency hides the latency of sequentially computing next references for each way in the eviction set, based on LLC cycle times from CACTI [42] (listed in Table I).

D. Streaming Rereference Matrix columns into the LLC

P-OPT stores current and next epoch columns of the Rereference Matrix in the LLC. At an epoch boundary, P-OPT streams in a new next epoch column and treats the previous next epoch column as the new current epoch column. To transfer Rereference Matrix entries from memory to LLC, P-OPT uses

a dedicated hardware unit called the streaming engine similar to existing commodity data streaming hardware (Intel DDIO [16], [22] allows ethernet controllers to directly write data into an LLC partition). The programmer invokes the streaming engine at every epoch boundary using a new `stream_nextrefs` instruction. The instruction swaps pointers to the current and next epoch (Figure 8) and streams in the next epoch column of the Rereference Matrix into the LLC locations pointed by `set-base` and `way-base` for the next epoch. Graph applications need to be restructured slightly to ensure that the streaming engine is invoked *between* two epochs (to ensure that all epochs operate on accurate Rereference Matrix data). Doing so does not impose a performance penalty because the streaming engine is guaranteed peak DRAM bandwidth to transfer Rereference Matrix data between epochs. Moreover, streaming engine latency is not a performance problem because epoch boundaries are infrequent.

E. Supporting NUCA Last Level Caches

While our discussion so far assumed a monolithic, UCA LLC, P-OPT is also efficient for the increasingly common NUCA LLCs [33]. We consider Static NUCA (S-NUCA) [37] with addresses statically partitioned across physically-distributed banks. The key NUCA challenge is to ensure that Rereference Matrix accesses during replacement are always bank-local. A typical S-NUCA system stripes consecutive cache lines across banks ($\text{bankID} = (\text{addr} \gg 6) \% \text{numBanks}$). Striping both Rereference Matrix and `irregData` cache line across banks cannot guarantee bank-local accesses to Rereference Matrix data at replacement time because a single cache line of Rereference Matrix data contains next references for 64 `irregData` cachelines (Figure 8). Ensuring bank-local Rereference Matrix accesses requires that for every Rereference Matrix cache line mapped to a bank, *all* 64 of its corresponding `irregData` cache lines must also map to the same bank.

P-OPT uses a modified mapping to distribute Rereference Matrix entries and `irregData` across NUCA banks. If P-OPT stripes Rereference Matrix cache lines across banks, the system must interleave `irregData` in blocks of 64 cache lines across NUCA banks (i.e. $\text{bankID} = (\text{addr} \gg (6+6)) \% \text{numBanks}$). P-OPT implements this modified mapping policy for `irregData` using Reactive-NUCA [25] support. Reactive-NUCA allows different address mapping policies for different pages of data through simple hardware and OS mechanisms. P-OPT uses the modified mapping policy only for `irregData` (which P-OPT assigns to a single 1GB Huge Page) and uses the default, cache line striped S-NUCA policy for all other data (including Rereference Matrix data).

P-OPT needs minor hardware changes for NUCA LLCs. First, P-OPT needs a per-bank copy of the registers used to track Rereference Matrix columns (`set-base`, `way-base`, `currPtr`, `nextPtr` in Figure 8). Second, the `irreg_base`, `irreg_bound`, and `currVertex` registers are global values that need to be shared or replicated across NUCA banks. Last, P-OPT needs per-bank `next-ref` engine and `next-ref`

buffers, because multiple banks may be concurrently evicting cache lines.

F. Generalizing P-OPT

With simple extensions, P-OPT supports multi-threading, multiple irregularly-accessed datastreams, and context switches. **Supporting Parallel Execution:** P-OPT supports parallel multi-threaded execution. In a multi-threaded execution, multiple active vertices are being traversed at a time (i.e., a unique `currDstId` for each thread) and P-OPT needs to select one of the active vertices for next reference computation (Algorithm 2; Lines 8-12). Thanks to pervasive, existing load balancing support in graph processing frameworks, different threads already process vertices in a narrow range. To guarantee that all threads always process vertices in the same epoch, P-OPT requires slight modification of the application to execute epochs serially (vertices within an epoch are executed in parallel). Executing epochs serially allows P-OPT to share the same Rereference Matrix columns across all threads. Due to the relatively small number of epochs (256 in the default P-OPT configuration) each epoch consists of a large number of vertices and restricting parallelism to only within epochs does not significantly impact performance. We empirically determined that assigning `currDstID` to be the vertex being processed by a software-designated main thread is an effective policy; providing similar LLC miss rates with P-OPT and T-OPT for multi-threaded graph applications as for serial executions.

Handling Multiple Irregular Data Streams: P-OPT can support multiple irregular data structures using three architecture changes. First, P-OPT holds a separate Rereference Matrix for each irregular data structure (only if the irregular data structures span different number of cache lines, otherwise a single Rereference Matrix can be shared). Second, P-OPT reserves the minimum number of ways in the LLC to hold the Rereference Matrix data for all the different irregular data structures. The system maintains a separate `set-base` and `way-base` register for each irregular data structure. Third, P-OPT maintains an `irreg_base` and `irreg_bound` register for each irregular data structure to use the right Rereference Matrix data corresponding to each data structure. We observe that tracking two irregular data structures – `frontier` and `srcData/dstData` (for pull/push executions) – covers many important graph applications. If an application has more irregular data streams (which is rare), a programmer could restructure the code to use an Array-of-Structures (AoS) format, combining all irregular accesses to a single array.

Virtualization: The Rereference Matrix in P-OPT only tracks reuse among graph application data. If applications share LLC, P-OPT may unfairly prefer caching graph data over other data. To remain fair, we assume per-process way-partitioning (i.e., via Intel CAT [27]) and that P-OPT only replaces data in the graph-process-designated LLC ways. P-OPT supports context switches, by saving its registers (`set-base`, `way-base`, `irreg_base`, `irreg_bound`, `currVertex`) with the process context. On resumption, P-OPT invokes the streaming engine to refetch Rereference Matrix contents

into reserved LLC ways. Static partitioning of the cache ensures that P-OPT does not monopolize the shared LLC in the presence of multiple co-running applications. Alternatively, P-OPT can be synergistic with existing *application-aware* shared cache management policies [29], [30], [48]. Studying these interactions are beyond the scope of this paper and left for future work.

G. Implementation Complexity

P-OPT has low architectural complexity. P-OPT stores the replacement metadata (Rereference Matrix columns) within the LLC and, hence, does not require *additional* storage for tracking next references. P-OPT adds `next-ref` buffers to temporarily store next references during replacement. The size of `next-ref` buffers state is bounded by the maximum cache-level parallelism at the LLC. For example, an 8-core architecture supporting 10 outstanding L1 misses (i.e. 10 L1 MSHRs) allows 80 concurrent LLC accesses. Each `next-ref` buffer tracks 1B per LLC way. For a 16-way LLC, each `next-ref` buffer tracks 16B of information. Therefore, a worst case maximum size for `next-ref` buffers is 1.25KB (80 * 16B). In practise, fewer `next-ref` buffers would be sufficient because graph applications lack memory-level parallelism [9], [56]. The `next-ref` engine is a simple FSM that only needs support for integer division and basic bit manipulation.

VI. EXPERIMENTAL SETUP

Platform details: We use the Sniper [13] simulator to measure performance, using its default Becton microarchitecture configuration (which is based on Intel Nehalem). Table I describes our baseline multi-core architecture, with cache timing from CACTI [42]. We disable prefetching in our study because prior work [8] observed that conventional stream prefetchers are ill-suited to handle the irregular memory accesses dominating graph applications. We made several improvements to sniper to better model P-OPT’s performance effects. We ensure that a graph application in P-OPT sees reduced effective Last Level Cache capacity and apply P-OPT’s modified S-NUCA policy for irregular data structures. We model contention between demand accesses and Rereference Matrix accesses within the NUCA banks. When reporting P-OPT performance numbers, we also account for the latency of the `streaming` engine to bring Rereference Matrix columns into the LLC before every epoch. To faithfully model this stop-the-world event, we slightly modify parallel graph applications to process epochs serially and use parallelism only within epochs (only P-OPT executions use the modified version while all the other policies operate on unmodified versions of parallel graph applications).

For faster design space exploration, we built a Pin [36]-based cache simulator⁴ to model the cache hierarchy in Table I and to evaluate various LLC replacement policies. The P-OPT and T-OPT results reported earlier in the paper came from

Cores	8 OoO-cores, 2.266GHz, 4-wide issue, 128-entry ROB, Pentium M Branch Predictor
L1(D/I)	32KB, 8-way set associative, Bit-PLRU replacement policy, Load-to-use = 3 cycles
L2	256KB, 8-way set associative, Bit-PLRU replacement policy, Load-to-use = 8 cycles
LLC	3MB/core, 16-way set associative, DRRIP replacement [30], Load-to-use = 21 cycles (local NUCA bank), NUCA bank cycle time = 7 cycles
NoC	Ring interconnect, 2 cycles hop-latency, 64 bits/cycle per-direction link B/W, MESI coherence
DRAM	173ns base access latency

TABLE I: **Simulation parameters**

this cache simulator. We validated our cache simulator against Sniper (LLC statistics from the cache simulator were within 5% of Sniper’s values). Unless specified, the cache-only simulator models serial execution of graph kernels to avoid thread-scheduling noise in Pin. The Sniper simulations evaluate parallel graph applications.

Workloads: We use five graph applications from GAP [9] and Ligra [50]. To avoid framework overheads, we re-wrote Ligra benchmarks as stand-alone applications (which yielded an average speedup of 1.55x over the original implementation).

	PR [9]	CC [9]	PR- δ [50]	Radii [50]	MIS [50]
irregData ElemSz	4B	4B	8B & 1bit	8B & 1bit	4B & 1bit
Execution style	Pull-Only	Push-Only	Pull-Mostly	Pull-Mostly	Pull-Mostly
Transpose	CSR	CSC	CSR	CSR	CSR
Uses frontier	N	N	Y	Y	Y

TABLE II: **Applications**

These applications have a diverse set of graph access patterns and properties (Table II). PageRank (PR) iteratively updates per-vertex ranks until convergence. Connected Components (CC) applies the Shiloach-Vishkin algorithm to compute largest connected components. PageRank-delta (PR- δ) is a frontier-based version of PageRank that only updates vertices that have not converged. Radii is a frontier-based application using concurrent BFS traversals to approximate a graph’s radius. Maximal Independent Set (MIS) iteratively processes vertex subsets to estimate the maximal independent set. PageRank-delta, Radii, and Maximal Independent Set use direction-switching [11] and frontiers encoded as bit-vectors. To reduce simulation time, we simulate one PageRank iteration (it shows no performance variation across iterations). For other applications, we use *iteration sampling* like prior work [40], [41] and simulate a subset of pull iterations in detail.

Input Graphs: We run our analyses on the graphs listed in Table III. The graphs are diverse in size and degree-distributions (power-law, community, normal, bounded-degree). We do not simulate Radii on HBUBL because its high diameter causes Radii to never switch to a pull iteration.

	DBP	UK-02	KRON	URAND	HBUBL
# Vertices (in M)	18.27	18.52	33.55M	33.55M	21.20
# Edges (in M)	136.53	292.24	133.51	134.22	63.58

TABLE III: **Input Graphs:** All graphs exceed the LLC size.

⁴<https://github.com/CMUAbstract/POPT-CacheSim-HPCA21>

VII. EVALUATION

We evaluate P-OPT, showing significant performance and locality improvements across a range of workloads and compare P-OPT to prior work on efficient caching for graph workloads.

A. P-OPT Improves Performance

Figure 10 shows performance and cache locality improvements achieved by P-OPT and an idealized T-OPT compared to the LRU and DRRIP replacement policies. As discussed in Section II-B, the state-of-the-art DRRIP replacement policy offers an average performance improvement of only 9% relative to the simple LRU policy due to its inability to capture graph application-specific reuse patterns. P-OPT outperforms DRRIP across the board, with average speedup of 22% and LLC miss reduction of 24%. Furthermore, P-OPT’s mean speedup is within 12% of the ideal speedup (with T-OPT).

Figure 10 shows four key findings. First, P-OPT is effective for applications with dense frontiers (PageRank and Connected Components) *and* sparse frontiers (Radii, Maximal Independent Set, and PageRank-delta). P-OPT offers higher speedup for PageRank and Connected Components because P-OPT needs to only store the Rereference Matrix data for a single irregular data structure (other applications need Rereference Matrix data for `srcData` and `frontier`). Second, P-OPT improves performance and locality for pull and push executions. Third, P-OPT provides benefits for a diverse set of graphs. KRON is one exception with both P-OPT and T-OPT offering slightly smaller improvement over DRRIP. These synthetic KRON graphs have highly skewed degree distributions. The more skewed the distribution, the more likely it is for hub vertices to hit by chance in cache; DRRIP has miss rate of 40% for KRON compared to a miss rate of 70% for other graphs. Finally, P-OPT’s speedup compared to DRRIP (22%) is significantly higher than state-of-the-art policies like Hawkeye and SHiP. Hawkeye and SHiP report average speedups of just 2.54% and 1.78% over DRRIP [28], [53]. While Hawkeye and SHiP provide small benefits, P-OPT leverages graph structure and offers a significant improvement over DRRIP.

B. P-OPT Scales with Graph Size

P-OPT remains performant as graph size increases. P-OPT stores the current and next epoch columns of the Rereference Matrix in LLC (Figure 9). Larger graphs need to reserve more LLC ways to store Rereference Matrix columns because the irregular data spans more cache lines. We evaluate a P-OPT variant, P-OPT-Single-Epoch (P-OPT-SE), that computes next references using only the current epoch column of the Rereference Matrix. P-OPT-SE encodes information about the next epoch within the current epoch column by repurposing the *second* most significant bit of an entry to track if the cache line is accessed in the next epoch (Figure 6). P-OPT-SE stores only the current epoch column in LLC. However, the reduced cache footprint in P-OPT-SE comes at the expense of reduced next reference quality. Down two bits per entry, the range of

next references tracked in P-OPT-SE is halved from 128 to 64 — P-OPT-SE is forced to use coarser quantization.

In Figure 11, we compare P-OPT-SE (one column, two reserved bits) to P-OPT (two columns, one reserved bit) for PageRank on a set of graphs. With fewer than 32 million vertices, P-OPT has better LLC locality. For these graphs, P-OPT reserves fewer than 3 ways of 16 and the benefit of better replacement information (i.e. current *and* next epoch) overshadows the reduction in effective LLC capacity. However, in larger graphs, P-OPT-SE has better locality because of P-OPT’s high reduction in effective LLC capacity. The result highlights the tension between next reference quantization and the effective LLC capacity; to improve upon P-OPT’s performance gains, future solutions must reduce the metadata footprint without significantly compromising the quality of replacement metadata.

C. P-OPT compared to prior optimizations

We compared P-OPT to prior work on locality optimizations for graph analytics.

1) *Graph-agnostic improvements with P-OPT*: Like prior work [20], [40], P-OPT observes that cache locality is key to improving graph processing performance. Unlike prior work, P-OPT is graph-agnostic, not reliant on specific structure or vertex ordering of a graph.

GRASP [20] is a replacement policy for graphs with very skewed degree distributions. GRASP expects a pre-processed input vertex array and GRASP uses Degree-Based Grouping (DBG) [19] to order vertices. We reordered our graphs using the author’s DBG implementation and implemented GRASP in our cache simulator, based on code from the authors. Figure 12(a) shows locality improvements from GRASP and P-OPT for PageRank on DBG-ordered graphs. P-OPT outperforms GRASP in three ways. First, GRASP works well for graphs with skewed degree distributions, but is less effective for other inputs; the best result for GRASP is for the highly skewed GPL graph. P-OPT is agnostic to graph structure, offering consistent improvement. Second, even for skewed graphs, P-OPT has higher LLC miss reduction than GRASP because GRASP is heuristic-based, assuming vertices with similar degrees have similar reuse. P-OPT, instead, approximates ideal next reference values capturing dynamically varied patterns of reuse. Last, GRASP requires the input graph to be reordered (using DBG) whereas P-OPT is applicable across any vertex ordering.

HATS-BDFS [40] is a dynamic vertex-scheduling architecture that improves graph cache locality. HATS runs hardware Bounded Depth First Search (BDFS) to schedule vertices, yielding locality improvements in graphs with community structure [34]. We implemented in our cache simulator an aggressive HATS-BDFS that assumes no overhead for BDFS vertex scheduling. Figure 12(b) compares P-OPT on the standard vertex schedule (“Vertex Ordered” per HATS [40]) against HATS-BDFS. The data shows that HATS-BDFS’s improvements are sensitive to graph structure. For its target use-cases (i.e., community-structured graphs – UK-02 and ARAB), BDFS offers locality improvements, even outpacing T-OPT

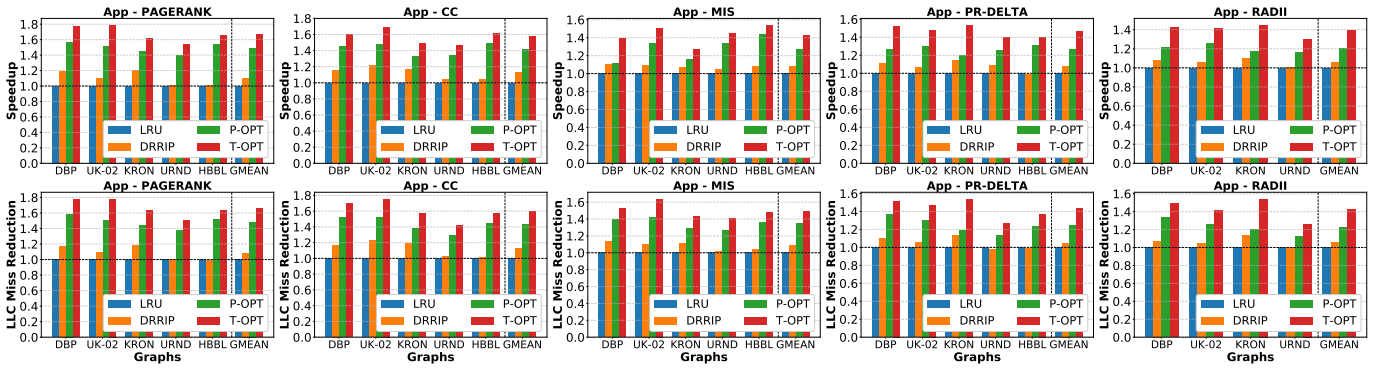


Fig. 10: **Speedups and LLC miss reductions with P-OPT and T-OPT:** *The T-OPT results represent an upper bound on performance/locality because T-OPT makes optimal replacement decisions using precise re-reference information without incurring any cost for accessing metadata. P-OPT is able to achieve performance close to T-OPT by quantizing the re-reference information and reserving a small portion of the LLC to store the (quantized) replacement metadata.*

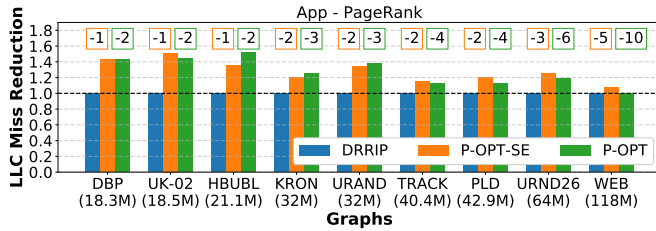
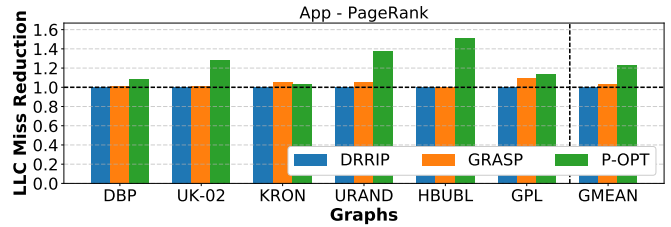


Fig. 11: **LLC miss reductions with P-OPT and P-OPT-SE:** *Boxes above bar groups indicate the number of LLC ways reserved to store next references. Graphs are listed in increasing order of number of vertices.*

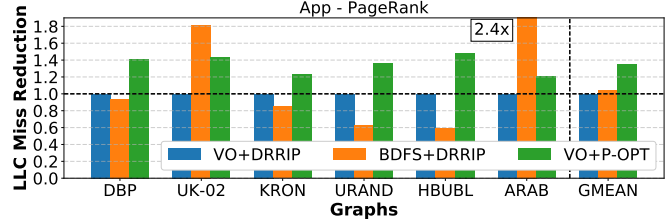
because BFS improves locality at all cache levels. However, for graphs without community structure (even power-law graphs such as DBP and KRON), BFS *increases* LLC misses. In contrast, P-OPT offers consistent LLC locality improvements, leading to a higher mean LLC miss reduction compared to HATS-BDFS.

2) *Optimizations Complementary to P-OPT:* P-OPT complements software graph locality optimizations – CSR-Segmenting [57] and Propagation Blocking [10].

CSR-Segmenting [57] is 1D tiling for graphs. We applied the CSR-segmenting optimization to the PageRank application to study how P-OPT and CSR-segmenting interact. Figure 13 shows relative cache performance of DRRIP and P-OPT for two large graphs as tile count increases, with results normalized to untilted DRRIP execution. Tiling improves P-OPT’s miss reduction over DRRIP. P-OPT benefits because tiling reduces the address range of random access allowing P-OPT to store only a tile of a Rereference Matrix column in LLC. Compared to DRRIP, P-OPT provides the same miss reduction with fewer tiles: For URAND64, P-OPT with two tiles has the same LLC miss reduction as DRRIP with 10 tiles. Thus, as tiling improves P-OPT, P-OPT amplifies the efficiency of tiling by reducing required tile count and, hence, the preprocessing costs (Preprocessing cost scales with tile count because each tile requires building a CSR).



(a) P-OPT compared to GRASP



(b) P-OPT compared to HATS-BDFS

Fig. 12: **P-OPT offers graph-agnostic improvements:** *In contrast to prior locality optimizations for graph workloads, P-OPT’s benefits are not restricted to specific structural properties or vertex orderings of input graphs.*

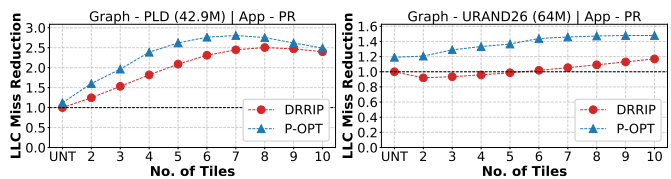


Fig. 13: **P-OPT and Tiling are mutually-enabling optimizations:** *Tiling allows P-OPT to reserve fewer LLC ways while P-OPT can reduce the preprocessing cost of tiling.*

Propagation Blocking (PB) [10] is a software graph locality optimization. Recent work [41] showed further improvements at the expense of additional hardware support for PB. PHI [41] improves graph applications with commutative vertex updates by aggregating updates in-cache to reduce DRAM traffic.

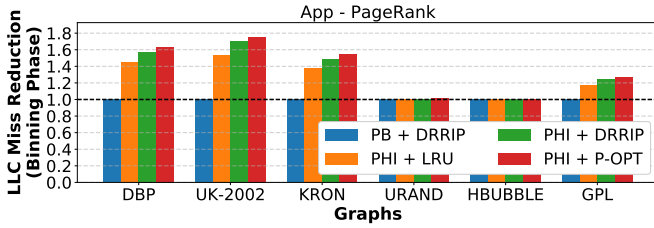


Fig. 14: **P-OPT is complementary to PHI and PB**

PHI is complementary to replacement and provides higher benefits with better replacement policies (e.g., DRRIP over LRU). We studied the PB and PHI interactions with P-OPT by implementing PHI in our cache simulator (targeting a multi-core setup). Figure 14 shows results for four setups on PB’s dominant execution phase (Binning). The data show that PHI improves locality over software PB (PB+DRRIP) and that PHI’s effectiveness improves with better replacement. The original PHI work did not evaluate on non-power-law graphs (e.g., URAND and HBUBL), which have worse private cache locality, impeding PHI’s update aggregation, and leading to little benefit. P-OPT, in contrast, is effective for these graphs, even when PHI is not.

D. Sensitivity Studies

Sensitivity to quantization level: We assumed 8-bit next reference entries up to this point. Figure 15 shows P-OPT’s performance with 4-bit, 8-bit, and 16-bit quantization in the Rereference Matrix. This dataset omits the costs of storing Rereference Matrix columns in LLC, reporting limit-case locality improvements for a given quantization level. Due to quantization, multiple lines might have the same rereference value during replacement leading to a tie (as described in Section V-C, ties are resolved by a baseline replacement policy; our evaluation assumes DRRIP). On average, we observe that for P-OPT with 4b, 8b, and 16b quantization of rereferences, 41%, 12%, and 0% of all LLC replacements respectively result in a tie. The already low percentage of replacement ties at 8b quantization explains why P-OPT sees little benefit with higher precision.

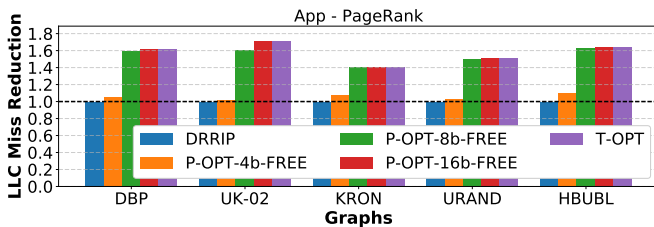


Fig. 15: **P-OPT at different levels of quantization:** With 8-bit quantization, P-OPT is able to provide a close approximation of the ideal (T-OPT).

Sensitivity to LLC parameters: We measured P-OPT’s sensitivity to LLC capacity and associativity. Figure 16 shows data for PageRank across all graphs. The benefit offered by P-OPT over DRRIP increases with LLC capacity because the

fraction of LLC consumed for Rereference Matrix columns reduces. P-OPT also offers higher miss reduction with higher LLC associativity. As associativity increases, P-OPT has more options for replacement and makes a better choice by considering next references of all ways in the eviction set.

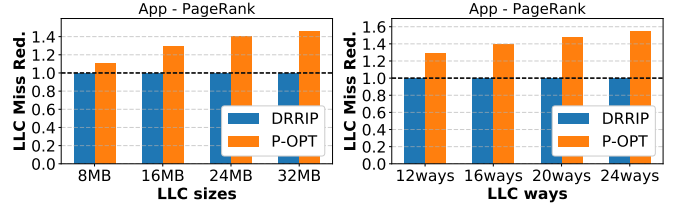


Fig. 16: **Sensitivity to LLC size and associativity:** P-OPT’s effectiveness increases with LLC size and associativity.

Preprocessing cost of P-OPT: P-OPT uses a Rereference Matrix to guide cache replacement and the Rereference Matrix is built from the transpose. The main performance results (Figure 10) omitted preprocessing costs because the Rereference Matrix is algorithm agnostic and needs to be created only once for a graph. We experimentally determined that building the Rereference Matrix imposes low overhead on a real 14-core Intel Xeon processor (we used 8 threads and Intel CAT to set the LLC to 24.5MB to mirror the simulated architecture in Table I). Table IV shows time spent building the Rereference Matrix compared to a baseline execution of PageRank. On average, constructing the Rereference Matrix accounts for 19.8% of PageRank runtime⁵. Figure 10 shows that without the Rereference Matrix construction cost, P-OPT offers a mean performance improvement of 36% over DRRIP⁶ for PageRank. Since the Rereference Matrix is algorithm agnostic, the preprocessing cost of P-OPT can be easily amortized by reusing the Rereference Matrix across multiple applications running on the same graph. However, even in scenarios where the Rereference Matrix construction cost cannot be amortized (e.g., single-shot graph analytics), the relatively small cost of constructing the Rereference Matrix allows P-OPT to provide a net speedup even after including the preprocessing cost.

	DBP	UK-02	KRON	URND	HBUBL
POPT Preprocessing Time	0.99s	1.25s	1.59s	1.77s	0.92s
PageRank Execution Time	8.83s	24.64s	4.84s	11.06s	0.89s

TABLE IV: **Relative preprocessing cost for P-OPT**

VIII. RELATED WORK

We compared P-OPT to the most closely related works in Sections II and VII. We include additional comparisons spanning three areas – cache replacement, irregular-data prefetching, and custom architectures for graph processing.

Replacement Policies: Hawkeye and SHiP outperform many classes of replacement policies [28], [53]. One such class

⁵HBUBL is an exception because PageRank converges unusually quickly (3 iterations)

⁶Server-class processors have been shown to use a variant of DRRIP [52])

of policies are policies like SDBP [32] and Leeway [21] that perform Dead-Block Prediction (DBP) (i.e. find cache lines that will receive no further accesses). P-OPT can more accurately identify dead lines because it tracks next references of irregular lines (Indeed, P-OPT outperforms Hawkeye and GRASP which were shown to be better than SDBP and Leeway respectively). By using close approximation of precise next references (Section VII-D), P-OPT is expected to outperform heuristic-based reuse distance predictions [17], [31].

Irregular Data Prefetching: IMP [54], HATS-VO [40], and DROPLET [8] are recent prefetchers that were designed primarily to handle irregular accesses in graph processing and sparse linear algebra applications. All three schemes are effective at reducing latency of irregular accesses but not necessarily memory traffic. P-OPT reduces memory traffic through better LLC locality, making better use of the available DRAM bandwidth. We note that next references in a graph's transpose could also be used for timely prefetching of irregular data. We leave the exploration of new prefetching mechanisms derived from the Rereference Matrix and the interplay of P-OPT with hardware-based [4], [8], [40], [54] or software-based [5] irregular prefetching for future work.

Custom architectures for graph processing: Minnow [56] is an architecture for efficient worklist management and optimizes worklist-based graph applications [47]. OMEGA [3] is a scratchpad-based architecture for graph processing on power-law input graphs. Custom accelerators [24], [43] have been proposed that optimize graph framework operations to accelerate common sub-computations across all applications using the framework. P-OPT observes the pervasiveness of poor cache locality in graph applications and leverages the readily-available transpose to guide better cache replacement. SpArch [59], an SpGeMM accelerator, proposed dedicated hardware to run ahead (upto a fixed depth) and compute next references for irregular data. P-OPT also uses next references for better replacement but relies on the transpose to more efficiently access next references.

IX. CONCLUSIONS

The main insight of this work is that a graph's transpose succinctly encodes the next reference information of all graph data, which enables transpose-driven Belady's MIN replacement policy (T-OPT) for graph analytics workloads. We present P-OPT, a practical implementation of T-OPT, that exploits quantization to efficiently store and access the next reference information to make better replacement decisions. Evaluations across a range of workloads and input graphs show that P-OPT provides significant performance and locality improvements compared to prior state-of-the-art replacement policies and locality optimizations for graph analytics.

ACKNOWLEDGMENT

Thanks to the anonymous reviewers for their helpful feedback and to Nathan Beckmann for his feedback on an early version of the manuscript. Special thanks to Hailang Liou for his valuable contributions during the early phases of this project

and help in developing the core mechanisms of P-OPT. This work was supported in part by National Science Foundation 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] "Cache Replacement Championship," <https://crc2.ece.tamu.edu/>, 2020, [Online]; accessed 31-July-2020].
- [3] 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)*. IEEE, 2018, pp. 134–145.
- [4] S. Ainsworth and T. M. Jones, "Graph prefetching using data structure knowledge," in *Proceedings of the 2016 International Conference on Supercomputing*, 2016, pp. 1–11.
- [5] S. Ainsworth and T. M. Jones, "Software prefetching for indirect memory accesses," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2017, pp. 305–317.
- [6] 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)*. IEEE, 2018, pp. 203–214.
- [7] 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>
- [8] 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.
- [9] S. Beamer, K. Asanovic, and D. Patterson, "Locality exists in graph processing: Workload characterization on an ivy bridge server," in *2015 IEEE International Symposium on Workload Characterization*, 2015, pp. 56–65.
- [10] S. Beamer, K. Asanović, and D. Patterson, "Reducing pagerank communication via propagation blocking," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017, pp. 820–831.
- [11] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," *Scientific Programming*, vol. 21, no. 3-4, pp. 137–148, 2013.
- [12] T. D. Bui, S. Ravi, and V. Ramavajjala, "Neural graph learning: Training neural networks using graphs," in *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, ser. WSDM '18. New York, NY, USA: ACM, 2018, pp. 64–71. [Online]. Available: <http://doi.acm.org/10.1145/3159652.3159731>
- [13] 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*, 2011, pp. 1–12.
- [14] M. Daga and J. L. Greathouse, "Structural agnostic spmv: Adapting csr-adaptive for irregular matrices," in *2015 IEEE 22nd International conference on high performance computing (HiPC)*. IEEE, 2015, pp. 64–74.
- [15] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.
- [16] I. D. Direct, "I/o technology (intel ddi) a primer," 2012.
- [17] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum, "Improving cache management policies using dynamic reuse distances," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2012, pp. 389–400.
- [18] D. Easley and J. Kleinberg, *Networks, crowds, and markets: Reasoning about a highly connected world*. Cambridge University Press, 2010.
- [19] P. Faldu, J. Diamond, and B. Grot, "A closer look at lightweight graph reordering," in *2019 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2019, pp. 1–13.

- [20] P. Faldu, J. Diamond, and B. Grot, "Domain-specialized cache management for graph analytics," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 234–248.
- [21] P. Faldu and B. Grot, "Leeway: Addressing variability in dead-block prediction for last-level caches," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2017, pp. 180–193.
- [22] A. Farshin, A. Roozbeh, G. Q. Maguire Jr, and D. Kostić, "Make the most out of last level cache in intel processors," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–17.
- [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] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.
- [25] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive nuca: Near-optimal block placement and replication in distributed caches," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, p. 184–195, Jun. 2009. [Online]. Available: <https://doi.org/10.1145/1555815.1555779>
- [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] C. Intel, "Improving real-time performance by utilizing cache allocation technology," *Intel Corporation*, April, 2015.
- [28] A. Jain and C. Lin, "Back to the future: leveraging belady's algorithm for improved cache replacement," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 78–89.
- [29] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely Jr, and J. Emer, "Adaptive insertion policies for managing shared caches," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 208–219.
- [30] 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.
- [31] G. Keramidas, P. Petoumenos, and S. Kaxiras, "Cache replacement based on reuse-distance prediction," in *2007 25th International Conference on Computer Design*. IEEE, 2007, pp. 245–250.
- [32] S. M. Khan, Y. Tian, and D. A. Jimenez, "Sampling dead block prediction for last-level caches," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2010, pp. 175–186.
- [33] C. Kim, D. Burger, and S. W. Keckler, "Nonuniform cache architectures for wire-delay dominated on-chip caches," *IEEE Micro*, vol. 23, no. 6, pp. 99–107, 2003.
- [34] V. Latora and M. Marchiori, "Efficient behavior of small-world networks," *Physical review letters*, vol. 87, no. 19, p. 198701, 2001.
- [35] 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.
- [36] 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," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.
- [37] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse engineering intel last-level cache complex addressing using performance counters," in *International Symposium on Recent Advances in Intrusion Detection*. Springer, 2015, pp. 48–65.
- [38] F. McSherry, M. Isard, and D. G. Murray, "Scalability! but at what cost?" in *HotOS*, 2015.
- [39] D. Merrill and M. Garland, "Merge-based sparse matrix-vector multiplication (spmv) using the csr storage format," *ACM SIGPLAN Notices*, vol. 51, no. 8, pp. 1–2, 2016.
- [40] 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.
- [41] 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.
- [42] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP Laboratories*, pp. 22–31, 2009.
- [43] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 166–177. [Online]. Available: <https://doi.org/10.1109/ISCA.2016.24>
- [44] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.
- [45] A. Panwar, S. Bansal, and K. Gopinath, "Hawkeye: Efficient fine-grained os support for huge pages," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 347–360.
- [46] 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.
- [47] 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.
- [48] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 2006, pp. 423–432.
- [49] 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.
- [50] 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.
- [51] 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," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1214–1225, 2015.
- [52] H. Wong, "Intel Ivy Bridge cache replacement policy," jan 2013. [Online]. Available: <http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/>
- [53] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely Jr, and J. Emer, "Ship: Signature-based hit predictor for high performance caching," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 430–441.
- [54] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "Imp: Indirect memory prefetcher," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 178–190.
- [55] 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.
- [56] 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.
- [57] 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.
- [58] 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.
- [59] Z. Zhang, H. Wang, S. Han, and W. J. Dally, "Sparch: Efficient architecture for sparse matrix multiplication," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 261–274.