2022 IEEE International Symposium on Workload Characterization (IISWC) | 978-1-6654-8798-6/22/$31.00 ©2022 IEEE | DOI: 10.1109/IISWC55918.2022.00026

The Implications of Page Size Management on Graph Analytics

Aninda Manocha[1] Zi Yan[2] Esin Tureci[1]
Juan Luis Aragón[3] David Nellans[2] Margaret Martonosi[1]

[1]*Princeton University*
[2]*NVIDIA*
[3]*University of Murcia*

Abstract

Graph representations of data are ubiquitous in analytic applications. However, graph workloads are notorious for having irregular memory access patterns with variable access frequency per address, which cause high translation lookaside buffer (TLB) miss rates and significant address translation overheads during workload execution. Furthermore, these access patterns sparsely span a large address space, yielding memory footprints greater than total TLB coverage by orders of magnitude. It is widely recognized that employing huge pages can alleviate some of these bottlenecks. However, in real systems, huge pages are not always available and the OS often provisions huge pages suboptimally, significantly reducing peak application performance. State-of-the-art huge page management techniques employ heuristics, such as huge page region utilization, to guide page size decisions. However, these heuristics are often only optimal for specific memory access patterns, or footprint sizes, and do not sufficiently adapt to dynamic workload characteristics.

This work performs a comprehensive characterization of the effects of page size allocation policy and page placement on graph application throughput. We show that when system memory is nearly full or fragmented (the common case in real systems), huge page resources available to an application are limited and their utility must be maximized. We demonstrate that (1) awareness of single-use memory can eliminate the use of precious huge page resources for data that receives little benefit and (2) coupling degree-aware preprocessing of graph data with programmer-guided use of huge pages boosts performance by $1.26 - 1.57\times$ *over using 4KB pages alone, while achieving* $77.3 - 96.3\%$ *the performance of unbounded huge page usage and requiring only* $0.58 - 2.92\%$ *of the memory resources. This manual, domain-specific optimization of huge page efficiency in memory constrained systems demonstrates that huge pages are a new class of resource that must be intelligently managed by programmers or next-generation OS policies to optimize application performance.*

1. Introduction

Graph data structures are increasingly utilized in big data analytics to relate entities, such as individuals in a social network or web pages on the World Wide Web, in an efficient manner. Many important graph data are constantly growing in size and are sparse by nature, as most vertices are only connected to a small fraction of the entire network. Consequently, graph analytic workloads often have substantial memory footprints, e.g. 10-500 GB, that are accessed in cache-unfriendly access patterns, which hurt hit rates and hamper performance. Prior works have targeted this memory bottleneck by increasing the latency tolerance and memory bandwidth efficiency of CPU architectures for graph analytics through prefetching [5, 48], program decoupling [34, 39], tailored caching techniques (including cache partitioning [9, 13, 51], partial cachelines [22, 26, 33, 55, 56], specialized cache replacement policies [3, 47, 49]), and domain-specific memory subsystems with specialized engines or accelerators [1, 37, 38]. However, to the best of our knowledge this work is the first to explore how to optimize OS page allocation and size for graph data structures to maximize the CPU's virtual memory address translation performance.

Page-based virtual memory hierarchies have historically been optimized for dense, regular access patterns that exhibit high spatial locality. Thus, while many general-purpose applications incur negligible overhead from translation look-aside buffer (TLB) lookups, graph analytic workloads often experience high TLB miss rates that impose significant runtime overheads, as shown in Figures 1 and 2. In modern operating systems (OSes), huge pages enable virtual-to-physical page mappings to be stored at a larger granularity, e.g. 2MB or 1GB, within the page tables. In turn, the processor's TLBs increase their reach across the application's memory footprint, which decreases address translation overheads during execution.

However, huge pages require additional CPU time to create and prepare and are difficult to manage as a system resource [23, 41, 53]. Modern OSes provide mechanisms, such as Linux's transparent huge page (THP) policy, that automatically employ huge pages to improve TLB coverage. This work shows that using THPs achieves significant performance gains for graph workloads when there are abundant contiguous 2MB physical memory regions, but common system-level effects causing memory pressure significantly limit huge page availability. As a result, huge pages often provide little benefit over 4KB base pages in realistic machine scenarios, despite the additional CPU overhead incurred for huge page creation.

978-1-6654-8798-6/22/$31.00 ©2022 IEEE
DOI 10.1109/IISWC55918.2022.00026

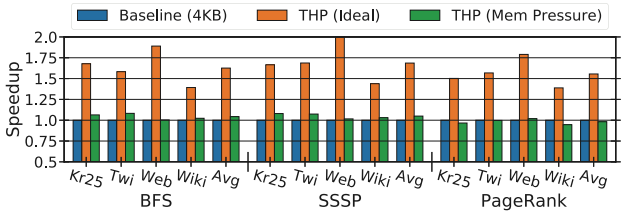

Figure 1: Graph workloads achieve significant performance improvement with huge pages (2MB) compared to only using base (4KB) pages. However, with memory pressure in real systems much of the huge page benefit vanishes.

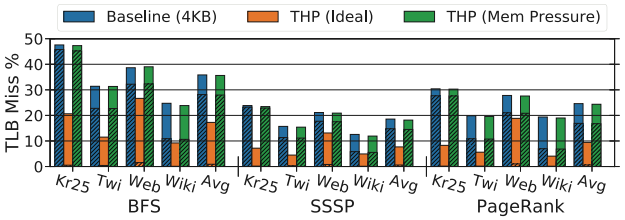

Figure 2: When using 4KB pages (blue), graph analytics experience high data TLB (solid) misses and second-level TLB misses (striped). When Linux's THP policy is enabled and huge pages are readily available (orange), both TLB miss rates are reduced significantly. However, when memory pressure (green) limits the number of available huge pages, Linux's policy offers no TLB miss reduction.

Figure 1 presents a speedup comparison of several graph workloads when employing 4KB pages only (blue), an idealized situation where the system is freshly booted without other applications running (orange), and a realistic situation where other workloads are also running on the system, emulated by limiting available memory to only 3GB greater than the application working set size and fragmenting 50% of the free memory (green). On average there is 1.63× runtime speedup achievable when there is an abundance of huge pages available, indicating there is a good performance improvement opportunity via huge page usage for graph applications. However, when there is system memory pressure, huge pages only provide a 1.02× speedup over 4KB pages.

This work extends state-of-the-art in memory management policies for graph workloads by examining how the memory allocation policy combined with application knowledge and explicit page size management can improve workload performance for analytic practitioners. Our contributions are summarized as follows:

- We provide thorough characterization of the understudied impacts of virtual memory and page management specifically on graph workload performance. This work elucidates where existing OS mechanisms fall short because they lack awareness to graph characteristics and demonstrates that Linux's THP policy cannot achieve peak performance when free system memory is near the size of the graph workload memory footprint or is just 20% fragmented.
- We examine graph application access patterns and provide new insights that enable programmers to maximize the benefits of using huge pages. We show that the irregularly accessed *property array* is almost solely responsible for high TLB miss rates on modern CPUs. Within this structure, memory access frequencies are highly correlated with vertex connectivity; we uncover a new opportunity to maximize performance while minimizing the number of huge pages needed by the system.
- Based on these observations we design a methodology for programmers to selectively and intelligently utilize huge pages for graph workloads. We propose repurposing lightweight preprocessing to coalesce "hot" (based on vertex degree) but sparse data into dense memory regions. With knowledge of where this "hot" data is located within the virtual address space, we selectively employ huge pages at application level within just a small memory region, e.g. 0.58 – 2.92% of the total footprint. This demonstrates 1.26 – 1.57× speedup over Linux's THP policy in the presence of memory pressure and fragmentation, coming within 77.3 – 96.3% of the idealized performance.

2. Background and Motivation

We now present an overview of graph applications and how their programming model affects their access patterns into the memory system, as well as details on Linux's THP policy. This is necessary to understand how to exploit graph analytic workload properties to improve address translation performance.

2.1. Overview of Graph Analytic Workloads

Graph analytic workloads have three important program aspects to consider: 1) data structures used to store graph information, 2) loading and initialization of these structures, and 3) the algorithms that operate on this data. We first walk through these aspects, then discuss the address translation problem.

2.1.1. Data Structures. Graph data is typically stored in the Compressed Sparse Row (CSR) format, as it is the most memory-efficient representation of sparse data. Fig. 3 presents an example CSR representation of a network G (purple). First, the **vertex array** (gold) stores the cumulative number of neighbors each vertex has (when traversing in order of vertex IDs). Second, the **edge array** (green), indexed by vertex array entries, stores the IDs of each vertex's neighbors. Third, the **values array** (blue) stores the values of each edge in the network (if edges are weighted). An additional array, the **property array** (red), tracks output data that is dynamically read and updated for each vertex, depending on the objective of the algorithm, e.g., shortest paths from a given vertex.

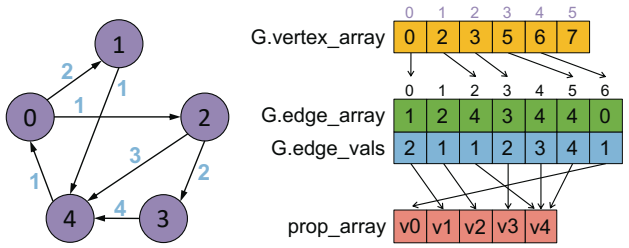

Figure 3: A sparse network is most efficiently stored as 3 dense arrays in CSR format. The *property array* tracks output data for a graph algorithm and is frequently and irregularly accessed to read and update vertex data.

```
1  void init(csr G, unsigned long *prop_arr, ...)
2     G.vertex_array = read_vertex_file(...);
3     G.edge_array = read_edges_file(...);
4     G.edge_vals = read_values_file(...);
5     prop_arr = init_vertex_data();
6  void run(csr G, unsigned long *prop_arr, ...)
7     while (worklist.size > 0)
8        for (v = 0; v < worklist.size; v++)
9           process_vertex(v);
10          for (e = G.vertex_array[v]; e < G.
                vertex_array[v+1]; e++)
11             process_edge(e);
12             neib_id = G.edge_array[e];
13             data = prop_arr[neib_id];
14             if (update_neib(data))
15                add_to_worklist(neib)
```

Figure 4: Typical flow of a graph processing workload where data is initially loaded into memory from storage, then executed over as a second step. The highlighted property array access is responsible for the majority of virtual memory system overheads.

2.1.2. Data Loading and Initialization. Figure 4 presents the pseudocode of the programming model for push-based graph processing kernels, which is the most work-efficient representation of many graph applications. First, all graph data is loaded during an initialization phase (lines 1-5). CSR and property array data are allocated and CSR data are read into memory from files. In addition, the property array (`prop_array`) is populated with initial vertex values, e.g. –1 to represent a vertex that has not been visited.

2.1.3. Graph Algorithms. Once all data has been loaded and initialized, the graph algorithm begins execution. Many graph analytic implementations are both *iterative* and *frontier-based*. The algorithm iterates through a worklist of vertices to process one at a time. The algorithm runs iteratively through frontiers, until no new vertices are added to the worklist, i.e. it is empty (line 7). Vertex processing consists of any necessary computations on a given vertex's data and iterating through its neighbors to determine if updates need to be made.

When processing neighbors, graph algorithms read and perform conditional updates of their property array data (line 13). For example, in BFS, neighbors are updated if they have not been visited. The neighbor ID retrieved from the edge array indexes into the property array (highlighted in gray), performing pointer-indirect accesses frequently in the innermost loop of the algorithm. *This access pattern is primarily responsible for memory system bottlenecks in graph analytic workloads.* These accesses are not only cache-unfriendly in a traditional memory hierarchy, but also have a significant detrimental effect on the CPU's address translation hierarchy, namely two-level TLBs. Even if a less storage-efficient data representation were used, graph application performance would continue to be bottlenecked by pointer indirect memory accesses, as the algorithm's graph traversal depends on the sparse and irregular structure of a network.

2.2. Address Translation Problems

Fig. 2 presents the TLB miss rates of the corresponding configurations, applications, and datasets presented in Fig. 1. The height of each bar represents the first-level data TLB (DTLB) miss rate. A DTLB miss can either hit in the second-level TLB (STLB) or miss and cause a page table walk, indicated by the height of the shaded (striped) bars. If an application only employs base 4KB pages (blue), it experiences high DTLB miss rates ranging from 12.6–47.6% (avg. 26.3%). Most DTLB misses result in STLB misses, incurring costly page table walks to CPU caches and DRAM that have long latencies. Address translation thus becomes a bottleneck.

To overcome this bottleneck, most CPUs today support huge pages in hardware with OS support to utilize huge pages. When a system is freshly booted and Linux's THP policy is enabled, it *aggressively allocates huge pages during application initialization* and nearly all data resides in huge pages on the system. This decreases TLB miss rates significantly to 4 – 26.7% (avg. 11.5%), under half the miss rate of only using 4KB pages. This means huge pages are a good solution to the address translation problem for graph applications, but this improvement is not observed in practice, as shown in Figure 1. Even with moderate amounts of memory pressure and fragmentation, Linux's policy has negligible impact on TLB miss rates. To understand this, we examine how huge pages are implemented and allocated within an OS.

2.3. Linux Transparent Huge Page (THP) Support

In Linux, huge pages can be used explicitly via *hugetlbfs* [32] or implicitly via Transparent Huge Page Support [31]. The latter is more programmer-friendly as it does not require boot-time or runtime page reservations, explicit source code modifications, or memory allocation API interceptions like *hugetlbfs* does. As a result, we focus on THPs to minimize address translation overheads in graph analytic applications.

2.3.1. How Linux's THP Policy Works. Before we dive into the performance optimization achieved by using Linux's policy, we first discuss how huge pages are created and destroyed:

Huge Page Allocation: Huge pages can be created at page fault time when an application first accesses a virtual address without any backing physical page. The Linux kernel receives the page fault triggered by the application, checks the huge page eligibility of the faulting virtual address, then allocates a 2MB huge page in physical memory to back the virtual address if possible. Several factors affect the huge page eligibility of a virtual address, including whether the virtual address is within a 2MB-aligned region, the region is bigger than 2MB, and the OS configuration allows huge page creation at page fault time.

Huge Page Promotion: Huge pages can also be created in place of existing virtual address regions backed by normal 4KB pages, which we call *promotion*. The kernel promotes a 2MB virtual address region by allocating a huge page, copying data from the region to the huge page, and updating the page table entries of the region. This provides flexibility for when huge pages are created, but this process is costly due to data copy time and page table maintenance overheads. To avoid impacting applications directly, Linux runs a kernel daemon, *khugepaged*, to periodically scan application virtual address regions and promote eligible regions in the background.

Huge Page Demotion: When a huge page is only partially mapped in an application, it is underutilized. Linux can demote huge pages to reclaim precious memory resources and does so by splitting a huge page into 512 normal 4KB pages, then updating page table entries to point to the new split pages so that unmapped 4KB pages can be reclaimed. But when a huge page is fully mapped, Linux does not perform any demotions.

2.3.2. Huge Page Inefficiencies. Greedy huge page usage can minimize address translation overheads. However, in reality, constrained memory (not enough free memory available) and/or fragmented memory (free pages are not contiguous when interleaved with in-use pages) can substantially reduce the number of huge pages available for applications and increase the time the kernel spends on huge page creation. This happens when the system has run for a period of time and used pages across the entire physical memory space. When memory is constrained, the Linux kernel is able to reclaim free memory to create new huge pages by swapping in-use data to disk, but this takes additional CPU time and can penalize the application runtime when huge pages are created at page fault time. When memory is fragmented, the number of available huge pages is decreased proportionally to the fragmented memory area. In both scenarios, naïve use of huge pages can hurt application performance. We will later explain further the sources of huge page inefficiency in detail in Section 4.

3. Experimental Methodology

We perform all experiments on a real system and carefully configure the system to minimize environmental inference:

TABLE 1: Evaluation System Parameters.

Processor	Intel Xeon CPU E5-2667 v3 @ 3.20 GHz (Haswell) 2 sockets, 8 cores/socket, 2 threads/core
Operating System	CentOS 7 - Linux v5.15
L1 Data TLB	4KB: 64 entries, 4-way set associativity 2MB: 32 entries, 4-way set associativity 1GB: 4 entries, 4-way set associativity
L1 Inst. TLB	4KB: 64 entries, 8-way set associativity 2MB: 8 entries, full set associativity
L2 TLB	4KB & 2MB: 1024 entries, 8-way set associativity
Memory	64GB DDR4 (per socket)

3.1. System Setup

We perform all evaluations using Linux kernel v5.15 on a 2-socket machine with Intel Xeon processors, where each socket has 64GB of RAM. Tab. 1 presents the system parameters in more detail. In our baseline configuration, THPs are disabled system-wide and applications run using 4KB base pages only.

Our machine has 2 non-uniform memory access (NUMA) nodes, where data access latency and bandwidth differ between local and remote NUMA nodes. Without careful control, the OS arbitrarily allocates memory from either node. This causes non-deterministic runtime results due to non-uniform data access at each run, which unnecessarily complicates our performance analysis. To ensure deterministic results, we use the `membind` flag with `numactl` [29, 30] to bind the process running the application to CPU 0 and all memory allocations to NUMA node 1. We perform this memory binding for all application runs, with and without THPs enabled.

We note that processors can differ in their TLB parameters, e.g. newer processors may have larger TLB sizes. However, even with more capacity, the TLB's total coverage is still significantly smaller than the memory footprint of many irregular applications and address translation bottlenecks remain. Furthermore, TLB size cannot be increased indefinitely due to limitations on access latency and energy consumption. We have performed the same performance characterizations detailed in this work on a newer Broadwell CPU and observed the same performance trends.

3.2. Applications and Datasets

We study the performance of three of the most commonly used graph processing algorithms that form the basis of a wide range of applications. Many other graph algorithms are built on top of variants of these core workloads.

Breadth First Search (BFS): Given a starting (root) vertex, determine the minimum number of hops to all vertices. In addition to its direct use in network analysis, e.g. LinkedIn degree separation, BFS forms the basic building block of many other graph applications such as Graph Neural Networks [54], Connected Components, and Betweenness Centrality [10]. This algorithm only utilizes the vertex array and edge array as inputs and updates the

TABLE 2: Evaluation Applications and Inputs.

Application	Input	Nodes	Edges	Footprint
Breadth First Search (BFS)	Kronecker25 (Kr25)	34M	1.05B	8.5GB
	Twitter (Twit)	53M	1.94B	16GB
	Sd1 Arc (Web)	95M	1.96B	16.5GB
	Wikipedia (Wiki)	12M	378M	3GB
Single Source Shortest Paths (SSSP)	Kronecker25 (Kr25)	34M	1.05B	12.5GB
	Twitter (Twit)	53M	1.94B	24GB
	Sd1 Arc (Web)	95M	1.96B	25GB
	Wikipedia (Wiki)	12M	378M	5GB
PageRank (PR)	Kronecker25 (Kr25)	34M	1.05B	9GB
	Twitter (Twit)	53M	1.94B	16GB
	Sd1 Arc (Web)	95M	1.96B	17GB
	Wikipedia (Wiki)	12M	378M	3GB

property array via pointer indirect accesses with a varying frequency equal to the number of incoming neighbors of each vertex.

PageRank (PR): Determine the "rank" or importance of all vertices (e.g. pages), where vertex scores are distributed to outgoing neighbors and updated until all scores converge (do not change by less ϵ). Variants of PR are used in ranking algorithms, e.g. of webpages, keywords, etc. Similar to BFS, PR accesses only the vertex array and edge array in addition to the property array. However, the number of accesses to the property array is also determined by the number of iterations until the convergence of rank values and therefore depends on in- or out-degrees of vertices as well as the threshold value ϵ.

Single-Source Shortest Paths (SSSP): Given a root vertex, determine the minimum distance (sum of edge weights) to all vertices. SSSP is utilized in navigation and transportation problems as well as network utilization and its more general form is the *k*-shortest paths algorithm. SSSP utilizes the values array in addition to the arrays utilized by BFS and PR. The property array is the most frequently and irregularly accessed, however, edge and values arrays are also accessed repeatedly at varying frequencies, depending on the edge values as well as the connectivity patterns.

We perform our analysis with a synthetic power-law network, Kronecker [8, 24], and 3 real-world social and web networks, Twitter, Sd1 Arc, and Wikipedia [12, 25], creating 12 application/dataset configurations. Tab. 2 details the network parameters and footprints of these configurations.

4. Characterizing THP Performance on Graph Analytic Workloads

This section first analyzes where THPs can most effectively provide performance improvements for graph analytic data structures. Next, we describe sources of huge page resource limitations. Then, we demonstrate how Linux's lack of application and data structure knowledge can harm its THP policy's performance gains when memory resources are limited.

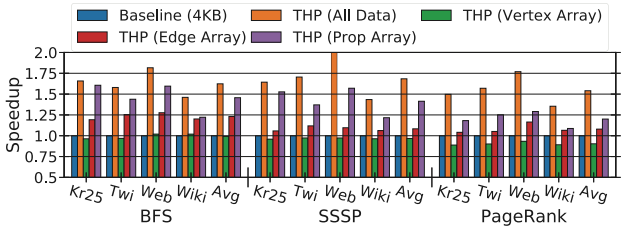

Figure 5: Speedup comparison between system-wide THPs and huge pages selectively applied to individual data structures. The property array benefits the most from huge pages due to being accessed both frequently and sparsely over a large virtual address space. Thus applying huge pages selectively to this array closely matches system-wide THP performance.

4.1. Graph Data Structure Analysis

Graph analytics utilize dense arrays to store vertex or edge locations, or values, and thus different data structures experience varying degrees of access frequency and irregularity. Fig. 4 highlights that memory accesses occur most frequently to the edge and property arrays, but the edge array is accessed sequentially and the property array is accessed irregularly. We therefore conduct an investigation of each data structure's sensitivity to huge pages by selectively applying THPs to each data structure using the `madvise` system call with the `MADV_HUGEPAGE` flag at address `addr` (start of array) with size `size` (array size in bytes).

THP Performance Analysis Fig. 5 compares the application speedups over the baseline (4KB pages) across different THP configurations when running BFS. To evaluate how data structure performances compare to idealized THP performance, there is no memory pressure in this experiment. Applying THPs to the property array (purple) yields a greater performance improvement than applying THPs to the vertex (green) or edge array (red) and also nearly matches the performance of system-wide THPs (orange). With 4KB pages only, sparse neighbor data updates can span a very large virtual address range, which leads to severe thrashing in the TLBs. *Therefore, the property array, which experiences a majority of the TLB misses, reaps the most benefit from huge pages.*

4.2. Huge Page Resource Limitations

Figure 1 shows that huge pages can improve graph application performance by reducing address translation overheads in an ideal system. However, memory pressure that appears in real systems can greatly decrease their effectiveness. Through profiling and analysis we have identified and classified two sources of inefficiency in applying huge pages to graph analytics: (1) memory capacity constraints and (2) memory fragmentation caused by movable and non-movable memory allocations.

At fresh boot time, almost all system memory is free and the OS can allocate as many huge pages as possible. As the

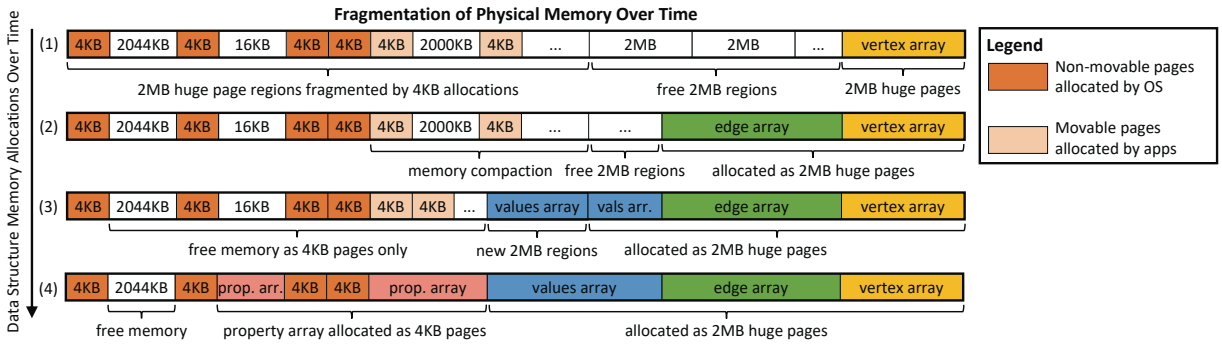

Figure 6: (1) When memory is fragmented by movable or non-movable pages, huge page regions become limited. (2) As graph data structures are initialized on demand, huge pages are used. (3) If no huge pages exist, the OS performs expensive memory compaction and reclamation to create new contiguous regions. (4) However, under heavy memory pressure the OS cannot create new huge pages, leaving no option but to place data into remaining 4KB pages.

system runs, the OS assigns various tasks to memory. With many applications running simultaneously, free memory becomes constrained; contiguous regions of memory naturally become fragmented by page allocations. Fragmentation arises from (1) **movable** pages for most user space memory and (2) **non-movable** pages for memory directly used by the kernel. Memory compaction can mitigate the former by grouping free pages together, but not the latter, which typically worsens over time.

Fig. 6 illustrates how different sources of memory fragmentation interfere with huge page efficiency. The first row demonstrates how movable (light orange) and non-movable (dark orange) pages fragment physical memory, while the OS can still allocate huge pages as long as free 2MB huge page regions are available. As graph analytics allocate memory for the vertex, edge, and values arrays, the OS quickly uses up the huge page regions, as shown in the second and third rows. Once all free 2MB regions are used up, the OS requires extra effort, such as memory compaction and/or page reclamation, to create huge pages in place of regions that are fragmented by movable pages, as shown in the third row.

Once the movable pages have been migrated and/or reclaimed, the newly created huge page regions are utilized by the remainder of the values array and property array. Eventually, all 2MB huge page regions are used up and the OS cannot allocate any more huge pages because the remaining free memory is fragmented by non-movable pages, a scenario which has been studied by prior works [17, 20, 41, 53]. As a result, the OS can only provide 4KB pages for the remainder of the graph data to be allocated, i.e. the property array, as shown in the fourth row. In summary, when memory is constrained, fragmentation interferes with huge page allocation for graph data. Since the property array benefits most from huge pages, Linux's THP policy can lose significant performance opportunities.

4.3. Competition for Memory Resources

Creating huge pages requires 2MB ranges of contiguous physical memory that are readily available when the OS has abundant free memory to use. However, graph processing often occurs on machines where the in-memory graph data storage approaches the total system memory capacity. Similarly, in virtualized environments, cloud service providers attempt to maximize machine utilization and containerized applications are packed to maximize utilization of system resources such as CPU underutilization and memory capacity [6], leading to scarce free system memory.

In addition to external causes of memory pressure, graph analytic workloads often load a considerable amount of data from disk initially. When data is stored in disk or another secondary storage device and then accessed by an application, the OS caches the data in memory (in a page cache) to avoid needing to read the data from disk in the future. However, this can reduce the amount of free memory available to the application itself. During our constrained memory experiments, fewer huge pages are created when the page cache occupies free memory during data loading time. Data loaded during initialization is read into memory and the original data in the page cache is not used again, resulting in free memory that cannot be reclaimed in time for efficient application utilization. To avoid this type of single-use memory interference, the page cache should be eliminated when memory is constrained.

Linux provides a global knob, where a root user can write `1` to `/proc/sys/vm/drop_caches` to drop all page cache pages, however, this knob is too coarse grained. The page cache can be controlled at a finer granularity by using (1) direct I/O to bypass the page cache or (2) *tmpfs* to control the page cache placement. Using *tmpfs* to store graph data files in the remote NUMA node separates the page cache from the local NUMA node where the application runs and exclusively allocates memory. This is faster than direct I/O, as data is read from the remote NUMA memory instead of disk. While the exact methodology an analytics developer uses to avoid huge page allocation

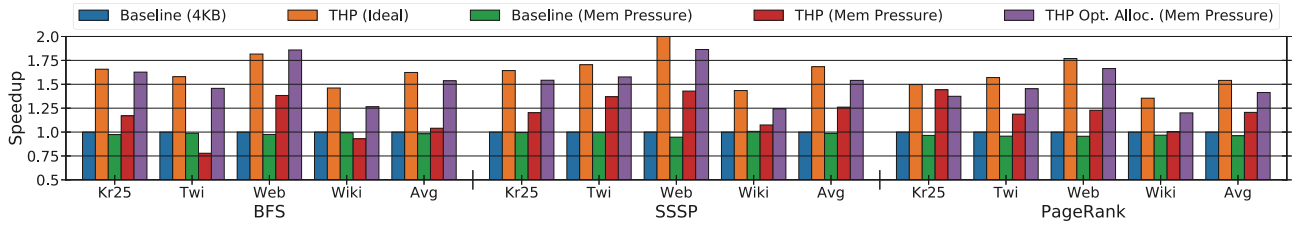

Figure 7: Performance comparisons between 4KB pages and THPs with (only 0.5GB of extra memory available relative to application footprint) and without memory pressure. THP performance is very sensitive to memory pressure, while the small page allocations are not.

interference can vary, it is imperative that developers are aware of such sources of memory interference.

In addition, initialization can involve temporary data structures, such as arrays for converting file data, that are not used during application execution. Such data structures reduce available memory temporarily and can hinder huge page creation, which has long running application effects. Graph analytic application developers must be acutely aware of both memory allocation order as well as the mechanisms in which they are loading data to avoid squandering huge page resources.

4.3.1. THP Performance Under Memory Pressure.

Graph applications that are memory-intensive run alongside other applications in datacenters and it is not uncommon for datacenters to oversubscribe memory. Although in practice the amount of memory pressure can vary significantly throughout application execution, we utilize a synthetic approach to introduce memory pressure at a variety of levels in order to precisely quantify its effect.

For our evaluations we eliminate the page cache and temporary data interference described above to analyze how Linux's policy affects performance in several constrained memory scenarios. To constrain memory, we utilize the *memhog* program to occupy a specified amount of memory, M, on the same NUMA node as the application [27]. We determine M based on the working set size (WSS) of the application/dataset configuration and the degree to which we want to constrain the available memory. For example, to constrain BFS running on the Kronecker network (8.5GB footprint) by $1\times$, we run *memhog* with 55.5GB on NUMA node 1 (64GB total memory). To prevent the OS from swapping out memory allocated by *memhog*, we use *mlock* [28] to pin the program's memory in physical memory. Therefore, the application can only use the 8.5GB free memory left.

Fig. 7 presents the performance impacts of using Linux's THP policy for all applications and datasets when there is high memory pressure (+0.5GB relative to the WSS) and the memory allocation order can either be natural (property array allocated last) or optimized for graph analytics (property array allocated first). All applications exhibit similar trends. When there is no memory pressure, Linux THPs offer significant performance gains (orange). However, these gains are significantly reduced when there is only 0.5GB of extra available memory in the system relative to the application's memory demands (red). Meanwhile, the baseline performance remains unaffected (green). When memory is constrained and the property array is allocated last, this performance critical data structure is allocated with fewer huge pages. Therefore, Linux THPs lose the opportunity to increase performance.

On the other hand, when the order of memory allocation is optimized for graph analytics to allocate the property array first (purple), this data is prioritized for huge page allocation and THP performance gains are much more robust. The performance of Linux THPs with optimized memory allocation, even when there is high memory pressure, nearly matches the ideal performance. This is because the OS is able to allocate huge pages for the entire property array before huge pages become limited while allocating data for CSR arrays. As a result, Linux THPs save the primary TLB misses that result from property array accesses.

To systematically investigate the performance impacts of memory pressure, we swept through 7 different levels of free memory relative to the application's WSS ranging from 0GB beyond the WSS to 3GB in 512MB increments. We also investigated when memory is oversubscribed by 0.5GB to verify that all page management schemes suffer when swapping occurs. We observe three distinct phases when sweeping through these levels of memory pressure:

Low Memory Pressure Linux's THP policy achieves significant performance gains when memory pressure is low. We observed that in our experiments, at least 2.5GB of additional memory available (relative to the WSS) are necessary to achieve the unbounded performance shown in Fig. 1. For the applications and datasets we evaluate, 2.5GB is sufficient for compaction and reclamation to operate and fragmentation that naturally occurs from application execution does not interfere with Linux's ability to allocate and use huge pages. However, the amount of memory necessary for compaction/reclaim may vary with application demands and system parameters.

Moderate Memory Pressure Linux's THP policy performance becomes suboptimal as memory pressure increases to a moderate amount. We observe that performance gains drop by 30% on average when there are 0-2GB of extra memory available. System memory is naturally fragmented and as memory becomes more constrained, fragmentation

inhibits huge page creation. The OS can no longer effortlessly create and use huge pages; extra OS kernel activities, such as memory defragmentation, are needed and performed to make contiguous huge page regions available in physical memory. We systematically characterize the performance impacts of memory fragmentation in the next section.

High Memory Pressure When memory is oversubscribed, page swapping occurs, which requires I/O operations to secondary storage and is much more costly relative to all memory management operations that occur within RAM. Therefore, swapping dominates application runtime. When memory is truly constrained such that there is no additional memory available or there is even a deficit of 0.5GB, the performance of base pages only and Linux's THP policy both degrade by an order of magnitude, resulting in 24.6× and 23.6× slowdowns, respectively, relative to the baseline performance with no memory pressure.

In summary, our characterizations demonstrate that memory pressure can significantly hurt huge page performance opportunities. Memory fragmentation that naturally occurs due to application and OS execution can limit the number of huge pages available for Linux's greedy policy. With low memory pressure, fragmentation does not interfere with THP performance because memory compaction and reclamation can easily operate. However, as memory pressure increases, the process of locating free huge page regions becomes more time consuming and there are not enough regions to provide for the entire application. Thus, moderate to high memory pressure necessitates careful consideration of huge page usage.

4.4. Memory Fragmentation

As memory becomes constrained, fragmentation becomes more apparent. Once free huge page regions run out, the OS must scan fragmented memory to look for compaction opportunities. Memory can be fragmented by movable and non-movable pages, which affect huge page creation differently.

4.4.1. THP Performance Under Fragmentation. In our experiments, we focus on memory fragmentation caused by non-movable pages, because memory compaction and/or page reclaim can alleviate fragmentation caused by movable pages. Because graph analytic workloads initialize all data at the beginning, memory compaction and page reclaim can help create huge pages early in the application execution without impacting the actual algorithm's performance. However, non-movable page fragmentations are always present, limiting the number of huge pages that can be allocated to applications.

To understand the impact of fragmented memory scenarios on huge page availability, we first use *memhog* to reduce the available memory on the node as previously described, then fragment the remaining memory with non-movable pages. We reduce the available memory to provide the application with WSS+3GB. Sec. 4.3 shows that THPs achieve substantial performance gains with this much memory available and thus it is a suitable baseline to evaluate fragmentation.

We define fragmentation level as the percentage of the available memory where there does not exist a contiguous 2MB region. We use a custom program, *frag*, to fragment memory based on a specified level F. This program first allocates 2MB pages, using the kernel function `alloc_pages_node()`, until F% of the available memory has been allocated. It then splits each of these 2MB allocated pages, so they are treated as 512 4KB pages. This allows pages to be freed as 4KB segments, rather than as 2MB. Lastly, the program iterates through each 2MB region (512 4KB pages) and frees pages 2-512. Therefore, the first 4KB page at every 2MB-aligned memory region remains allocated while the remaining memory is available. These pages were allocated with `alloc_pages_node()` without the `___GFP_MOVABLE` flag [21] to make them non-movable.

4.4.2. THP Performance Under Memory Fragmentation. Fig. 8 presents the performance impacts of 50% non-movable memory fragmentation in the presence of low memory pressure (WSS+3GB) for all applications and datasets and the memory allocation order is either natural or optimized for graph analytics. All applications again exhibit similar trends. When there is no fragmentation, THPs offer significant performance gains (orange). When the system memory is 50% fragmented, the OS cannot easily find contiguous 2MB huge page regions to create huge pages and THP benefits are significantly reduced (red). Meanwhile, the baseline performance remains unaffected (green).

Similar to the constrained memory scenario, when the memory allocation order is optimized for graph analytics such that that property array is allocated first, huge pages are prioritized for this data structure that incurs the most TLB misses. As a result, THPs provide performance gains despite memory being fragmented. However, unlike in the constrained memory scenario, where the available memory can be used for huge pages due to its contiguity, THP performance steadily declines with more fragmentation due to non-movable pages, which limits the number of huge pages available for the property array.

4.4.3. Sensitivity to Memory Fragmentation Levels. We sweep through 4 levels of memory fragmentations with a 3GB surplus of memory in relation to the WSS: 0%, 25%, 50%, and 75%. Fig. 9 displays the performance impacts these fragmentation levels have on the baseline and THP performances for BFS. We observe that for all datasets, there is a significant drop in THP performance when the available memory is just 25% fragmented (orange). This is because there are not enough huge pages to allocate for the property array. The corresponding TLB miss rates consequently increase. However, optimizing the memory allocation order regains some performance (green) and THPs can provide significant runtime improvements over the baseline, even when memory is 75% fragmented.

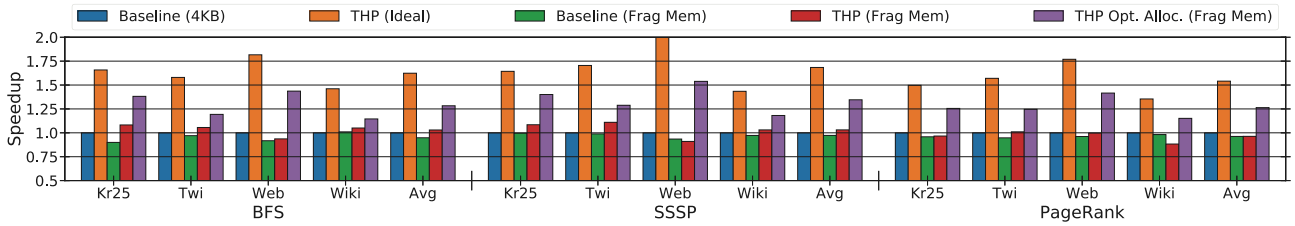

Figure 8: Speedup comparisons when employing THPs without and with 50% memory fragmentation and low memory pressure. THP performance drops significantly at moderate memory fragmentation, whereas THPs with optimized allocation order maintains 25% performance improvement over 4KB pages.

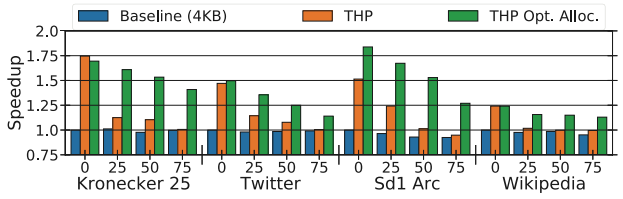

Figure 9: Sensitivity analysis to memory fragmentation for BFS runtime without THPs (baseline) and with natural and optimized THP memory allocation order. As the memory fragmentation level increases from 0-75%, THP performance degrades rapidly, suffering with only 25% memory fragmentation. Optimized allocation maintains good performance even at high fragmentation levels.

Our characterizations demonstrate that fragmentation can impact Linux's THP policy performance when memory is constrained, or fragmented by non-movable kernel allocations. In these scenarios, THPs may offer no performance gains. However, prioritizing the property array for huge page usage via optimized allocation order can regain some or all of the performance lost. Sec. 5 discusses how huge pages can be used more intelligently for the property array.

4.5. Summary

Our characterizations motivate the need for application-aware page size management. First, we find that the most frequent and irregular memory accesses correspond to vertex neighbor data updates in a single *property array* that is primarily responsible for the application's overall high TLB miss rate. Only using huge pages for this array can substantially decrease application TLB miss rates by 12.52%, achieving 82.9% of the performance achieved by using huge pages for the entire application. This allows the OS to only use 2.92%, on average, of the huge pages needed for the entire application.

Second, we show huge pages should not be naïvely allocated for the entire application memory because their resources are precious, especially when the system is under memory pressure. We show that when there is high memory pressure or fragmentation, the OS is not able to allocate huge pages to all application memory. Allocating huge pages without careful consideration until they are used up can significantly decrease the performance gains of Linux's THP policy by 1.39× and 1.59×. However, optimizing memory allocation to prioritize huge pages for the property array regains most of the available performance, achieving 92.1% and 80% of the ideal performance under high memory pressure and 50% fragmentation, respectively. Thus, provisioning huge pages to applications based on their program characteristics can greatly improve huge page efficiency.

5. Tailoring Huge Page Usage to Graph Analytics

It is key to leverage application and data structure knowledge to *selectively* utilize huge pages where they have the greatest performance impact, i.e. the property array, which substantially reduces the number of huge pages necessary for the application. Utilizing huge pages for all other arrays can lead to diminishing returns when there is memory pressure. There are two levels of refinement: (1) tailoring huge page usage to only be within the frequently and irregularly accessed property array and (2) selectively applying THPs to "hot" pages within this data structure. We discuss the details below.

5.1. Hot Page Identification

5.1.1. Variable Access Reuse. Within the property array, different vertices experience varying amounts of reuse. The distribution of access frequencies highly correlates with vertex connectivity. Notably "hot" vertices have high access frequencies and consequently higher reuse, yet even hot pages incur frequent TLB misses because their reuse distances are often much larger than TLB sizes. As a result, the TLB miss latencies for hot data accumulate and lead to significant address translation overheads. This problem worsens as the data size increases since the TLB size is fixed. Huge pages can better capture the reuse of hot data. It is imperative, especially for very large graph data, to prioritize the hottest data for huge page usage in order to eliminate as many TLB misses as possible.

The hot data have a very small collective footprint relative to the total memory footprint of the application. However, because individual hot vertices can be interspersed throughout the virtual address space, a single TLB entry often cannot capture many of them. Therefore, preprocessing needs to coalesce the hot data into a dense memory region, where their entire footprint can easily be covered with just a few huge pages, depending on the network size.

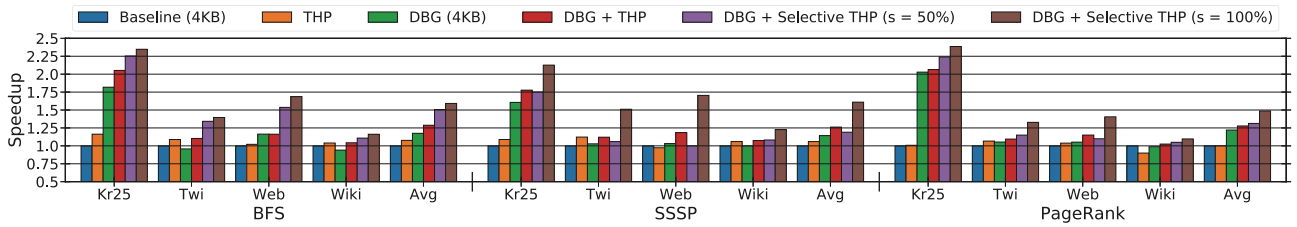

Figure 10: Speedup comparisons with system-wide THPs (applied to all data structures), degree-based grouping (DBG), and applying THPs to all data structures with DBG (DBG+THP) or selectively to a percentage (50% or 100%) of the property array while using DBG (DBG+Selective THP).

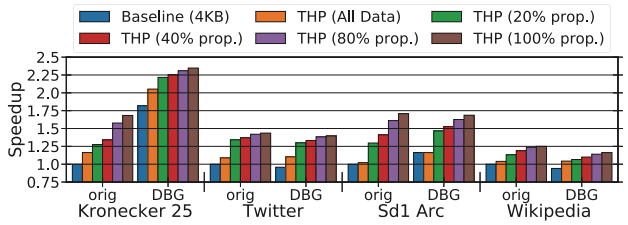

Figure 11: Speedup comparisons with system-wide THPs and huge pages used for 0 – 100% of the property array for BFS when memory is 50% fragmented. THPs for only 20% of the property array outperforms system-wide THPs.

5.1.2. Degree-Based Preprocessing. Faldu et al. proposed a lightweight reordering technique, *Degree-Based Grouping (DBG)* to reduce the memory footprint of "hot" vertices and improve the caching performance of graph analytics [14].

At a high level, DBG coarsely sorts vertex IDs by categorizing them into bins based on their degrees. Each bin has a minimum degree, e.g. vertices in the first bin must have a degree greater than or equal to $32d$, where d is the average degree of the network. The order in which vertices are arranged within each bin does not matter; the bins are simply used to differentiate between levels of degrees, or hotness. DBG uses 8 bins with the following minimum degrees: $32d$, $16d$, $8d$, $4d$, $2d$, d, $0.5d$, and 0. Due to the power-law nature of modern networks, a majority of vertices occupy the last bin, however, these bins sufficiently distinguish between hot and cold vertices. Once all vertices have been placed into bins, a mapping of all vertex IDs is generated by iterating through the bins in order from hottest to coldest.

Sorting Overheads DBG is considered a lightweight sorting algorithm because the graph structure is largely preserved. The graph only needs to be traversed 3 times: once to determine the degrees of all vertices, once to place each vertex in a bin, and once to generate a new ID for each vertex based on the mapping resulting from the preprocessing. The sizes of these traversals are linear with respect to the number of vertices in the network, whereas the application itself performs traversals of sizes based on the number of edges in the network. Because there are typically many more edges than vertices in real-world networks, this preprocessing often has low overhead.

We measure the preprocessing runtime overhead and find that in most cases, DBG has negligible impacts on the end-to-end application runtime. For SSSP and PR, DBG incurs up to a 2.36% overhead (1.32% average). For BFS, the application with the shortest runtime, DBG incurs up to a 16.5% overhead for the Wikipedia network and 13% on average. However, the ensuing caching improvements, both in the traditional and virtual memory hierarchy, more than offset this overhead. We perform preprocessing separately in order to not interfere with the available memory for huge pages and account for the preprocessing times when measuring application runtimes.

5.2. Selective THP Usage

Once hot data have been coalesced, fewer huge pages are necessary to sufficiently cover data regions where page size promotions are worthwhile. With this preprocessing knowledge, the programmer can perform application-aware huge page management by utilizing `madvise` and adjusting the `length` parameter to only apply THPs to s% of the property array, where s is a predetermined percentage.

Fig. 10 summarizes the performance impacts of 2 levels of selective THP use, 50% and 100% of the property array, combined with degree-based preprocessing in the presence of low memory pressure (+3GB) and 50% fragmentation for all datasets and applications. Preprocessing alone via DBG (green) provides some speedup for networks with little to no community structure, such as the Kronecker network, while many real-world networks, e.g. Twitter and Wikipedia, naturally have hot vertices in close proximity to one another, so DBG has little impact on the network structure and performance. For all configurations, THPs applied selectively to s = 100% of the property array (brown) outperforms DBG with (red) and without (green) system-wide THPs enabled. Selective THPs with s = 50% (purple) also outperforms DBG (green) and THPs applied system-wide (orange and red) for most configurations.

Sensitivity to THP Selectivity Levels We measure the sensitivity of selective THP performance by sweeping through backing 0 – 100% (increments of 20%) of the property array by huge pages with both the original and preprocessed (DBG) datasets. Fig. 11 demonstrates the

performance impacts of these selective THP configurations on BFS when there is low memory pressure (+3GB), but the free memory is 50% fragmented. For the original Kronecker and Sd1 Arc web networks, using more huge pages for the property array is always beneficial because the hot vertices are spread throughout the data structure. For the Twitter and Wikipedia networks, which naturally have community structure, and all preprocessed datasets, there are diminishing returns as more huge pages are used. This is because the hot data only occupies a small fraction of memory at the beginning of the property array; utilizing huge pages for this region successfully eliminates TLB misses and improves address translation overheads.

In all cases (with and without preprocessing), applying huge pages to just 20% of the property array yields a 1.15× speedup over THPs applied system-wide. Ultimately, by applying selective huge page allocation on top of lightweight data preprocessing, we are able to achieve 1.26 – 1.57× speedups over the baseline and 1.18 – 1.49× speedups over Linux's policy by improving huge page efficiency and reducing 2MB TLB thrashing, while using huge pages for just 0.58 – 2.92% of the total application memory. We note that these speedups we report are those solely attributed to improved TLB miss rates; any other improvement, e.g. on-chip cache performance improvement from degree-based sorting, is present in the baseline and with our page management strategy.

While these observations and manual tuning steps for analytic applications are effective, they are just the first step towards automatically identifying and exploiting the asymmetric value of huge page allocations in graph analytic applications. Our results demonstrate ample opportunity for future work on automated software and hardware co-designed runtime systems to exploit these trends.

6. Related Work

Transparent Huge Page Management: Various techniques aim to address the issues that arise from Linux's greedy huge page allocation strategy. Applications using huge pages in NUMA systems can suffer from imbalanced data access issues [11, 16]. Memory bloat is common and wastes free memory if not all data within a huge page region is used. Prior works balance performance and bloat by tracking memory accesses and demoting huge pages when the number of accessed constituent base pages is below a certain threshold [2, 23, 35, 36, 40, 45, 50, 58]. Memory fragmentation increases huge page allocation latency, as the kernel must perform extra work such as memory compaction. When applications page fault a huge page, the long allocation latency sits in the critical path, degrading overall application performance. Pre-zeroing, asynchronous promotion, user-directed promotion, and huge page preallocation are techniques to mitigate this problem [23, 36, 40, 46, 58]. Furthermore, 1GB huge pages are suitable for applications with larger than usual memory footprints [44].

Although state-of-the art huge page management offers improvements over Linux in the presence of system memory pressure, its performance remains suboptimal compared to the peak performance achievable by using huge pages. Rather than perform costly access tracking and kernel overheads to capture application-unaware heuristics [23, 40], our work highlights the opportunity of application and data access pattern knowledge in identifying data worth backing with huge pages.

TLB Improvements: A TLB is the hardware cache for page table entries used to reduce address translation time. TLB entries have limited size options (4KB, 2MB, and 1GB on x86_64) and the gap between them is very large (512× apart on x86_64). A variety of works focused on TLB optimization improve translation overheads by increasing memory region contiguity. CoLT and RangeTLB coalesce multiple page table entries of contiguous 4KB pages into a single TLB entry to increase TLB coverage [15, 19, 42, 43]. Software techniques also propose to maximize the size of contiguous memory regions in the system [15, 19, 53]. For applications with a small number of large contiguous memory ranges, direct segment proposes to use memory segments in the OS to provision these ranges and [base, size] format in the TLB to cache the address translations of these segments [7]. Midgard attacks the TLB coverage issue from a different angle; by introducing a new address space of virtual memory areas, it significantly reduces the amount of translations that need to be cached since virtual memory areas are usually much larger than normal page sizes and much smaller in numbers [18]. Overall, TLB optimization is orthogonal to page size management; these approaches complement our techniques.

Graph Sorting: Although complete regularization of graph data accesses is not possible for many graph algorithms, it is possible, yet challenging, to increase temporal or spatial cache locality through graph reordering. Reordering vertex information improves the locality of data in graph applications by coalescing addresses that are accessed more frequently to occupy the same cacheline or TLB entry (same virtual memory page). While degree-based graph sorting improves temporal locality [4, 57], it can come at the cost of destroying local "community structure," resulting in poor spatial locality. Similarly, graph sorting based on community structure increases spatial locality at the cost of poor temporal locality (in addition to a high preprocessing overhead) [52]. DBG [14] attempts to balance spatial and temporal locality optimization, where nodes are sorted based on degree in a course-grained fashion such that short distance spatial locality is preserved. This improves both cache and TLB miss rates for data in highly connected portions of memory, but large memory footprints (greater than the capacity of the caching structures) are still problematic for sorted datasets, especially when memory accesses are irregular. Our technique expands the total TLB reach in an intelligent and frugal manner, which is complementary to data reordering optimizations. Furthermore, our work uncovers the virtual memory impact of running graph algorithms on both unsorted and sorted networks.

7. Conclusion

Huge pages can effectively alleviate address translation overheads for memory-intensive graph applications in ideal scenarios having little to no memory pressure. However, application characteristics, namely irregular data access patterns and highly biased access frequency across all data, greatly impact the effectiveness of huge page utilization and modern OS huge page management is often far from optimal. When a system has limited and/or fragmented memory, huge page resources become precious and naïve huge page allocation policies cause huge page underutilization, resulting in wasted effort attempting to create huge pages. Despite prior research efforts to improve huge page efficiency, there remains room for improvement. State-of-the-art uses tuned heuristics, rather than access pattern characteristics, and thus cannot reliably identify data that benefits most from huge pages.

Our work demonstrates the value of application-aware huge page management. For graph analytics, aggregating frequently accessed data via preprocessing and allocating huge pages to the data structures that receive the most benefit realizes significant performance improvements, while only requiring huge pages for a small fraction of the application memory. Specifically, we achieve 77.3 – 96.3% of unbounded huge page performance and a speedup of 1.26 – 1.57× over using 4KB pages, while only requiring 0.58 – 2.92% of the application memory footprint be backed by huge pages. This is a realistic amount of available THPs to expect in real systems experiencing memory pressure.

Graph applications are representative of a worst-case performance scenario for virtual memory and thus there are few other known opportunities to improve the address translation bottleneck, and consequently overall performance, as significantly as through efficient huge page management. For this application domain, we conclude that huge page resources are the next performance frontier that needs to be managed by programmers, OSes, and next-generation automated systems, in order to leverage *application behavior knowledge* with real-time memory system resource tracking to obtain peak performance.

Acknowledgments

We thank the anonymous reviewers for their feedback. This research was supported in part by the DARPA SDH Program under agreement No. FA8650-18-2-7862 and NSF Award No. 1763838. Aninda Manocha was supported by the NSF Graduate Research Fellowship. Prof. Aragón was supported by Fundación Séneca-Agencia de Ciencia y Tecnología, Región de Murcia, Programa Jiménez de la Espada (21508/EE/21). The authors' views and conclusions contained herein should not be interpreted as representing the official policies or endorsements, either expressed or implied, of DARPA or NSF.

References

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

[2] N. Agarwal and T. F. Wenisch, "Thermostat: Application-transparent page management for two-tiered main memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017, pp. 631–644. [Online]. Available: http://doi.acm.org/10.1145/3037697.3037706

[3] V. Balaji, N. Crago, A. Jaleel, and B. Lucia, "Practical optimal cache replacement for graphs," in *Proceedings of the 2021 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2021.

[4] 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.

[5] 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 *Proceedings of the 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 373–386.

[6] S. A. Baset, L. Wang, and C. Tang, "Towards an understanding of oversubscription in cloud," in *2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE 12)*. San Jose, CA: USENIX Association, Apr. 2012. [Online]. Available: https://www.usenix.org/conference/hot-ice12/workshop-program/presentation/baset

[7] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 237–248. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485943

[8] S. Beamer, K. Asanović, and D. Patterson, "The GAP benchmark suite," 2015.

[9] N. Beckmann and D. Sanchez, "Jigsaw: Scalable software-defined caches," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013, pp. 213–224.

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

[11] J. Corbet, "AutoNUMA: the other approach to NUMA scheduling," http://lwn.net/Articles/488709/, 2012, [Online; accessed 04-Aug-2018].

[12] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, 2011.

[13] N. El-Sayed, A. Mukkara, P. Tsai, H. Kasture, X. Ma, and D. Sanchez, "KPart: A hybrid cache partitioning-sharing technique for commodity multicores," in *Proceedings of 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 104–117.

[14] P. Faldu, J. Diamond, and B. Grot, "A closer look at lightweight graph reordering," in *2019 IEEE International Symposium on Workload Characterization (IISWC)*. United States: Institute of Electrical and Electronics Engineers (IEEE), Mar. 2020, pp. 1–13, 2019 IEEE International Symposium on Workload Characterization, IISWC-2019 ; Conference date: 03-11-2019 Through 05-11-2019.

[15] J. Gandhi, V. Karakostas, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Ünsal, "Range translations for fast virtual memory," *IEEE Micro*, vol. 36, no. 3, pp. 118–126, 2016.

[16] F. Gaud, B. Lepers, J. Decouchant, J. Funston, A. Fedorova, and V. Quéma, "Large pages may be harmful on numa systems," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 231–242. [Online]. Available: http://dl.acm.org/citation.cfm?id=2643634.2643659

[17] M. Gorman and A. Whitcroft, "The what, the why and the where to of anti-fragmentation," in *Linux Symposium*, 2006, pp. 369–384.

[18] S. Gupta, A. Bhattacharyya, Y. Oh, A. Bhattacharjee, B. Falsafi, and M. Payer, *Rebooting Virtual Memory with Midgard*. IEEE Press, 2021, p. 512–525. [Online]. Available: https://doi.org/10.1109/ISCA52012.2021.00047

[19] F. Guvenilir and Y. N. Patt, "Tailored page sizes," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ser. ISCA '20. IEEE Press, 2020, p. 900–912. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00078

[20] M. S. Johnstone and P. R. Wilson, "The memory fragmentation problem: Solved?" in *Proceedings of the 1st International Symposium on Memory Management*, ser. ISMM '98. New York, NY, USA: ACM, 1998, pp. 26–36. [Online]. Available: http://doi.acm.org/10.1145/286860.286864

[21] L. Kernel, "Get free page (gfp) flags," https://elixir.bootlin.com/linux/v5.15/source/include/linux/gfp.h#L88, 2022.

[22] S. Kumar, H. Zhao, A. Shriraman, E. Matthews, S. Dwarkadas, and L. Shannon, "Amoeba-cache: Adaptive blocks for eliminating waste in the memory hierarchy," in *Proceedings of the 45th Annual International Symposium on Microarchitecture (MICRO)*, 2012, p. 376–388.

[23] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel, "Coordinated and efficient huge page management with ingens," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 705–721. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kwon

[24] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: An approach to modeling networks," *Journal of Machine Learning Reseach (JMLR)*, vol. 11, pp. 985–1042, Mar. 2010.

[25] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[26] L. Li, A. B. Hayes, S. L. Song, and E. Z. Zhang, "Tag-split cache for efficient GPGPU cache utilization," in *Proceedings of the 2016 International Conference on Supercomputing (ICS)*. ACM, 2016.

[27] Linux Kernel Documentation, "memhog(8) — Linux manual page," https://man7.org/linux/man-pages/man8/memhog.8.html.

[28] ——, "mlock(2) — Linux manual page," https://man7.org/linux/man-pages/man2/mlock.2.html.

[29] ——, "numa(3) — Linux manual page," https://man7.org/linux/man-pages/man3/numa.3.html.

[30] ——, "numactl - Control NUMA policy for processes or shared memory," https://linux.die.net/man/8/numactl.

[31] ——, "Transparent Hugepage Support," https://www.kernel.org/doc/html/latest/admin-guide/mm/transhuge.html.

[32] ——, "hugetlb page," https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt, 2022.

[33] A. Manocha, J. L. Aragón, and M. Martonosi, "Graphfire: Synergizing fetch, insertion, and replacement policies for graph analytics," *IEEE Transactions on Computers*, March 2022.

[34] A. Manocha, T. Sorensen, E. Tureci, O. Matthews, J. L. Aragón, and M. Martonosi, "GraphAttack: Optimizing data supply for graph applications on in-order multicore architectures," *ACM Transactions on Architecture and Code Optimization*, vol. 18, no. 4, pp. 1–26, Sep 2021.

[35] M. R. Meswani, S. Blagodurov, D. Roberts, J. o. Slice, M. Ignatowski, and G. H. Loh, "Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, February 2015, pp. 126–136.

[36] T. Michailidis, A. Delis, and M. Roussopoulos, "Mega: Overcoming traditional problems with os huge page management," in *Proceedings of the 12th ACM International Conference on Systems and Storage*, ser. SYSTOR '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 121–131. [Online]. Available: https://doi.org/10.1145/3319647.3325839

[37] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, "Exploiting locality in graph analytics through hardware-accelerated traversal scheduling," in *International symposium on Microarchitecture (MICRO)*, October 2018.

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

[39] Q. M. Nguyen and D. Sanchez, "Pipette: Improving core utilization on irregular applications through intra-core pipeline parallelism," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 596–608.

[40] 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*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 347–360. [Online]. Available: https://doi.org/10.1145/3297858.3304064

[41] A. Panwar, A. Prasad, and K. Gopinath, *Making Huge Pages Actually Useful*. New York, NY, USA: Association for Computing Machinery, 2018, p. 679–692. [Online]. Available: https://doi.org/10.1145/3173162.3173203

[42] C. H. Park, T. Heo, J. Jeong, and J. Huh, "Hybrid tlb coalescing: Improving tlb translation coverage under diverse fragmented memory allocations," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 444–456. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080217

[43] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "Colt: Coalesced large-reach tlbs," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45. Washington, DC, USA: IEEE Computer Society, 2012, pp. 258–269. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2012.32

[44] V. S. S. Ram, A. Panwar, and A. Basu, "Trident: Harnessing architectural resources for all page sizes in x86 processors," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1106–1120. [Online]. Available: https://doi.org/10.1145/3466752.3480062

[45] L. E. Ramos, E. Gorbatov, and R. Bianchini, "Page placement in hybrid memory systems," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '11. New York, NY, USA: ACM, 2011, pp. 85–95. [Online]. Available: http://doi.acm.org/10.1145/1995896.1995911

[46] D. Rientjes, "[rfc] hugepage collapse in process context," https://lore.kernel.org/linux-mm/d098c392-273a-36a4-1a29-59731cdf5d3d@google.com/, 2021.

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

[48] N. Talati, K. May, A. Behroozi, Y. Yang, K. Kaszyk, C. Vasiladiotis, T. Verma, L. Li, B. Nguyen, J. Sun, J. Morton, A. Ahmadi, T. Austin, M. O'Boyle, S. Mahlke, T. Mudge, and R. Drelinski, "Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2021.

[49] E. Teran, Z. Wang, and D. A. Jiménez, "Perceptron learning for reuse prediction," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[50] M. M. Tikir and J. K. Hollingsworth, "Hardware monitors for dynamic page migration," *J. Parallel Distrib. Comput.*, vol. 68, no. 9, pp. 1186–1200, Sep. 2008. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2008.05.006

[51] N. Vijaykumar, A. Jain, D. Majumdar, K. Hsieh, G. Pekhimenko, E. Ebrahimi, N. Hajinazar, P. B. Gibbons, and O. Mutlu, "A case for richer cross-layer abstractions: Bridging the semantic gap with expressive memory," in *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 207–220.

[52] H. Wei, J. X. Yu, C. Lu, and X. Lin, "Speedup graph processing by graph ordering," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1813–1828.

[53] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Translation ranger: Operating system support for contiguity-aware tlbs," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 698–710. [Online]. Available: https://doi.org/10.1145/3307650.3322223

[54] J. You, R. Ying, and J. Leskovec, "Position-aware graph neural networks," in *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. PMLR, 2019, pp. 7134–7143. [Online]. Available: http://proceedings.mlr.press/v97/you19b.html

[55] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "IMP: Indirect memory prefetcher," in *Proceedings of the 48th Annual International Symposium on Microarchitecture (MICRO)*, Dec 2015, pp. 178–190.

[56] C. Zhang, Y. Zeng, and X. Guo, "Scrabble: A fine-grained cache with adaptive merged block," *IEEE Transactions on Computers*, vol. 69, no. 1, pp. 112–125, 2020.

[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)*, 2017, pp. 293–302.

[58] W. Zhu, A. L. Cox, and S. Rixner, "A comprehensive analysis of superpage management mechanisms and policies," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 829–842. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/zhu-weixi

Appendix

1. Abstract

This artifact description provides information about the complete workflow necessary to generate and analyze the characterizations and results detailed in our manuscript, "The Implications of Page Size Management on Graph Analytics". We have provided public access to the graph applications and datasets (networks) we evaluated, the necessary software packages, and the experiment scripts we have written to systematically characterize virtual memory and system effects.

2. Artifact check-list (meta-information)

- **Algorithm:** Characterization of the impacts of huge page management techniques on graph analytic application runtime and TLB performance.
- **Program:** Breadth-First Search (BFS), Single-Source Shortest Paths (SSSP), and PageRank.
- **Compilation:** Intel C++ Compiler.
- **Binary:** Binaries for applications are generated via C++ compilation of source code, binaries for software tools (e.g. to limit or fragment memory) are provided, but can be regenerated using the source code provided.
- **Run-time environment:** Modified Linux v5.15 kernel. Linux's transparent huge page policy is enabled (system-wide) or set to madvise (programmer-directed).
- **Hardware:** Intel Xeon CPU E5-2667 v3 @ 3.20 GHz with 2 sockets, 8 cores/socket, and 2 threads/core.
- **Run-time state:** RAM is either freely available, limited but large contiguous chunks are available, or fragmented such that few large contiguous chunks are available.
- **Execution:** Linux shell scripts are provided to automate experiment execution (details below).
- **Metrics:** Application execution time speedup, Translation Lookaside Buffer (TLB) miss rate (% of TLB accesses that result in a miss in the first- and second-level TLBs).
- **Output:** Application execution time (seconds), TLB miss rate, and page table walk rate.
- **Experiments:** We measure the performance impacts of huge pages on graph application runtime and TLB performance. We investigate the application performance impacts of: (1) 4KB base pages only, (2) 2MB huge pages only, (3) 2MB huge pages for a single data structure while the remaining data are backed with 4KB pages, (4) 4KB base pages when there is memory pressure: limited free (0-3GB available) and/or fragmented memory (25-100%), (5) Linux's policy when there is memory pressure: limited free (0-3GB available) and/or fragmented memory (25-100%), (6) Linux's policy when there is memory pressure and the order of memory allocations is optimized for graph analytics, and (7) selective huge page use for data that frequently incurs TLB misses when there is memory pressure: limited free memory (3GB available) and fragmented memory (50%)
- **How much disk space required (approximately)?:** 100GB.
- **How much time is needed to prepare workflow (approximately)?:** 1 hour.
- **How much time is needed to complete experiments (approximately)?:** 2 weeks.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** CC-BY 4.0.
- **Data licenses (if publicly available)?:** CC-BY 4.0.
- **Workflow framework used?:** Linux shell scripts and Python v3.4.

3. Description

3.1. How to access. All application source code, links to public datasets and modified Linux kernel source code, software utilities, and workflow (automated experiment execution) scripts are available at https://github.com/amanocha/graphs_thp_management.

3.2. Hardware dependencies. We perform our characterizations and evaluations using an Intel Haswell Processor. To replicate these results, an x86 machine (support for 4KB and 2MB page sizes) with at least 2 NUMA nodes and at least 20GB of RAM available per node is necessary.

3.3. Software dependencies. We have modified the Linux v5.15 kernel source to provide support for programmer-driven huge page usage via system calls. This customized version is used for all experiments and is available at https://github.com/amanocha/graphs_thp_linux. It must be cloned, compiled, and installed onto the machine. Additionally, Python3 and `numactl` are needed to run experiments.

3.4. Data sets. We use the following datasets in our evaluations as well as their reordered variants:

1) Kronecker25 (synthetic power-law network)
2) Twitter (real social network)
3) Sd1 Arc (real web network)
4) Wikipedia (real social network)

All datasets are available at https://decades.cs.princeton.edu/datasets/big/. We perform reordering using the Degree-Based Grouping method (https://faldupriyank.com/papers/DBG_IISWC19.pdf). While we have provided the reordered datasets, we also provide the source code for regenerating them.

For more details, visit https://github.com/amanocha/graphs_thp_management#datasets.

4. Installation

Linux Source Code First clone our Linux kernel source code repository:

```
git clone https://github.com/amanocha/
    graphs_thp_linux.git
```

Then compile and install the code:

```
cd graphs_thp_linux
cp config .config
sudo yum group install "Development Tools"
sudo yum install ncurses-devel bison flex
    elfutils-libelf-devel openssl-devel
make -j [NUM_CORES]
sudo make modules_install
sudo make install
sudo reboot
```

After reboot verify that the code was installed correctly (you should see `5.15.0-rc6+` as the kernel version):

```
uname -r
```

Perf Linux's `perf` tool is needed to measure TLB miss rates. To install it:

```
cd graphs_thp_linux/tools/perf/
make -j [NUM_CORES]
cp perf /usr/bin/perf
```

Workflow To install our workflow code, clone our repository:

```
git clone https://github.com/amanocha/
    graphs_thp_management.git
```

Then run the following to set up software utilities and directories needed to run the experiments:

```
cd graphs_thp_management/numactl
bash make.sh
cd ../utils
make
cd ..
mkdir results
```

Datasets must be downloaded from https://decades.cs.princeton.edu/datasets/big/ and stored in a local folder, e.g. `local_data`. Then run the following commands to mount the datasets on a specific NUMA node `NUMA_NODE`:

```
cd graphs_thp_management
mkdir data
sudo mount -t tmpfs -o size=100g,mpol=bind: [
    NUMA_NODE] tmpfs data
cp -r [PATH_TO_DATASETS]/* data/
```

5. Experiment workflow

Details on experiment setup and all scripts are described at https://github.com/amanocha/graphs_thp_management#experiments. We provide scripts to characterize and evaluate the following:

TLB Miss Rates: To characterize the impacts of TLB misses and address translation overheads on graph analytics, run the following command (output will be stored in `results/ tlb_char/`):

```
sudo bash thp.sh 1 all all
```

Data Structure Analysis: To characterize the performance impacts of utilizing huge pages for individual data structures in graph analytics, namely the vertex array, edge array, and property array, run the following command (output will be stored in `results/data_struct/`):

```
sudo bash thp.sh 2 all all
```

Constrained Memory: To characterize the performance impacts of memory pressure on Linux THP performance, the scripting environment must first be configured. Open the file constrained.sh in a text editor and modify lines 7 and 8 based on the NUMA node ID and the amount of memory available (in MB) on the node (e.g. NUMA node 1 has 64GB of RAM):

```
NUMA_NODE=1 # NODE NUMBER
MAX_RAM=64000 # AMT OF MEM ON NODE
```

Then run the following (output will be stored in `results/constrained_mem/`):

```
sudo bash constrained.sh
```

Fragmented Memory: To characterize the performance impacts of memory fragmentation on Linux THP performance, the scripting environment must be first be configured. Open the file frag.sh in a text editor and modify lines 9 and 10 based on the NUMA node ID and the amount of memory available (in MB) on the node (e.g. NUMA node 1 has 64GB of RAM):

```
NUMA_NODE=1 # NODE NUMBER
MAX_RAM=64000 # AMT OF MEM ON NODE
```

Then run the following (output will be stored in `results/frag_mem/`):

```
sudo bash run_frag.sh 4
```

Selective THP: To characterize the performance impacts of selective THP usage, run the following (output will be stored in `results/select_thp/`):

```
sudo bash run_frag.sh 5
```

6. Evaluation and expected results

The experiment scripts automatically generate results. The directory structure of the results is described at https://github.com/amanocha/graphs_thp_management#results.

For each application/dataset configuration, there is an output folder `$APPLICATION_$DATASET` that contains application execution output in `app_output_x_i.txt`, where `x` is the transparent huge page setting and `i` is the iteration number (each experiment is run 3 times). The end of the output shows the following:

```
total kernel computation time: [TIME_IN_SEC]
user time: [USER_TIME_IN_SEC]
kernel time: [KERNEL_TIME_IN_SEC]
```

TLB output is recorded in `results_i.txt` (average is recorded in `results.txt`) and appears as follows:

```
TLB Miss Rate: [%]
STLB Miss Rate: [%]
Page Fault Rate: [%]
```

When there is no memory pressure, runtime performance should always improve over 4KB pages only when Linux's transparent huge page policy is used. As memory becomes more limited (but not oversubscribed) or more fragmented, then the performance improvements should decrease until there are essentially no improvements at all. Selective huge page usage should provide performance improvements regardless of whether there is system memory pressure.

7. Experiment customization

Linux shell scripts are customizable to adjust the amount of memory to limit and the level of memory fragmentation.

8. Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html