Large-scale Graph Processing on Commodity Systems: Understanding and Mitigating the Impact of Swapping

Alireza Haddadi
Dept. of IT, Uppsala University
Uppsala, Sweden
alireza.haddadi@it.uu.se

David Black-Schaffer
Dept. of IT, Uppsala University
Uppsala, Sweden
david.black-schaffer@it.uu.se

Chang Hyun Park
Dept. of IT, Uppsala University
Uppsala, Sweden
chang.hyun.park@it.uu.se

ABSTRACT

Graph workloads are critical in many areas. Unfortunately, graph sizes have been increasing faster than DRAM capacity. As a result, large-scale graph processing necessarily falls back to virtual memory paging, resulting in tremendous performance losses.

In this work we investigate how we can get the best possible performance on commodity systems from graphs that cannot fit into DRAM by understanding, and adjusting, how the virtual memory system and the graph characteristics interact. To do so, we first characterize the graph applications, system, and SSD behavior as a function of how much of the graph fits in DRAM. From this analysis we see that for multiple graph types, the system fails to fully utilize the bandwidth of the SSDs due to a lack of parallel page-in requests.

We use this insight to motivate overcommitting CPU threads for graph processing. This allows us to significantly increase the number of parallel page-in requests for several graph types, and recover much of the performance lost to paging. We show that overcommitting threads generally improves performance for various algorithms and graph types. However, we identify one graph that suffers from overcommitting threads, leading to the recommendation that overcommitting threads is generally good for performance, but there may be certain graph inputs that suffer from overcommitting threads.

CCS CONCEPTS

• **Software and its engineering** → **Virtual memory**.

KEYWORDS

graph processing, virtual memory, swapping, SSD, commodity system, thread overcommitting, characterization, operating system

1 INTRODUCTION

Graph processing workloads are widely used in industry and in supercomputing [17]. Today's *large-scale graphs* are often enormous in size (more than 100GBs [9]) and typically incur many irregular memory accesses, making them challenging for the locality-based caching and pattern-based prefetching of modern memory systems. Unfortunately, DRAM capacity has not kept up with the growth of graph sizes, meaning that large-scale graphs are impossible to keep in DRAM during processing. Prior works have studied *out-of-core computing* for graphs that do not fit in the DRAM. These approaches require specialized algorithmic changes [13, 14, 19, 20, 23, 25, 26] and/or specialized hardware to achieve out-of-core computing [16].

In this work, we investigate the processing of large-scale graphs with commodity graph algorithm implementations running on commodity systems. In these *commodity systems*, large-scale graph processing will necessarily result in accesses to the backing store due to limited DRAM capacity, and, with *commodity implementations*, these accesses will happen automatically through the operating system's virtual memory swapping. For our commodity system, we use SSDs for virtual memory, which can provide enormous throughput at low latency, but only if the operating system provides them with sufficiently many parallel requests. This potential for virtual memory performance through parallelism, combined with the diversity of the degree of parallelism in the graph workloads, frames our areas of investigation: What are the bottlenecks in current commodity systems? And, what changes can we make to commodity systems to improve performance?

To address these, we first explore the performance impact on large-scale graph processing of paging by varying the portion of the graph that fits into DRAM. As expected, performance can drop dramatically with swapping (Section 4). To understand where the inefficiency is coming from, we then take a closer look into the kernel and the SSD swap device to identify bottlenecks. From the investigation we make two observations: (1) cores are idling and (2) the SSD swap device is operating below its performance limits. We use these finding to propose the simple solution of *overcommitting graph processing threads* to keep the cores busy and generate more parallel swap requests to get closer to the performance limit of the SSD (Section 5).

Overcommitting threads is able to provide up to 4.18x speedup over the baseline. In another case, overcommitting threads is able to improve the performance from 44.8% of the performance of the baseline (with 100% memory available on system) to up to 76.93%, improving performance by 32.13%). We also show that we are able to better use the CPU and extract more I/O throughput from the swap device by issuing more IO requests.

Finally, we conclude the paper and discuss future research based on two interesting questions raised from our investigation (Section 7). (1) Overcommitting threads can start to saturate a single SSD device. Can adding more SSDs provide more swapping performance? (2) If a system is equipped with a low-latency SSD drive, will the kernel become the new bottleneck of handling page-faults? Will this justify making radical changes to the kernel paging mechanism?

2 BACKGROUND

2.1 Graph processing

Graph processing extracts insights (such as neighbors, connectivity, popularity, etc.) from data structures that encode the relationships between data entities. Graphs consist of vertices (or nodes) and edges between vertices to form a network of interconnected nodes. An edge can be either directed or undirected, depending on the meaning of the underlying data. In addition to vertices and edges,

graphs typically hold further information, such as property values for each vertex, or the edge weights.

To store the graphs in memory, a common structure is the compressed sparse row (CSR) format which can also be used together with the compressed sparse column (CSC) format [4]. This format consists of arrays of `vertices` and `edges`. Each entry in the vertices array points to the portion of the edge array that contains the edges connecting from that specific vertex, and each entry in the edge array is the index of the vertex the edge connects to. In addition to these two arrays, the graph algorithm and/or the graph itself typically use additional arrays to hold per-vertex (e.g., page-rank scores) or per-edge data (e.g., per edge weights).

Graph algorithms traverse the graph by following edge connections between vertices and edges through the arrays above. While iterating through the `vertices`, and `edges` can be streaming accesses, the final access of using the edge to reference the target vertex is often a random access. As memory systems are optimized for locality and regular access patterns, these random accesses can cause the processor to stall for up to 80% of the execution time [24]. When working with large-scale graphs that do not fit in the system memory, these random accesses take even longer due to the virtual memory swapping needed to bring them into DRAM from the storage device.

2.2 Virtual memory paging is slow

Virtual memory enables the operating system to decouple the userspace memory allocations (virtual addresses) from the physical machine memory addresses (DRAM locations). This decoupling allows the operating system to move around the userspace data to different physical location, without causing any correctness issues in the user program.

For workloads that do not fit into the DRAM, this decoupling enables paging or swapping[1], whereby parts of the userspace memory allocations are moved out of the main memory (DRAM) and into secondary storage (swap devices such as SSDs). On access to swapped out userspace data, the kernel fault handler is called and the kernel finds out which page the user program was attempting to access and pages in that address, makes a valid mapping in the page table and returns the execution to the user program. From then on, the user program continues to execute as if nothing happened.

However, handling a page-fault is enormously slower than a direct DRAM access, with the latency dominated by reading the page from the disk. Spinning magnetic hard disks took milliseconds to read in data from the disk, while the latest NAND flash technology in solid state drives (SSDs) take at least 45μs [22]. Although the latency has reduced two orders of magnitude from spinning disks, SSDs still take much longer to access than DRAM. Thus, even with faster SSDs, virtual memory swapping leads to significant performance degradation while the program waits for the data to be returned form the SSD.

2.3 SSDs are highly parallel

SSDs have impressive I/O throughput. For example, the SSD that is used in this work can provide up to 7000 MB/s of sequential read performance. However, as discussed above, the access latency of random accesses is in the order of tens of microseconds, which is orders of magnitude longer than DRAM access latencies. This can cause significant slowdowns, as threads of execution that cause the page-fault need to wait more than 45μs for the page-fault to resolve.

However, SSDs are capable of handling many independent requests in parallel. While this does not reduce the latency of individual requests, it can, dramatically, increase the overall throughput of requests handled if the application can provide sufficient parallelism. The reason for this parallelism is due to the organization of the SSDs, which consists internally of parallel channels, ways, and dies, and SSD controllers that take care of queueing and re-ordering accesses to maximize performance. Indeed, the NVMe protocol specifically allows the SSD controller to process requests out-of-order [18], and the controller picks out requests to issue to idle parallel components to maximize throughput. As many large-scale graphs also have significant parallelism in their accesses, in this work we seek to see if we can adapt commodity systems and algorithms to match the supported parallelism of the SSDs to improve throughput and mitigate the performance loss of swapping.

3 METHODOLOGY

The commodity system used in our evaluations was an Intel i7-9700 processor (8 cores with 8 threads) with 128GB of DDR4 DRAM and a Samsung 980 Pro 1TB NVMe SSD connected via the M.2 port, used exclusively as a swap device. We used Ubuntu 20.04.6 with Linux kernel version 6.3.0 and gcc 10.5 with the `-O3` flag to compile the graph benchmarks. Unless otherwise noted, we allowed OpenMP use all the eight cores of the system to spawn 8 user threads.

Table 1: The vertex and edge sizes of each graph input along with the type of each graph input.

Graph Input	Vertices	Edges	Graph type
twitter	61M	1,468M	Power-law
web	50M	1,930M	Power-law
road	23M	57M	Mesh
kron	134M	2,111M	Power-law
urand	134M	2,147M	Uniform-random

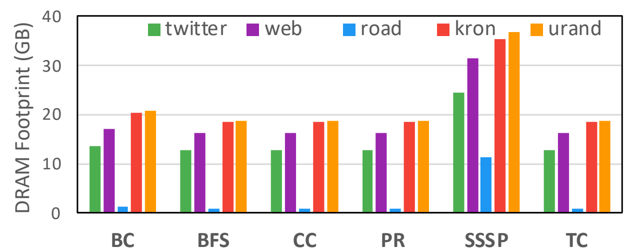

Figure 1: Memory footprint of each input graph when running a particular graph algorithm.

[1]In this work, we use the two terms, swapping and paging, interchangeably

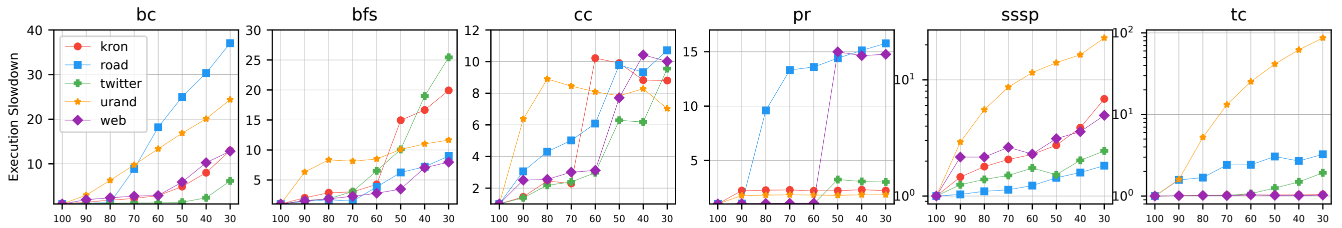

Figure 2: The relative execution time (y-axis) as less memory is available (x-axis, percent of footprint available in DRAM starting from fitting fully at 100%). The rest of the footprint will use the swap space on the swap device. Note that sssp and tc are plotted on log scales due to outliers.

The commodity implementation used in our evaluation was the GAP benchmark suite[7]. We evaluate all six graph algorithms, namely bc, bfs, cc, pr, sssp, and tc, provided in the suite, which are based on the most commonly evaluated graph kernels [5]. For each algorithm, we used the five input graphs offered by the GAP benchmark suite: twitter, web, road, kron, and urand. The first three input graphs are real-world graphs, and the latter two are synthetic graphs that model a power-law network and a uniformly random network, respectively. The input graphs are listed in Table 1 and their DRAM sizes in Figure 1. Prior to using the input graphs, we converted the input graphs into the serialized form (i.e. .sg or .wsg) to remove the graph building time from our experiments.

To control the amount of available DRAM for an application execution, we used the cgroups functionality to limit the total amount of page-cache, anonymous, file-backed and shared mappings that are consumed by a process. To understand the amount of memory needed by each benchmark, we first ran the algorithm/input pair without any memory limit and measured the peak memory usage of the program. These reference *footprints* are presented in Figure 1. We use the reference footprints to allow us to scale each benchmark's available memory by the same ratio (percentage).

4 CHARACTERIZING THE IMPACT OF SWAPPING

To evaluate the impact of large-scale graphs that do not fit into DRAM we executed six graph algorithms with five graphs each and reduced DRAM available to each benchmark. The resulting slowdown is shown in Figure 2, normalized to the runtime where the execution fits fully in DRAM (e.g., a value of 5 indicates a 5× slowdown).

These results show devastating performance losses as the percentage of the working set that fits in DRAM decreases. For example, the road graph running the bc algorithm slowed down to 37× when only 30% of the footprint fit in DRAM. Another interesting observation is that the tc algorithm does not suffer from performance loss (except for the urand graph). As the urand graph is a synthetic random graph, it would have the worst possible locality, suggesting that the tc algorithm is largely insensitive to paging as long as the graph has significant locality.

4.1 Kernel Behavior

Page-fault characterization: When the program attempts to access data that is not in the DRAM the CPU triggers a page-fault, which causes the OS to bring in the requested data from the swap device. To understand how the system time is being spent during execution with page faults, we used the perf record tool to sample and record what each core was doing.

While sampling the activity of all cores, we found that the conventional approach of using the HW performance counter cycles did not account for the idle periods. We attempted to mitigate this issue with the following two approaches, to no avail. First, we manually disabled the intel_idle driver [21] to prevent cores from entering low-power states, which could result in missing sampling events. Second, we addressed another potential source of energy optimization that could negatively affect the precision of our measurements: the CPU governor [8]. For some modes, the governor adjusts the frequency of cores based on the load of the core, in an effort to conserve energy.

We found that changing the CPU governor did not help and that disabling the idle driver (via setting the idle loop to poll) did increase the number of samples, but did not result in the expected number of samples as was measured by the kernel time accounting. This is because the Intel architectural performance counter for cycles does not count the halted cycles. Therefore, to sample the idle cycles of the cores, we decided to use the kernel SW event cpu-clock to sample both busy and idle cycles of each core.

We show a breakdown of the execution of the bfs algorithm processing the web graph with 30% of the memory footprint available as DRAM in Figure 3. The breakdown shows that of the full execution time (bottom gray bar), only 18% of the total CPU time is spent doing work in either the application or the OS (blue bfs bar), while the remaining 80% of the time the CPU cores are idle (gray Idle bar), waiting for I/O requests to complete. The idle time also includes the I/O time of reading the graph input data at the initialization phase of the workload; however, we believe this time should be relatively small due to the graph input loading benefiting from sequential access and readahead (i.e. next 32 page prefetching). The breakdown of the 18% time spent doing work, shows that 7.4% of the total time is spent on the user graph processing (green, User) while 10.7% is spent in kernel (yellow, System) as broken down in the top two bars. This clarifies the massive amount of time spent not doing application work due to paging: 10.7% in kernel work and 80% idle waiting for I/O.

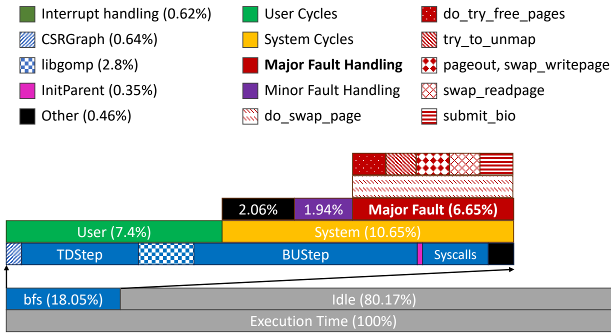

Figure 3: A Flamegraph of `bfs` running the `web` input when running with 30% of available memory. While everything else retains its proportional scale, the function `do_swap_page` is broken down to five key functions that are represented not by their scale, but rather by their order of execution, from left to right.

Out of the time spent in the kernel (System time, yellow), the majority, 6.7% of the total time, was spent servicing *major* page faults, page faults that require paging in from disk. Handling a major page-fault consists of invoking the `do_swap_page` which starts off by trying to get a free page to handle the page-fault (`do_try_free_pages`). If the free page allocation fails, a page is evicted based on the LRU chain and if the page needs to be written into the swap device the `swap_writepage` is invoked. Once the page to be evicted is ready to be unmapped, the `try_to_unmap` is invoked to free up a page to handle this page-fault. Using the free page, the `swap_readpage` is invoked to create a request to read in a page from the swap device. Finally, the `submit_bio` is invoked to issue the read request into the block I/O layer. While the time handling major page faults is the majority of the kernel time, 12× as much time is spent idling, waiting for the I/O to complete.

Readahead effect: An important optimization provided by the kernel is page-in *readahead*, which optimistically brings in the next few sequential pages on a page-in. On our machine, the default `readahead` degree is 32 for file-backed pages, meaning that up to 32 pages would be brought into the DRAM on each page-in. This value is 8 for pages that reside in swap space. Unless otherwise noted, the term `readahead` in our context refers to swap-backed `readaheads`. For `readahead` to be helpful, the application must have spatial locality of access across the prefetched pages. Some of our graph algorithms, such as page-rank (`pr`), have such an access pattern and benefit from `readahead`, as shown in Figure 4.

The left plot in Figure 4 shows the execution time breakdown with and without `readahead` for a selection of algorithms (`pr`, `bc`, and `bfs`) and graphs (`kron`, `road`, and `web`). For both `pr` executions, turning off `readahead` (right bars), resulted in noticeable increase in execution time. Most of the increase came from increase in `idle` time. This is because `readahead` was able to issue more *useful* `readaheads` into the SSD to make use of the idle SSD resource (more on this in Section 4.2). However, we also see that `readahead` helps `bc:web` less, and in the case of `bc:kron` and `bfs:web`, readahead hurts performance.

The middle graph in Figure 4 shows the normalized number of read/write requests sent to the SSD. We observe that the paging traffic is mostly reads, due to the fact that the majority of the graph data (vertices and edges arrays) stay unchanged, and each algorithm allocates other arrays to hold specific data that the algorithm uses.

We also observe that `readahead` batches up many potential read/write requests into larger I/O operations. The total number of pages transferred with and without `readahead` is similar (not shown in the graph for brevity). However, the number of I/O requests increases by nearly 3.4x when `readahead` is disabled. This demonstrates that `readahead` packs I/O requests into larger read/write requests.

The graph on the right of Figure 4 shows the total number of major page-faults normalized to the number of instructions executed. Major page-faults are the page-faults that require reading in pages from swap. This metric compares the intensity of the major page-faults across different algorithm/input pairs and also with and without `readahead`. By turning off `readahead`, we see large increases in major page-faults, especially for `pr`. We also see varying behavior within the same algorithm (`bc`) for different input graphs.

`Readahead` prefetches pages that are likely to be accessed in the future, changing a major fault into a minor page-faults when they are accessed. The time spent handling these minor page-faults is shown in Figure 3.

We find that the kernel `readahead` helps reduce the number of page-faults that require disk I/O (major page-faults), by prefetching in next sequential pages. This `readahead` is effective on some algorithms and the different input graphs can also affect the efficacy of the `readahead`.

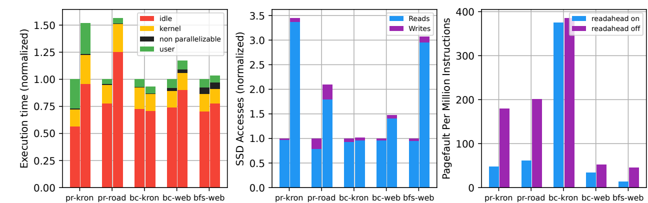

Figure 4: The effect of kernel `readahead` on the execution time (left graph), number of SSD I/O requests (middle graph), and the number of major page-faults per million userspace instructions (right graph). Left bars: `readahead` enabled. Right bars: `readahead` disabled.

Page-faults and swap traffic over time: To show the intensity of the page-fault throughout the application execution, Figure 5 shows the page-fault intensity (bottom) and virtual memory I/O to the swap device (top) of `bfs` running the `web` input over time. The first 19 seconds of the execution result in significant writes as the graph data is being read from the disk into the main memory, and due to limited memory, the loaded graph is then swapped out (written to) the swap device. However, after 20s, 99.7% of the swap device traffic is reads, as the graph data structure is not modified, and the graph processing allocates additional (and smaller) arrays to hold computed values. We believe these additional arrays correspond to the write traffic after the 20s point.

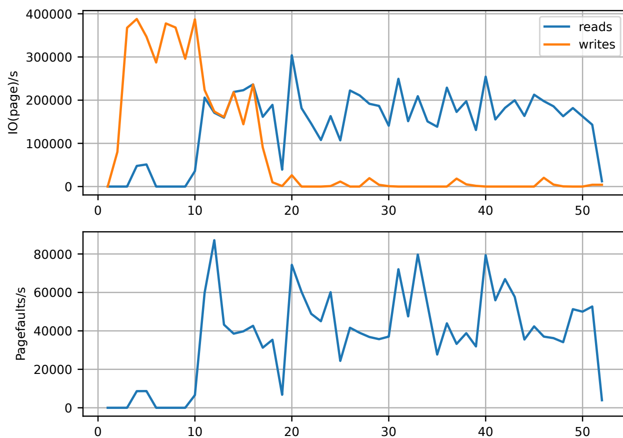

Figure 5: Reads/writes and page-fault intensity over time for `bfs` running `web` with 30% memory. Top graph shows 4KB Read/writes to swap device per second, and the bottom graph shows page-faults per second.

4.2 SSD Behavior

SSDs are highly parallel devices[12] composed internally of multiple independent channels, ways, and dies that can work independently to enhance the device I/O throughput through parallelism. Using the inherent parallelism of SSDs require the host system to generate and queue sufficient number of requests for the SSD and that the SSD has a sufficiently large queue to find enough independent requests to process in parallel. The SSD controller schedules the requests in the queue to maximize device performance.

To measure the capability of the SSD on our testbed, we used the `fio` tester [3] to measure the random access performance of the SSD. We attached to the SSD using SPDK and used the `SPDK plugin` for `fio` [1] to measure the raw performance. We also tried measuring the SSD performance using the `libaio` and `io_uring` APIs, but they were only capable of reaching 542K and 623K IOPS, respectively, below what the device is capable of. Recent work has also reported that kernel storage APIs do not extract the full performance of SSD drives [10]. Thus, we only present the SPDK measurements to show the raw SSD performance without associated kernel costs.

Figure 6 presents the random access performance of the SSD as a function of the number of requests in the SSD's request queue. We plot various types of random access: random writes, reads, read/write with 50:50 ratio and 99:1 ratio. The random read and random r/w with 99:1 ratio perform identically, likely due to the DRAM buffering of writes on the SSD. One key take-away from this measurement is that the SSD on our testbed performs at its peak performance when there are 128 or more requests in the request queue for all mixes except exclusively random writes, which saturates at 16.

From Figure 5, we can infer that during the first 10 seconds, the SSD could support a maximum throughput following the random write (blue) curve in Figure 6. From 10-20s, the SSD would exhibit throughput following the random read/write 50:50 (green), and from 20s until the end, the SSD would exhibit throughput similar to

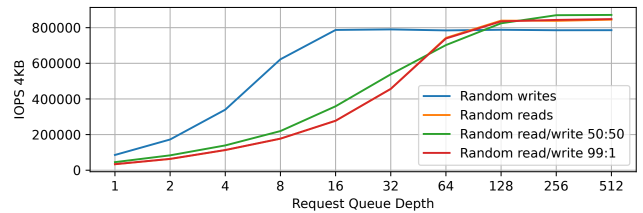

Figure 6: Random access (4KB) read performance of the NVMe SSD with varying request queue depth.

the read/write 99:1 (red) curve. However, from Figure 5, we can also see that the system is not coming close to saturating the SSD as it is averaging only 200k IOPS compared to the 400k IOPS the device is capable of. This suggests that the system is not maintaining 128 or more requests in the request queue and that there is the potential to obtain a swap throughput of nearly twice as many IOPS.

As we have seen that SSD throughput is highly sensitive to the number of queued I/O requests, we inspect the number of I/O requests outstanding for the swap device. Figure 7 shows the queue depth of the swap device for the range of algorithms and graphs. The key observation from this graph is that none of the algorithms are capable of filling the SSD's request queue depth to above 128. This also means that the system is likely not utilizing the SSD device to its full throughput capability.

Summary: With commodity algorithms on our commodity system, the CPU cores are spending significant amount of time idling waiting for swap requests and the SSD still has more throughput headroom due to too few I/O requests being generated. Therefore, as a simple remedy to this situation, we propose oversubscribing (overcommitting) the number of user threads onto physical CPU cores to both give the cores more work to do and generate more I/O requests.

5 OVERCOMMITTING TO KEEP THE CORES AND SSD BUSY

To keep the CPU cores working and to generate more requests to use more of the performance available on the SSD, we propose executing more user threads than the number of physical CPU cores, or overcommitting. In the base execution evaluated until now, the OpenMP runtime allocated one thread for each physical CPU core. However, we propose increasing the overcommitting ratio to make more work available for the CPU cores during page faults and to generate more page-faults to keep the SSD busy and use up the available SSD I/O throughput.

Performance of overcommitting threads: We present the effect of overcommitting threads in Figure 8. The figure presents the relative performance of the benchmarks as a function of available memory. (The red line $ocf = 1$ provides the same data as Figure 2, but inverted.) The additional lines ($ocf = 2, 4, 8, 16$) show the performance of the program when the overcommitting factor is increased to 2, 4, 8, and 16 threads per CPU core.

For most algorithms and input graphs, having a larger overcommit factor (i.e. purple, $ocf = 16$) shows the best performance. However, the `road` input graph shows a peculiar behavior for the

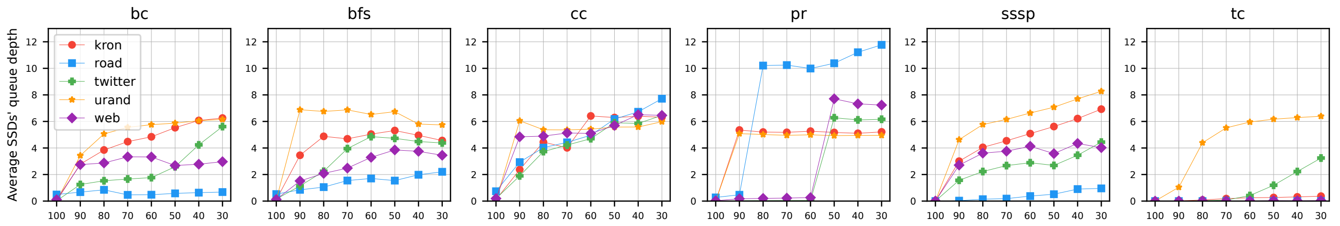

Figure 7: The average number of in-flight requests for the swap device request queue is shown on the y-axis as less memory is available (x-axis).

bc, bfs and sssp algorithms where overcommitting actually degrades performance, even when 100% memory is available. We have found that the road input graph is taking excessively long in the synchronization points of the benchmarks, which could be due to the high diameter of the input graph [6].

We find that the pr graph has an initial drop of performance and afterwards the performance drop levels off. Furthermore, for the kron, twitter, and urand benchmarks increasing the overcommit factor can reclaim a significant amount (up to 32.2%, kron) of the performance lost due to swapping.

In addition, there are many instances where the $ocf = 8$ and $ocf = 16$ show similar performance, namely kron:bc,cc, twitter:bc, urand:bc,cc, and web:cc and where higher overcommitment results in lower performance, such as kron:pr, road:bc,bfs,pr, twitter:pr, and urand:pr. We will discuss this behavior in the next section when we discuss the SSD behavior.

Performance improvement from overcommitting: Another visualization of performance is provided in Figure 9. This plot normalizes each ocf point along the x-axis to its corresponding $ocf = 1$ point. I.e., this plot shows the *performance improvement* of overcommitting threads compared to the baseline, for each memory availability point (x-axis). This plot would be useful to estimate the performance gains from overcommitting threads, when the system can only has a certain amount (e.g. $x = 50\%$) of the total required memory.

In cases where running graph analytics with graph inputs that surpass the DRAM size is inevitable, based on these experiments, we observe that overcommitting threads with a factor of 16 can provide up to 4.18x performance gain over the baseline execution (bc algorithm and web graph) with a geomean improvement of 1.65x.

5.1 Kernel Behavior

Effect of overcommitting on page-faults: Overcommitting threads implies that more threads of execution exist on the system at the same time. This leads to more page-faults occurring during a unit of time. Table 2 shows the increase in the number of page-faults per second when increasing the overcommit factor from 1 to 16. This demonstrates that overcommitting is effective in generating more page-faults, which will reduce idle time.

Effect of increased page-fault on execution time: Figure 10 shows the effect of an increased number of page-faults on execution time by measuring the execution time breakdown using the kernel time accounting.

Table 2: Increase in the number of page-fault per second due to overcommitting threads ($ocf = 16$) over the baseline ($ocf = 1$) when 30% of the data fits in the memory.

pr:kron	pr:road	bc:kron	bc:web	bfs:web
1.88x	1.86x	3.70x	4.29x	2.77x

With a decreasing amount of memory available on the system, the execution with no overcommitting of threads ($ocf = 1$, leftmost bar of each group of bars) shows large increases in the idle and kernel execution time. However, as the overcommitment factor increases (towards right in the group of bars) the idle time is significantly reduced. In the bc application processing the kron graph, the idle time is greatly reduced, however, the bfs algorithm processing the web graph leaves significant amount of idle time left. In both cases, we find that with less memory available the kernel time increases due to the increased number of page-faults and interrupts that need to be handled.

Another interesting observation is that with sufficient page-faults (generated by overcommitting threads), some workloads show that the kernel time is greater than the idle time, e.g., pr:kron, pr:road, and bc:kron. To improve these cases further, the kernel page-fault handling logic would need to be improved to further improve performance.

5.2 SSD Behavior

Overcommitting threads generates more page-faults, resulting in more I/O requests being queued into the SSD request queue. Figure 11 presents the average request queue length for the swap device as the available memory decreases (x-axis), and with varying overcommitting factors (different lines). There is a clear increasing trend of the number of requests in the IO queue with a larger overcommitting factor.

For some workloads bc:kron, bc:web, and bfs:web, the increase in the number of requests on the queue translate to performance improvement. The increase in IO requests on these workloads show a similar shape to the performance improvements of these workloads in Figure 9. This suggests that the workloads are able to extract more performance out of the swap device and this leads to the performance gains.

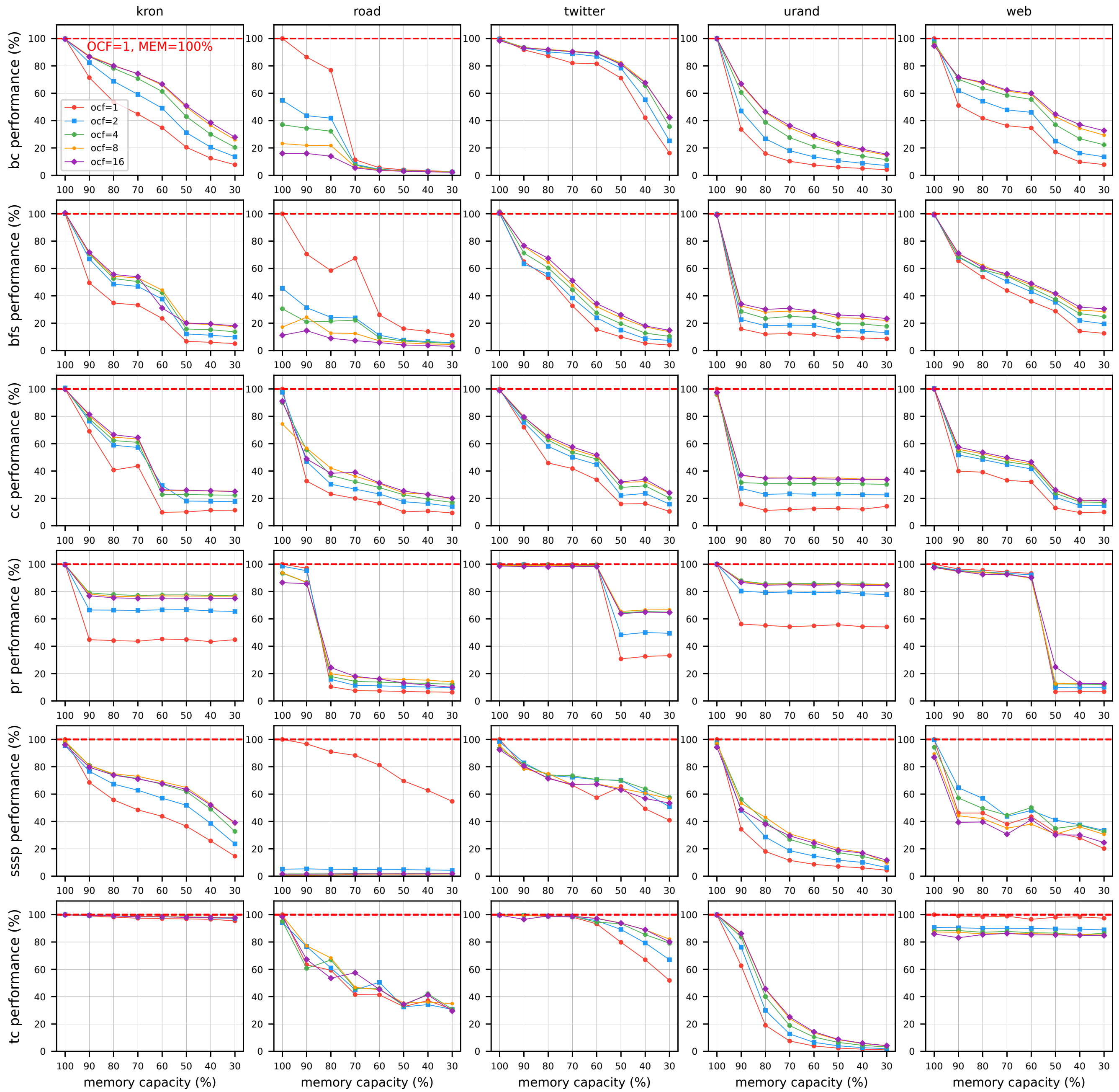

Figure 8: The performance relative to fitting completely in DRAM (percent of max performance on y-axis) as a function of available memory (x-axis). Varying thread overcommitting factors (ocf 1 through 16) are plotted. Each row represents a particular algorithm and each column a particular graph. We plot the *performance* relative to the reference execution (no overcommit, 100% memory in DRAM.)

Other workloads `pr:kron` and `pr:road` show slightly different results. The IO request queue depth increases with increasing overcommitting factors. However, for the overcommitting factor of 16, (purple line), we find that for both workloads, the performance drops as compared to smaller overcommitting factors (e.g. $ocf = 8$). For `pr:kron` this can be attributed to the increased `kernel` time going from $ocf = 8$ to $ocf = 16$ as shown in Figure 10. For this workload, overcommitting beyond $ocf = 4$ has no more effect on reducing the `idle` time. Instead, the larger number of threads increases the burden on the kernel and results in increased `kernel`

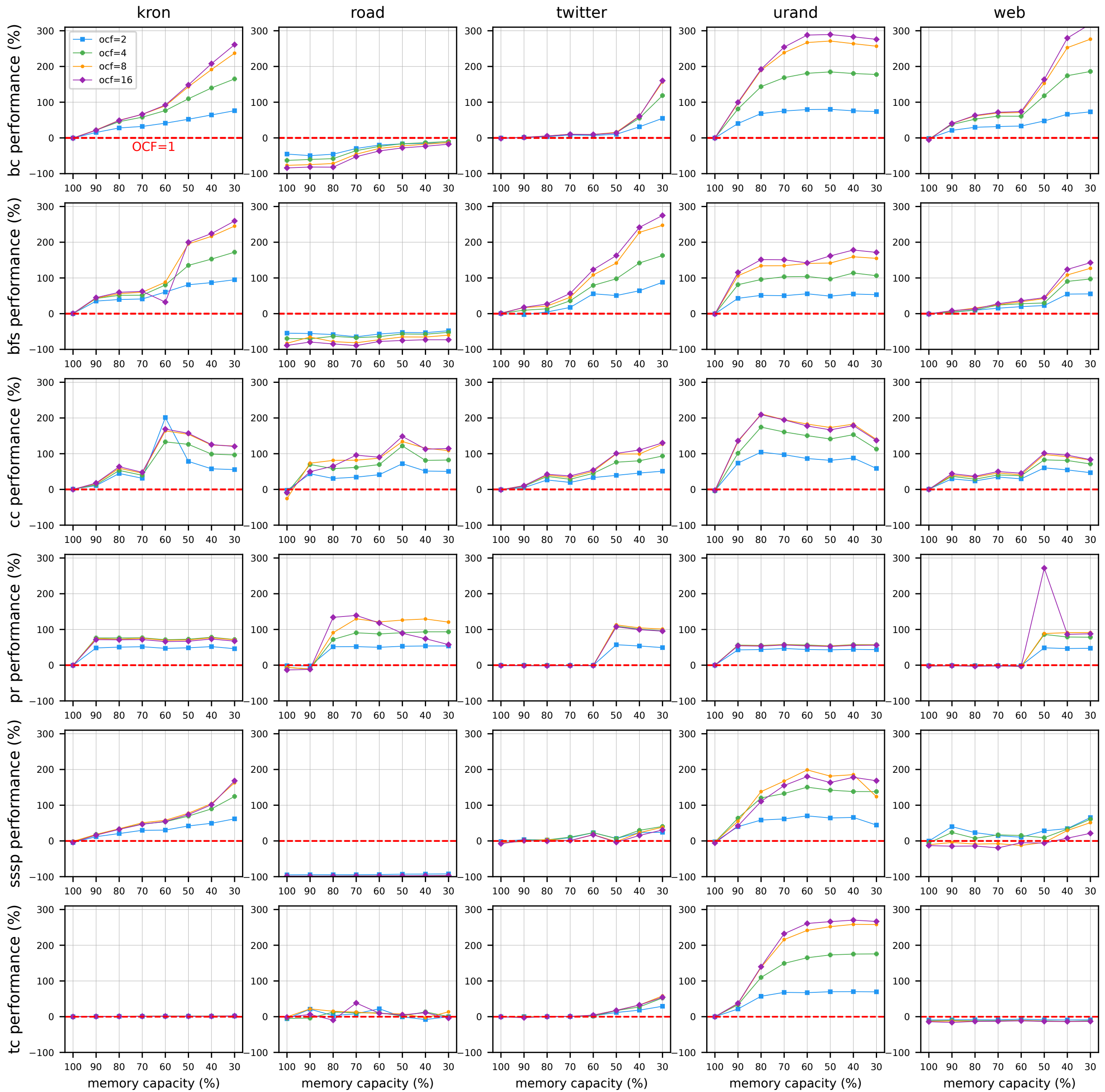

Figure 9: The performance improvement (y-axis, higher is better) from overcommitting normalized to no overcommitting at each available memory point. The red dashed line provides the baseline performance ($ocf = 1$) for the corresponding available memory (x-axis).

time, resulting in longer execution. For the case of `pr:road`, we find that the execution with overcommitting factor of 16 results in increased number of write requests (7.6% compared to $ocf = 8$), but similar number of pages written (0.3% difference) signifying that more random write requests were generated. Finally, the `idle` time increased by 2.1x as compared to $ocf = 8$. Based on the above information we speculate that the smaller chunks of random-write (as opposed to writing larger chunks of data) negatively affected the SSD performance and increased read latencies for swapping in data, resulting in longer system `idle` time.

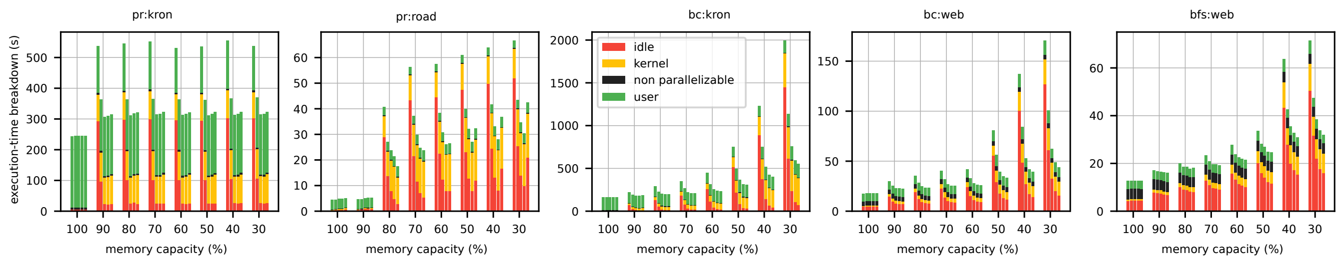

Figure 10: The execution time breakdown of algorithm:input with varying level of available memory (x-axis), and each bar representing varying overcommitting factor from 1× (leftmost bar) to 16× (rightmost bar).

5.3 Summary

Overcommitting threads improves performance by keeping the core busy with meaningful work by scheduling in another thread when a thread blocks due to a page-fault. This results in the system issuing more page-faults and due to the SSD being capable of servicing the increased number of I/O request concurrently, the system idle time is reduced, resulting in a more efficient graph execution. The reduction in idle time results in the performance improvements of higher overcommitting factors in Figure 8 and Figure 9.

6 RELATED WORK

Out-of-core processing: There have been numerous works on graph processing that have focused on out-of-core processing [13, 14, 16, 19, 20, 23, 25, 26]. Such efforts have been proposed to overcome the challenge of the graph data not fitting in memory, and the need to store such data in slower media such as SSDs, fabric-attached-memory, or tiered memory. FAM-Graph used graph processing characteristics to optimize memory tiering in fabric-attached-memory settings [23]. GridGraph [26], GraphChi [13] and Mosaic [14] design efficient graph blocking methods to improve the usage of data brought into the main memory. X-Stream [19] proposes an edge-centric computation algorithm to make better use of the sequential bandwidth of the storage device. XPGraph [20] optimizes the graph accessing to optimize for the characteristic of the tiered memory media. GraphSSD[16] proposed implementing a graph framework into the SSD to offload many graph related functionality away from the core and into the storage. The optimizations proposed in these works with regards to making better use of the characteristics of the storage media through customized hardware and algorithms could be relevant to our work on commodity algorithms and systems. FlashGraph [25] controls disk access from the userspace and overlaps computation with IO. It also differentiates data to be stored in memory and disk. Their approach of moving disk access into the userspace is relevant for our future research direction where we need to cut down on the page-fault handling time in the kernel. The largest difference between the prior work and our work is that we use a commodity graph processing algorithm and commodity hardware and operating system and investigate the system and SSD behavior to understand the bottlenecks of the entire system.

Kernel side improvements: Infiniswap [11] was a proposal to disaggregate memory through a cluster of RDMA attached servers, where data that does not fit in the memory is swapped out over RDMA into the DRAM of other nodes in the cluster. This approach is similar to our approach in that it runs unmodified benchmarks and hardware, but differs in that it swaps out data onto DRAM on other machines of the cluster. Fastswap [2] proposed a faster kernel swapping mechanism by identifying and fixing inefficiencies in the readahead and can be applied to our work. Manocha et al. studied the implication of large pages on graph processing and efficient use of large-pages using application knowledge [15]. The effect of large pages and swapping together is a research direction that needs to be investigated in the future.

7 CONCLUSION AND FUTURE RESEARCH

This paper investigated the effect of processing large-scale graphs that do not fit into DRAM using commodity algorithms and systems, which cause graph data to be swapped to a SSD device by the virtual memory system. Due to the long access latency of the SSD, this swapping resulted in severe performance penalties. To address this, we characterized the behavior of a range of commodity applications on our commodity system and identified that both the system cores and SSD were being underutilized. To better utilize the compute and storage resources, we propose a simple solution to overcommit threads onto physical cores. As a result, instead of physical cores idling when a thread of execution triggers a page-fault, another thread can be scheduled onto the core to do more meaningful work and generate more page-faults. This also increased number of page-faults sent to the SSD, which was potentially beneficial as we had observed that the SSD had a significant amount of headroom remaining in its random-access throughput. Thus, overcommitting threads makes better use of the CPU cores to keep doing meaningful work, while issuing more swapping request to the SSD to extract more swapping bandwidth for the system.

In this work we have demonstrated that we can achieve a speedup of up to 4.18x for large-scale graph processing that does not fit into DRAM on a commodity system with commodity algorithms by understanding and adjusting the thread overcommitting factor. Across our range of benchmarks, we show a geomean performance improvement of 1.65x. However, 3 of the benchmarks showed a performance decrease of 80.7%, indicating that while overcommitting is broadly helpful, there are cases where it does not benefit performance.

For future work, we plan on investigating the system performance when multiple swap devices are available on the system.

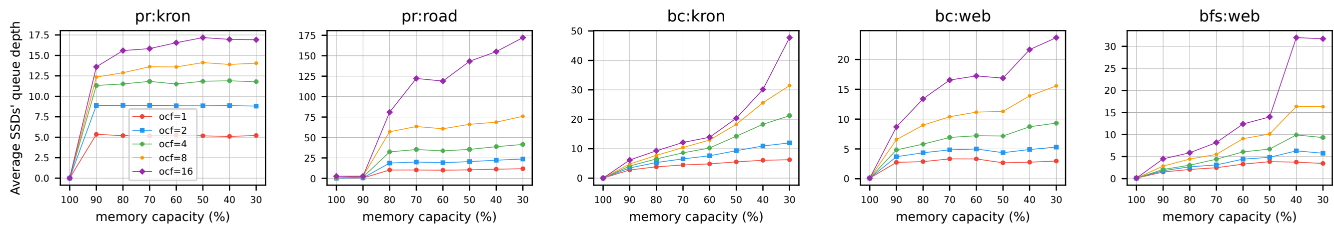

Figure 11: The average number of in-flight requests on the swap device request queue with decreasing available memory (x-axis) and varying overcommitting factors (different lines).

Our results have shown that performance peaks out at around $ocf = 16$. However, even at these overcommitting factors we still see significant amount of idle time, which we believe is due to IO throughput limitation of our single swap SSD. We will be investigating how more SSDs, and the I/O throughput they provide can help with the system swapping performance for graph applications. With regards to the kernel overhead, a question that needs further investigation is the scalability within the kernel shared data structures and routines for page faulting. This may be of particular interest when coupled with other storage technologies that provide even lower latency, making the relative impact of the kernel overhead greater.

ACKNOWLEDGMENTS

This work was supported by the Electronics and Telecommunications Research Institute (ETRI) grant funded by the Korean government (grant No. 23ZS1300), the Knut and Alice Wallenberg Foundation through the Wallenberg Academy Fellows Program (grant No. 2015.0153), the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant No. 715283), and the Swedish Research Council (grant No. 2019-02429).

REFERENCES

[1] 2023. FIO plugin. https://github.com/spdk/spdk/blob/master/examples/nvme/fio_plugin/README.md
[2] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can Far Memory Improve Job Throughput?. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 14, 16 pages. https://doi.org/10.1145/3342195.3387522
[3] Jens Axboe. 2023. Flexible I/O Tester. https://github.com/axboe/fio
[4] Vignesh Balaji, Neal Crago, Aamer Jaleel, and Brandon Lucia. 2021. P-OPT: Practical Optimal Cache Replacement for Graph Analytics. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 668–681. https://doi.org/10.1109/HPCA51647.2021.00062
[5] Scott Beamer. 2016. *Understanding and Improving Graph Algorithm Performance.* Ph. D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-153.html
[6] Scott Beamer, Krste Asanovic, and David Patterson. 2015. Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server. In *2015 IEEE International Symposium on Workload Characterization*. 56–65. https://doi.org/10.1109/IISWC.2015.12
[7] Scott Beamer, Krste Asanović, and David Patterson. 2017. The GAP Benchmark Suite. arXiv:1508.03619 [cs.DC]
[8] Dominik Brodowski, Nico Golde, Rafael J. Wysocki, and Viresh Kumar. 2017. CPU frequency and voltage scaling code in the Linux(TM) kernel. https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt
[9] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One Trillion Edges: Graph Processing at Facebook-Scale. *Proc. VLDB Endow.* 8, 12 (aug 2015), 1804–1815. https://doi.org/10.14778/2824032.2824077
[10] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. 2022. Understanding Modern Storage APIs: A Systematic Study of Libaio, SPDK, and Io_uring. In *Proceedings of the 15th ACM International Conference on Systems and Storage* (Haifa, Israel) *(SYSTOR '22)*. Association for Computing Machinery, New York, NY, USA, 120–127. https://doi.org/10.1145/3534056.3534945
[11] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 649–667. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu
[12] Myoungsoo Jung. 2019. SSD Architecture and System-level Controllers. https://ocw.snu.ac.kr/sites/default/files/NOTE/Week16.pdf
[13] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, 31–46. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/kyrola
[14] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a Trillion-Edge Graph on a Single Machine. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 527–543. https://doi.org/10.1145/3064176.3064191
[15] Aninda Manocha, Zi Yan, Esin Tureci, Juan Luis Aragón, David Nellans, and Margaret Martonosi. 2022. The Implications of Page Size Management on Graph Analytics. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*. 199–214. https://doi.org/10.1109/IISWC55918.2022.00026
[16] Kiran Kumar Matam, Gunjae Koo, Haipeng Zha, Hung-Wei Tseng, and Murali Annavaram. 2019. GraphSSD: Graph Semantics Aware SSD. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 116–128. https://doi.org/10.1145/3307650.3322275
[17] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. 2010. Introducing the graph 500. *Cray Users Group (CUG)* 19 (2010), 45–74.
[18] NVM Express, Inc. 2022. NVM Express Base Specification 2.0c. https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-2.0c-2022.10.04-Ratified.pdf
[19] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: Edge-Centric Graph Processing Using Streaming Partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 472–488. https://doi.org/10.1145/2517349.2522740
[20] Rui Wang, Shuibing He, Weixu Zong, Yongkun Li, and Yinlong Xu. 2022. XP-Graph: XPline-Friendly Persistent Memory Graph Stores for Large-Scale Evolving Graphs. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1308–1325. https://doi.org/10.1109/MICRO56248.2022.00091
[21] Rafael J. Wysocki. 2020. intel_idle CPU Idle Time Management Driver. https://docs.kernel.org/admin-guide/pm/intel_idle.html
[22] Jong Yuh, Jason Li, Heguang Li, Yoshihiro Oyama, Cynthia Hsu, Pradeep Anantula, Stanley Jeong, Anirudh Amarnath, Siddhesh Darne, Sneha Bhatia, Tianyu Tang, Aditya Arya, Naman Rastogi, Naoki Ookuma, Hiroyuki Mizukoshi, Alex Yap, Demin Wang, Steve Kim, Yonggang Wu, Min Peng, Jason Lu, Tommy Ip, Seema Malhotra, David Han, Masatoshi Okumura, Jiwen Liu, John Sohn, Hardwell Chibvongodze, Muralikrishna Balaga, Aki Matsuda, Chakshu Puri, Chen Chen, Indra K V, Chaitanya G, Venky Ramachandra, Yosuke Kato, Ravi Kumar, Huijuan Wang, Farookh Moogat, In-Soo Yoon, Kazushige Kanda, Takahiro Shimizu, Noboru Shibata, Takashi Shigeoka, Kosuke Yanagidaira, Takuyo Kodama, Ryo Fukuda, Yasuhiro Hirashima, and Mitsuhiro Abe. 2022. A 1-Tb 4b/Cell 4-Plane 162-Layer 3D Flash Memory With a 2.4-Gb/s I/O Speed Interface. In

2022 IEEE International Solid- State Circuits Conference (ISSCC), Vol. 65. 130–132. https://doi.org/10.1109/ISSCC42614.2022.9731110

[23] Daniel Zahka and Ada Gavrilovska. 2022. FAM-Graph: Graph Analytics on Disaggregated Memory. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 81–92. https://doi.org/10.1109/IPDPS53621.2022.00017

[24] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Matei Zaharia, and Saman Amarasinghe. 2016. Making Caches Work for Graph Analytics. arXiv:1608.01362v3 [cs.DC]

[25] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. 2015. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. USENIX Association, Santa Clara, CA, 45–58. https://www.usenix.org/conference/fast15/technical-sessions/presentation/zheng

[26] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 375–386. https://www.usenix.org/conference/atc15/technical-session/presentation/zhu