xHeap: Transparent Hugepage Optimizations for Memory Offloading

Ioannis Malliotakis[*†]
FORTH-ICS, Greece
jmal@ics.forth.gr

Anastasios Papagiannis
Isovalent, Greece
anastasios.papagiannis@isovalent.com

Manolis Marazakis[†]
FORTH-ICS, Greece
maraz@ics.forth.gr

Angelos Bilas[*]
FORTH-ICS, Greece
bilas@ics.forth.gr

Abstract

Increasing dataset sizes require larger application heaps. With recent technology limitations in DRAM scaling, modern datacenters resort to using the virtual memory mechanisms, i.e., the swapper (*swap*), to transparently extend application heaps over fast, block storage, such as NVMe SSDs or compressed DRAM. At the same time, larger heaps incur increased TLB miss and page fault handling CPU costs, necessitating the use of hugepages for their reduction, now viable given the advent of lower latency and higher throughput storage devices.

However, the use of Transparent Hugepages (THP) combined with memory offloading in Linux currently has significant limitations. Hugepage utilization behaviour is overly aggressive in committing memory, and too coarse-grained for memory offloading purposes. Additionally, hugepage promotions lack the responsiveness and concurrency necessary to provide timely hugepage benefits to offloaded applications.

To address these issues, we design *xHeap*, an alternative file-backed *mmio* path for the Linux kernel, which (a) decouples the use of memory for device-backed heaps from other functions in the kernel, (b) transparently supports regular pages and hugepages at fine granularity, and (c) provides concurrent, asynchronous promotions to adapt to runtime application behaviour.

We implement *xHeap* as a loadable module in Linux to minimize kernel modifications, maintaining complete interoperability with the kernel memory management subsystem, and evaluate it with a variety of workloads. We find that *xHeap* reduces dTLB miss cycles by up to 15×, and improves performance by up to 76%, when offloading 15-25% of application memory, compared to *swap* with THP.

ACM Reference Format:
Ioannis Malliotakis, Anastasios Papagiannis, Manolis Marazakis, and Angelos Bilas. 2026. *xHeap*: Transparent Hugepage Optimizations for Memory Offloading. In *6th Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems (CHEOPS '26), April 27–30, 2026, Edinburgh, Scotland Uk.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3805687.3806256

[*]Department of Computer Science, University of Crete, Greece
[†]Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS), Greece

1 Introduction

The current proliferation of data-intensive applications, with growing dataset sizes, places increasing strain on the memory (DRAM) of modern datacenter servers [7, 10, 37, 40, 64]. However, DRAM capacity scaling is limited due to technological challenges. Additionally, DRAM cost, both monetary and in power, represents an increasing percentage of overall datacenter expenses [64]. These factors motivate approaches to reduce DRAM needs by offloading portions of the application heap. Meta and Google use Transparent Memory Offloading (TMO) [64] and software-defined far memory [17] respectively, to offload up to 20% of application heaps over fast block-addressable devices, including NVMe SSDs and compressed DRAM, e.g., via *zram* [36]. At the same time, growing application heaps place increasing stress on the host CPU's TLB, incurring increased overhead in TLB misses on virtual address translations. Recent performance analysis on in-memory heaps at Google [25] and Meta [68] shows that up to 20% of total CPU cycles in the datacenter are spent on handling TLB misses. As such, the use of hugepages has been examined to improve host CPU efficiency. Hugepages are hardware-supported pages larger (2MB on x86_64, 1GB also supported) than the base (regular) page size. Given favourable access patterns, hugepages improve TLB coverage and reduce kernel processing costs for page faults and page batching operations.

In this work, we argue that hugepages should be utilized in memory offloading setups, to improve CPU efficiency with regards to TLB miss handling, while reducing application DRAM requirements. Recent advancements in device technology, such as high-throughput NVMe [1] or CXL [49] SSDs, which significantly reduce I/O overheads [2, 27, 65], further support this notion. However, hugepages cannot be used as a drop-in replacement for regular pages in the presence of memory offloading. Applications may exhibit sparse, fine-grained accesses over portions of their heap, resulting in low or high hugepage coverage [4, 6, 32]. For such applications,

using hugepages indiscriminately for the full heap throughout execution incurs memory bloat [29, 44, 70], which in turn leads to excessive I/O for device-backed heaps. Furthermore, application behaviour may change over time [41, 48], motivating the demotion of hugepages to regular to improve memory efficiency, or the timely promotion of an address range to a hugepage to improve CPU efficiency. The latter involves the replacement of all regular pages backing an aligned, hugepage-sized virtual address range, and their page table entries, with a hugepage and single hugepage table entry. Current OS-level support for hugepages combined with memory offloading remains problematic in this regard.

The current Linux mechanisms deployed for application-transparent memory offloading and hugepage utilization, are *swap* and Transparent HugePages (THP) respectively. Their combination, while possible thanks to kernel support for swapping out entire hugepages [26], leads to significant performance issues related to two aspects of hugepage utilization. **(1) Aggressive, coarse-grained hugepage fault handling:** The Linux kernel provides a single pair of flags (*MADV_(NO)HUGEPAGE*) for use by hugepage-aware applications (e.g., via *madvise*). These flags are set at coarse Virtual Memory Area (VMA) granularity, which prevents the selection of fine-grained virtual address ranges for hugepages, as the kernel imposes a limit of 64K VMAs per process. Without explicit application hugepage awareness, the decision to use hugepages boils down to a single, system-wide always/never switch. When enabled, the kernel aggressively serves page faults with hugepages so long as enough free memory is readily available, which can lead to thrashing with excessive swapping, and thus increased I/O traffic. **(2) High-latency, low-throughput hugepage promotions:** On memory pressure, page faults are served with regular pages, and a background kernel daemon, *khugepaged*, handles hugepage use. *khugepaged* linearly scans process address spaces and aggressively promotes virtual address ranges with at least one regular page table entry to hugepages. Promotion throughput is thus limited to *khugepaged*'s throughput. At the same time, *khugepaged* avoids promoting ranges with swapped out pages above a threshold [1] (64 by default) to limit I/O traffic, and sleeps [2] for 10s between scans to minimize CPU cost. Therefore, *khugepaged* fails to provide for timely hugepage promotions in the context of memory offloading.

To address these issues, we present *xHeap*, an alternative memory-mapped I/O path in the Linux kernel. *xHeap* provides transparent regular and hugepage support for device-backed heaps. It is implemented as a Linux kernel module to facilitate deployment while minimizing kernel modifications, requiring only a single added *if-check* in the kernel page fault path. The main design decisions and mechanisms of *xHeap* are:

1. **Decoupled page management:** *xHeap* uses separate preallocated regular and hugepage pools to minimize hugepage allocation stalls, prevent hugepage-incurred external memory fragmentation, and decouple itself from the Linux kernel page cache and page allocation, reclamation, and writeback paths.
2. **Fine-grained hugepage support:** *xHeap* augments its page fault path towards the fine-grained mixture of regular pages and hugepages. To do so, it enables hugepage selection on a page fault basis, contrary to the Linux coarse-grained hugepage utilization approach.
3. **Concurrent and asynchronous hugepage promotions:** To remain responsive to application pattern changes, and hide hugepage I/O stalls, *xHeap* implements asynchronous hugepage promotions. Unlike Linux THP, *xHeap* schedules promotions based on application page fault patterns, and utilizes the increased I/O capabilities of newer offloading mediums by allowing for multiple concurrent promotions.

We evaluate the performance of *xHeap* compared to *swap*, both on its own and combined with THP to allow for hugepage use. Although the choice of backing device technology is orthogonal to *xHeap*, we choose to use NVMe SSDs as they offer abundant capacity at low cost/GB, high throughput, and high concurrency. We use data-and-compute intensive workloads, namely data deduplication [9], support vector machine (SVM) [31] and particle transport simulation [61], offloading 15-25% of their max RSS, around the range of 20% as reported by TMO and software-defined far memory.

We find that, compared to *swap* with THP, *xHeap* with asynchronous hugepage promotions reduces data TLB miss cycle percentage by 2.5-15×, and improves overall application performance by up to 76%.

2 *xHeap* Design

xHeap is designed as an alternative *mmap* path for file-backed memory mappings, in the form of a Linux kernel loadable module. Figure 1 shows the placement of *xHeap* within the Linux kernel. *xHeap* may operate directly over a raw block storage device or over an existing filesystem. In the latter case, we use *wrapfs* [67] to intercept Linux VFS operations and redirect them to the underlying filesystem as appropriate. It is important to stress that *xHeap* is designed to be independent of the virtual memory mechanisms (regular and hugepage) of the kernel, however, remains fully interoperable with the existing kernel paths. This has three significant benefits: (a) it allows a server to simultaneously host applications with regular and device-backed heaps, (b) it reduces interference for applications that use device-backed heaps with respect to native kernel virtual memory support, and (c) it allows deploying *xHeap* with almost no kernel modifications, a significant challenge for the types of functions

[1] `/sys/kernel/mm/transparent_hugepage/khugepaged/max_ptes_swap`
[2] *Ibid.*/`scan_sleep_millisecs`

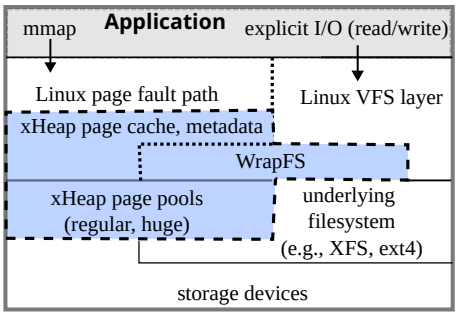

Figure 1. Placement of *xHeap* components within the Linux kernel. *xHeap* boundaries denoted by dashed lines. Dotted lines denote component interaction.

required and performed by the virtual memory management path.

xHeap is entirely transparent to applications, as long as the memory allocation library uses file-backed mappings to allocate virtual memory regions. We use *libvmmalloc* [51], but any other library can make use of *xHeap* via file-backed mappings. Once the *xHeap* module is loaded, any file-backed mapping established over the *xHeap*-managed device/files utilizes *xHeap*'s custom page fault, allocation, reclamation, and writeback paths. For completeness, *xHeap* also implements traditional file-based operations beyond *mmap*, such as buffered and direct `read`/`write` system calls.

Next, we present the main design decisions and challenges for using hugepages in device-backed heaps.

2.1 Decoupled page management

Similar to previous works [47, 62], *xHeap* reserves a pool of pages for use by applications. This allows *xHeap* to reduce interference with the kernel and implement its own page allocation and reclamation paths without modifying core components of the kernel memory management subsystem. The *xHeap* page pool is global, and shared by all applications extending their heap over *xHeap*. Its size is specified when loading the *xHeap* module as a parameter, and semantically corresponds to the total DRAM budget allotted to *xHeap* for heap extension. To minimize interaction with the Linux page allocator, *xHeap* allocates pages equal to the total pool size from the kernel on module initialization.

Preallocating pages for the page pool presents a design decision on how to split DRAM between regular pages and hugepages. A fully flexible pool, similar to the kernel buddy allocator, would initially allocate all memory in hugepage-sized chunks and then use these chunks either as single-unit hugepages or split them and use parts of them as regular pages as needed. However, splitting and partial reallocation of hugepages incurs external memory fragmentation [39]. As fragmentation increases over time, Linux requires costly compactions [12, 43] to create contiguous free memory for new hugepages. Compactions cause long application stalls [20, 45, 68] and may even fail to create hugepages [39], limiting the associated benefits.

xHeap uses two separate page pools, one for regular pages and one for hugepages. There is no page movement between the pools, and each pool makes separate decisions for page reclamation and writeback (Figure 2). This guarantees that hugepages are always available via reclamation within the hugepage pool without compactions. A set of kernel threads is responsible for page writeback in each pool. The page replacement policy is an approximation of CLOCK, with separate clock hands in per-core page queues, to minimize synchronization costs on heavy reclaim activity. Statically dividing the page pool may hinder performance for workloads with massively varying access patterns under *xHeap*. However, we find that it suffices to dedicate most of the *xHeap* DRAM budget to hugepages to maximize hugepage performance benefits, while leaving a small portion of regular pages to serve sparse accesses. Augmenting the *xHeap* page pool with compaction to allow for pool resizing is orthogonal to the current design. We note that resizing and compactions will also require an accompanying policy for determining the size of each pool. We leave the exploration of these issues as future work.

2.2 Fine-grained hugepage support

Fine-grained hugepage support requires the ability to handle multiple page sizes during application page faults. For this reason, *xHeap* provides a custom page fault path (Figure 2), which we discuss next.

xHeap initially registers two page fault handlers, one for regular pages and one for hugepages, and associates them with *xHeap* mappings. The kernel may then invoke either of these handlers as part of the common page fault path, depending on the state of the faulting address. The kernel first attempts to invoke the hugepage fault handler if the address maps to a hugepage table entry or when no page table entry exists. The *xHeap* hugepage fault handler uses a **policy** to choose whether to serve the page fault with a hugepage, or fall back to the regular fault handler and proceed with a regular page.

Selecting the page type to serve a page fault is critical in balancing the benefits of hugepages with the associated memory and I/O cost. *xHeap* fault handlers mix regular and hugepages within each VMA, transparently to applications and without incurring VMA split costs, which stall *all* application page faults regardless of faulting address. For hugepage-aware applications, the existing hugepage behaviour flags act as hints to the *xHeap* fault handlers. Previous works for in-memory and memory tiering scenarios, have approached the problem of selecting address ranges to back with hugepages with different methods, e.g., page access coverage metrics [29, 44], memory access sampling [4, 32], or page table scanning [34]. Any of these techniques may be

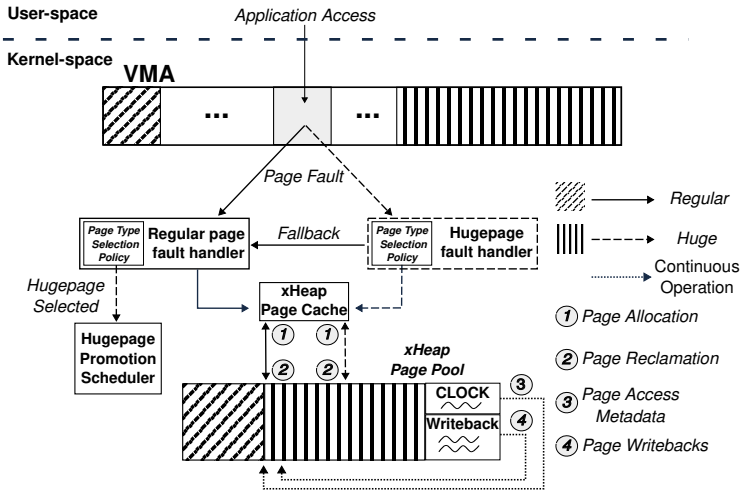

Figure 2. *xHeap* components in the page fault path.

used as policies for *xHeap*. In our work we focus on the required mechanisms to support such policies, and evaluate a simple page selection policy, discussed next in the context of hugepage promotions, and leave more sophisticated policies as future work. After selecting the page type, both handlers follow similar paths, with regards to page allocation, possible reclamation and I/O, and installing the page table entry.

2.3 Hugepage promotions

Application memory access patterns may change over time. As a result, sets of regular pages may need to be promoted to a hugepage or a sparsely accessed hugepage demoted to regular pages. We note that the demotion of a hugepage in *xHeap* is relatively straightforward: if a hugepage is rarely accessed, it is likely to be reclaimed and the address range partially refaulted with regular pages on subsequent accesses. However, providing application performance benefits through hugepage promotions presents significant challenges both in mechanism and policy, which we explore next.

Mechanism-wise, the challenge is for hugepage promotions to be timely enough so that applications enjoy hugepage benefits with minimal stall. Having a single thread, like *khugepaged*, scan terabyte-scale virtual address spaces for multiple processes diminishes the possibility that the promotions performed actually correspond to the current application memory access patterns. As such, *xHeap* schedules hugepage promotions at page fault time, driven by the page type selection policy. *xHeap* defers promotions outside page fault context, to allow the application to continue with execution and minimize stall, once the page fault has been served with a regular page. Even so, having a single thread process scheduled promotions limits scalability, especially for multithreaded applications, where multiple promotions may be concurrently scheduled to different parts of the address space. To that end, *xHeap* uses kernel workqueues [35], and places work items corresponding to hugepage promotions in the high priority system workqueue. This allows for multiple promotions to be processed concurrently, unlike with THP. To prevent redundant work from promotions scheduled by concurrent page faults in the same aligned region, *xHeap* uses a sharded promotion reservation list, containing descriptors for pending promotions.

Policy-wise, the decision to schedule a promotion reflects the complexities of the overall page type selection policy. Ideally, that is with ample memory and no regular pages in range, serving page faults directly with hugepages precludes the promotion cost and provides applications with immediate hugepage benefits. Otherwise, when a promotion is necessary to install a hugepage, its scheduling must balance the associated benefits and costs. Scheduling a promotion too early may lead to memory bloat and increased I/O traffic, while scheduling a promotion too late diminishes the potential for hugepage-derived TLB performance benefits. *xHeap* provides a page type selection policy, which balances the aggressive Linux hugepage fault behaviour, with Quicksilver's [70] modified FreeBSD hugepage promotion behaviour. So long as hugepages are available without reclamation, *xHeap* serves page faults with hugepages when possible (i.e., size and alignment restrictions hold, and no regular pages in range). Otherwise, *xHeap* schedules an asynchronous hugepage promotion once 64 regular pages within a candidate hugepage range have been faulted in, which indicates that the range is likely to benefit from a hugepage.

3 Evaluation Methodology

Our experimental testbed is a dual-socket server with two Intel Xeon E5-2630 CPUs, operating at 2.4GHz. Each socket has 8 cores with 2 threads per core for a total of 32 threads. The frequency scaling governor is set to "performance". The server is equipped with 256GB of DDR4 DRAM operating

Configuration	Abbreviation
swap	**S**
swapTHP	**ST**
xHeap	**X**

Table 1. Evaluated configuration abbreviations.

at 2400MHz frequency. The system memory is split equally into two NUMA nodes. We use four Samsung 970 EVO Plus NVMe SSDs in a RAID-0 setup with the XFS filesystem and an aggregate usable capacity of 7.3TB. Each device is capable of up to 3.5GB/s throughput on sequential reads and 3.3GB/s on sequential writes [59]. The server runs on Ubuntu version 20.04.6 and the Linux kernel version is 6.1.128.

We evaluate the following configurations: **swap:** The default Linux swapper, with THP disabled, set to operate over a swap partition on our NVMe device array. **swapTHP:** The same swapper configuration, with THP enabled and set to "always". Swapping entire hugepages out is enabled in the kernel configuration (`CONFIG_THP_SWAP`). ***xHeap*:** *xHeap* with hugepage fault handling and asynchronous hugepage promotions, using the page type selection policy discussed in Section 2. Hugepages use 98% of the *xHeap* pool. Table 1 shows the abbreviation used for each workload on our evaluation figures.

We use three types of compute applications which typically operate on large volumes of data: (a) Data deduplication from the PARSEC benchmark suite [9] to compress a domain-level web graph edge list by Common Crawl [13]. (b) Support Vector Machine (SVM) via a multi-core variant of LIBLINEAR [31], which performs CPU-based linear regression training on the KDD 2012 training dataset [3]. (c) A particle transport simulation workload with XSBench [61], which performs 1M cross-section lookups over a grid of nuclides. For *xHeap* we preload applications with *libvmmalloc* [51], which initially calls *mmap* to setup a jemalloc-based arena allocator over a backing file. It then intercepts `malloc` and related library calls, and returns virtual addresses corresponding to the memory-mapped file.

We set the DRAM budget per application between 75% and 85% of its max resident set size (RSS), as reported by calibration in-memory runs with THP disabled, in 5% increments. To limit the available memory to the selected budget, we use Linux cgroups for the swap configurations, and accordingly set the page pool size parameter for *xHeap*.

4 Experimental Results

In this section we present our experimental evaluation. We first examine the impact of *xHeap* on application data TLB (dTLB) miss cycles. Then, we look at overall application performance and CPU system time, when offloading 15-25% of application memory to storage.

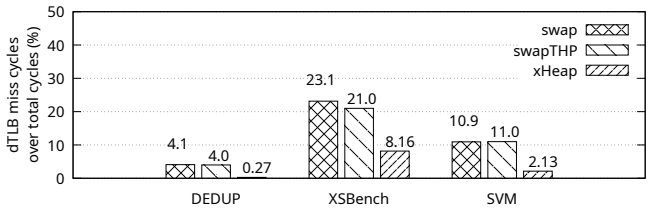

Figure 3. Percentage of dTLB miss cycles over total application cycles per workload.

4.1 *xHeap* reduces dTLB pressure

We first verify that *xHeap* remedies dTLB pressure when offloading memory. We run each workload with 75% of its max RSS in memory and use *perf* to measure the cycles spent handling dTLB misses, as a percentage of the total application cycles. Figure 3 shows these results.

Examining the *swap* configuration without hugepages verifies the TLB miss costs presented in previous works [25, 68], with an average dTLB miss cycle cost of 12.7% across all workloads, and up to 23% in the worst case (XSBench). dTLB benefits with THP are negligible for all workloads. This implies that khugepaged is unable to perform timely hugepage promotions in the presence of memory offloading. We attribute this to khugepaged's `max_ptes_swap` parameter, which avoids promotions in regions with more than 64 swapped out pages by default. This leads to inability to actively reduce dTLB miss costs at runtime. Instead, THP mainly gains dTLB benefits through aggresive allocation of hugepages on page faults during workload initialization. With memory offloading, these benefits are quickly negated. *xHeap* consistently reduces dTLB miss cycle percentage for all workloads compared to both *swap* configurations. Specifically, *xHeap* reduces dTLB miss cycle percentage by 2.83×(XSBench)-15.3×(DEDUP) over *swap*, and by 2.57×(XSBench)-15×(DEDUP) over *swap*THP. *xHeap* achieves this by promoting hugepages on demand, based on application page faults. This allows *xHeap* to adapt to application access patterns and optimize dTLB use.

4.2 *xHeap* enables hugepage benefits

We continue by verifying that *xHeap*'s hugepage use effectively lowers CPU system time and improves performance under offloading. For each workload we plot the execution time breakdown across memory budgets.

Figure 4 shows that data deduplication is amenable to memory offloading, as there is small variance in execution time across memory budgets for a single configuration. *xHeap* performs best for all memory budgets, with execution time lower by 1.5× compared to *swap*THP, which exhibits performance equal within error to *swap*. *swap*THP fails to utilize hugepages throughout execution in contrast to *xHeap*, which thus reduces CPU system time percentage by ~8.2-8.4×, from 26-27% to 2-3%.

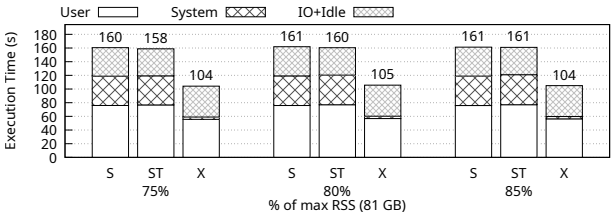

Figure 4. Execution time breakdown for data deduplication.

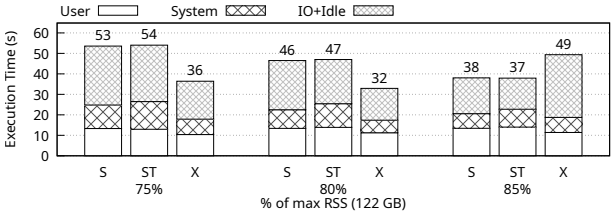

Figure 5. Execution time breakdown for XSBench.

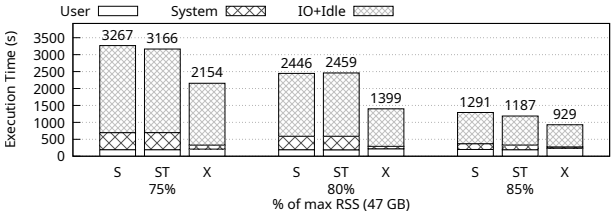

Figure 6. Execution time breakdown for SVM.

Figure 5 shows the execution time breakdown for XSBench. This workload demonstrates the pitfalls of indiscriminate hugepage usage for the entirety of application heaps. The sparse, random accesses by grid lookups incur frequent page faults with memory offloading. The *xHeap* page type selection policy mitigates this issue, and *xHeap* improves performance over *swap* and *swap*THP by 1.4× with 75 and 80% of the max RSS in memory, lowering CPU system time from 24.5% (*swap*THP) to 19%. With 85% of the heap in memory, *xHeap* exhibits higher page fault I/O cost due to its opportunistic fault-time hugepage allocation strategy, and performance drops by 1.3× compared to *swap*THP. Augmenting the *xHeap* page type selection policy to avoid using hugepages -even if available-, for major page faults over reclaimed hugepage-sized ranges would mitigate this issue, perhaps at the detriment of other workloads.

Figure 6 shows the execution time for SVM. This workload favours hugepage use with sequential accesses, in contrast to XSBench. As such, *xHeap* performs best out of all configurations. *swap*THP fails to utilize hugepages beyond workload initialization, and exhibits similar execution time and CPU system time (14% on average) to *swap*. In comparison, *xHeap* reduces CPU system time percentage by 2.9× (5% on average), and improves execution time by 1.28-1.76× across all memory budgets. Dynamic hugepage selection policies could enable *xHeap* to detect the hugepage-friendly pattern of SVM, and adapt by lowering its promotion threshold to further improve performance.

5 Related Work

Several efforts target the performance of *mmio* [11, 33, 46, 47, 50, 60, 62]. All efforts target application I/O, whereas *xHeap* targets improvements for device-backed heaps. We divide related work closer to *xHeap* in two categories, as follows.

Support for hugepages: Linux offers dynamic hugepage support for anonymous mappings with Transparent Hugepages (THP) [15] with significant synchronization constraints. Ingens [29] and HawkEye [44] mitigate application stalls resulting from aggressive THP behaviour by utilizing different page utilization metrics for selecting anonymous hugepages. HugeTLB pages [14] is a static Linux hugepage allocation mechanism for anonymous mappings, which requires explicit application support and does not allow hugepage promotion, reclamation, or swapout.

FreeBSD supports transparent hugepages for both file-backed and anonymous mappings on Intel [52] and ARM [53] architectures. Hugepages are tentatively reserved in memory and incrementally promoted through page faults. Demotion is also supported. The basis for this mechanism is first described by Navarro et al. [42]. Quicksilver [70] builds on FreeBSD hugepage support with extended promotion and demotion policies. Windows [66] supports static hugepages for volatile memory. Explicit application support is required, and no promotion or demotion mechanism is supported. *xHeap* offers application-transparent hugepage support similar to FreeBSD, and uses asynchronous hugepage promotions.

Other Linux-based work related to hugepages attempts to address memory fragmentation issues associated with hugepage allocations at the OS memory management level [20, 21, 45, 68] or the user-space memory allocator level [25, 38], architectural modifications to drive hugepage promotion decisions [40], and dynamic use of multiple hugepage sizes and compaction optimizations [54, 56]. All of these works only concern themselves with anonymous hugepages.

Support for heap extension: Previous works have utilized FLASH storage devices in the domain of specialized applications, e.g., graph analytics frameworks [57, 69, 71]. Other works also aimed at a unified DRAM/SSD interface to scale memory capacity heaps regardless of application type [2, 8, 18], however require application reprogramming or recompilation to utilize new semantics. *xHeap* aims at transparent and application agnostic device-backed heaps.

TMO [64] and DAOS [48] developed shared resource pressure metrics and memory access monitoring frameworks respectively in order to drive memory offloading decisions -either to storage or compressed swap in memory- and effectively utilize memory within the datacenter. Such works

are orthogonal to *xHeap* and can be utilized to drive similar decisions in *xHeap*, such as dynamic page pool resizing.

Tiering approaches seek to address memory capacity limitations by using a slower memory tier. This tier may occupy a portion of DRAM and store compressed data [17, 30], or reside in Non-volatile Memory (NVM) [4, 16, 24, 28, 32]. Recent works have also examined placing this tier in remote node memory using RDMA [5, 23, 55, 58, 63] and current work is exploring the use of the new CXL memory protocol to this end [19, 22]. Recent work examines how fast storage devices can be coupled with CXL to address memory wall issues [27, 65]. *xHeap* transparently extends the memory hierarchy with fast storage devices and similar to such works, can benefit from placement policies, access sampling, and latency hiding techniques.

6 Conclusions

Increasing application heap demands combined with DRAM scaling limitations have led to the adoption of *swap*-enabled heaps, while growing virtual address spaces necessitate the use of hugepages to mitigate TLB miss costs. Current approaches to transparent heap extension offer limited hugepage support, failing to reap hugepage benefits. In this paper we present *xHeap*, an alternative *mmio* path in Linux which enables the fine-grained mixture of regular and hugepages for device-backed heaps. *xHeap* provides support for (a) decoupling device-backed heaps from other, related, kernel functions, (b) transparent and fine-grained mixing of regular pages and hugepages, and (c) concurrent and asynchronous hugepage promotions which adapt to application access patterns. We evaluate heap extension over *xHeap* with a variety of workloads and identify significant performance improvements. Overall, we believe that the use of hugepages has the potential to improve performance under memory offloading significantly, while future work should examine page type selection policies and combating hugepage-incurred external memory fragmentation in depth.

Acknowledgements

We thankfully acknowledge the support of the European Commission under the European Pilot for Exascale (EUPEX) project (Grant Agreement ID: 101033975).

References

[1] Aorus gen5 12000 ssd 2tb specification. https://www.gigabyte.com/SSD/AORUS-Gen5-12000-SSD-2TB/sp#sp.

[2] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-mei Hwu. Flatflash: Exploiting the byte-accessibility of ssds within a unified memory-storage hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 971–985, New York, NY, USA, 2019. Association for Computing Machinery.

[3] Aden, Yi Wang. KDD Cup 2012, Track 2. https://kaggle.com/competitions/kddcup2012-track2.

[4] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. *SIGPLAN Not.*, 52(4):631–644, apr 2017.

[5] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[6] Rachata Ausavarungnirun, Timothy Merrifield, Jayneel Gandhi, and Christopher J. Rossbach. Prism: Architectural support for variable-granularity memory metadata. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, PACT '20, page 441–454, New York, NY, USA, 2020. Association for Computing Machinery.

[7] Manu Awasthi. Rethinking design metrics for datacenter dram. In *Proceedings of the 2015 International Symposium on Memory Systems*, MEMSYS '15, page 162–163, New York, NY, USA, 2015. Association for Computing Machinery.

[8] Anirudh Badam and Vivek S. Pai. Ssdalloc: Hybrid ssd/ram memory management made easy. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, page 211–224, USA, 2011. USENIX Association.

[9] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, page 72–81, New York, NY, USA, 2008. Association for Computing Machinery.

[10] Robert Birke, Lydia Y. Chen, and Evgenia Smirni. Data centers in the cloud: A large scale performance study. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 336–343, 2012.

[11] Jungsik Choi, Jiwon Kim, and Hwansoo Han. Efficient memory mapped file I/O for In-Memory file systems. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, Santa Clara, CA, July 2017. USENIX Association.

[12] Corbet, Jonathan. Memory Compaction. [Online].

[13] Common Crawl. Web graph release cc-main-2024-25-dec-jan-feb. https://data.commoncrawl.org/projects/hyperlinkgraph/cc-main-2024-25-dec-jan-feb/index.html, 2025. [Online].

[14] Linux Kernel Documentation. Hugetlb pages. https://www.kernel.org/doc/html/latest/admin-guide/mm/hugetlbpage.html.

[15] Linux Kernel Documentation. Transparent hugepage support. https://www.kernel.org/doc/html/latest/admin-guide/mm/transhuge.html.

[16] Subramanya R Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016.

[17] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, et al. Towards an adaptable systems architecture for memory tiering at warehouse-scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 727–741, 2023.

[18] Guanyu Feng, Huanqi Cao, Xiaowei Zhu, Bowen Yu, Yuanwei Wang, Zixuan Ma, Shengqi Chen, and Wenguang Chen. TriCache: A User-Transparent block cache enabling High-Performance Out-of-Core processing with In-Memory programs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 395–411, Carlsbad, CA, July 2022. USENIX Association.

[19] Yehonatan Fridman, Suprasad Mutalik Desai, Navneet Singh, Thomas Willhalm, and Gal Oren. Cxl memory as persistent memory for disaggregated hpc: A practical approach. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, SC-W '23, page 983–994, New

York, NY, USA, 2023. Association for Computing Machinery.

[20] Mel Gorman and Patrick Healy. Supporting superpage allocation without additional hardware support. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, page 41–50, New York, NY, USA, 2008. Association for Computing Machinery.

[21] Mel Gorman and Patrick Healy. Performance characteristics of explicit superpage support. In *Proceedings of the 2010 International Conference on Computer Architecture*, ISCA'10, page 293–310, Berlin, Heidelberg, 2010. Springer-Verlag.

[22] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct access, High-Performance memory disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 287–294, Carlsbad, CA, July 2022. USENIX Association.

[23] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, Boston, MA, March 2017. USENIX Association.

[24] Taekyung Heo, Yang Wang, Wei Cui, Jaehyuk Huh, and Lintao Zhang. Adaptive page migration policy with huge pages in tiered memory systems. *IEEE Transactions on Computers*, 71(1):53–68, 2022.

[25] A.H. Hunter, Chris Kennelly, Paul Turner, Darryl Gove, Tipp Moseley, and Parthasarathy Ranganathan. Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 257–273. USENIX Association, July 2021.

[26] Jonathan Corbet. The final step for huge-page swapping. https://lwn.net/Articles/758677/.

[27] Myoungsoo Jung. Hello bytes, bye blocks: Pcie storage meets compute express link for memory expansion (cxl-ssd). In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '22, page 45–51, New York, NY, USA, 2022. Association for Computing Machinery.

[28] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. Exploring the design space of page management for Multi-Tiered memory systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 715–728. USENIX Association, July 2021.

[29] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 705–721, USA, 2016. USENIX Association.

[30] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. Software-defined far memory in warehouse-scale computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 317–330, New York, NY, USA, 2019. Association for Computing Machinery.

[31] Mu-Chu Lee, Wei-Lin Chiang, and Chih-Jen Lin. Fast matrix-vector multiplications for large-scale logistic regression on shared-memory systems. In *2015 IEEE International Conference on Data Mining*, pages 835–840. IEEE, 2015.

[32] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. Memtis: Efficient memory tiering with dynamic page classification and page size determination. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 17–34, New York, NY, USA, 2023. Association for Computing Machinery.

[33] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. Virtual-memory assisted buffer management. *Proc. ACM Manag. Data*, 1(1), may 2023.

[34] Chuandong Li, Sai Sha, Yangqing Zeng, Xiran Yang, Yingwei Luo, Xiaolin Wang, Zhenlin Wang, and Diyu Zhou. Taming hot bloat under virtualization with HUGESCOPE. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 999–1012, Santa Clara, CA, July 2024. USENIX Association.

[35] Linux Kernel Documentation. Workqueue. https://www.kernel.org/doc/html/latest/core-api/workqueue.html.

[36] Linux Kernel Documentation. zram: Compressed RAM-based block devices. https://www.kernel.org/doc/html/latest/admin-guide/blockdev/zram.html.

[37] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. Imbalance in the cloud: An analysis on alibaba cluster trace. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 2884–2892, 2017.

[38] Martin Maas, Chris Kennelly, Khanh Nguyen, Darryl Gove, Kathryn S. McKinley, and Paul Turner. Adaptive huge-page subrelease for non-moving memory allocators in warehouse-scale computers. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2021, page 28–38, New York, NY, USA, 2021. Association for Computing Machinery.

[39] Mark Mansi and Michael M Swift. Characterizing physical memory fragmentation. *arXiv preprint arXiv:2401.03523*, 2024.

[40] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. Tpp: Transparent page placement for cxl-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 742–755, New York, NY, USA, 2023. Association for Computing Machinery.

[41] Alan Nair, Sandeep Kumar, Aravinda Prasad, Ying Huang, Andy Rudoff, and Sreenivas Subramoney. Telescope: Telemetry for gargantuan memory footprint applications. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 409–424, Santa Clara, CA, July 2024. USENIX Association.

[42] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (Copyright Restrictions Prevent ACM from Being Able to Make the PDFs for This Conference Available for Downloading)*, OSDI '02, page 89–104, USA, 2002. USENIX Association.

[43] Nitin Gupta. Proactive compaction for the kernel.

[44] Ashish Panwar, Sorav 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*, ASPLOS '19, New York, NY, USA, 2019. Association for Computing Machinery.

[45] Ashish Panwar, Aravinda Prasad, and K. Gopinath. Making huge pages actually useful. *SIGPLAN Not.*, 53(2):679–692, mar 2018.

[46] Anastasios Papagiannis, Manolis Marazakis, and Angelos Bilas. Memory-mapped i/o on steroids. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 277–293, New York, NY, USA, 2021. Association for Computing Machinery.

[47] Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. Optimizing memory-mapped i/o for fast storage devices. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'20, USA, 2020. USENIX Association.

[48] SeongJae Park, Madhuparna Bhowmik, and Alexandru Uta. Daos: Data access-aware operating system. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '22, page 4–15, New York, NY, USA, 2022. Association for Computing Machinery.

[49] Shuyi Pei and Rekha Pitchumani. Cmm-h (cxl memory module – hybrid): Rethinking storage for the memory-centric computing era. https://semiconductor.samsung.com/us/news-events/tech-blog/ret

hinking-storage-for-the-memory-centric-computing-era/.

[50] Ivy B. Peng, Maya B. Gokhale, Karim Youssef, Keita Iwabuchi, and Roger Pearce. Enabling scalable and extensible memory-mapped datastores in userspace. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):866–877, 2022.

[51] Persistent Memory Development Kit (PMDK). Volatile persistent memory allocator. https://github.com/pmem/vmem/.

[52] The FreeBSD Project. Freebsd 7.2-release release notes. https://www.freebsd.org/releases/7.2R/relnotes-detailed/, 2009.

[53] The FreeBSD Project. Freebsd 10.0-release release notes. https://www.freebsd.org/releases/10.0R/relnotes/, 2014.

[54] Stratos Psomadakis, Chloe Alverti, Vasileios Karakostas, Christos Katsakioris, Dimitrios Siakavaras, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. Elastic translations: Fast virtual memory with multiple translation sizes. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 17–35, 2024.

[55] Yifan Qiao, Chenxi Wang, Zhenyuan Ruan, Adam Belay, Qingda Lu, Yiying Zhang, Miryung Kim, and Guoqing Harry Xu. Hermit:{Low-Latency},{High-Throughput}, and transparent remote memory via {Feedback-Directed} asynchrony. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 181–198, 2023.

[56] Venkat Sri Sai Ram, Ashish Panwar, and Arkaprava Basu. Trident: Harnessing architectural resources for all page sizes in x86 processors. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1106–1120, 2021.

[57] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488, 2013.

[58] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-Performance, Application-Integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 315–332. USENIX Association, November 2020.

[59] Samsung Electronics Corporation. Samsung V-NAND SSD 970 EVO Plus 2021 Data Sheet. https://download.semiconductor.samsung.com/resources/data-sheet/Samsung_NVMe_SSD_970_EVO_Plus_Data_Sheet_Rev.3.0_10129514051212.pdf.

[60] Nae Young Song, Yongseok Son, Hyuck Han, and Heon Young Yeom. Efficient memory-mapped i/o on fast storage device. *ACM Trans. Storage*, 12(4), may 2016.

[61] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. Xsbench-the development and verification of a performance abstraction for monte carlo reactor analysis. *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, 2014.

[62] Brian Van Essen, Henry Hsieh, Sasha Ames, Roger Pearce, and Maya Gokhale. Di-mmap—a scalable memory-map runtime for out-of-core data-intensive applications. *Cluster Computing*, 18(1):15–28, 2015.

[63] Chenxi Wang, Yifan Qiao, Haoran Ma, Shi Liu, Wenguang Chen, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Canvas: Isolated and adaptive swapping for Multi-Applications on remote memory. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 161–179, Boston, MA, April 2023. USENIX Association.

[64] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. Tmo: Transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 609–621, New York, NY, USA, 2022. Association for Computing Machinery.

[65] Shao-Peng Yang, Minjae Kim, Sanghyun Nam, Juhyung Park, Jin yong Choi, Eyee Hyun Nam, Eunji Lee, Sungjin Lee, and Bryan S. Kim. Overcoming the memory wall with CXL-Enabled SSDs. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 601–617, Boston, MA, July 2023. USENIX Association.

[66] Pavel Yosifovich, David A Solomon, and Alex Ionescu. *Windows Internals, Part 1: System architecture, processes, threads, memory management, and more*. Microsoft Press, 2017.

[67] Erez Zadok and Ion Badulescu. A stackable file system interface for linux. In *LinuxExpo Conference Proceedings*, volume 94, pages 141–151, 1999.

[68] Kaiyang Zhao, Kaiwen Xue, Ziqi Wang, Dan Schatzberg, Leon Yang, Antonis Manousis, Johannes Weiner, Rik Van Riel, Bikash Sharma, Chunqiang Tang, et al. Contiguitas: The pursuit of physical memory contiguity in datacenters. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–15, 2023.

[69] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S Szalay. {FlashGraph}: Processing {Billion-Node} graphs on an array of commodity {SSDs}. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 45–58, 2015.

[70] Weixi Zhu, Alan L. Cox, and Scott Rixner. A comprehensive analysis of superpage management mechanisms and policies. In *Proc. of the 2020 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'20, USA, 2020. USENIX Association.

[71] Xiaowei Zhu, Wentao Han, and Wenguang Chen. {GridGraph}:{Large-Scale} graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 375–386, 2015.