Understanding Transparent Superpage Management

WEIXI ZHU, ALAN L. COX, AND SCOTT RIXNER

Weixi Zhu is currently a fifth-year CS PhD student at Rice University who is advised by Professor Scott Rixner and works closely with Professor Alan Cox. His research area is memory systems aimed at improving their flexibility and performance for both generic and domain-specific architectures. He received his BS in the National Elite Program (computer science) at Nanjing University in 2016 and defended his MS thesis at the Computer Science Department of Rice University in 2018. wxchu@rice.edu

Alan L. Cox is a professor of computer science at Rice University and a long-time contributor to the FreeBSD project. Over the years, his research has sought to address fundamental problems at the intersection of operating systems, computer architecture, and networking. Prior to joining Rice, he earned his BS at Carnegie Mellon University and his PhD at the University of Rochester. alc@rice.edu

Scott Rixner is a professor of computer science at Rice University. His research spans virtualization, operating systems, and computer architecture, with a specific focus on memory systems and networking. His work has led to 11 patents and has been implemented within several open source systems. He is also well versed in the internals of the Python programming language, as he has developed Python interpreters for both embedded systems and web browsers. Prior to joining Rice, he received his PhD from MIT. rixner@rice.edu

Superpages (2 MB pages) can reduce the address translation overhead for large-memory workloads in modern computer systems. We clearly outline the sequence of events in the life of a superpage and explore the design space of when and how to trigger and respond to those events. We provide a framework that enables better understanding of superpage management and the trade-offs involved in different design decisions. Quicksilver, our novel superpage management system, is designed based on the insights obtained by using this framework to improve superpage management.

The memory capacity of modern machines continues to expand at a rapid pace. There is also a growing class of "large memory" data-oriented applications—including in-memory databases, data analysis tools, and scientific computation—that can productively utilize all available memory resources. These large memory applications can process data at scales of terabytes or even petabytes, which cannot fit in the memory. Therefore, they either use out-of-core computation frameworks or build their own heuristics to efficiently cache disk data to avoid the unexpected performance impacts of swapping. As a result, these applications have very large memory footprints, which makes address translation performance critical.

The use of *superpages*, or "huge pages," can reduce the cost of virtual-to-physical address translation. For example, the x86-64 architecture supports 2 MB superpages. Using these 2 MB mappings eliminates one level of the page walk traversal and enables more efficient use of TLB (translation lookaside buffer) entries. Intel's most recent processors can hold 1536 mappings in the TLB. The 2 MB superpages can therefore increase TLB coverage from around 6 MB (0.009% of the memory in a system with 64 GB of DRAM) to 3 GB (4.7%). While this is still a small fraction of the total physical memory capacity of a large machine, it is far more likely to capture an application's short-term working set.

The challenge, however, is for the operating system (OS) to transparently manage memory resources in order to maximize superpage use. Modern systems do not necessarily accomplish this well, which has led to many suggestions that transparent huge page (THP) support be turned off in Linux for performance-critical applications. A better solution, however, is to understand the benefits and limitations of existing superpage management policies in order to redesign and improve them.

We carefully explain and analyze the life cycle of a superpage and present several novel observations about the mechanisms used for superpage management. These observations motivate Quicksilver (https://github.com/rice-systems/quicksilver) [9], an innovative design for transparent superpage management based upon FreeBSD's reservation-based physical superpage allocator. The proposed design achieves the benefits of aggressive superpage allocation but mitigates the memory bloat and fragmentation issues that arise from underutilized superpages. The system is able to match or beat the performance of existing systems in both lightly and heavily fragmented scenarios. For example, when using synchronous page preparation, the system achieves 2× speedups over Linux on PageRank using GraphChi on a heavily fragmented system. On Redis, the system is able to maintain Redis throughput and

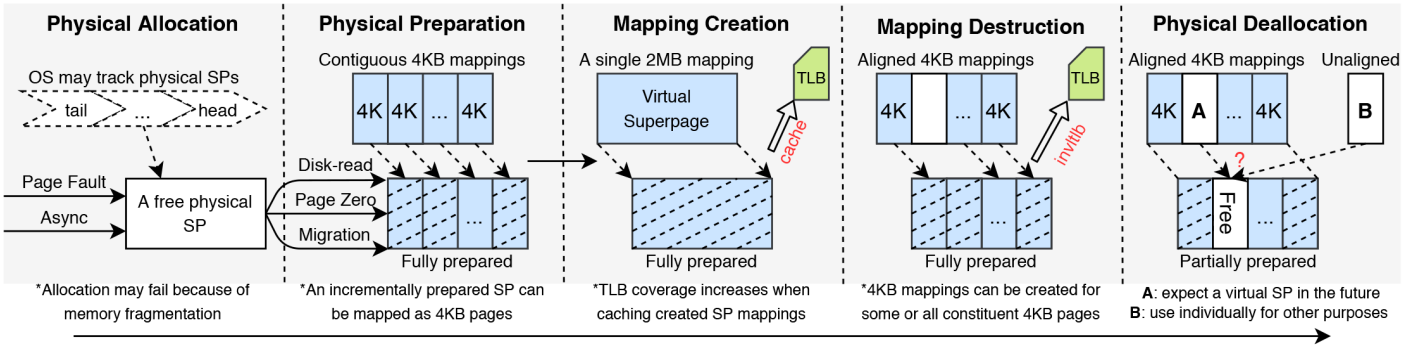

Figure 1: The five events in the life of a superpage (SP)

tail latency as fragmentation increases, whereas the throughput of other systems degrades and tail latency increases. Finally, the system is able to achieve these performance improvements without excessive memory bloat.

Transparent Superpage Management

Kernels manage superpages transparently via these five events:

1. *Physical superpage allocation*: acquisition of a free physical superpage
2. *Physical superpage preparation*: incremental or full preparation of the initial data for an allocated physical superpage
3. *Superpage mapping creation*: creation of a virtual superpage in a process's address space and mapping it to a fully prepared physical superpage
4. *Superpage mapping destruction*: destruction of a virtual superpage mapping
5. *Physical superpage deallocation*: partial or full deallocation of an allocated physical superpage

The five events follow an order that indicates their prerequisites. However, the triggers and handlers for each of these events are determined by the OS and vary across OSes. Figure 1 illustrates the lifetime of a superpage in terms of these five events.

As shown in the figure, the first step in the process is physical superpage allocation. The OS can choose to allocate a physical superpage to back any 2 MB-aligned virtual memory region. A physical superpage could be allocated synchronously upon a page fault or asynchronously via a background task. In order to allocate a physical superpage, the physical memory allocator must have an available, aligned 2 MB region. Under severe memory fragmentation, such regions may not be available.

The second step is to prepare the physical superpage with its initial data. A physical superpage can be prepared in one of three ways. First, if the virtual memory region is anonymous, that is, not backed by a file, then the superpage simply needs to be zeroed. Second, if the virtual memory region is a memory-mapped file, then the data must be read from the file. Finally, if the virtual memory region is currently mapped to independent 4 KB pages, then the contents of those existing pages must be copied into the physical superpage. In this case, the 4 KB pages within the superpage that were not already mapped would need to be prepared appropriately, either via zeroing or reading from the backing file.

Physical superpages can be prepared all at once or incrementally. As each 4 KB page is prepared, it can also be temporarily mapped as a 4 KB page. At a minimum, on a page fault, the 4 KB page that triggered the fault must be prepared immediately in order to allow the application to resume. However, upon a page fault, the OS can choose to prepare the entire physical superpage, only prepare the relevant 4 KB page, or prepare the relevant 4 KB page, allow the application to resume, and prepare the remaining pages later (either asynchronously or when they are accessed).

Once a physical superpage has been fully prepared, the third step is to map that superpage into a process's virtual address space in order to achieve address translation benefits. Before the superpage is mapped, the physical memory can still be accessed via 4 KB mappings; afterwards, the OS loses the ability to track accesses and modifications at a 4 KB granularity. Therefore, an OS may delay the creation of a superpage mapping if only some of the constituent pages are dirty in order to avoid unnecessary I/O in the future.

Superpage mappings are often created upon a page fault, on either the initial fault to the memory region or a subsequent fault after the entire superpage has been prepared. However, if the physical superpage preparation is asynchronous, then its superpage mapping can also be created asynchronously. Note that on some architectures—for example, ARM—any 4 KB mappings that were previously created must first be destroyed.

Fourth, superpage mappings can be destroyed at any time, but must be destroyed whenever any part of the virtual superpage is freed or has its protection changed. After the superpage mapping is destroyed, 4 KB mappings must be recreated for any constituent pages that have not been freed.

	Linux [3]	FreeBSD [6]	Ingens [4]	HawkEye [7]	Quicksilver [9]
Allocation	On first page fault (defragmenting if necessary) and asynchronously for regions with one 4 KB mapping	Created ("reserved") on the first page fault	Asynchronously for regions with 460 4 KB mappings, prioritizing processes with fewer superpages	Asynchronously for regions with one 4 KB mapping, prioritizing heavily utilized regions and processes with big memory usage and high TLB overheads	Created ("reserved") on the first page fault
Preparation	Immediately prepares entire superpage by zeroing or migration	Incrementally prepares in-place 4 KB pages on page faults	Immediately prepares entire superpage by zeroing and migration	Immediately prepares entire superpage by zeroing and migration	Incrementally prepares until a threshold is reached (e.g., 64 in-place 4 KB pages), then prepares the remainder entirely
Mapping	Immediately after allocation and full preparation	Upon the page fault that finishes all preparation	Immediately after allocation and full preparation	Immediately after allocation and full preparation	Upon the page fault that finishes all preparation
Unmapping	When virtual memory is freed, or the mapping is changed, in whole or in part	When virtual memory is freed, or the mapping is changed, in whole or in part	When virtual memory is freed, or the mapping is changed, in whole or in part	When virtual memory is freed, or the mapping is changed, in whole or in part	When virtual memory is freed, or the mapping is changed, in whole or in part
Deallocation	As soon as the superpage is unmapped	Defers as long as possible	As soon as the superpage is unmapped	As soon as the superpage is unmapped	Defers until the superpage is inactive

Table 1: Comparison of modern superpage management designs

Finally, a physical superpage is deallocated when an application frees some or all of the virtual superpage, when an application terminates, or when the OS needs to reclaim memory. If a superpage mapping exists, it must be destroyed before the physical superpage can be deallocated. Then, either the entire 2 MB can be returned to the physical memory allocator or the physical superpage can be "broken" into 4 KB pages. If the physical superpage is broken into its constituent 4 KB pages, the OS can return a subset of those pages to the physical memory allocator. However, returning only a subset of the constituent pages increases memory fragmentation, decreasing the likelihood of future physical superpage allocations.

Superpage Management Designs

Table 1 presents a comparison of superpage management designs, showing how they handle the five events that occur in the lifetime of a superpage. The table shows two existing operating systems—Linux and FreeBSD—and three state-of-the-art research prototypes—Ingens, HawkEye, and Quicksilver.

Note that the primary differences among these systems are in how they allocate and prepare superpages. There are three key mechanisms that are used to allocate superpages: first-touch, reservations, and asynchronous daemons. The first-touch policy, used exclusively by Linux, allocates, prepares, and maps superpages on the first page fault to a 2 MB-aligned virtual memory region. Linux goes so far as to compact memory if a physical superpage is not currently available in order to attempt to obtain one. This maximizes address translation benefits, as memory is defragmented upon allocation and the superpage mapping is created immediately. However, this also increases page fault latency. In contrast, the reservation-based policy used by FreeBSD and Quicksilver simply *reserves* a physical superpage on the first page fault to a 2 MB-aligned virtual memory region. A physical superpage is allocated for that region, but it is not immediately prepared and mapped. This leads to faster page fault handling, but does not immediately achieve address translation benefits. However, there are benefits to delaying preparation and mapping. If not all of the constituent pages are accessed, then they can be

Workload	Linux-4 KB	Linux-noKhugepaged	Linux
Del-70	11.6 GB	11.7 GB	19.8 GB
Range-XL	14.4 GB	25.7 GB	30.7 GB

Table 2: Redis memory consumption. Linux-noKhugepaged disables khugepaged.

quickly reclaimed under memory pressure, and resources were not wasted on preparation for ultimately untouched pages.

Quicksilver strikes a balance between incremental and all-at-once preparation. Reservations are initially prepared incrementally. This minimizes the initial page fault latency, but loses immediate address translation benefits. Therefore, Quicksilver has an additional threshold, *t*. Once *t* 4 KB pages get prepared, it prepares the remainder of the superpage all-at-once, either synchronously (Sync-*t*) or asynchronously (Async-*t*). This design choice reduces memory bloat, as will be discussed in Observation 1 in the next section, because it does not immediately prepare and map the superpage. However, it enables address translation benefits sooner than waiting for the entire superpage to be accessed.

Linux, Ingens, and HawkEye all utilize asynchronous daemons to allocate, prepare, and map superpages in the background.

Linux's khugepaged is indiscriminate as it scans memory and creates superpages for any aligned 2 MB anonymous virtual memory region that contains at least one dirty 4 KB mapping. As with Linux's first-touch policy, if no free physical superpage exists, it will defragment memory in an attempt to create one. Ingens' and HawkEye's asynchronous daemons both improve upon Linux's indiscriminate allocation policy.

To prevent excessive memory bloat, Ingens increases the threshold of 4 KB pages used to trigger creation of a superpage from one single page to 90%, meaning there must be at least 460 4 KB mappings in a 2 MB region in order to create a superpage for that region. Ingens also prioritizes processes with fewer superpages in order to improve overall fairness. In addition, Ingens actively compacts non-referenced memory in the background.

HawkEye uses the same threshold as Linux: one dirty page. Under memory pressure, it scans mapped superpages and makes their zero-filled 4 KB pages copy-on-write to a canonical zero page to reclaim free memory. HawkEye also maintains a list of candidate 2 MB-aligned regions, but further weights them by the regions' spatial and temporal utilization and the processes' memory consumption and TLB overheads. HawkEye then creates a superpage mapping for the most heavily weighted region in an attempt to make the most profitable promotions first.

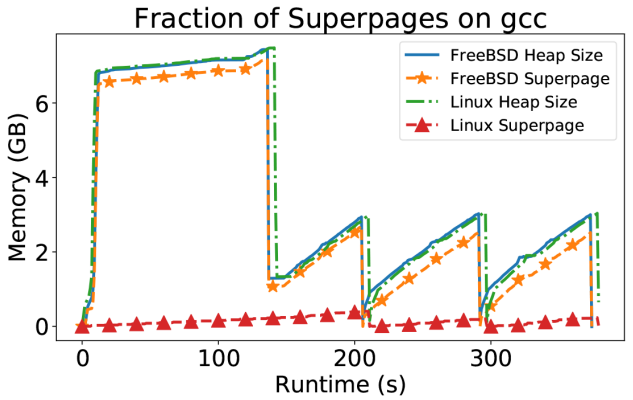

Figure 2: Linux's first-touch policy fails to create superpages.

Analysis of Existing Designs

In this section, we analyze the designs for transparent superpage management described in the previous section and present several novel observations about them. Details on the experimental setup can be found in [9].

Observation 1: Coupling physical allocation, preparation, and mapping of superpages leads to memory bloat and fewer superpage mappings. It also is not compatible with transparent use of multiple superpage sizes.

Linux's first-touch policy couples physical superpage allocation, preparation, and superpage mapping creation together. As a result, it enjoys two obvious benefits: it provides immediate address translation benefits, and it eliminates a large number of page faults. Therefore, it is usually the best policy when there is abundant contiguous free memory.

However, this coupled policy has several drawbacks. First, it can bloat memory and waste time preparing underutilized superpages. In a microbenchmark that sparsely touches 30 GB of anonymous memory, Linux's first-touch policy spends 1.4 sec and consumes 30 GB compared to 0.06 sec and 0.2 GB when disabling transparent huge pages. While such a case is rare when applications use malloc to dynamically allocate memory, it may still happen in a long-running server (for example, Redis). Table 2 shows Redis performance on two workloads: Del-70, which randomly deletes 70% of objects after inserting them, and Range-XL, which inserts randomly sized objects between 256 bytes and 1 MB. The table shows that Linux's first-touch policy bloats memory by 78% compared to Linux with superpages disabled (Linux-4 KB) on the workload Range-XL.

Second, it misses chances to create superpage mappings when virtual memory grows. During a page fault, Linux cannot create a superpage mapping beyond the heap's end, so it installs a 4 KB page, which later prevents creation of a superpage mapping when the heap grows. Figure 2 shows such behavior for gcc [2], which

Page Size	Anonymous	NVMe Disk	Spinning Disk
2 MB	91 µs	**1.7 ms**	**11 ms**
1 GB	**46 ms**	**0.9 sec**	**7.7 sec**

Table 3: Page fault latency. Bold numbers are estimates.

includes three compilations. Linux's first-touch policy creates a few superpage mappings early in each compilation but fails to create more as the heap grows. Instead, promotion-based policies can create more superpages, as seen with FreeBSD and Linux's khugepaged.

Third, it cannot be extended to larger anonymous or file-backed superpages. Table 3 estimates the page-fault latency on both 1 GB anonymous superpages and 2 MB and 1 GB file-backed superpages. Faulting a 2 MB file-backed superpage on the NVMe disk costs 1.7 ms and faulting a 1 GB anonymous superpage takes 46 ms. These numbers may cause latency spikes in server applications. Furthermore, it cannot easily determine which page size to use on first touch. This is arguably more of an immediate problem on ARM processors, which support both 64 KB and 2 MB superpages.

Observation 2: Asynchronous, out-of-place promotion alleviates latency spikes but delays physical superpage allocations.

Promotion-based policies can use 4 KB mappings and later replace them with a superpage mapping. This allows for potentially better-informed decisions about superpage mapping creation and can easily be extended to support multiple sizes of superpages. Specifically, there are two kinds of promotion policies, named out-of-place promotion and in-place promotion. They differ in whether previously prepared 4 KB pages require migration when preparing a physical superpage.

Under out-of-place promotion, a physical superpage is not allocated in advance; on a page fault, a 4 KB physical page is allocated that may neither be contiguous nor aligned with its neighbors. When the OS decides to create a superpage mapping, it must allocate a physical superpage, migrate mapped 4 KB physical pages, and zero the remaining ones. At this time, previously created 4 KB mappings are no longer valid.

Linux, Ingens, and HawkEye perform asynchronous, out-of-place promotion to hide the cost of page migration. As discussed in the previous section, Linux includes khugepaged as a supplement to create superpage mappings. The steady, slow increase of Linux's superpages in Figure 2 is from khugepaged's out-of-place promotions. However, khugepaged can easily bloat memory. Table 2 shows a memory bloat from 11.6 GB to 19.8 GB on workload Del-70. On workload Range-XL, it bloats memory from 25.7 GB to 30.7 GB.

Ingens and HawkEye disable Linux's first-touch policy and instead improve the behavior and functionality of khugepaged. Under memory fragmentation, Linux tries to compact memory when it fails to allocate superpages, which blocks the ongoing page fault and leads to latency spikes. Ingens and HawkEye enhance khugepaged and use it as their primary superpage management mechanism.

However, out-of-place promotion delays physical superpage allocations and, ultimately, superpage mapping creations, because the OS must scan page tables to find candidate 2 MB regions and schedule the background tasks to promote them. Table 4 compares in-place promotion (FreeBSD) with out-of-place promotion (Ingens and HawkEye) on applications where superpage creation speed is critical. Both PageRank using GraphChi (GraphChi-PR) [5] and BlockSVM [8] represent important real-life applications, using fast algorithms to process big data that cannot fit in memory. To better illustrate the problem, in Table 4 Ingens* and HawkEye* were tuned to be more aggressive, so that all 2 MB regions containing at least one dirty 4 KB mapping are candidates for promotion. Specifically, Ingens* uses a 0% utilization threshold instead of 90%, and HawkEye* uses a 100% maximum CPU budget to promote superpages. However, Table 4 shows that FreeBSD consistently outperforms both of them. In other words, the most conservative in-place promotion policy creates superpage mappings faster than the most aggressive out-of-place promotion policy.

Observation 3: Reservation-based policies enable speculative physical page allocation, which enables the use of multiple page sizes, in-place promotion, and obviates the need for asynchronous, out-of-place promotion.

In-place promotion does not require page migration. It creates a physical superpage on the first touch, then incrementally prepares and maps its constituent 4 KB pages without page allocation. Therefore, the allocation of a physical superpage is immediate, but its superpage mapping creation is delayed. To bypass 4 KB page allocations, it requires a bookkeeping system to track allocated physical superpages: for example, FreeBSD's reservation system.

FreeBSD's reservation system immediately allocates physical superpages but delays superpage mapping creation, sacrificing some address translation benefits. Navarro et al. reported negligible overheads from the reservation system [6]. Table 4 shows that Linux consistently outperforms FreeBSD when memory is unfragmented, though Linux and FreeBSD both created similar numbers of anonymous superpage mappings.

However, FreeBSD aggressively allocates physical superpages for anonymous memory. Upon a page fault of anonymous memory, it always speculatively allocates a physical superpage,

Workloads	Ingens	Ingens*	HawkEye	HawkEye*	FreeBSD
GraphChi-PR	0.58	0.58	0.53	0.60	0.77
BlockSVM	0.81	0.79	0.73	0.81	0.96

Table 4: Speedup over Linux with unfragmented memory. All systems have worse performance than Linux. The Ingens* and HawkEye* versions are aggressively tuned.

	Linux-4 KB	Linux
Frag-0	1.04 GB/s (5.6 ms)	1.34 GB/s (4.1 ms)
Frag-50	1.04 GB/s (5.7 ms)	0.92 GB/s (10.2 ms)

Table 5: Mean throughput and 95th latency of Redis Cold workload. Frag-X has X% fragmented memory.

expecting the heap to grow. This eliminates one of the primary needs for khugepaged in Linux. In Figure 2, FreeBSD has most of the memory quickly mapped as superpages, because most speculatively allocated physical superpages end up as fully prepared pages.

Observation 4: Reservations and delaying partial deallocation of physical superpages fight fragmentation.

Superpages are easily fragmented on a long-running server. A few 4 KB pages can consume a physical superpage, which benefits little if mapped as a superpage. Existing systems deal with memory fragmentation in three ways.

Linux compacts memory immediately when it fails to allocate a superpage. It tries to greedily use superpages but risks blocking a page fault. Table 5 evaluated the performance of Redis on a Cold workload, where an empty instance is populated with 16 GB of 4 KB objects. Under fragmentation (Frag-50), Linux obtains slightly higher throughput but much higher tail latency than Linux-4 KB.

FreeBSD delays the partial deallocation of a physical superpage to increase the likelihood of reclaiming a free physical superpage. When individual 4 KB pages get freed sooner, they land in a lower-ordered buddy queue and are more likely to be quickly reallocated for other purposes. Therefore, performing partial deallocations only when necessary due to memory pressure decreases fragmentation.

Ingens actively defragments memory in the background to avoid blocking page faults. It preferably migrates non-referenced memory, so that it minimizes the interference with running applications. As a result, Ingens generates fewer latency spikes compared with Linux [4]. These migrations, however, do consume processor and memory resources.

Evaluation

This section provides a brief evaluation of several variants of Quicksilver (Sync-*t* and Async-*t*) against Linux, FreeBSD, Ingens, HawkEye, and their aggressively tuned variants. A more detailed evaluation can be found in [9].

Unfragmented Performance

Sync-1 uses the same superpage preparation and mapping policy for anonymous memory as Linux. With no fragmentation, they perform similarly. However, there are two notable differences. First, Sync-1 speculatively allocates superpages for growing heaps, which allows it to outperform Linux on canneal [1] and gcc [2]. Their similar speedups on reservation-based systems validate Observation 3. Second, Sync-1 creates file-backed superpages and outperforms Linux on GraphChi-PR.

With no fragmentation, FreeBSD outperforms Ingens and HawkEye. This validates Observation 2, as the issue is that out-of-place promotion is slower. Furthermore, on the Redis Cold workload, Ingens and HawkEye even show a degradation over Linux without using superpages.

Sync-64 typically outperforms Async-64 because Async-64 zeros pages in the background, which can cause interference. The comparable performance of Sync-64 and Sync-1 shows that less aggressive preparation and mapping policies can achieve comparable results to immediately mapping superpages on first touch.

Performance under Fragmentation

Linux has a higher tail latency on a Redis Cold workload under fragmentation than Linux without superpages because its on-allocation defragmentation significantly increases page fault latency. In contrast, FreeBSD does not actively defragment memory, so it generates no latency spikes. Ingens and HawkEye offload superpage allocation from page faults and compact memory in the background, so they reduce interference and generate few latency spikes on the Redis Cold workload. Furthermore, their speedup over Linux increases as fragmentation increases.

The four variants of Quicksilver all consistently perform well under fragmentation because their background defragmentation not only avoids increasing page fault latency, but also succeeds in recovering unfragmented performance. Specifically, on the Redis Cold workload with Frag-100, Sync-1 maintained the

	GraphChi-PR	canneal	DSjeng	XZ
Ingens	1.13	1.00	1.01	1.02
HawkEye	1.11	1.01	0.97	1.02
FreeBSD	1.10	1.05	1.04	1.02
Sync-1	2.18	1.12	1.10	1.14
Sync-64	2.11	1.12	1.11	1.14
Async-64	1.68	1.12	1.11	1.13
Async-256	1.65	1.16	1.08	1.13

Table 6: Performance speedup over Linux in a fully fragmented system (Frag-100)

highest throughput (1.31 GB/s) while providing low (4.5 ms) tail latency. This outperforms Linux, the second best system, which only achieved 1.07 GB/s with 5.6 ms tail latency.

Table 6 shows some select results across the systems discussed in the paper in a fully fragmented system (DSjeng and XZ are from SPEC CPU2017 [2]). Note that Quicksilver outperforms the other systems under high fragmentation across a wide range of workloads, but these applications show some of the greatest benefits.

GraphChi-PR is an important real-world workload, and Sync-1 is able to achieve a 2.18× speedup over Linux, far greater than any of the other systems. To better understand that speedup, consider the other variants of Quicksilver on GraphChi-PR. First, in a fully fragmented system, Async-256 performs well because its preemptive and asynchronous superpage deallocation allows many more superpage allocations than the non-Quicksilver systems. Quicksilver is able to defragment memory more efficiently by identifying inactive fragmented superpages. Furthermore, the in-place promotions contribute to the 1.65 speedup of Async-256, which is already much higher than all of the other non-Quicksilver systems. The more aggressive promotion threshold of Async-64 leads to a slightly higher 1.68 speedup.

Second, Sync-64 outperforms Async-64 with a speedup of 2.11. Again, the asynchronous deallocation is beneficial. However, in addition, the synchronous all-at-once preparation implemented by bulk zeroing in Sync-64 efficiently removes the delay of creating superpages. With the same number of superpages created, Sync-64 is able to reduce page walk pending cycles by 76%. Finally, Sync-1 obtains the highest speedup of 2.18 with a more aggressive promotion threshold. While the speedups on the other applications are not as dramatic, the underlying trends are the same.

Conclusion

The solution to perceived performance issues with transparent superpages is not to disable them. Rather it is to carefully understand how superpage management systems work so that they can be improved. The explicit enumeration of the five events involved in the life of a superpage provides a framework around which to compare and contrast superpage management policies. This framework and analysis yielded several key observations about superpage management that motivated Quicksilver's innovative design. Quicksilver achieves the benefits of aggressive superpage allocation, while mitigating the memory bloat and fragmentation issues that arise from underutilized superpages. Both the Sync-1 and Sync-64 variants of Quicksilver are able to match or beat the performance of existing systems in both lightly and heavily fragmented scenarios, in terms of application performance, tail latency, and memory bloat.

References

[1] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*, pp. 72–81: https://dl.acm.org/doi/10.1145/1454115.1454128.

[2] J. Bucek, K.-D. Lange, and J. V. Kistowski, "SPEC CPU2017: Next-Generation Compute Benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE '18)*, pp. 41–42: https://dl.acm.org/doi/pdf/10.1145/3185768.3185771.

[3] M. Gorman and P. Healy, "Supporting Superpage Allocation without Additional Hardware Support," in *Proceedings of the 7th International Symposium on Memory Management (ISMM '08)*, pp. 41–50: https://dl.acm.org/doi/10.1145/1375634.1375641.

[4] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel, "Coordinated and Efficient Huge Page Management with Ingens," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, pp. 705–721: https://www.usenix.org/system/files/conference/osdi16/osdi16-kwon.pdf.

[5] A. Kyrola, G. E. Blelloch, and C. Guestrin, "GraphChi: Large-Scale Graph Computation on Just a PC," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation, (OSDI '12)*, pp. 31–46: https://www.usenix.org/system/files/conference/osdi12/osdi12-final-126.pdf.

[6] J. Navarro, S. Iyer, P. Druschel, and A. L. Cox, "Practical, Transparent Operating System Support for Superpages," in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, pp. 89–104: https://www.usenix.org/legacy/events/osdi02/tech/full_papers/navarro/navarro.pdf.

[7] A. Panwar, S. Bansal, and K. Gopinath, "HawkEye: Efficient Fine-Grained OS Support for Huge Pages," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*, pp. 347–360: https://dl.acm.org/doi/10.1145/3297858.3304064.

[8] H.-F. Yu, C.-J. Hsieh, K.-W. Chang, and C.-J. Lin, "Large Linear Classification When Data Cannot Fit in Memory," in *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, pp. 2777–2782: https://www.ijcai.org/Proceedings/11/Papers/462.pdf.

[9] W. Zhu, A. L. Cox, and S. Rixner, "A Comprehensive Analysis of Superpage Management Mechanisms and Policies," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*, pp. 829–842: https://www.usenix.org/system/files/atc20-zhu-weixi_0.pdf.