A Comprehensive Analysis of Superpage Management Mechanisms and Policies

Weixi Zhu, Alan L. Cox, and Scott Rixner, *Rice University*

https://www.usenix.org/conference/atc20/presentation/zhu-weixi

This paper is included in the Proceedings of the 2020 USENIX Annual Technical Conference.

July 15–17, 2020

978-1-939133-14-4

A Comprehensive Analysis of Superpage Management Mechanisms and Policies

Weixi Zhu, Alan L. Cox and Scott Rixner
Rice University
{wxzhu, alc, rixner}@rice.edu

Abstract

Superpages (2MB pages) can reduce the address translation overhead for large-memory workloads in modern computer systems. This paper clearly outlines the sequence of events in the life of a superpage and explores the design space of when and how to trigger and respond to those events. This provides a framework that enables better understanding of superpage management and the trade-offs involved in different design decisions. Under this framework, this paper discusses why state-of-the-art designs exhibit different performance characteristics in terms of runtime, latency and memory consumption. This paper illuminates the root causes of latency spikes and memory bloat and introduces Quicksilver, a novel superpage management design that addresses these issues while maintaining address translation performance.

1 Introduction

The physical memory size of modern computers continues to grow at a rapid pace. Furthermore, there is an ever expanding class of "large-memory" data-oriented applications — including databases, data analysis tools, and scientific computations — that can productively utilize all of this memory. While some of these applications expect the entirety of their data to reside within physical memory, others process data at a scale that far exceeds its size. These others either use out-of-core computation frameworks or implement schemes for caching data from secondary storage that avoid swapping by the virtual memory system. In either case, these applications have large memory footprints, so the cost of virtual-to-physical address translation significantly impacts their performance.

The use of *superpages*, or "huge pages", can reduce the cost of virtual-to-physical address translation. For example, the x86-64 architecture supports 2MB superpages. Using these superpages (1) eliminates one level from the hierarchical page table, thereby reducing the expected number of memory accesses to resolve a TLB miss, and (2) enables more efficient use of the TLB's limited number of entries. Intel's recent processors can store up to 1536 4KB or 2MB page mappings in their TLBs. Superpages can therefore increase these TLBs' coverage from around 6MB (0.009% of the physical memory in a computer with 64GB of DRAM) to 3GB. While this is still a small fraction of the computer's physical memory, it is far more likely to capture an application's short-term working set. The benefits of this increased coverage are obvious. The challenge, however, is for the operating system (OS) to transparently manage superpages in an effective manner.

This paper first defines the five distinct events in the life cycle of a transparently managed superpage, and then it analyzes the various state-of-the-art approaches to handling each event. Briefly, the events are as follows. First, a physical superpage must be allocated. Throughout this paper, unless stated otherwise, "superpage" refers to a 2MB page, so this is the act of acquiring a contiguous, aligned 2MB region from the physical memory allocator. Second, the physical superpage must be prepared. For anonymous memory, the entire 2MB region must be zeroed. For file-backed memory, the entire 2MB region must be read from secondary storage. Third, a superpage mapping from a 2MB aligned virtual memory region to the physical superpage must be created. Fourth, the mapping must be destroyed. Finally, the physical memory must be deallocated. FreeBSD, Linux's Transparent Huge Pages (THP), and two recently proposed systems (Ingens [20] and HawkEye [24]) differ in when these events are triggered (for instance, these events can be independent, grouped, synchronous, asynchronous, etc.) and the granularity of the operations (for instance, some operations can be performed incrementally or all at once). This classification of the events enables a more principled comparison of the policies, behaviors, and performance of these different systems.

This paper also presents several new observations about transparent superpage management. First, coupling physical allocation, preparation, and mapping of superpages, as is done in Linux's THP, leads to memory bloat and fewer superpage mappings. Second, while alleviating tail latency problems in server workloads, state-of-the-art asynchronous, "out-of-place" promotion delays physical superpage allocation and reduces address translation benefits. Third, speculatively allocating physical superpages enables "in-place" promotion and obviates the need for asynchronous, out-of-place promotion. Fourth, in combination, reserving physical superpages and delaying partial deallocation of those superpages as long as possible fights fragmentation, leading to more superpage usage and address translation benefits. Finally, bulk zeroing is more efficient on modern processors than repeated 4KB zeroing. These observations are supported by evidence presented throughout the paper.

Finally, this paper introduces Quicksilver[1], an innovative transparent superpage management system that is based on FreeBSD's reservation-based physical memory allocator. Quicksilver achieves the benefits of aggressive superpage allocation, but mitigates the memory bloat and fragmentation issues that arise from underutilized superpages. Quicksilver is able to match or beat the performance of existing systems in scenarios with either lightly or heavily fragmented memory. For example, when using synchronous preparation, on a heavily fragmented system it achieves a 2x speedup over Linux for GraphChi performing PageRank on a dataset that exceeds the physical memory size. Furthermore, on Redis, Quicksilver is able to maintain the same throughput and tail latency as fragmentation increases, whereas the throughput of other systems degrades and tail latency increases. Finally, Quicksilver is able to limit memory bloat as well as Ingens [20], which is a recent research prototype specifically designed to combat memory bloat.

2 Transparent Superpage Management

Managing superpages transparently to the application involves five distinct events: physical superpage allocation, physical superpage preparation, superpage mapping creation, superpage mapping destruction, and physical superpage deallocation. Figure 1 illustrates the life cycle of a superpage in terms of these events. This section discusses the trade-offs between the possible choices, including those made by production and prototype systems [5, 16, 20, 23, 24], for when to trigger and how to handle these events.

2.1 Physical Superpage Allocation

The OS can choose to allocate a physical superpage to back any 2MB-aligned virtual memory region. A physical superpage could be allocated synchronously upon a page fault, or asynchronously via a background task. If there are free physical superpages, synchronous allocation is a relatively inexpensive operation given the widespread use of buddy allocators for physical memory management.

However, in order to allocate a physical superpage, the physical memory allocator must have an available, aligned, 2MB region. Under severe memory fragmentation, such regions may not be available. A memory manager could attempt to keep as many such regions available as possible (or create them when needed) using smart allocation policies or memory migration. If no such region is available or can be created, then the system must fall back to allocating 4KB pages.

Even after 4KB pages have been allocated for a virtual memory region, it is still possible to allocate a physical superpage for that region asynchronously. In the background, the OS can use migration to create free physical superpages or wait for them to be freed by applications. Once a free physical superpage exists, it could be allocated to a previously accessed virtual memory region. At that point, all previously allocated 4KB pages would need to be migrated into the newly acquired physical superpage.

2.2 Physical Superpage Preparation

Once a physical superpage has been allocated, it must be prepared with its initial data before it can be mapped. A physical superpage can be prepared in one of three ways. First, if the virtual memory region is anonymous, i.e., not backed by a file, then the page 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 4KB pages, then the contents of those existing pages must be copied into the physical superpage. Note that any constituent pages 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. Furthermore, as they are prepared, they can have some, or all, of their constituent pages mapped as 4KB pages (each constituent page that is mapped must have already been prepared). At a minimum, on a page fault, the 4KB 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 required 4KB page, or prepare the required 4KB page, allow the application to resume, and prepare the remaining 4KB pages later (either asynchronously or when they are accessed).

The three types of preparation — zeroing, copying, and file reading — have different costs, and so may impact the choice of when and how much of a physical superpage to prepare. Incremental preparation decreases page fault latency and minimizes unnecessary preparation for 4KB pages that may ultimately never get accessed. However, as the page is incrementally prepared, the constituent pages will be using 4KB mappings. In contrast, all at once preparation eliminates future page faults to the virtual memory region and allows for the immediate creation of a superpage mapping.

2.3 Superpage Mapping Creation

Once a physical superpage has been fully prepared, it must then be mapped as such in order to achieve address transla-

[1] https://github.com/rice-systems/quicksilver

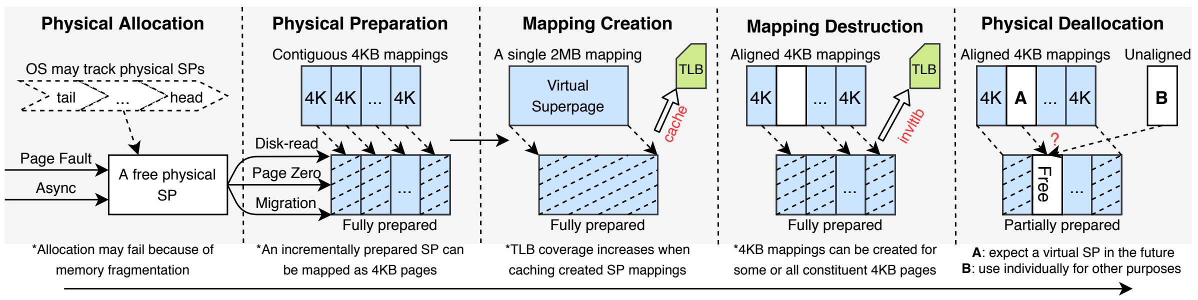

Figure 1: The five events in the life cycle of a superpage (SP).

tion benefits. Before the superpage is mapped, the physical memory can still be accessed via 4KB mappings; afterwards, the OS loses the ability to track accesses and modifications at a 4KB 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 future I/O.

A superpage mapping is typically 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 may also be created asynchronously. Note that on some architectures, *e.g.*, ARM, any 4KB mappings that were previously created must first be destroyed.

2.4 Superpage Mapping Destruction

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, 4KB mappings must be recreated for any constituent pages that have not been freed.

With superpage mappings, the OS cannot track whether constituent pages are accessed or modified. Therefore, in some scenarios, the OS may choose to preemptively destroy a superpage mapping and substitute 512 4KB mappings for it to enable finer-grained memory management. For example, when a clean superpage is first modified, the OS could choose to destroy the superpage mapping in order to only mark the single modified 4KB page as dirty, potentially reducing future I/O operations. This would require the OS to make a read-only superpage mapping and use the page fault caused by the write access to destroy the mapping and replace it with 4KB mappings. Similarly, the OS could choose to destroy a superpage mapping when under memory pressure to enable swapping pages at a finer granularity.

2.5 Physical Superpage Deallocation

Generally, 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 2MB can be returned to the physical memory allocator or the physical superpage can be "broken" into 4KB pages. If the physical superpage is broken into its constituent 4KB 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.

Before part or all of a physical superpage is returned to the physical memory allocator, any constituent pages that have been prepared but not freed must be preserved. Preservation typically happens in one of three ways. In-use pages can be kept rather than returned to the allocator, and 4KB mappings can be created to those pages. Alternatively, the in-use pages can be copied to other physical pages, allowing the entire physical superpage to be returned. The last option is to write the in-use pages to secondary storage before returning them.

3 State-of-the-art Designs

This section compares the state-of-the-art designs for transparent superpage management in FreeBSD, Linux, and recent research prototypes (Ingens [20] and HawkEye [24]), with a particular focus on how they manage the events described in the previous section.

3.1 FreeBSD

FreeBSD supports transparent superpages for all kinds of memory, including memory-mapped files and executables. It decouples physical superpage allocation from preparation by using a reservation-based memory allocator [23,29]. FreeBSD tries to allocate ("reserves") a physical superpage upon the first page fault to any aligned 2MB region. If physical superpages are available, they are allocated for any memory-mapped file exceeding 2MB in size. Anonymous memory always uses superpages if available, regardless of size, as anonymous memory is expected to grow.

Once a physical superpage is allocated for anonymous memory, only the 4KB page that caused the page fault is prepared, and a reservation entry is created to track all of the constituent pages. Any subsequent page fault to that 2MB region skips page allocation and simply prepares one additional 4KB page of the physical superpage. The physical superpage preparation finishes once all of its constituents have been prepared. For file-backed memory, the process is the same, except memory is prepared in 64KB batches to minimize I/O overhead.

FreeBSD creates superpage mappings synchronously during page faults. FreeBSD only creates a superpage mapping if the characteristics (*e.g.*, protection and modified state) of all the constituent 4KB mappings are the same. Identical protections are required for correctness; identical dirty states ensure that FreeBSD will not do unnecessary I/O to preserve the contents of the page when it is later deallocated.

Superpage mappings are destroyed on partial memory protection changes and partial unmappings. FreeBSD also preemptively destroys clean superpage mappings before modification. As a result, only one 4KB mapping is marked as dirty, instead of the entire superpage. Once the last clean 4KB page is modified, a dirty superpage mapping gets created.

FreeBSD defers physical superpage deallocation as long as possible in order to minimize memory fragmentation and preserve the availability of free physical superpages. However, under memory pressure, FreeBSD looks for a partially prepared physical superpage and breaks the corresponding reservation to allow the unused memory within that 2MB physical memory region to be reclaimed for other uses.

3.2 Linux

Linux's THP only uses superpages for anonymous memory and tries to allocate a physical superpage on the first page fault to a 2MB-aligned virtual memory region. If allocation fails and defragmentation is enabled (the default), it immediately does memory compaction via page migration to create a free physical superpage. This blocks the faulting process, increasing page fault latency. Under severe fragmentation, migration may still fail to create a free physical superpage.

Linux does all-at-once physical superpage preparation: the entire physical superpage is always zeroed right after being allocated. This increases the initial page fault latency, but may reduce the average latency [24]. After this preparation, a superpage mapping is immediately created. The superpage mapping will be destroyed if some or all of the superpage is unmapped or has its protection settings changed. Once some or all of the superpage has been freed, the physical superpage is deallocated and free memory is immediately reclaimed.

This "first-touch" superpage policy only allocates physical superpages at the time of the first page fault. However, Linux also includes a kernel daemon called "khugepaged", which asynchronously scans the system page tables. When it finds an aligned 2MB anonymous virtual memory region that contains at least one dirty 4KB mapping, khugepaged tries to allocate a physical superpage. If a free physical superpage exists, it acquires it; otherwise, it calls Linux's memory compaction to reclaim one by migrating pages.

Before preparing this physical superpage, khugepaged blocks accesses to the virtual 2MB region by blocking page faults within the region and destroying the existing 4KB mappings. It then prepares all of the physical superpage's constituent 4KB pages, one at a time. For a previously mapped page, the contents are copied. Previously unmapped pages are zeroed. Finally, it installs a superpage mapping.

Khugepaged's preparation is more costly than the first-touch preparation that occurs in a page fault. It blocks accesses to the 2MB region, causes TLB shootdowns, and pollutes CPU caches. As a result, it is allowed by default to allocate at most 8 superpages every 10 seconds (1.6 MB/s).

When an application partially frees memory within a superpage without unmapping the virtual memory (*e.g.*, `MADV_DONTNEED`), it triggers the destruction of the superpage mapping and the deallocation of the physical superpage. The remaining in-use memory then gets mapped as 4KB pages. However, when khugepaged scans this 2MB region, it will unnecessarily migrate the mapped memory into another allocated superpage and effectively reallocate the freed memory. It is precisely this behavior of khugepaged which has led to the severe memory bloating reported in recent work [20, 24].

3.3 Ingens and HawkEye

Recent state-of-the-art prototypes (Ingens [20] and HawkEye [24]) attempt to mitigate the page fault latency spikes incurred by Linux's first-touch superpage policy as well as the memory bloat incurred by khugepaged — behaviors which have led many to suggest that Linux's transparent superpage support be disabled for best performance.

Both systems disable Linux's first-touch policy, instead allocating, preparing, and mapping only a single 4KB page on a page fault. They then effectively modify khugepaged to more aggressively manage superpages.

Khugepaged's behavior differs in default Linux, Ingens, and HawkEye in terms of order, threshold, and rate for superpage creation. To prevent excessive memory bloat, Ingens increases the threshold to trigger creation of a superpage from one in-use 4KB page to 90% in-use, meaning there must be at least 460 4KB mappings in a 2MB region in order to create a superpage for that region.

Ingens maintains a list of candidate 2MB-aligned regions on page faults. As long as the list is not empty, Ingens keeps creating superpage mappings. However, asynchronous superpage creation introduces a fairness problem that the scanning order of page tables can lead to long delays for some processes. To alleviate this, Ingens prioritizes processes with fewer superpages. In addition, Ingens actively compacts non-referenced memory at an aggressive rate.

HawkEye uses the same threshold as default khugepaged: one 4KB page. Under memory pressure, it scans mapped superpages and makes their zero-filled 4KB pages copy-on-write to a single zero-filled page to reclaim memory.

HawkEye also maintains a list of candidate 2MB-aligned regions, but further weights them by their memory utilization, the process's resident size, sampled access frequency and

TLB overheads. HawkEye then creates a superpage mapping for the one with the most weight that is believed to bring the highest TLB overhead, called fine-grained superpage management in the paper [24]. It attempts to obtain considerable address translation benefits with fewer superpages.

HawkEye's fine-grained superpage management further consumes CPU resources besides the migration-based superpage mapping creations. To avoid interference with running processes, it uses the same promotion rate (1.6MB/s) as Linux's default khugepaged.

4 Analysis of Existing Designs

This section analyzes the designs for transparent superpage management described in the previous section and presents several novel observations about them. These observations motivate the design of Quicksilver.

Platforms. All designs were evaluated on an Intel E3-1245 v6-based server with maximum turbo performance and hyper-threading enabled. This server has 4 physical cores, 32GB DDR4 2400 ECC RAM, and a 256GB NVMe SSD. Linux version 4.3 was used, as both Ingens and HawkEye are based on that version. FreeBSD version 11.2 was used, upon which Quicksilver is built. Swapping is disabled under every OS.

Benchmarks. A large variety of benchmarks are evaluated. GUPS performs 2^{32} serial random memory accesses to 2^{30} 64-bit integers (8GB) [13]. Graphchi-PR, BlockSVM and ANN use out-of-core implementations to solve big-data tasks [21, 32]. Graphchi-PR computes 3 iterations of PageRank on the preprocessed Twitter-2010 dataset [19]. BlockSVM trains a classification model on the kdd2010-bridge dataset [28]. ANN randomly queries nearest neighbors on 2GB preprocessed hash tables. XSBench is a parallel computation kernel of the Monte Carlo neutron transport algorithm [30]. Canneal and freqmine are PARSEC benchmarks with large memory footprints [10]. Gcc, mcf, DSjeng and XZ are SPEC CPU2017 benchmarks with large memory footprints [11]. Buildkernel compiles the FreeBSD 11.2 kernel.

Graphchi-PR, XSBench, canneal and Buildkernel are multi-threaded to fully utilize CPU resources. Cold and Warm are Redis workloads benchmarking throughput and tail latency from a separate client machine with 8 threads and 16 request pipelines. The Cold workload populates an empty Redis instance with 16GB of 4KB objects. The Warm workload queries the fully populated 16GB Redis instance with a set/get ratio of 5:5 using 4KB objects. Del-70, Del-50, Range-S and Range-XL are Redis workloads benchmarking memory consumption. Del-70 and Del-50 insert 2 million 8KB objects and randomly delete 70% and 50% of them, respectively. Range-S and Range-XL insert randomly sized objects from small and large size ranges, respectively. Detailed benchmark settings and scripts can be found in the Quicksilver repository.

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.

Workload	Linux-4KB	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 1: Redis memory consumption. Linux-noKhugepaged disables khugepaged.

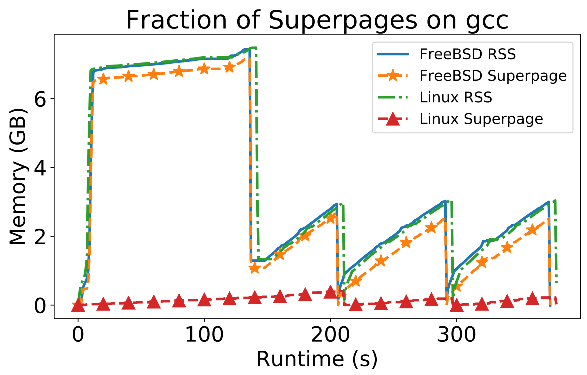

Figure 2: Linux's first touch policy fails to create superpages.

Linux's first-touch policy couples physical superpage allocation, preparation and superpage mapping creation together. As a result, it enjoys two obvious benefits. First, it provides immediate address translation benefits, including shorter page walk time and increased TLB efficiency. Second, it eliminates a large number of page faults for a heavily utilized superpage. Therefore, it is usually the best mapping policy when there is abundant contiguous free memory.

However, this coupled policy has several drawbacks. First, it can easily bloat memory and waste time preparing underutilized superpages. In a microbenchmark that sparsely touches 30GB of anonymous memory, Linux's first-touch policy takes 1.4s to run and consumes 30GB compared to 0.06s and 0.2GB when disabling transparent huge pages. While such a corner case is rare when applications use `malloc` to dynamically allocate memory, it may still happen in a long-running server, *e.g.*, Redis. Table 1 shows that Linux's first touch policy bloats memory by 78% compared to Linux-4KB on the workload Range-XL, which inserts objects of random sizes ranging from 256B to 1MB.

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 4KB page which later prevents creation of a superpage mapping when the heap grows. Figure 2 shows such behavior for gcc [11], which 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,

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

Table 2: Page fault latency. Bold numbers are estimations.

e.g., FreeBSD and Linux's khugepaged.

Third, it cannot be extended to larger anonymous or file-backed superpages. Table 2 estimates the page fault latency on both 1GB anonymous superpages and 2MB/1GB file-backed superpages. Faulting a 2MB file-backed superpage on the NVMe disk costs 1.7ms and faulting a 1GB anonymous superpage takes 46ms. 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 64KB and 2MB superpages.

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

Promotion-based policies can use 4KB 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 4KB 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 4KB 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 4KB physical pages and zero the remaining ones. At this time, previously created 4KB mappings are no longer valid.

Linux and recent prototypes [20,24] perform asynchronous, out-of-place promotion to hide the cost of page migration. As discussed in Section 3.2, Linux includes khugepaged as a supplement to create superpage mappings for growing heaps. 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 1 shows a memory bloat from 11.6GB to 19.8GB on workload Del-70, which randomly deletes 70% of the objects. On workload Range-XL, it bloats memory from 25.7GB to 30.7GB.

Ingens and HawkEye [20, 24] disable Linux's first-touch policy and instead improve the behavior and functionality of khugepaged, motivated by avoiding latency spikes in server workloads. 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 enhanced khugepaged and offloaded superpage allocations from the critical path, alleviating such latency spikes. So khugepaged works as their primary superpage management mechanism.

Workloads	Ingens	Ingens*	HawkEye	HawkEye*	FreeBSD
GUPS	0.87	0.84	0.28	0.88	0.96
Graphchi-PR	0.58	0.58	0.53	0.60	0.77
BlockSVM	0.81	0.79	0.73	0.81	0.96

Table 3: Speedup over Linux with unfragmented memory. All systems have worse performance than Linux.

However, out-of-place promotion delays physical superpage allocations and ultimately superpage mapping creation, because the OS must scan page tables to find candidate 2MB regions and schedule the background tasks to promote them. Table 3 compares in-place promotion (FreeBSD) with out-of-place promotion (Ingens and HawkEye) on applications where superpage creation speed is critical. While GUPS only involves random accesses, both Graphchi-PR and BlockSVM [21, 32] represent important real-life applications – using fast algorithms to process big data that cannot fit in memory. To better illustrate the problem, Ingens* and HawkEye* were tuned to be more aggressive, so that all 2MB regions containing at least one dirty 4KB mapping are candidates for promotion. Specifically, Ingens* uses a 0% utilization threshold instead of 90%; HawkEye* uses a 100% maximum CPU budget to promote superpages. However, Table 3 shows that FreeBSD consistently and significantly 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 4KB pages without page allocation. Therefore, the allocation of a physical superpage is immediate, but its superpage mapping creation is delayed. To bypass 4KB page allocations, it requires a bookkeeping system to track allocated physical superpages, *e.g.*, FreeBSD's reservation system. On x86-64, after it substitutes a superpage mapping for the 4KB mappings, it need not flush the previous 4KB mappings from the TLB.

FreeBSD implements an in-place promotion policy based on its reservation system as described in Section 3.1. It conservatively creates superpage mappings to avoid making performance worse. Navarro, *et al.* reported negligible overheads from the reservation system [23].

FreeBSD immediately allocates physical superpages but delays superpage mapping creation, sacrificing some address translation benefits. Table 3 shows that Linux consistently outperforms FreeBSD when memory is unfragmented, though

	Linux-4KB	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 4: Mean throughput and 95th latency of Redis Cold workload.

they 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, 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 4KB 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 4 evaluated the performance of Redis. Under fragmentation, Linux obtains slightly higher throughput but much higher tail latency than Linux-4KB.

FreeBSD delays the partial deallocation of a physical superpage to increase the likelihood of reclaiming a free physical superpage. When individual 4KB 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 [20]. These migrations, however, do consume processor and memory resources.

Observation 5: Bulk zeroing is more efficient on modern processors than repeated 4KB zeroing.

Modern OSes have abandoned asynchronous page zeroing because it usually degrades performance in a multiprocess situation. Furthermore, the introduction of ERMS (Enhanced REP MOVSB/STOSB) has accelerated page zeroing. However, existing OSes fail to fully exploit the benefits of ERMS support, because they still zero pages 4KB at a time. Modern CPUs can zero a 2MB page much faster with bulk zeroing, which calls the assembly language page zeroing code at a size larger than 4KB. Table 5 compares 2MB zeroing speed on

CPU (GHz)	DRAM	temporal			non-temporal		
Bulk Size:	(MHz)	4KB	32KB	2MB	4KB	32KB	2MB
E3-1231v3 (3.40)	1600	92	88	87	114	99	97
E3-1245v6 (3.70)	2400	84	67	65	92	74	71
E5-2640v3 (2.60)	1866	355	287	280	154	112	106
E5-2640v4 (2.40)	2133	409	334	325	163	113	106
R7-2700X (4.30)	2666	185	183	159	99	60	53

Table 5: 2MB page zeroing time (us) drops consistently using a larger bulk size.

five modern machines. Existing OSes take 84–409us to zero a 2MB superpage. After using a larger bulk size, the range is improved to 67–334us. Furthermore, these machines have a consistently short non-temporal (`moventi` or `clzero`) bulk zeroing latency (53–106us). The AMD Ryzen 7 2700X CPU achieves 53us with the highest CPU and DRAM frequency and its specific `clzero` implementation.

5 Design and Implementation

This section describes Quicksilver, an improved transparent superpage management system based upon the observations from the previous section. To benefit from in-place promotions, Quicksilver is built upon FreeBSD's reservation-based superpage management strategy.

5.1 Design

Aggressive Physical Superpage Allocation. Section 4 shows that aggressive allocation on first touch (as done by Linux and FreeBSD) is effective. Moreover, Observation 3 shows that FreeBSD's reservation system allows speculative physical allocation for anonymous memory and creates even more superpages than Linux, as shown in Figure 2. Since it also supports multiple superpage sizes and avoids memory bloating, Quicksilver retains FreeBSD's reservation system: allocating physical superpages when virtual memory regions that may use superpages are first accessed. Allocation is performed synchronously to avoid page migrations.

The drawbacks of FreeBSD's use of reservations are twofold. First, FreeBSD delays preparation and mapping of superpages, resulting in lower performance than Linux in some scenarios, as shown in Table 3. However, this is not inherent in the use of reservations for allocation, but rather should be addressed via preparation and mapping policies. Second, holding underutilized physical superpages in reservations can prevent future superpage allocations. However, this is better resolved via deallocation policies that recognize and recover from such situations.

Hybrid Physical Superpage Preparation. 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 prompt address translation benefits. Therefore, Quicksilver has an additional threshold, t. Once t 4KB pages get prepared, it prepares the remainder of the superpage all-at-once.

This reduces bloat, as discussed in Observation 1, 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. The use of a threshold is further based on previous work showing that the utilization of physical superpages is largely bimodal [34]. Once more than about 64 4KB pages have been accessed, it is very likely that the physical superpage will eventually be fully populated (or very nearly so). Therefore, at that point, it is very likely to be beneficial to prepare the remainder of the page and create a superpage mapping for it. Motivated by Observation 5, bulk zeroing is used to accelerate page zeroing when zero-filling the remainder of the superpage.

Relaxed Superpage Mapping Creation. Once an entire physical superpage has been prepared, there is little downside to immediately creating a superpage mapping for anonymous memory, which is rarely, if ever, swapped in modern systems. Therefore, Quicksilver relaxes FreeBSD's current design — which does not create a superpage mapping if the accessed or modified states of the constituent pages differ — to always create a mapping once the physical superpage has been fully prepared, as do Linux, Ingens, and HawkEye.

For file-backed superpages, Quicksilver retains FreeBSD's write-protection mechanism to avoid extra disk I/O, but no longer examines if all constituent pages are accessed. Because memory-mapped files are usually prefetched 64KB at a time, file-backed superpages may not be fully accessed when they get fully prepared. By allowing different access bits, more file-backed superpage mappings can be created. Note that Linux and its variants do not use superpages at all for files.

On-demand Superpage Mapping Destruction. There is no reason to destroy a superpage mapping unless some or all of the memory within the superpage is freed, its protection is changed, or the physical superpage must be deallocated to reclaim memory. Therefore, Quicksilver maintains FreeBSD's policy of only destroying mappings in the aforementioned situations.

Preemptive Physical Superpage Deallocation. As discussed in Observation 4, delaying partial deallocation of physical superpages effectively limits fragmentation. However, to maximize the effectiveness of synchronous physical superpage allocation, there must be available superpages to allocate. Superpage availability can have a considerable impact on performance as was shown in Table 4. Therefore, Quicksilver maintains a target number of free physical superpages.

Underutilized reservations that are inactive for a long period are preemptively deallocated. These partially prepared physical superpages are not yet mapped as superpages, so the deallocation reduces memory bloat and recovers memory contiguity. Such preemptive deallocation copes well with hybrid preparation under a population threshold t. As a result, preemptive deallocation usually evacuates underutilized and less frequently accessed superpages.

This approach has three advantages. First, fewer pages are migrated. Second, the preemptive migration happens in the background, so it does not happen on the critical path of any OS function executed by the application. Finally, it is likely to have minimal impact on running processes, as it is operating on pages that come from less frequently accessed superpages.

5.2 Implementation

Quicksilver was implemented within FreeBSD 11.2. Quicksilver focuses on anonymous memory, with FreeBSD's superpage support for file-backed memory slightly improved (access bit equivalence is no longer required for promotion). This section further describes the page zeroing mechanism and the migration/deallocation daemon.

Hybrid Preparation. A physical superpage is incrementally prepared until it reaches a population threshold, t. Then the remainder of the physical superpage is prepared by zero-filling it. The system can do this either synchronously or asynchronously, named Sync-t and Async-t. Specifically, Async-t periodically scans the linked list of partially populated physical reservations and starts zero-filling from the most active ones reaching the population threshold t. Therefore, it incurs no fairness issue because the order is determined by physical allocation activity, not process IDs.

In both cases, zero-filling uses non-temporal stores. When using Sync-t, pages are zeroed using the largest bulk size possible, as motivated by Observation 5. Since zeroing is done by the page fault handler, the page fault handler can create a superpage mapping immediately after zeroing is complete. When using Async-t, 4KB pages are zeroed individually. While this yields lower zeroing performance, it reduces lock contention when operating on pages. Since zeroing is done asynchronously and independently of any process's virtual address space, a superpage mapping is not created until the first soft page fault after all pages have been zeroed.

Relaxed Mapping Creation. For anonymous memory, the superpage mapping creation condition is relaxed to ignore checking for dirty and access bits. This allows a superpage mapping to be created immediately after Sync-t completes the zero-filling (these pages are clean). For file-backed memory, superpage mappings are created on a soft page fault of a file-backed physical superpage. Default FreeBSD skips mapping creation because the access bits are inferred not to all be set when prefaulting the prefetched disk data. After relaxing the access bit checking, Quicksilver tries to create a superpage mapping at that point.

Preemptive Deallocation. Physical superpages are often underutilized [20, 34]. Given Observation 4, the system delays

Threads	Linux		FreeBSD	
	default	aggressive	default	emulate Linux ELF
1	1.05	1.19	1.15	1.16
8	**1.07**	0.91	1.11	**1.18**

Table 6: Canneal performance speedup. Only bold numbers are comparable.

Linux		FreeBSD		
default	aggressive	default	patched [1]	match Linux
1.01	1.24	1.02	1.07	**1.19**
0.4 K	8.2 K	0.0 K	1.2 K	**8.2 K**

Table 7: Throughput speedup and number of created superpage mappings of a Redis server populated by Del-70. Only bold numbers are comparable.

partial deallocation of physical superpages. However, to ensure that there are sufficient free physical superpages for future allocations, Quicksilver uses an evacuation daemon to reclaim free physical superpages by preemptively deallocating underutilized physical superpages.

The daemon maintains a target number of free physical superpages. It periodically scans the list of partially populated reservations and examines their inactive time, during which they are neither populated nor deallocated. If they are inactive for a long time, *e.g.*, 5 seconds, the daemon then reclaims a free physical superpage by migrating out its constituent 4KB pages. To avoid contention with running applications, the daemon is restricted to use a maximum memory bandwidth of 1GB/s. This is less than 5% of the evaluated machine.

6 Methodology

Fragmentation. Three fragmentation levels are modeled to mimic long-running servers, named Frag-0, Frag-50 and Frag-100. They represent situations from non-fragmented to severely-fragmented. Specifically, Frag-50 leaves 50% of the application's maximum resident memory as free superpages.

The three fragmentation levels are crafted by a user-space tool which works under a first-touch physical superpage allocation policy (available in both Linux and FreeBSD). It first fragments superpages until there is memory pressure, then starts over and fragments a target number of superpages. Unlike a previous memory fragmentation method [24] that only performs the latter step, Linux's memory compaction usually fails either in page faults or khugepaged. To fragment a superpage, the tool touches part of a 2MB-aligned virtual region and unmaps the untouched part. This will trigger a physical superpage allocation and force a partial deallocation, fragmenting that physical superpage.

Library Differences. FreeBSD dynamically links executable files with its natively shipped libc, while Linux uses GNU libc. This makes any performance comparison between FreeBSD and Linux unfair, because a different implementation of a standard function may change performance significantly. For example, the libc string library in FreeBSD 11.2 does not use ERMS optimizations, so memory copy-intense applications have worse performance. To remove this difference, applications were compiled and statically linked on Linux and then run on FreeBSD using FreeBSD's Linux system call emulation. Table 6 shows that natively compiled canneal on FreeBSD runs slower than emulated canneal, because of slower memory copying in dynamic array resizing. Although a libc library with ERMS optimizations could be ported from FreeBSD 12.0, this methodology ensures that the exact same binaries are run on all systems, eliminating any possible library differences.

There are three exceptions. GraphChi-PR uses `dlopen` to dynamically link the openmp library, so it cannot be statically compiled. Redis calls gettimeofday() very frequently, causing huge emulation overhead. Therefore, they are compiled natively on FreeBSD-based systems after porting the libc library from FreeBSD 12.0. They therefore may have minor library-induced performance differences between the Linux-based and FreeBSD-based systems. Lastly, FreeBSD's Linux emulation caused significant performance degradation on GUPS, because of cache misses resulting from an unaligned dynamically allocated data structure. To fix this, GUPS was modified to use `malloc_aligned`.

System Tuning. When there are idle CPUs, tuning Linux's khugepaged to be more aggressive can obtain better performance. Table 6 shows this in a single-threaded case. This tuning also yields higher throughput for single-threaded Redis, as shown in Table 7. However, performance degrades when the application uses all CPUs and competes with khugepaged, so Linux remains unchanged for the remainder of the evaluation.

FreeBSD 11.2 has suboptimal Redis performance due to three reasons. First, it uses a network socket buffer size suitable for 1Gbps NICs. Second, its libc has no ERMS optimizations, while memory copying dominates Redis's performance. Third, it is unlikely to repromote superpages after `MADV_FREE` (Redis uses `MADV_FREE` on FreeBSD to save page faults). Therefore, FreeBSD was tuned to use the correct buffer size for a 40Gbps NIC, and the libc library was ported from FreeBSD 12.0. Additionally, a recent patch [1] to FreeBSD was applied to increase the likelihood of superpage repromotion, creating 1.2K more superpage mappings in Table 7. The dirty bit requirement for anonymous memory was relaxed to match Linux's performance, creating 8.2K superpage mappings.

Ingens and HawkEye are evaluated with their default settings. Ingens promotes superpages with a 90%-utilization threshold. HawkEye promotes superpages at the speed of 1.6MB/s guided by performance counters. Ingens* and HawkEye* are aggressively tuned variants. Specifically, Ingens* uses a utilization threshold of 0% instead of 90% and enables 1GB/s proactive memory compaction. HawkEye* uses a 100% CPU maximum with a promotion threshold of 1.

7 Evaluation

Four variants of Quicksilver are considered, named Sync-1, Sync-64, Async-64 and Async-256. They all handle the five superpage events similarly except for superpage preparation. Therefore, for clarity they are named after their preparation policies. These four variants represent reasonable design points in the Sync-*t* and Async-*t* space, and use the same 1GB/s active defragmentation daemon. They share the same library and system tunings with FreeBSD. All performance numbers are the mean of three runs.

7.1 Non-fragmented (Frag-0) Performance

Sync-1 vs. Linux. Sync-1 uses the same superpage preparation and mapping policy for anonymous memory as Linux. With no fragmentation, Tables 8 and 9 show that 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 and gcc. Their similar speedups on reservation-based systems validate Observation 3. Second, Sync-1 creates file-backed superpages and outperforms Linux on ANN and Graphchi-PR.

Promotion Speed. Under Frag-0, FreeBSD often outperforms Ingens, HawkEye and their aggressively tuned variants, as shown in Table 8. This validates Observation 2, as the issue is that out-of-place promotion has a slower promotion speed. Furthermore, as shown in Table 9, on the Redis Cold workload, Ingens, HawkEye and their aggressively tuned variants even show a slight degradation compared to Linux-4KB. These systems introduce some noticeable interference with running applications when they manage superpages in the background.

Sync-64 mostly 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.

7.2 Performance Under Fragmentation

Table 9 shows that Linux obtains a much higher tail latency on the Redis Cold workload under Frag-50/100 than Linux-4KB, 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, as shown in Table 8, their speedup over Linux increases as fragmentation increases. However, HawkEye does not achieve the same speedups on XSBench that were reported in the original paper [24], because in these application runs, most of its memory compaction fails and its important data was not allocated at the high end of the address space.

The four variants of Quicksilver all consistently perform well on both non-server and server workloads, because their background defragmentation not only avoids increasing page fault latency, but also succeeds in recovering unfragmented performance. Specifically, on the Redis Cold workload, Sync-1 maintained the highest throughput (1.31 GB/s) while providing low (4.5 ms) tail latency under Frag-100. However, the per-second background scanning of the evacuation daemon may fail to improve performance when applications quickly touch all of their memory in the beginning (*e.g.* GUPS and ANN). As a result, there is high performance variation on GUPS and ANN performance is not improved over other systems, as shown in Table 8.

Graphchi-PR. On all applications in Table 8, the Sync-*t* and Async-*t* systems all match or outperform Linux. Since Graphchi-PR is an important and real-world task, it is selected to elaborate how the design choices described in Section 5 contribute to the 2.18 speedup of Sync-1 under Frag-100.

Under Frag-100, Async-64 obtains a speedup of 1.68, which is higher than the 1.15 speedup obtained by Ingens* on Graphchi-PR. When Graphchi-PR terminated, Ingens* has a total of 1,926 (mean of 3 runs) free physical superpages, but Async-64 has 11,955 free physical superpages. Although they have the same memory bandwidth budget (1GB/s) for active defragmentation, Quicksilver's evacuation daemon defragments memory more efficiently by identifying inactive fragmented superpages. The in-place promotions further contribute to the higher speedup of Async-64. When memory is not fragmented, Async-64 obtains a speedup of 0.83, higher than all other non-Quicksilver systems.

Sync-64 obtains an even higher speedup of 2.11. The shared evacuation daemon allows both Async-64 and Sync-64 to allocate a similar number of superpages, but 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, Sync-64 is able to reduce page walk pending cycles by 76%. The highest speedup is obtained by Sync-1 with a more aggressive promotion threshold.

7.3 Memory Bloat

All systems suffer less than 1% memory bloat compared to Linux-4KB on the applications shown in Table 8. However, long-running servers may still suffer from memory bloat. When applications frequently allocate and deallocate memory, an aggressive superpage preparation policy may preemptively prepare a superpage and sacrifice free memory for minor address translation benefits, ultimately creating false memory pressure.

Frag-0	GUPS	Graphchi-PR	BlockSVM	XSBench	ANN	canneal	freqmine	gcc	mcf	DSjeng	XZ
Ingens	0.87	0.58	0.81	0.98	1.00	0.95	0.99	1.00	0.99	0.99	0.96
Ingens*	0.84	0.58	0.79	0.97	0.97	0.92	0.99	1.01	0.96	0.99	0.92
HawkEye	0.28	0.53	0.73	0.88	1.00	0.95	0.99	0.99	0.94	0.86	0.90
HawkEye*	0.88	0.60	0.81	0.98	1.00	0.97	1.00	0.99	0.97	0.99	0.94
FreeBSD	0.96	0.77	0.96	0.99	0.98	1.14	1.00	1.05	0.99	1.00	0.99
Sync-1	0.99	1.07	1.00	1.00	1.07	1.14	0.99	1.05	1.00	1.00	1.00
Sync-64	0.98	1.05	1.00	1.00	1.08	1.14	0.99	1.05	1.00	1.00	1.00
Async-64	0.96	0.83	0.97	0.99	1.08	1.14	1.00	1.05	1.00	1.00	0.99
Async-256	0.96	0.82	0.97	0.99	1.08	1.14	0.99	1.05	0.99	1.00	0.99

Frag-50	GUPS	Graphchi-PR	BlockSVM	XSBench	ANN	canneal	freqmine	gcc	mcf	DSjeng	XZ
Ingens	0.98	0.71	0.82	1.01	1.00	0.99	1.00	1.00	1.00	1.00	0.99
Ingens*	1.24	0.73	0.86	1.00	0.98	1.00	0.99	1.02	0.99	1.04	0.97
HawkEye	0.62	0.68	0.77	0.91	1.00	0.96	1.00	0.99	0.97	0.92	0.94
HawkEye*	0.89	0.68	0.80	1.00	0.99	0.99	1.00	0.99	0.99	0.98	0.99
FreeBSD	0.98	0.94	0.89	1.02	0.97	1.01	1.00	1.05	1.01	1.02	1.01
Sync-1	2.04(0.08)	1.37	1.04	1.03	1.04	1.17	1.00	1.05	1.03	1.05	1.05
Sync-64	2.01	1.32	1.06	1.03	1.04	1.18	1.00	1.05	1.03	1.06	1.05
Async-64	2.11	1.06	1.02	1.03	1.03	1.17	1.00	1.05	1.03	1.06	1.04
Async-256	2.11	1.05	1.02	1.03	1.03	1.17	1.00	1.05	1.03	1.06	1.04

Frag-100	GUPS	Graphchi-PR	BlockSVM	XSBench	ANN	canneal	freqmine	gcc	mcf	DSjeng	XZ
Ingens	1.02	1.13	0.86	1.04	1.00	1.00	1.00	1.00	1.01	1.01	1.02
Ingens*	1.30	1.15	0.88	1.13	0.99	1.06	1.00	1.02	1.03	1.08	1.06
HawkEye	0.97	1.11	0.85	1.03	1.00	1.01	1.00	1.00	0.99	0.97	1.02
HawkEye*	0.96	1.11	0.84	1.03	1.00	1.01	1.00	0.99	0.99	0.97	1.01
FreeBSD	0.96	1.10	0.85	1.04	0.98	1.05	1.00	1.00	1.00	1.04	1.02
Sync-1	2.35(0.30)	2.18	1.12	1.07	1.04	1.12	1.00	1.05	1.02	1.10	1.14
Sync-64	2.29(0.14)	2.11	1.13	1.07	1.01	1.12	1.00	1.05	1.05	1.11	1.14
Async-64	1.91(0.21)	1.68	1.11	1.06	0.98	1.12	1.00	1.05	1.05	1.11	1.13
Async-256	2.10(0.22)	1.65	1.10	1.08	0.98	1.16	1.00	1.06	1.05	1.08	1.13

Table 8: Performance speedup over Linux under three fragmentation levels. Red boxes indicate that the system performs worse than Linux on that application. The normalized standard deviation of runtime is no greater than 5% unless specified in parentheses.

Table 10 compares the memory consumption of four Redis workloads. Among these workloads, Linux bloats memory the most, consistent with previous findings [20]. However, Sync-1 exhibits lower memory consumption than Linux despite similar policies. In fact, it is khugepaged that bloats memory. When a partially deallocated superpage is scanned, it allocates the memory back to recreate a superpage, undermining the application's efforts to free and defragment memory.

All systems other than Linux limit memory consumption for the first three workloads; they only really differ on Range-XL. HawkEye, FreeBSD, and Async-256 exhibit the lowest memory consumption on Range-XL, whereas the other systems bloat memory by 40–60%. HawkEye stops allocating superpages when the TLB overhead is minor, FreeBSD only promotes fully utilized superpages, and Async-256 has a conservative promotion threshold.

Sync-1 vs. Sync-64 Besides bloating memory, aggressive preparation policies may cause excessive creation of superpages. This is common when many small processes get forked. For example, Table 11 shows what happens in a 9-threaded compilation of the FreeBSD kernel. Sync-1 creates more than 200k superpages, while the less aggressive Sync-64 only creates around 100k. Over half of the superpages created by Sync-1 had less than 13% utilization. Consequently, Sync-1 spends 13.9% more system time preparing them, which outweighs their benefits. In a long running server, using an aggressive policy like Sync-1 could waste both power and memory contiguity by creating underutilized superpages. In contrast, Sync-64 avoids such cases and suffers from less performance degradation than Sync-1 in both Table 8 and Table 9. Therefore, it is more preferable for long-running servers.

8 Related Work

Direct segments have been proposed as a supplement to existing page-based address translation for large-memory applications [9, 14, 18]. While they are effective at reducing the cost of address translation, they are limited to systems that allocate nearly all of the system memory to a single application with the same access rights. While these ideas can be generalized to some degree, they ultimately limit the flexibility of the OS to allocate and use physical memory.

Automatic TLB entry coalescing to increase the effective

Cold	Linux-4KB	Linux	Ingens	Ingens*	HawkEye	HawkEye*	FreeBSD	Sync-1	Sync-64	Async-64	Async-256
Frag-0	1.04(5.6)	1.34(4.1)	1.00(5.9)	0.98(6.3)	1.00(5.9)	1.00(5.8)	1.11(6.1)	1.26(4.5)	1.20(4.8)	1.10(6.0)	1.11(6.0)
Frag-50	1.04(5.7)	0.92(10.2)	0.95(5.9)	0.97(5.9)	1.02(5.9)	1.03(5.8)	1.04(6.2)	1.25(4.5)	1.27(4.7)	1.09(6.0)	1.09(6.0)
Frag-100	1.07(5.6)	0.81(9.9)	0.94(6.1)	0.97(6.1)	1.00(5.8)	1.02(5.8)	0.98(6.5)	1.31(4.5)	1.26(4.6)	1.14(5.9)	1.08(5.9)
Warm	Linux-4KB	Linux	Ingens	Ingens*	HawkEye	HawkEye*	FreeBSD	Sync-1	Sync-64	Async-64	Async-256
Frag-0	1.06(6.5)	1.32(5.2)	1.23(5.7)	1.21(5.8)	1.03(6.7)	1.06(6.5)	1.30(5.6)	1.32(5.5)	1.31(5.5)	1.31(5.5)	1.30(5.6)
Frag-50	1.07(6.5)	1.17(5.9)	1.09(6.4)	1.19(5.8)	1.03(6.7)	1.05(6.7)	1.18(6.1)	1.32(5.5)	1.32(5.5)	1.31(5.5)	1.31(5.5)
Frag-100	1.07(6.5)	1.16(5.9)	1.01(6.9)	1.09(6.4)	1.05(6.6)	1.07(6.5)	1.10(6.6)	1.33(5.4)	1.34(5.5)	1.33(5.4)	1.31(5.5)

Table 9: Redis throughput (GB/s) and 95th latency (ms) of workloads Cold and Warm. Numbers in parentheses are 95th latencies. The maximum standard deviation is 0.04GB/s for throughput and 0.57ms for 95th latency.

Workload	Linux-4KB	Linux	Ingens	Ingens*	HawkEye	HawkEye*	FreeBSD	Sync-1	Sync-64	Async-64	Async-256
Del-70	11.6	19.8	11.6	11.7	11.6	11.6	11.6	11.6	11.6	11.6	11.6
Del-50	16.7	19.8	16.8	16.8	16.7	16.9	16.7	16.8	16.8	16.8	16.8
Range-S	14.3	15.6	16.0	15.6	14.9	14.5	14.3	15.6	15.6	15.3	15.1
Range-XL	14.4	30.7	22.7	23.3	15.7	20.6	14.9	23.1	20.9	19.5	15.9

Table 10: Redis memory consumption (GB) of four workloads. Khugepaged further bloats memory in Linux.

Buildkernel	real	user	sys	# SP	# PF
Sync-1	197.7	1409.4	89.4	200.5 K	5.3 M
Sync-64	196.9	1408.8	78.5	99.6 K	10.3 M
FreeBSD	203.7	1436.7	98.0	36.9 K	30.2 M

Table 11: Runtime (seconds) and numbers of superpages and page faults of compiling the FreeBSD 11.2 kernel.

reach of the TLB has been proposed and implemented [26,27]. Essentially, a page walk will load multiple 4KB mappings found in the same cache line. If these mappings refer to contiguous pages and have identical access privileges, then they are merged into a single TLB entry. Although TLB entry coalescing occurs automatically in hardware, it nonetheless requires the OS to allocate physically contiguous memory. AMD Ryzen processors do such coalescing [12].

A large body of work has shown that using superpages can reduce the cost of address translation. Originally, OS support for superpages required the administrator to manually control the use of superpages. For example, Linux has long supported *persistent huge pages* [4]. A huge page pool with a static number of huge pages must be allocated by the administrator before running applications. The persistent huge pages are pinned in memory and can only be used via specific flags to `mmap` system calls. Superpage support in Windows and OS X are similar to Linux persistent huge pages [3,6].

To eliminate the need for manual control, FreeBSD, Linux, and many research prototypes have explored transparent superpage support, as described in Section 3. This support has been extensively described and studied [16,17,20,23,24,29]. As this transparent support for superpages has become widely available in production OSes, many people have argued that effectively handling all of the issues that can arise still requires further improvements to OS memory management support [15–17,20,22,24,25]. For example, some of these people have worked to improve Linux's superpage management by decreasing memory fragmentation and more carefully allocating physical superpages using Linux's idle page tracking mechanisms [20,22,24,25,31]. Others have shown that it is beneficial to decrease memory fragmentation and increase the contiguity of physical memory. To achieve this, several efforts have focused on minimizing migration and reducing its performance impact, while still attempting to reduce fragmentation and increase contiguity [7,8,22,25,31]. The deprecated lumpy reclaim from Linux was also developed to increase contiguity [2]. It reclaims a 2MB superpage by finding an inactive 4KB page and swaps out all dirty 4KB pages inside the 2MB block. Because these dirty 4KB pages may also contain active ones, swapping them out may hurt performance instead. Besides efforts on anonymous superpages, Zhou, *et al.* augmented FreeBSD to synchronously page-in code and pad code sections to create more code superpages [33].

9 Conclusions

This paper has performed a comprehensive analysis of superpage management mechanisms and policies. 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 five 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. However, Sync-64 is preferable for long-running servers, as it does not aggressively create underutilized superpages.

References

[1] FreeBSD MADV_FREE heuristics. https://svnweb.freebsd.org/base?view=revision&revision=350463. Viewed 2020-05-31.

[2] Linux's lumpy reclaim. https://lkml.org/lkml/2012/3/28/323. Viewed 2020-05-31.

[3] OS X superpage support. https://www.unix.com/man-page/osx/2/mmap/. Viewed 2020-05-31.

[4] Persistent huge pages in Linux. https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt. Viewed 2020-05-31.

[5] Transparent huge pages in Linux. https://www.kernel.org/doc/Documentation/vm/transhuge.txt. Viewed 2020-05-31.

[6] Windows large page support. https://docs.microsoft.com/en-us/windows/desktop/memory/large-page-support. Viewed 2020-05-31.

[7] Neha Agarwal and Thomas F Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *ACM SIGARCH Computer Architecture News*, volume 45, pages 631–644. ACM, 2017.

[8] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J Rossbach, and Onur Mutlu. Mosaic: Enabling application-transparent support for multiple page sizes in throughput processors. *ACM SIGOPS Operating Systems Review*, 51(1):27–44, 2018.

[9] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient virtual memory for big memory servers. In *The 40th Annual International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, June 23-27, 2013*, pages 237–248, 2013.

[10] 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*, pages 72–81. ACM, 2008.

[11] James Bucek, Klaus-Dieter Lange, et al. SPEC CPU2017: next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 41–42. ACM, 2018.

[12] Mike Clark. A new x86 core architecture for the next generation of computing. In *Hot Chips 28 Symposium (HCS), 2016 IEEE*, pages 1–19. IEEE, 2016.

[13] II Earl Joseph. Gups (giga-updates per second) benchmark. *URL http://www. dgate. org/~ brg/files/dis/gups*, 2000.

[14] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Efficient memory virtualization: Reducing dimensionality of nested page walks. In *47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2014, Cambridge, United Kingdom, December 13-17, 2014*, pages 178–189, 2014.

[15] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. Large pages may be harmful on NUMA systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 231–242, 2014.

[16] Mel Gorman and Patrick Healy. Supporting superpage allocation without additional hardware support. In *Proceedings of the 7th international symposium on Memory management*, pages 41–50. ACM, 2008.

[17] Mel Gorman and Patrick Healy. Performance characteristics of explicit superpage support. In *International Symposium on Computer Architecture*, pages 293–310. Springer, 2010.

[18] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman S. Unsal. Redundant memory mappings for fast access to large memories. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*, pages 66–78, 2015.

[19] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, pages 591–600. AcM, 2010.

[20] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with ingens. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 705–721, 2016.

[21] Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 31–46, 2012.

[22] Theodore Michailidis, Alex Delis, and Mema Roussopoulos. Mega: overcoming traditional problems with os huge page management. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 121–131. ACM, 2019.

[23] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan L. Cox. Practical, transparent operating system support for superpages. In *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA, December 9-11, 2002*, 2002.

[24] 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*, pages 347–360. ACM, 2019.

[25] Ashish Panwar, Aravinda Prasad, and K Gopinath. Making huge pages actually useful. In *ACM SIGPLAN Notices*, volume 53, pages 679–692. ACM, 2018.

[26] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H. Loh. Increasing TLB reach by exploiting clustering in page translations. In *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*, pages 558–567, 2014.

[27] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. Colt: Coalesced large-reach tlbs. In *45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2012, Vancouver, BC, Canada, December 1-5, 2012*, pages 258–269, 2012.

[28] J Stamper, A Niculescu-Mizil, S Ritter, GJ Gordon, and KR Koedinger. Bridge to algebra 2008–2009. *Challenge data set from KDD Cup*, 2010.

[29] Madhusudhan Talluri and Mark D. Hill. Surpassing the TLB performance of superpages with less operating system support. In *ASPLOS-VI Proceedings - Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, USA, October 4-7, 1994.*, pages 171–182, 1994.

[30] 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.

[31] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Translation ranger: operating system support for contiguity-aware tlbs. In *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 698–710, 2019.

[32] Hsiang-Fu Yu, Cho-Jui Hsieh, Kai-Wei Chang, and Chih-Jen Lin. Large linear classification when data cannot fit in memory. In *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

[33] Yufeng Zhou, Xiaowan Dong, Alan L Cox, and Sandhya Dwarkadas. On the impact of instruction address translation overhead. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 106–116. IEEE, 2019.

[34] Weixi Zhu. Exploring superpage promotion policies for efficient address translation. Master's thesis, Rice University, 6100 Main St, Houston, TX 77005, 2019.