Coordinated and Efficient Huge Page Management with Ingens

Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach[1], Emmett Witchel
The University of Texas at Austin
[1]*The University of Texas at Austin and VMware Research Group*

Abstract

Modern computing is hungry for RAM, with today's enormous capacities eagerly consumed by diverse workloads. Hardware address translation overheads have grown with memory capacity, motivating hardware manufacturers to provide TLBs with thousands of entries for large page sizes (called huge pages). Operating systems and hypervisors support huge pages with a hodge-podge of best-effort algorithms and spot fixes that made sense for architectures with limited huge page support, but the time has come for a more fundamental redesign.

Ingens is a framework for huge page support that relies on a handful of basic primitives to provide transparent huge page support in a principled, coordinated way. By managing contiguity as a first-class resource and by tracking utilization and access frequency of memory pages, Ingens is able to eliminate a number of fairness and performance pathologies that plague current systems. Experiments with our prototype demonstrate fairness improvements, performance improvements (up to 18%), tail-latency reduction (up to 41%), and reduction of memory bloat from 69% to less than 1% for important applications like Web services (e.g., the Cloudstone benchmark) and the Redis key-value store.

1 Introduction

Modern computing platforms can support terabytes of RAM and workloads able to take advantage of such large memories are now commonplace [51]. However, increased capacity represents a significant challenge for address translation. All modern processors use page tables for address translation and TLBs to cache virtual-to-physical mappings. Because TLB capacities cannot scale at the same rate as DRAM, TLB misses and address translation can incur crippling performance penalties for large memory workloads [44, 53] when these workloads use traditional page sizes (i.e., 4KB). Hardware-supported address virtualization (e.g., AMD's nested page tables) increases average-case address translation overhead because multi-dimensional page tables amplify worst-case translation costs by 6× [59]. Hardware manufacturers have addressed increasing DRAM capacity with better support for larger page sizes, or *huge pages*, which reduce address translation overheads by reducing the frequency of TLB misses. However, the success of these mechanisms is critically dependent on the ability of the operating systems and hypervisors to manage huge pages.

While huge pages have been commonly supported in hardware since the 90s [75, 76], until recently, processors have had a very small number of TLB entries reserved for huge pages, limiting their usability. Newer architectures support thousands of huge page entries in dual-level TLBs (e.g., 1,536 in Intel's Skylake [1]), which is a major change: the onus of better huge page support has shifted from the hardware to the system software. There is now both an urgent need and an opportunity to modernize memory management.

Operating system memory management has generally responded to huge page hardware with best-effort algorithms and spot fixes, choosing to keep their management algorithms focused on the 4KB page (which we call a *base page*). For example, Linux and KVM (Linux's in-kernel hypervisor) adequately support many large-memory workloads (i.e., ones with simple, static memory allocation behavior), but a variety of common workloads are exposed to unacceptable performance overheads, wasted memory capacity, and unfair performance variability when using huge pages. These problems are common and severe enough that administrators generally disable huge pages (e.g., MongoDB, Couchbase, Redis, SAP, Splunk, etc.) despite their obvious average-case performance advantages [24, 9, 11, 30, 26, 32, 34, 37]. Other operating systems have similar or even more severe problems supporting huge pages (see §2.2 and §3.4).

Ingens[1] is a memory manager for the operating system and hypervisor that replaces the best-effort mechanisms

[1]Ingens is Latin for huge.

and spot-fixes of the past with a coordinated, unified approach to huge pages; one that is better targeted to the increased TLB capacity in modern processors. Ingens does not interfere with workloads that perform well with current huge page support: the prototype adds 0.7% overhead on average (Table 4). Ingens addresses the following problems endemic to current huge page support, and we quantify the impact of these problems on real workloads using our prototype.

- **Latency.** Huge pages expose applications to high latency variation and increased tail latency (§3.1). Ingens improves the Cloudstone benchmark [77] by 18% and reduces 90th percentile tail-latency by 41%.
- **Bloat.** Huge pages can make a process or virtual machine (VM) occupy a large amount of physical memory while much of that memory remains unusable due to internal fragmentation (§3.2). For Redis, Linux bloats memory use by 69%, while Ingens bloats by just 0.8%.
- **Unfairness.** Simple, greedy allocation of huge pages is unfair, causing large and persistent performance variation across identical processes or VMs (§3.5). Ingens makes huge page allocation fair (e.g., Figure 5).
- **High-performance memory savings.** Services that reduce memory consumption, such as kernel same-page merging (KSM), can prevent a VM from using huge pages (§3.6). On one workload (Figure 11), Linux saves 9.2% of memory but slows down the programs by 6.8–19%. Ingens saves 71.3% of the memory that Linux/KVM can save with only a 1.5–2.6% slowdown.

Ingens is a memory management redesign that brings performance, memory savings and fairness to memory-intensive applications with dynamic memory behavior. It is based on two principles: (1) memory contiguity is an explicit resource to be allocated across processes and (2) good information about spatial and temporal access patterns is essential to managing contiguity; it allows the OS to tell/predict when contiguity is/will be profitably used. The measured performance of the Ingens prototype on realistic workloads validates the approach.

2 Background

Current trends in memory management hardware are making it critical that system software support huge pages efficiently and flexibly. This section considers those trends along with the challenges huge page support creates for the OS and hypervisor. We provide an overview of huge page support in modern operating systems and conclude with experiments that show the performance benefits for the state-of-the-art in huge page management.

2.1 Virtual memory hardware trends

Virtual memory decouples the address space used by programs from that exported by physical memory (RAM). A page table maps virtual to physical page number, with recently used page table entries cached in the hardware translation lookaside buffer (TLB). Increasing the page size increases TLB *reach* (the amount of data covered by translations cached in the TLB), but larger pages require larger regions of contiguous physical memory. Large pages can suffer from internal fragmentation (unused portions within the unit of allocation) and can also increase external fragmentation (reducing the remaining supply of contiguous physical memory). Using larger pages requires more active memory management from the system software to increase available contiguity and avoid fragmentation.

Seminal work in huge page management recognized the importance of explicitly managing memory contiguity in the OS [68] and formed the basis for huge page support in FreeBSD. Innovations of Ingens relative to previous work are considered in detail in Section 3.4; here we survey recent hardware trends that make the need for system support of huge pages more urgent.

DRAM Growth. Larger DRAM sizes have led to deeper page tables, increasing the number of memory references needed to look up a virtual page number. x86 uses a 4-level page table with a worst case of four page table memory references to perform a single address translation.

Hardware memory virtualization. Extended page tables (Intel) or nested page tables (AMD) require additional indirection for each stage of memory address translation, making the process of resolving a virtual page number even more complex. With extended page tables, both the guest OS and host hypervisor perform virtual to physical translations to satisfy a single request. During translation, guest physical addresses are treated as host virtual addresses, which use hardware page-table walkers to perform the entire translation. Each layer of lookup in the guest can require a multi-level translation in the host, amplifying the maximum cost to 24 lookups [59, 40], and increasing average latencies [67].

Increased TLB reach. Recently, Intel has moved to a two-level TLB design, and in the past few years has provided a significant number of second-level TLB entries for huge pages, going from zero for Sandy Bridge and Ivy Bridge to 1,024 for Haswell [2] (2013) and 1,536 for Skylake [1] (2015).

Better hardware support for multiple page sizes creates an opportunity for the OS and the hypervisor, but it puts stress on the current memory management algorithms. In addition to managing the complexity of different page granularities, system software must generate and maintain significant memory contiguity to use larger page sizes.

Name	Suite/Application	Description
429.mcf	SPEC CPU 2006 [33]	Single-threaded scientific computation
Canneal	PARSEC 3.0 [28]	Parallel scientific computation
SVM [64]	Liblinear [22]	Machine learning, Support vector machine
Tunkrank [8]	PowerGraph [55]	Large scale in-memory graph analytics
Nutch [19]	Hadoop [4]	Web search indexing using MapReduce
MovieRecmd [25]	Spark/MLlib [5]	Machine learning, Movie recommendation
Olio	Cloudstone [8]	Social-event Web service (ngnix/php/mysql)
Redis	Redis [29]	In-memory Key-value store
MongoDB	MongoDB [23]	In-memory NoSQL database

Table 1: Summary of memory intensive workloads.

Issue	OS	Hyp
Page fault latency (§3.1)	O	
Bloat (§3.2)	O	
Fragmentation (§3.3)	O	O
Unfair allocation (§3.5)	O	O
Memory sharing (§3.6)		O

Table 2: Summary of issues in Linux as the guest OS and KVM as the host hypervisor.

2.2 Operating system support for huge pages

Early operating system support for huge pages provided a separate interface for explicit huge page allocation from a dedicated huge page pool configured by the system administrator. Windows and OS X continue to have this level of support. In Windows, applications must use an explicit memory allocation API for huge page allocation [21] and Windows recommends that applications allocate huge pages all at once when they begin. OS X applications also must set an explicit flag in the memory allocation API to use huge pages [15].

Initial huge page support in Linux used a similar separate interface for huge page allocation that a developer must invoke explicitly (called `hugetlbfs`). Developers did not like the burden of this alternate API and kernel developers wanted to bring the benefits of huge pages to legacy applications and applications with dynamic memory behavior [6, 36]. Hence, the primary way huge pages are allocated in Linux today is *transparently* by the kernel.

Transparent support is vital. Transparent huge page support [80, 68] is the only practical way to bring the benefits of huge pages to all applications, which can remain unchanged while the system provides them with the often significant performance advantages of huge pages. With transparent huge page support, the kernel allocates memory to applications using base pages. We say the kernel **promotes** a sequence of 512 properly aligned pages to a huge page (and demotes a huge page into 512 base pages).

Transparent management of huge pages best supports the multi-programmed and dynamic workloads typical of web applications and analytics where memory is contended and access patterns are often unpredictable. To the contrary, when a single big-memory application is the only important program running, the application can simply map a large region and keep it mapped for the duration of execution, for example fast network functions using Intel's Data Plane Development Kit [10]. These simple programs are well supported by even the rudimentary huge page support in Windows and OS X. However, multi-programmed workloads and workloads with more complex memory behavior are common in enterprise and cloud computing, so Ingens focuses on OS support for these more challenging cases. While transparent huge page support is far more developer-friendly than explicit allocation, it creates memory management challenges in the operating system that Ingens addresses.

Linux running on Intel processors currently has the best transparent huge page support among commodity OSes so we base our prototype on it and most of our discussion focuses on Linux. We quantify Linux's performance advantages in Section 3.4. The design of Ingens focuses on 4 KB (base) and 2 MB (huge) pages because these are most useful to applications with dynamic memory behavior (1 GB are usually too large for user data structures).

Linux is greedy and aggressive. Linux's huge page management algorithms are greedy: it promotes huge pages in the page fault handler based on local information. Linux is also aggressive: it will always try to allocate a huge page. Huge pages require 2 MB of contiguous free physical memory but sometimes contiguous physical memory is in short supply (e.g., when memory is fragmented). Linux's approach to huge page allocation works well for simple applications that allocate a large memory region and use it uniformly, but we demonstrate many applications that have more complex behavior and are penalized by Linux's greedy and aggressive promotion of huge pages (§3). Ingens recognizes that memory contiguity is a valuable resource and explicitly manages it.

2.3 Hypervisor support for huge pages

Ingens focuses on the case where Linux is used both as the guest operating system and as the host hypervisor (i.e., KVM [62]). The Linux/KVM pair is widely used in cloud deployments [27, 16, 3]. In the hypervisor, Ingens supports host huge pages mapped from guest physical memory. When promoting guest physical memory, Ingens modifies the extended page table to use huge pages because it is acting as a hypervisor, not as an operating

Workloads	h_B g_H	h_H g_B	h_H g_H
429.mcf	1.18	1.13	1.43
Canneal	1.11	1.10	1.32
SVM	1.14	1.17	1.53
Tunkrank	1.11	1.11	1.30
Nutch	1.01	1.07	1.12
MovieRecmd	1.03	1.02	1.11
Olio	1.43	1.08	1.46
Redis	1.12	1.04	1.20
MongoDB	1.08	1.22	1.37

Table 3: Application speed up for huge page (2 MB) support relative to host (h) and guest (g) using base (4 KB) pages. For example, h_B means the host uses base pages and h_H means the host uses both base and huge pages.

system.

Because operating system and hypervisor memory management are unified in Linux, Ingens adopts the unified model. Some of the problems with huge pages that we describe in Section 3 only apply to the OS and some only to the hypervisor (summarized in Table 2). For example, addressing memory sharing vs. performance (§3.6) requires only hypervisor modifications and would be as successful for a Windows guest as it is for a Linux guest. We leave for future work determining the most efficient way to implement Ingens for operating systems and hypervisors that do not share memory management code.

2.4 Performance improvement from huge pages

Table 1 describes a variety of memory-intensive real-world applications including web infrastructure such as key/value stores and databases, as well as scientific applications, data analytics and recommendation systems. Measurements with hardware performance counters show they all spend a significant portion of their execution time doing page walks. For example, when using base pages for both guest and host, we measure 429.mcf spending 47.5% of its execution time doing page walks (24.2% for the extended page table and 23.3% for the guest page table). On the other hand, 429.mcf spends only 4.2% of its execution time walking page tables when using huge pages for both the guest and host.

We execute all workloads in a KVM virtual machine running Linux with default transparent huge page support [80] for both the application (in the guest OS) and the virtual machine (in the host OS). The hardware configuration is detailed in Section 6.

Table 3 shows the performance improvements gained with transparent huge page support for both the guest and the host operating system. The table shows speedup normalized to the case where both host and guest use only base pages. In every case, huge page support helps performance, often significantly (up to 53%). The largest speedup is always attained when both host and guest use huge pages.

These results show the value of huge page support and show that Linux's memory manager can obtain that benefit under simple operating conditions. However, a variety of more challenging circumstances expose the limitations of Linux's memory management.

3 Current huge page problems

This section quantifies the limitations in performance and fairness for the state-of-the-art in transparent huge page management. We examine virtualized systems with Linux/KVM as the guest OS and hypervisor. The variety and severity of the limitations motivate our redesign of page management. All data is collected using the experimental setup described in Section 2.4.

3.1 Page fault latency and synchronous promotion

When a process faults on an anonymous memory region, the page fault handler allocates physical memory to back the page. Both base and huge pages share this code path. Linux is greedy and aggressive in its allocation of huge pages, so if an application faults on a base page, Linux will immediately try to upgrade the request and allocate a huge page if it can.

This greedy approach fundamentally increases page fault latency for two reasons. First, Linux must zero pages before returning them to the user. Huge pages are 512× larger than base pages, and thus are much slower to clear. Second, huge page allocation requires 2 MB of physically contiguous memory. When memory is fragmented, the OS often must compact memory to generate that much contiguity. Previous work shows that memory quickly fragments in multi-tenant cloud environments [41]. When memory is fragmented, Linux will often synchronously compact memory in the page fault handler, increasing average and tail latency.

To measure these effects, we compare page fault latency when huge pages are enabled and disabled, in fragmented and non-fragmented settings. We quantify fragmentation using the *free memory fragmentation index* (FMFI) [58], a value between 0 (unfragmented) and 1 (highly fragmented). A microbenchmark maps 10 GB of anonymous virtual memory and reads it sequentially.

When memory is unfragmented (FMFI < 0.1), page clearing overheads increase average page fault latency from 3.6 μs for base pages only to 378 μs for huge pages (105× slower). When memory is heavily fragmented, (FMFI $= 0.9$), the 3.6 μs average latency for base pages grows to 8.1 μs (2.1× slower) for base and huge pages. Average latency is lower in the fragmented case because 98% of the allocations fall back to base pages (e.g. because memory is too fragmented to allocate a huge page).

SVM	Synchronous	Asynchronous
Exec. time (sec)	178 (1.30×)	228 (1.02×)
Huge page	4.8 GB	468 MB
Promotion speed	immediate	1.6 MB/s

Table 4: Comparison of synchronous promotion and asynchronous promotion when both host and guest use huge pages. The parenthesis is speedup compared to not using huge pages. We use the default asynchronous promotion speed of Ubuntu 14.04.

Workload	Using huge pages	Not using huge pages
Redis	20.7 GB (1.69×)	12.2 GB
MongoDB	12.4 GB (1.23×)	10.1 GB

Table 5: Physical memory size of Redis and MongoDB.

Compacting and zeroing memory in the page fault handler penalizes applications that are sensitive to average latency and to tail latency, such as Web services.

To avoid this additional page fault latency, Linux can promote huge pages asynchronously, based on a configurable asynchronous promotion speed (in MB/s). Table 4 shows performance measurements for asynchronous-only huge page promotion when executing SVM in a virtual machine. Asynchronous-only promotion turns a 30% speedup into a 2% speedup: it does not promote fast enough. Simply increasing the promotion speed does not solve the problem. Earlier implementations of Linux did more aggressive asynchronous promotion, incurring unacceptably high CPU utilization for memory scanning and compaction. The CPU use of aggressive promotion reduced or in some cases erased the performance benefits of huge pages, causing users to disable transparent huge page support in practice [17, 14, 13, 7].

3.2 Increased memory footprint (bloat)

Huge pages improve performance, but applications do not always fully utilize the huge pages allocated to them. Linux greedily allocates huge pages even though underutilized huge pages create internal fragmentation. A huge page might eliminate TLB misses, but the cost is that a process using less than a full huge page has to reserve the entire region.

Table 5 shows memory bloat from huge pages when running Redis and MongoDB, each within their own virtual machine. For Redis, we populate 2 million keys with 8 KB objects and then delete 70% of the keys randomly. Redis frees the memory backing the deleted objects which leaves physical memory sparsely allocated. Linux promotes the sparsely allocated memory to huge pages, creating internal fragmentation and causing Redis to use 69% more memory compared to not using huge pages. We demonstrate the same problem in MongoDB, making 10 million `get` requests for 15 million 1 KB objects which are initially in persistent storage. MongoDB allocates the objects sparsely in a large virtual address space. Linux promotes huge pages including unused memory, and as a result, MongoDB uses 23% more memory relative to running without huge page support.

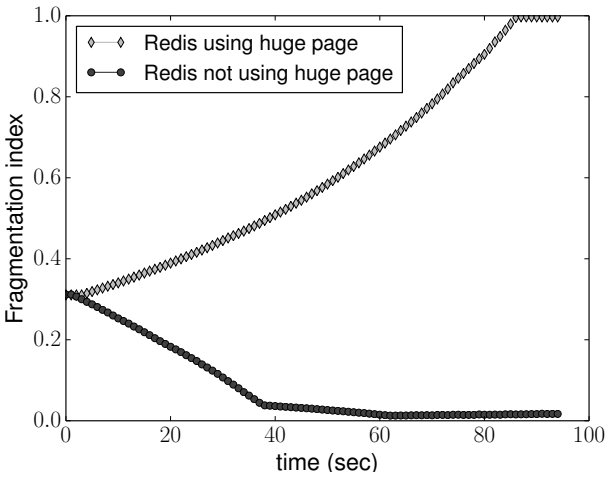


Figure 1: Fragmentation index in Linux when running a Redis server, with Linux using (and not using) huge pages. The System has 24 GB memory. Redis uses 13 GB, other processes use 5 GB, and system has 6 GB free memory.

Greedy and aggressive allocation of huge pages makes it impossible to predict an application's total memory usage in production because memory usage depends on huge page use, which in turn depends on memory fragmentation and the allocation pattern of applications. Table 5 shows if an administrator provisions 18 GB memory (1.5× over-provisioning relative to using only base pages), Redis starts swapping when it uses huge pages, negating the benefits of caching objects in memory [31].

While these experiments illustrate the potential impact of bloat for a handful of workloads, it is important to note that the problem is fundamental to Linux's current design. Memory bloating can happen in any working set, memory, and TLB size: application-level memory usage can conspire with aggressive promotion to create internal fragmentation that the OS cannot address. In such situations, such applications will eventually put the system under memory pressure regardless of physical memory size.

3.3 Huge pages increase fragmentation

One common theme in analyzing page fault latency (§3.1) and memory bloat (§3.2) is Linux's greedy allocation and promotion of huge pages. We now measure how aggressive promotion of huge pages quickly consumes available physical memory contiguity, which then increases memory fragmentation for the remaining physical memory.

OS	SVM	Canneal	Redis
FreeBSD	1.28	1.13	1.02
Linux	1.30	1.21	1.15

Table 6: Performance speedup when using huge page in different operating systems.

Increasing fragmentation is the precondition for problems with page fault latency and memory bloat, so greedy promotion creates a vicious cycle. We again rely on the free memory fragmentation index, or FMFI to quantify the relationship between huge page allocation and fragmentation.

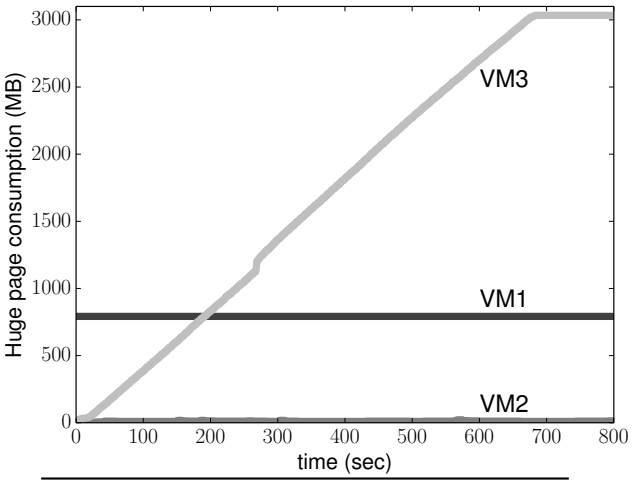
Figure 1 shows the fragmentation index over time when running the popular key-value store application Redis in a virtual machine. Initially, the system is lightly fragmented (FMFI = 0.3) by other processes. Through the measurement period, Redis clients populate the server with 13 GB of key/value pairs. Redis rapidly consumes contiguous memory as Linux allocates huge pages to it, increasing the fragmentation index. When the FMFI is equal to 1, the remaining physical memory is so fragmented, Linux starts memory compaction to allocate huge pages.

3.4 Comparison with FreeBSD huge page support

FreeBSD supports transparent huge pages using reservation-based huge page allocation [68]. When applications start accessing a 2 MB virtual address region, the page fault handler reserves contiguous memory, but does not promote the region to a huge page. It allocates base pages from the reserved memory for subsequent page faults in the region. FreeBSD monitors page utilization of the region and promotes it to a huge page only when all base pages of the reserved memory are allocated. FreeBSD is therefore slower to promote huge pages than Linux and promotion requires complete utilization of a 2 MB region.

FreeBSD supports huge pages for file-cached pages. x86 hardware maintains access/dirty bits for entire huge pages—any read or write will set the huge page's access/dirty bit. FreeBSD wants to avoid increasing IO traffic when evicting from the page cache or swapping. Therefore it is conservative about creating writable huge pages. When FreeBSD promotes a huge page, it marks it read-only, with writes demoting the huge page. Only when all pages in the region are modified will FreeBSD then promote the region to a writable huge page. The read-only promotion design does not increase IO traffic from the page cache because huge pages consist of either all clean (read-only) or all modified base pages.

FreeBSD promotion of huge pages is more conservative than in Linux, which reduces memory bloating, but yields slower performance. Table 6 compares the performance benefits of huge pages in FreeBSD and Linux. Applications with dense, uniform access memory patterns (e.g., SVM) enjoy similar speedups on Linux and FreeBSD. However, FreeBSD does not support asynchronous promotion, so applications which allocate memory gradually (e.g., Canneal) show less benefit. Redis makes frequent hash table updates and exhibits many read-only huge page demotions in FreeBSD. Consequently, Redis also shows limited speedup compared with Linux.



SVM	VM1	VM2	VM3
Exec. time (sec)	533 (1.12×)	589 (1.24×)	475

Figure 2: Unfair allocation of huge pages in KVM. Three virtual machines run concurrently, each executing SVM. The line graph is huge page size (MB) over time and the table shows execution time of SVM for 2 iterations.

3.5 Unfair performance

All of our measurements are on virtual machines where Linux is the guest operating system, and KVM (Linux's in-kernel hypervisor) is the host hypervisor. Ingens modifies the memory management code of both Linux and KVM. The previous sections focused on problems with operating system memory management, the remaining sections describe problems with KVM memory management.

Unfair huge page allocation can lead to unfair performance differences when huge pages become scarce. Linux does not fairly redistribute contiguity, which can lead to unfair performance imbalance. To demonstrate this problem, we run 4 virtual machines in a setting where memory is initially fragmented (FMFI = 0.85). Each VM uses 8 GB of memory. VM0 starts first and obtains all huge pages that are available (3 GB). Later, VM1 starts and begins allocating memory, during which VM2 and VM3 start. VM0 then terminates, releasing its 3 GB of huge pages. We measure how Linux redistributes that contiguity to the remaining identical VMs.

The graph in Figure 2 shows the amount of huge page

Policy	Mem saving	Performance slowdown	H/M
No sharing	–	429.mcf: 278 SVM: 191 Tunkrank: 236	429.mcf: 99% SVM: 99% Tunkrank: 99%
KVM (Linux)	1.19 GB (9.2%)	429.mcf: 331 (19.0%) SVM: 204 (6.8%) Tunkrank: 268 (13.5%)	429.mcf: 66% SVM: 90% Tunkrank: 69%
Huge page sharing	199 MB (1.5%)	429.mcf: 278 (0.0%) SVM: 194 (1.5%) Tunkrank: 238 (0.8%)	429.mcf: 99% SVM: 99% Tunkrank: 99%

Table 7: Memory saving and performance trade off for a multi-process workload. Each row is an experiment where all workloads run concurrently in separate virtual machines. H/M - huge page ratio out of total memory used. Parentheses in the Mem saving column expresses the memory saved as a percentage of the total memory (13 GB) allocated to all three virtual machines.

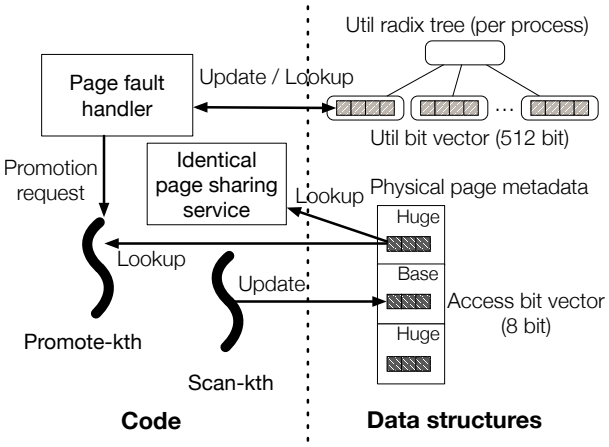


Figure 3: Important code and data structures in the Ingens memory manager.

memory allocated to VM1, VM2, and VM3 (all running SVM) over time, starting 10 seconds before the termination of VM0. When VM1 allocates memory, Linux compacts memory for huge page allocation, but compaction begins to fail at 810 MB. VM2 and VM3 start without huge pages. When VM0 terminates 10 seconds into the experiment, Linux allocates all 3 GB of recently freed huge pages to VM3 through asynchronous promotion. This creates significant and persistent performance inequality among the VMs. The table in Figure 2 shows the variation in performance (NB: to avoid IO measurement noise, data loading time is excluded from the measurement). In a cloud provider scenario, with purchased VM instances of the same type, users have good reason to expect similar performance from identical virtual machine instances, but VM2 is 24% slower than VM3.

3.6 Memory sharing vs. performance

Modern hypervisors detect and share memory pages from different virtual machines whose contents are identical [81, 63]. The ability to share identical memory reduces the memory consumed by guest VMs, increasing VM consolidation ratios. In KVM, identical page sharing in the host is done transparently in units of base pages. If the contents of a base page are duplicated in a different VM, but the duplicated base page is contained within a huge page, KVM will split the huge page into base pages to enable sharing. This policy prioritizes reducing memory footprint over preservation of huge pages, so it penalizes performance.

Another possible policy, which we call *huge page sharing*, would not split huge pages. A base page is not allowed to share pages belonging to a huge page to prevent the demotion of the huge page but it can share base pages. In contrast, a huge page is only allowed to share huge pages. We implement huge page sharing to compare with KVM and the result is shown in Table 7. We fit the virtual machine memory size to the working set size of each workload to avoid spurious sharing of zeroed pages. KVM saves 9.2% of memory but the workloads show a slowdown of up to 19.0% because TLB misses are increased by splitting huge pages (the percentage of huge pages in use (H/M) goes down to 66%). On the other hand, while huge page sharing preserves good performance, it provides only reduced memory consumption by 1.5%. This tradeoff between performance and memory savings is avoidable. Identical page sharing services can and should be coordinated with huge page management to obtain both performance and memory saving benefits.

4 Design

Ingens's goal is to enable transparent huge page support that reduces latency, latency variability and bloat while providing meaningful fairness guarantees and reasonable tradeoffs between high performance and memory savings. Ingens builds on a handful of basic primitives to achieve these goals: utilization tracking, access frequency tracking, and contiguity monitoring.

While the discussion in this section is mostly expressed in terms of process behavior, Ingens techniques apply equally to processes and to virtual machines. Figure 3 shows the major data structures and code paths of Ingens, which we describe in this section.

4.1 Monitoring space and time

Ingens unifies and coordinates huge page management by introducing two efficient mechanisms to measure the utilization of huge-page sized regions (space) and how frequently huge-page sized regions are accessed (time). Ingens collects this information efficiently and then leverages it throughout the kernel to make policy decisions, using two bitvectors. We describe both.

Util bitvector. The util bitvector records which base pages are used within each huge-page sized memory region (an aligned 2 MB region containing 512 base pages). Each bit set in the util bitvector indicates that the corresponding base page is in use. The bitvector is stored in a radix tree and Ingens uses a huge-page number as the key to lookup a bitvector. The page fault handler updates the util bitvector.

Access bitvector. The access bitvector records the recent access history of a process to its pages (base or huge). Scan-kth periodically scans a process' hardware access bits in its page table to maintain per-page (base or huge) access frequency information, stored as an 8-bit vector within Linux' page metadata. Ingens computes the exponential moving average (EMA) [12] from the bitvector which we define as follows:

$$F_t = \alpha(\textit{weight(util bitvector)}) + (1-\alpha)F_{t-1} \quad (1)$$

The *weight* is the sum of set bits in the bitvector, F_t is the access frequency value at time t, and α is a parameter. Based on a sensitivity analysis using our workloads, we set α to 0.4, meaning Ingens considers the page "frequently accessed" when $F_t \geq 3 \times \textit{bitvector size}/4$ (i.e., 6 in our case).

We can experimentally verify the accuracy of the frequency information by checking whether pages classified as frequently accessed have their access bit set in the next scan interval: in most workloads we find the misprediction ratio to be under 3%, although random access patterns (e.g. Redis, MongoDB) can yield higher error rates depending on the dynamic request pattern.

4.2 Fast page faults

To keep the page fault handling path fast, Ingens decouples promotion decisions (policy) from huge page allocation (mechanism). The page fault handler decides when to promote a huge page and signals a background thread (called `Promote-kth`) to do the promotion (and allocation if necessary) asynchronously (Figure 3). Promote-kth compacts memory if necessary and promotes the pages identified by the page fault handler. The Ingens page fault handler never does a high-latency huge page allocation. When Promote-kth starts executing, it has a list of viable candidates for promotion; after promoting them, it resumes its scan of virtual memory to find additional candidates.

4.3 Utilization-based promotion (mitigate bloat)

Ingens explicitly and conservatively manages memory contiguity as a resource, allocating contiguous memory only when it decides a process (or VM) will use most of the allocated region based on utilization. Ingens allocates only base pages in the page fault handler and tracks base page allocations in the util bitvector. If a huge page region accumulates enough allocated base pages (90% in our prototype), the page fault handler wakes up Promote-kth to promote the base pages to a huge page.

Utilization tracking lets Ingens mitigate memory bloating. Because Ingens allocates contiguous resources only for highly utilized virtual address regions, it can control internal fragmentation. The utilization threshold provides an upper bound on memory bloat. For example, if an administrator sets the threshold to 90%, processes can use only 10% more memory in the worst case compared to a system using base pages only. The administrator can simply provision 10% additional memory to avoid unexpected swapping.

Utilization-based demotion (performance). Processes can free a base page, usually by calling `free`. If a freed base page is contained within a huge page, Linux demotes the huge page instantly. For example, Redis frees objects when deleting keys which results in a system call to free the memory. Redis uses jemalloc [20], whose `free` implementation makes an `madvise` system call with the `MADV_DONTNEED` flag to release the memory[2]. Linux demotes the huge page that contains the freed base page[3].

Demoting in-use huge pages hurts performance. Consequently, Ingens defers the demotion of high utilization huge pages. When a base page is freed within a huge page, Ingens clears the bit for the page in the util bitvector. When utilization drops below a threshold, Ingens demotes the huge page and frees the base pages whose bits are clear in the util bitvector.

4.4 Proactive batched compaction (reduce fragmentation)

Maintaining available free contiguous memory is important to satisfy large size allocation requests required when Ingens decides to promote a region to a huge page, or to satisfy other system-level contiguity in service of, for example, device drivers or user-level DMA. To this end, Ingens monitors the fragmentation state of physical memory and proactively compacts memory to reduce the latency of large contiguous allocations.

Ingens's goal is to control memory fragmentation by keeping FMFI below a threshold (that defaults to 0.8). Proactive compaction happens in Promote-kth after performing periodic scanning. Aggressive proactive compaction causes high CPU utilization, interfering with user applications. Ingens limits the maximum amount of compacted memory to 100 MB for each compaction. Compaction moves pages, which necessitates TLB invalidations. Ingens does not move frequently accessed pages to

[2] TCMalloc [35] also functions this way.

[3] Kernel version 4.5 introduces a new mechanism to free memory efficiently, called `MADV_FREE` but it also demotes huge pages instantly and causes the same memory bloating problem as `MADV_DONTNEED`.

reduce the performance impact of compaction.

4.5 Balance page sharing with performance

Ingens uses access frequency information to balance identical page sharing with application performance. It decides whether or not huge pages should be demoted to enable sharing of identical base pages contained within the huge page. In contrast to KVM, which always prioritizes memory savings over contiguity, Ingens implements a policy that avoids demoting frequently accessed huge pages. When encountering a matching identical base-page sized region within a huge page, Ingens denies sharing if that huge page is frequently accessed, otherwise it allows the huge page to be demoted for sharing.

For page sharing, the kernel marks a shared page read-only. When a process writes the page, the kernel stops sharing the page and allocates a new page to the process (similar to a copy-on-write mechanism). Ingens checks the utilization for the huge page region enclosing the new page and if it is highly utilized, it promotes the page (while Linux would wait for asychronous promotion).

4.6 Proportional promotion manages contiguity

Ingens monitors and distributes memory contiguity fairly among processes and VMs, employing techniques for proportional fair sharing of memory with an idleness penalty [81]. Each process has a share priority for memory that begins at an arbitrary but standard value (e.g, 10,000). Ingens allocates huge pages in proportion to the share value. Ingens counts infrequently accessed pages as idle memory and imposes a penalty for the idle memory. An application that has received many huge pages but is not using them actively does not get more.

We adapt ESX's adjusted shares-per-page ratio [81] to express our per-process memory promotion metric mathematically as follows.

$$\mathscr{M} = \frac{S}{H \cdot (f + \tau(1-f))} \tag{2}$$

where S is a process' (or virtual machine's or container's) huge page share priority and H is the number of bytes backed by huge pages allocated to the process. $(f + \tau(1-f))$ is a penalty factor for idle huge pages. f is the fraction of idle huge pages relative to the total number of huge pages used by this process ($0 \leq f \leq 1$) and τ, with $0 < \tau \leq 1$, is a parameter to control the idleness penalty. Larger values of $\mathscr{M}$ receive higher priority for huge page promotion.

Intuitively, if two processes' S value are similar and one process has fewer huge pages (H is smaller), then the kernel prioritizes promotion (or allocation and promotion) of huge pages for that process. If S and H values are similar among a group of processes, the process with the largest fraction of idle pages has the smaller $\mathscr{M}$, and hence the lowest priority for obtaining new huge pages. $\tau = 1$ means $\mathscr{M}$ disregards idle memory while τ close to 0 means $\mathscr{M}$'s value is inversely proportional to the amount of idle memory.

A kernel thread (called `Scan-kth`) periodically profiles the idle fraction of huge pages in each process and updates the value of $\mathscr{M}$ for fair promotion.

4.7 Fair promotion

Promote-kth performs fair allocation of contiguity using the promotion metric. When contiguity is contended, fairness is achieved when all processes have a priority-proportional share of the available contiguity. Mathematically this is achieved by minimizing $\mathscr{O}$, defined as follows:

$$\mathscr{O} = \sum_i (\mathscr{M}_i - \bar{\mathscr{M}})^2 \tag{3}$$

The $\mathscr{M}_i$ indicates the promotion metric of process/VM i and $\bar{\mathscr{M}}$ is the mean of all process' promotion metrics. Intuitively, the formula characterizes how much process' contiguity allocation ($\mathscr{M}_i$) deviates from a fair state ($\bar{\mathscr{M}}$): in a perfectly fair state, all the $\mathscr{M}_i$ equal $\bar{\mathscr{M}}$, yielding a 0-valued $\mathscr{O}$.

In practice, to optimize $\mathscr{O}$, it suffices to iteratively select the process with the biggest $\mathscr{M}_i$, scan its address space to promote huge pages, and update $\mathscr{M}_i$ and $\mathscr{O}$. Iteration stops when $\mathscr{O}$ is close to 0 or when Promote-kth cannot generate any additional huge pages (e.g., all process are completely backed by huge pages).

An important benefit of this approach is that it does not require a performance model and it applies equally well to processes and virtual machines.

5 Implementation

Ingens is implemented in Linux 4.3.0 and contains new mechanisms to support page utilization and access frequency tracking. It also uses Linux infrastructure for huge page page table mappings and memory compaction.

5.1 Huge page promotion

Promote-kth runs as a background kernel thread and schedules huge page promotions (replacing Linux's `khugepaged`). Promote-kth maintains two priority lists: `high` and `normal`. The high priority list is a global list containing promotion requests from the page fault handler and the normal priority list is a per-application list filled in as Promote-kth periodically scans the address space. The page fault handler or a periodic timer wakes Promote-kth, which then examines the two lists and promotes in priority order.

Ingens does not reserve contiguous memory in the page fault handler. When the page fault handler requests a huge page promotion, the physical memory backing the base pages might not be contiguous. In this case, Promote-kth allocates a new 2 MB contiguous physical memory region,

copies the data from the discontiguous physical memory, and maps the contiguous physical memory into the process' virtual address space. After promotion, Promote-kth frees the original discontiguous physical memory.

An application's virtual address space can grow, shrink, or be merged with other virtual address regions. These changes make new opportunities for huge page promotion which both Linux and Ingens detect by periodically scanning address spaces in the normal priority list (Linux in `khugepaged`, Ingens in Promote-kth). For example, a virtual address region that is smaller than the size of a huge page might merge with another region, allowing it to be part of a huge page.

Promote-kth compares the promotion metric (§4.6) of each application and selects the process with the highest deviation from a fair state (§4.7). It scans 16 MB of pages and sleeps for 10 seconds which is also Linux's default settings (i.e., the 1.6 MB/s in Table 4). After scanning a process' entire address space, Promote-kth records the number of promoted huge pages and if an application has too few promotions (zero in the prototype), Promote-kth excludes the application from the normal priority list for 120 seconds. This mechanism prevents an adversarial application that can monopolize Promote-kth. Such an application would have a small number of huge pages and would appear to be a good candidate to scan to increase fairness (§4.7)).

5.2 Access frequency tracking

In 2015, Linux added an access bit tracking framework [70] for version 4.3. The kernel adds an idle flag for each physical page and uses hardware access bits to track when a page remains unused. If the hardware sets an access bit, the kernel clears the idle bit. The framework provides APIs to query the idle flags and clear the access bit. Scan-kth uses this framework to find idle memory during a periodic scan of application memory. The default period is 2 seconds. Scan-kth clears the access bits at the beginning of the profiling period and queries the idle flag at the end.

In the x86 architecture, clearing the access bit causes a TLB invalidation for the corresponding page. Consequently, frequent periodic scanning can have a negative performance impact. To ameliorate this problem, Ingens supports frequency-aware profiling and sampling. When Scan-kth needs to clear the access bit of a page, it checks whether the page is frequently accessed or not. If it is not frequently accessed, Scan-kth clears the access bit, otherwise it clears it with 20% probability. Ingens uses an efficient hardware-based random number generator [18].

To verify that sampling reduces worst case overheads, we run a synthetic benchmark which reads 10 GB memory randomly without any computation, and measure the execution time for one million iterations. When Ingens resets all access bits, the execution time of the workload is degraded by 29%. Sampling-based scanning reduces the overhead to 8%. In contrast to this worst-case microbenchmark, Section 6 shows that slowdowns of Ingens on real workloads average 1%.

5.3 Limitations and future work

Linux supports transparent huge pages only for anonymous memory because huge page support for page cache pages can significantly increase I/O traffic, potentially offsetting the benefits of huge pages. If Linux adds huge pages to the page cache, it will make sense to extend Ingens to manage them with the goal of improving the read-only page cache support (implemented in FreeBSD [68]), while avoiding significant increases in I/O traffic for write-back of huge pages which are sparsely modified.

Hardware support for finer-grain tracking of access and dirty bits for huge pages would benefit Ingens. Hardware-managed access and dirty bits for all base pages within a huge page region could avoid wasted I/O on write-back of dirty pages, and enable much better informed decisions about when to demote a huge page or when huge pages can be reclaimed fairly under memory pressure.

NUMA considerations. Ingens maintains Linux's NUMA heuristics, preferring pages from a node's local NUMA region, and refusing to allocate a huge page from a different NUMA domain. All of our measurements are within a single NUMA region.

Previous work has shown that if memory is shared across NUMA nodes, huge pages may contribute to memory request imbalance across different memory controllers and reduced locality of accesses, decreasing their performance benefit [54]. This happens due to page-level false sharing, where unrelated data is accessed on the same page, and the hot page effect, which is exacerbated by the large page size. The authors propose extensions to Linux' huge page allocation mechanism to balance huge pages among NUMA domains and to split huge pages if false sharing is detected or if they become too hot. These extensions integrate nicely with Ingens. Scan-kth can already measure page access frequencies and Promote-kth can check whether huge pages need to be demoted.

6 Evaluation

We evaluate Ingens using the applications in Table 1, comparing against the performance of Linux's huge page support which is state-of-the-art. Experiments are performed on two Intel Xeon E5-2640 v3 2.60GHz CPUs (Haswell) with 64 GB memory and two 256 MB SSDs. We use Linux 4.3 and Ubuntu 14.04 for both the guest and host system. Intel supports multiple hardware page sizes of 4 KB, 2 MB and 1 GB; our experiments use only 4 KB and 2 MB huge pages. We set the number of vCPUs equal to the number of application threads.

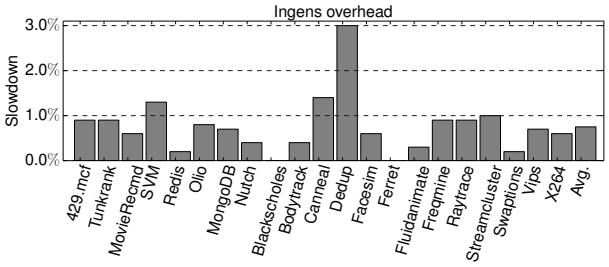


Figure 4: Performance slowdown of utilization-based promotion relative to Linux when memory is not fragmented.

Background task	CPU utilization
Proactive compaction	1.3%
Access bit tracking	11.4%

Table 8: CPU utilization of background tasks in Ingens. For access bit tracking, Scan-kth scans memory of MongoDB that uses 10.7GB memory.

We characterize the overheads of Ingens's basic mechanisms such as access tracking and utilization-based huge page promotion. We evaluate the performance of utilization-based promotion and demotion and Ingens ability to provide fairness across applications using huge pages. Finally, we show that Ingens's access frequency-based same page merging achieves good memory savings while preserving most of the performance benefit of huge pages. We use a single configuration to evaluate Ingens which is consistent with our examples in Sections 4 and 5: utilization threshold is 90%, Scan-kth period is 10s, access frequency tracking interval is 2 sec, and sampling ratio is 20%. Proactive batched compaction happens when FMFI is below 0.8, with an interval of 5 seconds; the maximum amount of compacted memory is 100MB; and a page is frequently accessed if $F_t \geq 6$.

6.1 Ingens overhead

Figure 4 shows the overheads introduced by Ingens for memory intensive workloads. To evaluate the performance of utilization-based huge page promotion in the unfragmented case, we run a number of benchmarks and compare their run time with Linux. Ingens's utilization-based huge page promotion slows applications down 3.0% in the worst case and 0.7% on average. The slowdowns stem primarily from Ingens not promoting huge pages as aggressively as Linux, so the workload executes with slower base pages for a short time until Ingens promotes huge pages. A secondary overhead stems from the computation of huge page utilization.

To verify that Ingens does not interfere with the performance of "normal" workloads, we measure an average performance penalty of 0.8% across the entire PARSEC 3.0 benchmark suite.

Linux	Ingens
922.3	1091.9 (1.18×)

(a) Throughput of full operation mix (requests/sec and speedup normalized to Linux).

	Event view		Homepage visit		Tag search	
	Linux	Ingens	Linux	Ingens	Linux	Ingens
Average	478	338	236	207	289	240
90th	605	354	372	226	417	299
MAX	694	649	379	385	518	507

(b) Latency (millisecond) of read-dominant operations.

Table 9: Performance result of Cloudstone WEB 2.0 Benchmark (Olio) when memory is fragmented.

Table 8 shows the CPU utilization of background tasks in Ingens. We measure the CPU utilization across 1 second intervals and take the average. For proactive compaction, we set Ingens to compact 100 MB of memory every 2 seconds (which is more aggressive than the default of 5 seconds). CPU overhead of access bit tracking depends on how many pages are scanned, so we measure the CPU utilization of Scan-kth while running MongoDB using 10.7 GB of memory.

6.2 Utilization-based promotion

To evaluate Ingens's utilization-based huge page promotion, we compare a mix of operations from the Cloudstone WEB 2.0 benchmark, which simulates a social event website. Cloudstone models a LAMP stack, consisting of a web server (nginx), PHP, and MySQL. We run Cloudstone in a KVM virtual machine and use the Rain workload generator [45] for load.

A study of the top million websites showed that in 2015 the average size exceeded 2 MB [50]. In light of this, we modify Cloudstone to serve some web pages that use about 2 MB of memory, enabling the benchmark to make better use of huge pages. The Cloudstone benchmark consists of 7 web pages, and we only modify the homepage and a page that displays social event details to use 2 MB memory. The other pages remain unchanged.

We compare throughput and latency for Cloudstone on Linux and Ingens when memory is fragmented from prior activity (FMFI = 0.9). To cause fragmentation, we run a program that allocates a large region of memory and then partially frees it.

We use Cloudstone's default operation mix: 85% read (viewing events, visiting homepage, and searching event by tag), 10% login, and 5% write (adding new events and inviting people). Our test database has 7,000 events, 2,000 people, and 900 tags. Table 9 (a) shows the throughput attained by the benchmark running on Linux and Ingens. Ingens's utilization-based promotion achieves a speedup of 1.18× over Linux. Table 9 (b) shows average and tail

Linux-nohuge	Linux	Ingens-90%	Ingens-70%	Ingens-50%
12.2 GB	20.7 GB	12.3 GB	12.9 GB	17.8 GB

(a) Redis memory consumption in different configurations. The percentage in the label is a utilization threshold.

	Throughput	90th lat.	99th lat.	99.9th lat.
Linux-nohuge	19.0K	4	5	109
Linux	21.7K	3	4	8
Ingens-90%	20.9K	3	4	64
Ingens-70%	21.1K	3	4	55
Ingens-50%	21.6K	3	4	23

(b) Redis GET Performance: Throughput (operations/sec) and latency (millisecond).

Table 10: Redis memory use and performance.

latency of the read operations in the benchmark. Ingens reduces an average latency up to 29.2% over Linux. In the tail, the reduction improves further, up to 41.4% at the 90th percentile.

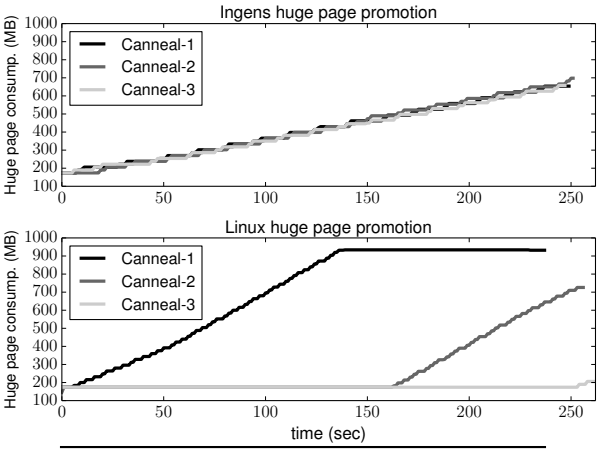
Performance for Ingens improves because it reduces the average page-fault latency by not compacting memory synchronously in the page fault handler. We measure 461,383 page compactions throughout the run time of the benchmark in Linux when memory is fragmented.

When memory is not fragmented, Ingens reduces throughput by 13.4% and increases latency up to 18.1% compared with Linux. The benchmark contains many short-lived requests and Linux's greedy huge page allocation pays off by drastically reducing the total number of page faults. Ingens is less aggressive about huge page allocation to avoid memory bloat, so it incurs many more page faults.

Ingens copes with this performance problem with an adaptive policy. When memory fragmentation is below 0.5 Ingens mimics Linux's aggressive huge page allocation. This policy restores Ingens's performance to Linux's levels. However, while bloat (§3.2) is not a problem for this workload, the adaptive policy increases risk of bloat in the general case. Like any management problem, it might not be possible to find a single policy that has every desirable property for a given workload. We verified that this policy performs similarly to the default policy used in Table 4, but it is most appropriate for workloads with many short-lived processes.

6.3 Memory bloating evalution

To evaluate Ingens's ability to minimize memory bloating without impacting performance, we evaluate the memory use and throughput of a benchmark using the Redis key-value store. Redis is known to be susceptible to memory bloat, as its memory allocations are often sparse. To create a sparse address space in our benchmark, we first populate Redis with 2 million keys, each with 8 KB objects and



	Canneal-1	Canneal-2	Canneal-3
Linux	181	192	195
Ingens	186	186	187

Figure 5: Huge page consumption (MB) and execution time (second). 3 instances of canneal (Parsec 3.0 benchmark) run concurrently and Promote-kth promotes huge pages. Execution time in the table excludes data loading time.

then delete 70% of the key space using a random pattern. We then measure the GET performance using the benchmark tool shipped with Redis. For Ingens, we evaluate different utilization thresholds for huge page promotion.

Table 10 shows that memory use for the 90% and 70% utilization-based configurations is very close to the case where only base pages are used. Only at 50% utilization does Ingens approach the memory use of Linux's aggressive huge page promotion.

The throughput and latency of the utilization-based approach is very close to using only huge pages. Only in the 99.9th percentile does Ingens deviate from Linux using huge pages only, while still delivering much better tail latency than Linux using base pages only.

6.4 Fair huge page promotion

Ingens guarantees a fair distribution of huge pages. If applications have the same share priority (§4.6), Ingens provides the same amount of huge pages. To evaluate fairness, we run a set of three identical applications concurrently with the same share priority and idleness parameter, and measure the amount of huge pages each one holds at any point in time.

Figure 5 shows that Linux does not allocate huge pages fairly, it simply allocates huge pages to the first application that can use them (Canneal-1). In fact, Linux asynchronously promotes huge pages by scanning linearly through each application's address space, only considering the next application when it is finished with the current application. Time 160 is when Linux has pro-

Policy	Mem saving	Performance slowdown	H/M
KVM (Linux)	1438 MB (9.6%)	Tunkrank: 274 (12.7%) MovieRecmd: 210 (6.5%) SVM: 232 (20.2%)	Tunkrank: 66% MovieRecmd: 10% SVM: 72%
Huge page sharing	317 MB (2.1%)	Tunkrank: 243 MovieRecmd: 197 SVM: 193	Tunkrank: 99% MovieRecmd: 99% SVM: 99%
Ingens	1026 MB (6.8%)	Tunkrank: 247 (1.6%) MovieRecmd: 200 (1.5%) SVM: 198 (2.5%)	Tunkrank: 90% MovieRecmd: 79% SVM: 94%

Table 11: Memory saving (MB) and performance (second) trade off. H/M - huge page ratio out of total memory used. Parentheses in the Mem saving column expresses the memory saved as a percentage of the total memory (15 GB) allocated to all three virtual machines.

moted almost all of Canneal-1's address space to huge pages so only then does it begin to allocate huge pages to Canneal-2.

In contrast, Ingens promotes huge pages based on the fairness objective described in Section 4.7 and thus equally distributes the available huge pages to each application. Fair distribution of huge pages translates to fair end-to-end execution time as well. All applications finish at the same time in Ingens, while Canneal-1 finishes well before 2 and 3 on Linux.

6.5 Trade off of memory saving and performance

Finally, we evaluate the memory and performance trade-offs of identical page sharing. We run a workload mix of three different applications, each in its own virtual machine. We measure their memory use and performance slowdown under three different OS configurations: (1) KVM with aggressive page sharing, where huge pages are demoted if underlying base pages can be shared. (2) KVM where only pages of the same type may be shared and huge pages are never broken up (huge page sharing). (3) Ingens, where only infrequently used huge pages are demoted for page sharing. To avoid unused memory saving, we intentionally fit guest physical memory size to memory usages of the workloads.

Table 11 shows that KVM's aggressive page sharing saves the most memory (9.6%), but also cedes the most performance (between 6.5% and 20.2% slowdown) when compared to huge page sharing. When sharing only pages of the same type, it saves memory only 2.1%. Finally, Ingens allows us to save 6.8% of memory, while only slowing down the application up to 2.5%. The main reason for the low performance degradation is that the ratio of huge pages to total pages remains high in Ingens, due to its access frequency-based approach to huge page demotion and instant promotion when Ingens stops page sharing.

7 Related work

Virtual memory is an active research area. Our evidence of performance degradation from address translation overheads is well-corroborated [44, 53, 47, 67].

Operating system support. Navarro et al. [68] implement OS support for multiple page sizes with contiguity-awareness and fragmentation reduction as primary concerns. They propose reservation-based allocation, allocating contiguous ranges of pages in advance, and deferring promotion. Many of their ideas are widely used [80], and it forms the basis of FreeBSD's huge page support. Ingens's utilization-based promotion uses a util bitvector that is similar to the population map [68]. In contrast to that work, Ingens does not use reservation-based allocation, decouples huge page allocation from promotion decisions, and redistributes contiguity fairly when it becomes available (e.g., after process termination). Ingens has higher performance because it promotes more huge pages; it does not require promoted pages to be read-only or completely modified (§3.4). Features in modern systems such as memory compaction and same-page merging [63] pose new challenges not addressed by this previous work.

Gorman et al. [56] propose a placement policy for an OS's physical page allocator that mitigates fragmentation and promotes contiguity by grouping pages according to relocatability. Subsequent work [57] proposes a software-exposed interface for applications to explicitly request huge pages like `libhugetlbfs` [65]. The foci of Ingens, including trade-offs between memory sharing and performance, and unfair allocation of huge pages are unaddressed by previous work.

Hardware support. TLB miss overheads can be reduced by accelerating page table walks [42, 46] or reducing their frequency [52]; by reducing the number of TLB misses (e.g. through prefetching [48, 60, 74], prediction [69], or structural change to the TLB [79, 72, 71] or TLB hierarchy [47, 66, 78, 39, 38, 61, 44, 53]). Multi-page mapping techniques [79, 72, 71] map multiple pages with a single TLB entry, improving TLB reach by a small factor (e.g. to 8 or 16); much greater improvements to TLB reach are needed to deal with modern memory sizes. Direct segments [44, 53] extend standard paging with a large segment to map the majority of an address space to a contiguous physical memory region, but require application modifications and are limited to workloads able to a single large segment. Redundant memory mappings (RMM) [61] extend TLB reach by mapping *ranges* of virtually and physically contiguous pages in a range TLB. The level of additional architectural support is significant, while Ingens works on current hardware.

A number of related works propose hardware support to recover and expose contiguity. GLUE [73] groups

contiguous, aligned small page translations under a single speculative huge page translation in the TLB. Speculative translations, (similar to SpecTLB [43]) can be verified by off-critical-path page-table walks, reducing effective page-table walk latency. GTSM [49] provides hardware support to leverage contiguity of physical memory extents even when pages have been retired due to bit errors. Were such features to become available, hardware mechanisms for preserving contiguity could reduce overheads induced by proactive compaction in Ingens.

Architectural assists are ultimately complementary to our own work. Hardware support can help, but higher-level coordination of hardware mechanisms by software is a fundamental necessity. Additionally, as none of these assists are likely to be realized in imminently available hardware, using techniques such as those we propose in Ingens are a *de facto* necessity.

8 Conclusion

Hardware vendors are betting on huge pages to make address translation overheads acceptable as memory capacities continue to grow. Ingens provides principled, coordinated transparent huge page support for the operating system and hypervisor, enabling challenging workloads to achieve the expected benefits of huge pages, without harming workloads that are well served by state-of the art huge page support. Ingens reduces tail-latency and bloat, while improving fairness and performance.

Acknowledgement

For their insights and comments we thank readers Mark Silberstein, Nadav Amit, Reza Taheri, Kathryn S. McKinley, the anonymous reviewers, and our shepherd Sasha Fedorova. We acknowledge funding from NSF grants CNS-1228843 and CNS-1618563.

References

[1] http://www.7-cpu.com/cpu/Skylake.html. [Accessed April, 2016].

[2] http://www.7-cpu.com/cpu/Haswell.html. [Accessed April, 2016].

[3] Apache Cloudstack. https://en.wikipedia.org/wiki/Apache_CloudStack. [Accessed April, 2016].

[4] Apache Hadoop. http://hadoop.apache.org/. [Accessed April, 2016].

[5] Apache Spark. http://spark.apache.org/docs/latest/index.html. [Accessed April, 2016].

[6] Application-friendly kernel interfaces. https://lwn.net/Articles/227818/. [March, 2007].

[7] Cloudera recommends turning off memory compaction due to high CPU utilization. http://www.cloudera.com/documentation/enterprise/latest/topics/cdh_admin_performance.html. [Accessed April, 2016].

[8] Cloudsuite. http://parsa.epfl.ch/cloudsuite/graph.html. [Accessed April, 2016].

[9] CouchBase recommends disabling huge pages. http://blog.couchbase.com/often-overlooked-linux-os-tweaks. [March, 2014].

[10] Data Plane Development Kit. http://www.dpdk.org/. [Accessed April-2016].

[11] DokuDB recommends disabling huge pages. https://www.percona.com/blog/2014/07/23/why-tokudb-hates-transparent-hugepages/. [July, 2014].

[12] Exponential moving average. https://en.wikipedia.org/wiki/Moving_average#Exponential_moving_average. [Accessed April, 2016].

[13] High CPU utilization in Hadoop due to transparent huge pages. https://www.ghostar.org/2015/02/transparent-huge-pages-on-hadoop-makes-me-sad/. [February, 2015].

[14] High CPU utilization in Mysql due to transparent huge pages. http://developer.okta.com/blog/2015/05/22/tcmalloc. [May, 2015].

[15] Huge page support in Mac OS X. https://developer.apple.com/legacy/library/documentation/Darwin/Reference/ManPages/man2/mmap.2.html. [Accessed April-2016].

[16] IBM cloud with KVM hypervisor. http://www.networkworld.com/article/2230172/opensource-subnet/red-hat-s-kvm-virtualization-proves-itself-in-ibm-s-cloud.html. [March, 2010].

[17] IBM recommends turning off huge pages due to high CPU utilization. http://www-01.ibm.com/support/docview.wss?uid=swg21677458. [July, 2014].

[18] Intel hardware random number generator. https://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide. [May, 2014].

[19] Intel HiBench. https://github.com/intel-hadoop/HiBench/tree/master/workloads. [Accessed April, 2016].

[20] Jemalloc. http://www.canonware.com/jemalloc/. [Accessed April-2016].

[21] Large-page support in Windows. https://msdn.microsoft.com/en-us/library/windows/desktop/aa366720(v=vs.85).aspx. [Accessed April-2016].

[22] Liblinear. https://www.csie.ntu.edu.tw/~cjlin/liblinear/. [Accessed April, 2016].

[23] MongoDB. https://www.mongodb.com/. [Accessed April, 2016].

[24] MongoDB recommends disabling huge pages. https://docs.mongodb.org/manual/tutorial/transparent-huge-pages/. [Accessed April, 2016].

[25] Movie recommendation with Spark. http://ampcamp.berkeley.edu/big-data-mini-course/movie-recommendation-with-mllib.html. [Accessed April, 2016].

[26] NuoDB recommends disabling huge pages. http://www.nuodb.com/techblog/linux-transparent-huge-pages-jemalloc-and-nuodb. [May, 2014].

[27] OpenStack. https://openvirtualizationalliance.org/what-kvm/openstack. [Accessed April-2016].

[28] PARSEC 3.0 benchmark suite. http://parsec.cs.princeton.edu/. [Accessed April, 2016].

[29] Redis. http://redis.io/. [Accessed April, 2016].

[30] Redis recommends disabling huge pages. http://redis.io/topics/latency. [Accessed April, 2016].

[31] Redis SSD swap discussion. http://antirez.com/news/52. [March, 2013].

[32] SAP IQ recommends disabling huge pages. http://scn.sap.com/people/markmumy/blog/2014/05/22/sap-iq-and-linux-hugepagestransparent-hugepages. [May, 2014].

[33] SPEC CPU 2006. https://www.spec.org/cpu2006/. [Accessed April, 2016].

[34] Splunk recommends disabling huge pages. http://docs.splunk.com/Documentation/Splunk/6.1.3/ReleaseNotes/SplunkandTHP. [December, 2013].

[35] Thread-caching malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html. [Accessed April-2016].

[36] Transparent huge pages in 2.6.38. https://lwn.net/Articles/423584/. [January, 2011].

[37] VoltDB recommends disabling huge pages. https://docs.voltdb.com/AdminGuide/adminmemmgt.php. [Accessed April, 2016].

[38] Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh. Revisiting hardware-assisted page walks for virtualized systems. In *International Symposium on Computer Architecture (ISCA)*, 2012.

[39] Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh. Fast two-level address translation for virtualized systems. In *IEEE Transactions on Computers*, 2015.

[40] AMD. *AMD-V Nested Paging*, 2010. http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf.

[41] Jean Araujo, Rubens Matos, Paulo Maciel, Rivalino Matias, and Ibrahim Beicker. Experimental evaluation of software aging effects on the eucalyptus cloud computing infrastructure. In *Middleware Industry Track Workshop*, 2011.

[42] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation caching: Skip, don't walk (the page table). In *International Symposium on Computer Architecture (ISCA)*, 2010.

[43] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Spectlb: A mechanism for speculative address translation. In *International Symposium on Computer Architecture (ISCA)*, 2011.

[44] Arkapravu Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient virtual memory for big memory servers. In *International Symposium on Computer Architecture (ISCA)*, 2013.

[45] Aaron Beitch, Brandon Liu, Timothy Yung, Rean Griffith, Armando Fox, and David Patterson. Rain: A workload generation toolkit for cloud computing applications. In *U.C. Berkeley Technical Publications (UCB/EECS-2010-14)*, 2010.

[46] Abhishek Bhattacharjee. Large-reach memory management unit caches. In *International Symposium on Microarchitecture*, 2013.

[47] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. Shared last-level TLBs for chip multiprocessors. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2011.

[48] Abhishek Bhattacharjee and Margaret Martonosi. Characterizing the TLB behavior of emerging parallel workloads on chip multiprocessors. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2009.

[49] Yu Du, Miao Zhou, B.R. Childers, D. Mosse, and R. Melhem. Supporting superpages in non-contiguous physical memory. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[50] Tammy Everts. The average web page is more than 2 MB size. `https://www.soasta.com/blog/page-bloat-average-web-page-2-mb/`. [June, 2015].

[51] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 37–48, New York, NY, USA, 2012. ACM.

[52] Jayneel Gandhi, , Mark D. Hill, and Michael M. Swift. Exceeding the best of nested and shadow paging. In *International Symposium on Computer Architecture (ISCA)*, 2016.

[53] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Efficient memory virtualization. In *International Symposium on Microarchitecture*, 2014.

[54] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. Large pages may be harmful on numa systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 231–242, Berkeley, CA, USA, 2014. USENIX Association.

[55] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, Hollywood, CA, 2012. USENIX.

[56] Mel Gorman and Patrick Healy. Supporting superpage allocation without additional hardware support. In *Proceedings of the 7th International Symposium on Memory Management*, 2008.

[57] Mel Gorman and Patrick Healy. Performance characteristics of explicit superpage support. In *Workshorp on the Interaction between Operating Systems and Computer Architecture (WIOSCA)*, 2010.

[58] Mel Gorman and Andy Whitcroft. The what, the why and the where to of anti-fragmentation. In *Linux Symposium*, 2005.

[59] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developers Manual*, 2016. `https://www-ssl.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf`.

[60] Gokul B. Kandiraju and Anand Sivasubramaniam. Going the distance for TLB prefetching: An application-driven study. In *International Symposium on Computer Architecture (ISCA)*, 2002.

[61] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrin Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman nsal. Redundant memory mappings for fast access to large memories. In *International Symposium on Computer Architecture (ISCA)*, 2015.

[62] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM: The linux virtual machine monitor. In *Linux Symposium*, 2007.

[63] Kernel Same-page Merging. `https://en.wikipedia.org/wiki/Kernel_same-page_merging`. [Accessed April, 2016].

[64] Ching-Pei Lee and Chih-Jen Lin. Large-scale linear RankSVM. *Neural Comput.*, 26(4):781–817, April 2014.

[65] Huge Pages Part 2 (Interfaces). `https://lwn.net/Articles/375096/`. [February, 2010].

[66] Daniel Lustig, Abhishek Bhattacharjee, and Margaret Martonosi. TLB improvements for chip multiprocessors: Inter-core cooperative prefetchers and shared last-level TLBs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2013.

[67] Timothy Merrifield and H. Reza Taheri. Performance implications of extended page tables on virtualized x86 processors. In *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '16, pages 25–35, New York, NY, USA, 2016. ACM.

[68] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[69] M.-M. Papadopoulou, Xin Tong, A. Seznec, and A. Moshovos. Prediction-based superpage-friendly TLB designs. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[70] Idle Page Tracking. `http://lxr.free-electrons.com/source/Documentation/vm/idle_page_tracking.txt`. [November, 2015].

[71] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H. Loh. Increasing TLB reach by exploiting clustering in page translations. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[72] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. CoLT: Coalesced large-reach TLBs. In *International Symposium on Microarchitecture*, 2012.

[73] Binh Pham, Jan Vesely, Gabriel Loh, and Abhishek Bhattacharjee. Large pages and lightweight memory management in virtualized systems: Can you have it both ways? In *International Symposium on Microarchitecture*, 2015.

[74] Ashley Saulsbury, Fredrik Dahlgren, and Per Stenström. Recency-based TLB preloading. In *International Symposium on Computer Architecture (ISCA)*, 2000.

[75] Tom Shanley. *Pentium Pro Processor System Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1996.

[76] Richard L. Sites and Richard T. Witek. *ALPHA architecture reference manual*. Digital Press, Boston, Oxford, Melbourne, 1998.

[77] Will Sobel, Shanti Subramanyam, Akara Sucharitakul, Jimmy Nguyen, Hubert Wong, Arthur Klepchukov, Sheetal Patil, O Fox, and David Patterson. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0, 2008.

[78] Shekhar Srikantaiah and Mahmut Kandemir. Synergistic tlbs for high performance address translation in chip multiprocessors. In *International Symposium on Microarchitecture*, 2010.

[79] M. Talluri and M. D. Hill. Surpassing the TLB performance of superpages with less operating system support. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1994.

[80] Transparent Hugepages. `https://lwn.net/Articles/359158/`. [October, 2009].

[81] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.