Ingens

Coordinated and Efficient huge page management

Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel

Kishalay Raj
Rachit Arora

Introduction

Defining the background:

- Increased RAM capacities
- Limited TLB capacity
- Base pages limit TLB reach
- Poor huge page support in S/W

Defining the problem:

- Latency
- Memory Bloat
- Unfairness
- Memory savings

State-of-the-art

- Increasing RAM capacities (over terabytes).
  - Challenge: Address translation.
- TLBs cache the virtual-to-physical mappings.
- The TLB capacities haven't scaled like RAM understandably.
- Result : High performance penalty, TLB misses force page table walks.
- Hardware supports huge pages more than ever.
- Operating System/Hypervisor support for huge pages limited.
  - Hodge-Podge of best-effort algorithms and spot fixes.

Ingens

- Ingens : A coordinated, unified approach to huge pages.
- Manages memory contiguity as a first-class resource.
- Tracks utilization and access frequency of pages in memory.

Ingens : Impact on real workload problems

- Latency
    - Huge Pages increase tail latency and cause high latency variation.
    - Improves Cloudstone benchmark by 18% , reduces 90th percentile latency by 41%.

- Bloat
    - Process/VM allocated huge page memory is often internally fragmented.
    - For Redis, Linux bloats memory by 69% vs 0.8% with Ingens.

Ingens : Impact on real workload problems

- Unfairness
  - Linux : Simple, greedy allocation of huge pages.
  - Causes large, persistent performance differences across processes/VMs.
  - Ingens ensure fairness.
- Performance vs Memory Savings
  - Kernel Same-Page Merging (KSM) and other services reduce memory consumption.
  - These services may also prevent a VM from using huge pages containing shared code.
  - Ingens manages saving 71.3% of the memory saved by Linux/KVM while reduce performance slowdown from 6.8-19% to 1.5-2.6%.

Background

- Virtual Memory Hardware Trends
- OS support for huge pages
- Hypervisor support for huge pages
- Performance improvement from huge pages

Virtual Memory Hardware Trends

- DRAM Growth
  - Increased DRAM sizes imply deeper page tables.
  - Increased latency for page table walks.
    - X86 : 4 level page table => Four memory references per translation.
- Hardware Memory Virtualization
  - Extended (Intel) or Nested (AMD) page tables further aggravate VA translation complexity.
  - Both host and guest OS perform VA-> PA translations for a single request.
  - Each layer of lookup in guest => Multi level translations in host.
  - Maximum lookup cost can go as high as 24 lookups. => Higher latency.

Virtual Memory Hardware Trends

- Increased TLB Reach
  - Intel’s new two level TLB design, second level huge table entries have increased.
  - Eg. Haswell : 1024 entries, Skylake : 1536 entries.
  - Increased page size support (2MB vs 4KB) increases TLB reach.
  - Larger pages require contiguity => Risk fragmentation (internal and external).
  - System software must generate, manage and maintain significant memory continuity.
    - Memory contiguity as a first-class resource.

Operating System Support for Huge Pages

- Transparent support is vital
  - Only way to extend huge page support to all applications with dynamic memory behaviour.
  - Kernel autonomously can
    - Promote 512 contiguous and aligned base pages to a huge page.
    - Demote a huge page to 512 independent base pages.
  - Ingens builds on Linux running on Intel processors.
    - Best transparent huge page support currently.
    - Only 4KB and 2MB page sizes are dealt. (1 GB pages are too large for most cases).

Operating System Support for Huge Pages

- Linux Huge Page Management : Greedy and Aggressive
  - Promotes pages in the page fault handler based on local information.
  - Always tries to allocate a huge page to a process if possible.
  - The approach can work only for processes with uniform memory access patterns.
  - Many applications don’t fit this behaviour and are consequently penalized.
  - Results in an intuition: Contiguity should be explicitly managed.
    - Valuable OS resource.

Hypervisor Support for Huge Pages

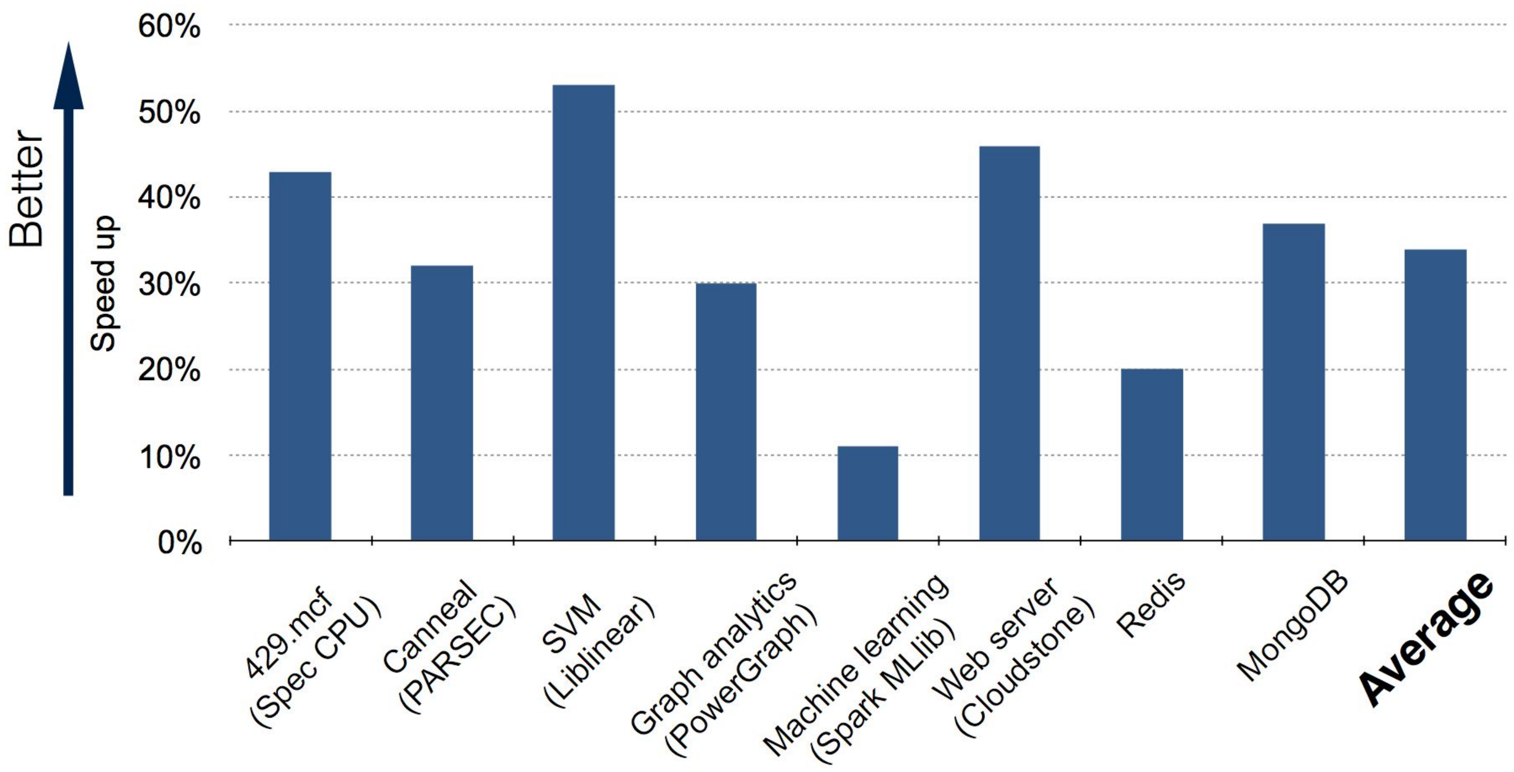
- Focus : Linux used as both the guest OS and host hypervisor.
- Ingens supports host huge pages mapped from guest physical memory.
- Linux : Unified OS and hypervisor memory management.
- We next explore current huge page problems.
  - Some of these problems only apply to the OS.
  - Similarly some only apply to the hypervisor.
- Ingens : Future Scope of work.
  - Support for OS and hypervisor pair that does not share memory management.

Performance Improvement : Huge Pages

- Application speed up recorded with huge page support on linux shows that :
  - There are observable speed ups on enabling huge pages on host as well as guest.
  - Therefore, largest speedup attained when both host and guest use huge pages.
- Therefore huge page support holds a lot of value.
- However, a variety of more challenging workloads and circumstances:
  - Expose the limitations of Linux's memory management.
  - Motivate the need for better and more principled huge page management.

Performance Improvement : Huge Pages

- Application speed up over using base pages only



Current Huge Page Problems

- Page fault latency, synchronous promotion
- Increased memory footprint
- Fragmentation
- Unfair performance
- Memory sharing issues

Page Fault Latency

- Process faults on a memory region
  - => Page fault handler allocates physical memory to the page.
- However, Linux is greedy
  - On a page fault on a base page, it tries to upgrade and allocate a huge page if possible.
- This **synchronous promotion** increases latency because :
  - Linux must zero out a page before returning them to user.
  - Huge pages are 512x larger than base pages (thus, slower to clear).
  - With fragmented memory, contiguous 2MB regions become much harder to find than 4KB.
  - OS must even resort to compacting memory to generate contiguity.
    - Synchronous compaction of memory in page fault handler.
    - Increased average and tail latency.

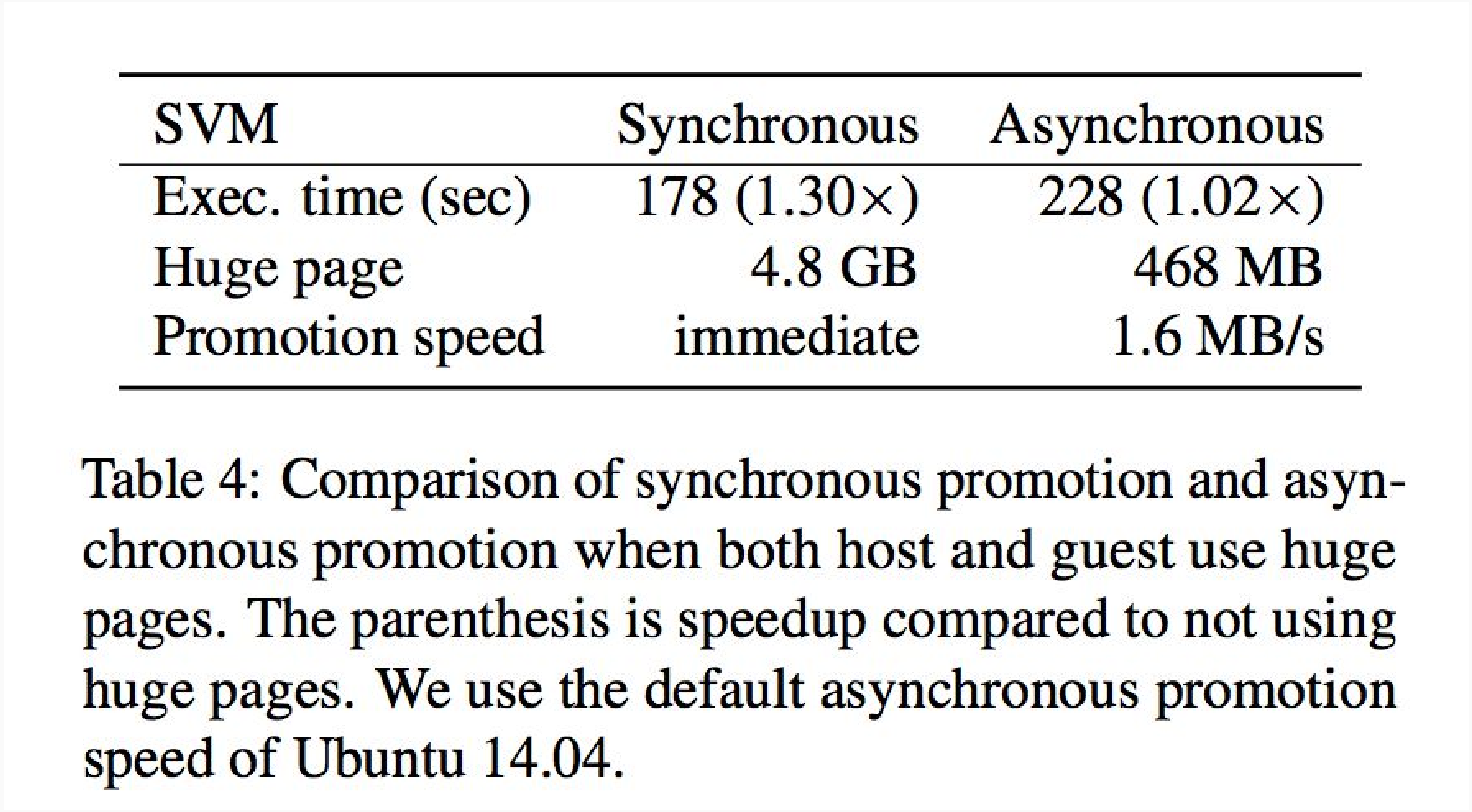
Page Fault Latency

- Free memory fragmentation index (FMFI) :
    - 0 (unfragmented) to 1 (highly fragmented).
- For unfragmented memory (FMFI < 0.1)
    - Page clearing overheads increase page fault latency.
    - 3.6 us for base pages, 378 us (105x slower) for huge pages.
- For fragmented memory (FMFI = 0.9)
    - 8.1 us average latency for base and huge pages. (2.1x 3.6 us)
    - Average latency is lower because 98% allocations are base pages.
    - Why? The memory is too fragmented for huge page allocation.

Page Fault Latency

- Synchronous huge page promotion with page faults:
  - Increases average and tail latency.
  - Penalizes time-sensitive applications. Eg. Web services.
- Alternate : Asynchronous huge page promotion.
  - Configured to work at a promotion speed (in MB/s)
  - Not fast enough in practice at 1.6 MB/s.
- Faster asynchronous promotion => BAD IDEA.
  - Unacceptably high CPU utilization for memory scanning and compaction.
  - Aggressive CPU use reduces/cancels any performance benefits of huge pages.

Page Fault Latency



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.

Increased Memory Footprint

- Application in general may not utilize their allocated huge pages uniformly.
- Linux continues to allocate huge pages irrespective of utilization.
- Leads to several huge pages that are **internally fragmented**.
    - Huge pages reserve large memory regions.
- Promotion of sparsely allocated memory to huge pages => **Bloating!**
- Experiments with:
    - Redis : 2 million keys allocated, 70 % freed => 69% bloating over base page only allocation.
    - MongoDB : Sparse allocation in VA space => 23% bloating over base page only allocation.

Increased Memory Footprint

- Impossible to predict total memory usage of application in production.
  - Memory usage depends on the huge page use.
  - Huge page use depends on memory fragmentation and allocation pattern.
- Eg. Previous instance of Redis application (1.69x bloating).
  - Base page memory usage of 12.2 GB for the job.
  - An 18 GB provision for huge page supported run (1.5x) over-provisioning would cause swapping.
- Linux transparent huge page support susceptible to unacceptable bloating.

Fragmentation

- Greedy promotion quickly consumes available physical memory contiguity.
- Therefore memory becomes increasingly fragmented.
  - => Precondition for page fault latency, memory bloat. VICIOUS CYCLE!

- Redis application tested with initial FMFI = 0.3.
  - Clients populate the server with 13 GB of key/value pairs
  - With huge pages, FMFI quickly rises to 1 and Linux resorts to memory compaction.

Fragmentation

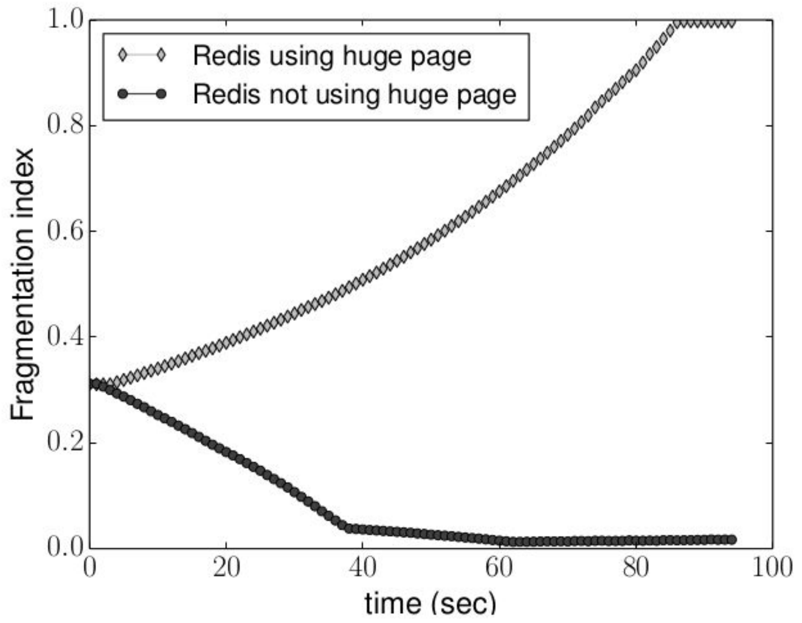
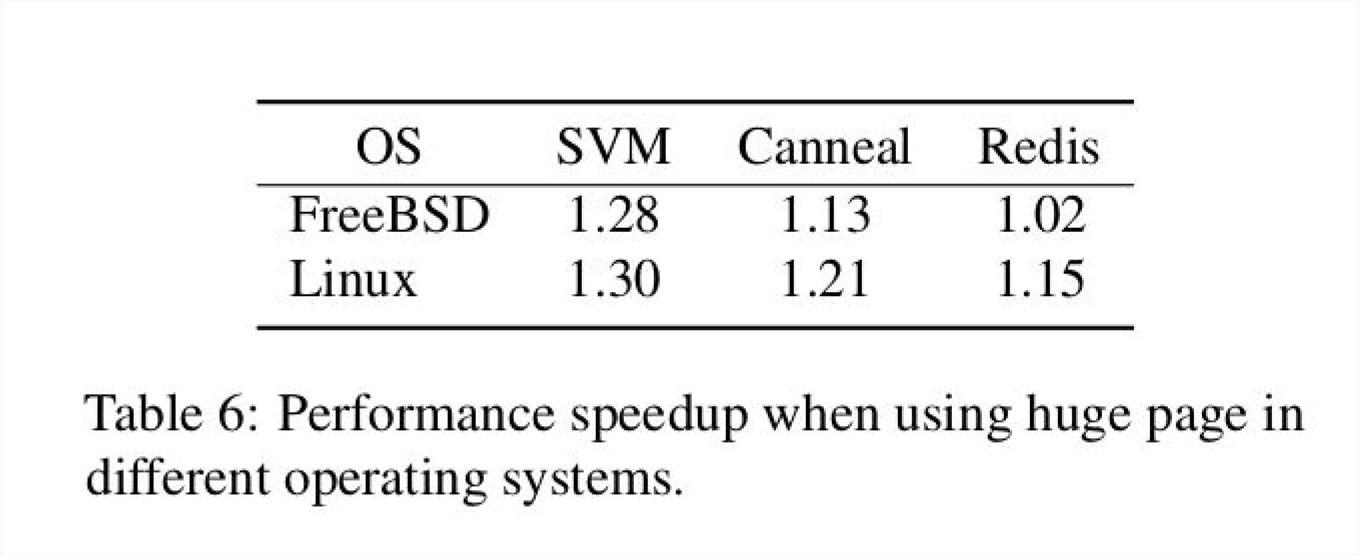


Figure 1: Fragmentation index in Linux when running a Redis server, with Linux using (and not using) huge pages. The System has 24 GB memory. Redis uses 13 GB, other processes use 5 GB, and system has 6 GB free memory.

FreeBSD Huge Page Support

- Reserves contiguous 2MB memory but does not promote immediately.
  - Monitors page utilization while only allocating base pages on page faults.
  - Promotes a reserved region to huge page when the 2MB region in completely utilized.
- FreeBSD supports huge pages for file-cached pages.
- X86 maintains access/dirty bits for the entire huge page.
  - Eviction or swapping of huge page => Increased I/O traffic.
- FreeBSD is conservative about writable huge pages. Marks them read only.
- On a write, huge page demoted to base pages.
- Promoted again only if all base pages are modified.
- Conservative, Asynchronous approach limits speedups.

FreeBSD Huge Page Support



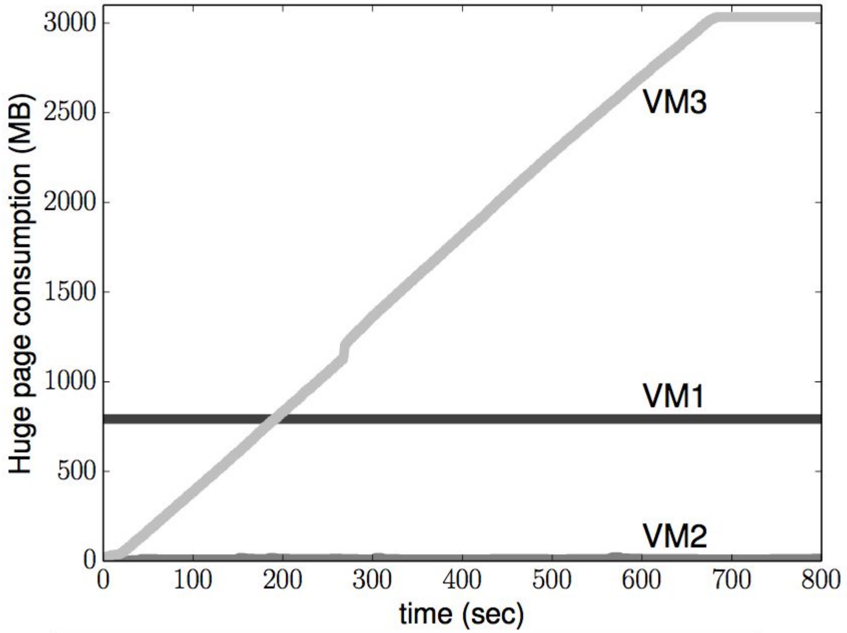
OS	SVM	Canneal	Redis
FreeBSD	1.28	1.13	1.02
Linux	1.30	1.21	1.15

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

Unfair Performance

- Unfair huge page allocation :
  - => Unequal speedups.
  - => Unfair performance differences.
- Aggravated by high fragmentation.

- VM0 starts first in system with FMFI 0.85.
- VM1 starts next.
- VM2 and VM3 follow.
- VM0 terminates and frees its memory.
- Clearly, Linux allocates all new huge pages to VM3.
- VM2 24% slower than VM3 as a result.



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.

Memory Sharing vs Performance

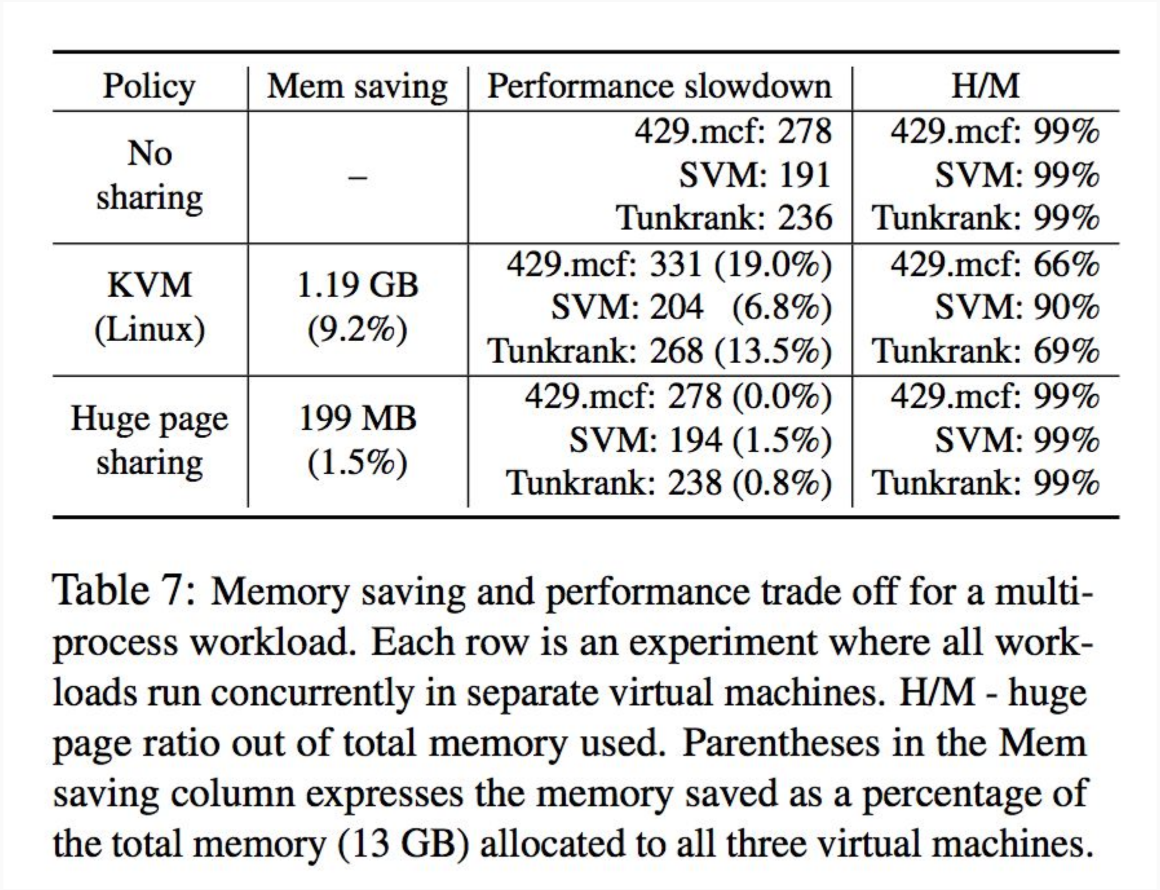
- Sharing identical memory/code reduces guest VM memory footprint.
- KVM hypervisor : Identical page sharing supported with base pages only.
- If a huge page contains shareable base page, KVM demotes it.
- Clearly KVM penalizes performance to save memory.

Alternate Policy : Huge Page Sharing

- Base pages can be shared only with other base pages (no huge pages).
- Huge pages can be shared with other huge pages.
- Prioritizes performance over memory footprint.

Clearly, performance vs memory tradeoff is a common theme in systems.

Memory Sharing vs Performance



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.

Design of Ingens

Basic primitives:

- Utilization tracking
- Access frequency tracking
- Contiguity monitoring

Goals of the design:

- Reduce page fault latency
- Reduce memory bloating
- Fair huge page allocation

Monitoring Space and Time

- Util bitvector :
  - The util bitvector records which base pages are used within each huge-page region
  - The page fault handler updates the util bitvector
- Access bitvector :
  - The access bitvector records the recent access history of a process to its pages
  - Ingens computes the exponential moving average (EMA):
    - F_t = a(weight(util bitvector)) + (1-a)F_{t-1}
    - a = 0.4 for the experiments
  - Experimental verification of frequency information, using the access bit

Fast page faults

- Ingens **decouples promotion decisions** from huge page allocation
- 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**
- Promote-kth **compacts memory if necessary** and promotes the pages identified by the page fault handler

Mitigating memory bloat

- Ingens manages **memory contiguity as a resource**
- It 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 base pages, it wakes up **promote-Kth to promote the base pages to a huge page**
- The **utilization threshold** gives an upper bound on memory bloat
- Similar mechanism for **utilization based** demotion

Proactive compaction and page sharing

- Ingens monitors the fragmentation state of memory
- Proactively compacts memory to reduce the latency of large allocations
- Compaction -> TLB invalidations
- Don’t move frequently accessed pages
    - Minimize impact of TLB invalidations
- Ingens denies sharing if that huge page is frequently accessed, otherwise it allows the huge page to be demoted for sharing

Memory contiguity management

- Fair allocation of memory contiguity across processes
- Share priority for every process
- Ingens imposes a penalty for idle memory
- Per page memory promotion metric:
  - $$\mathscr{M} = \frac{S}{H \cdot (f + \tau(1 - f))}$$
- Scan Kth profiles idle fraction of huge pages for fair promotion

Fair promotion

- Fairness is achieved when all processes have a priority proportional share of the available contiguity
- Mathematically this is achieved by minimizing theta
    - $$\Theta = \sum_i (\mathcal{M}_i - \bar{\mathcal{M}})^2$$
- In practice, an approximate minimization suffices

Implementation

- Huge page promotions
- Access frequency tracking
- Limitations

Huge page promotion

- Promote kth runs as a background kernel thread
- Promote kth maintains two priority lists:
  - High : Promotion requests from page fault handler
  - Low : Promotion requests filled as Promote kth scans memory
- Virtual memory can grow, shrink or be merged
- Promote-kth compares the promotion metric of each application and selects the process with the highest deviation from a fair state
- Remove seemingly adversarial applications

Access frequency tracking

- Scan kth uses the Linux access bit tracking framework to find idle memory
- Default time period for performing the scan is of 2 seconds
- Clearing access bit -> TLB invalidation
- To ameliorate this:
  - Frequency aware profiling and sampling
  - Not frequently accessed : scan Kth clears the access bit
  - Otherwise it clears it with 20% probability
  - Experimental validation for this strategy

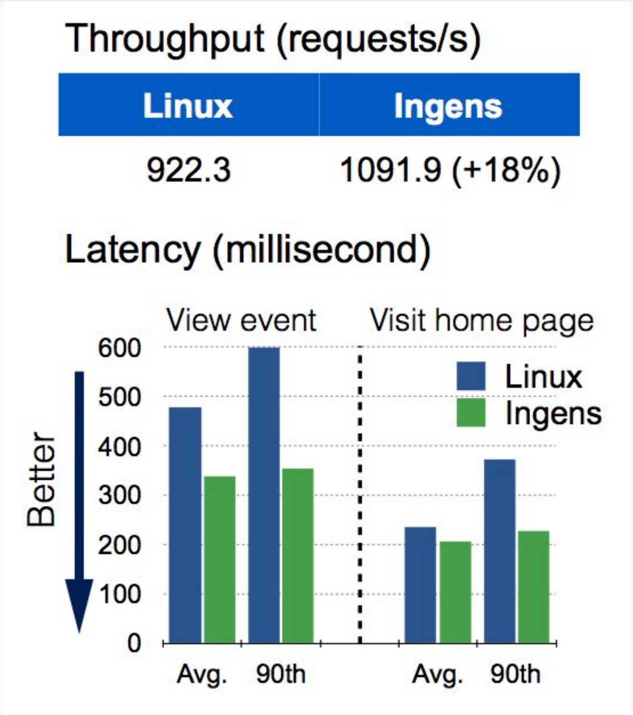
Limitations

- Linux does not support huge pages for page cache pages
  - Makes sense to extend Ingens to manage them
- Hardware support for finer-grain tracking of access and dirty bits for huge pages would greatly benefit Ingens
- All measurements are within a single NUMA domain
  - Huge pages may lead to memory request imbalance.
  - Further work needed to balance huge pages among NUMA domains and split huge pages if false sharing is detected or if they become too hot

Evaluation

- Page fault latency evaluation
- Memory bloating evaluation
- Ingens overhead
- Fair huge page promotion
- Trade off of memory saving and performance

Page fault latency evaluation

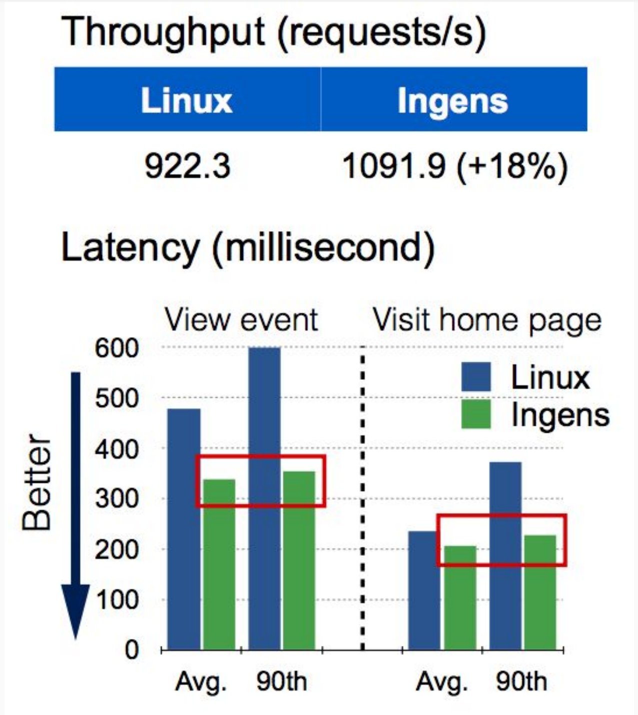

Throughput (requests/s)

Linux	Ingens
922.3	1091.9 (+18%)

Latency (millisecond)

- Fragmented memory
- Cloudstone workload (latency sensitive)
  - 85% read, 10% login, 5% write workloads
- Ingens reduces:
  - Average latency up to 29.2%
  - Tail latency up to 41.4%

Page fault latency evaluation



Throughput (requests/s)

Linux	Ingens
922.3	1091.9 (+18%)

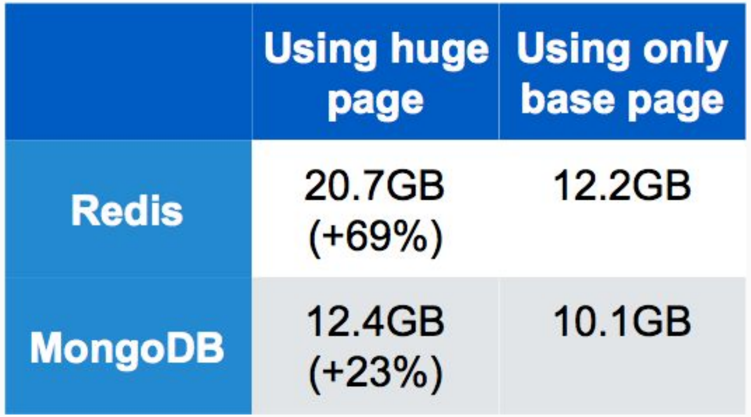
Latency (millisecond)

- Fragmented memory
- Cloudstone workload (latency sensitive)
  - 85% read, 10% login, 5% write workloads
- Ingens reduces:
  - Average latency up to 29.2%
  - Tail latency up to 41.4%

Memory bloating evaluation

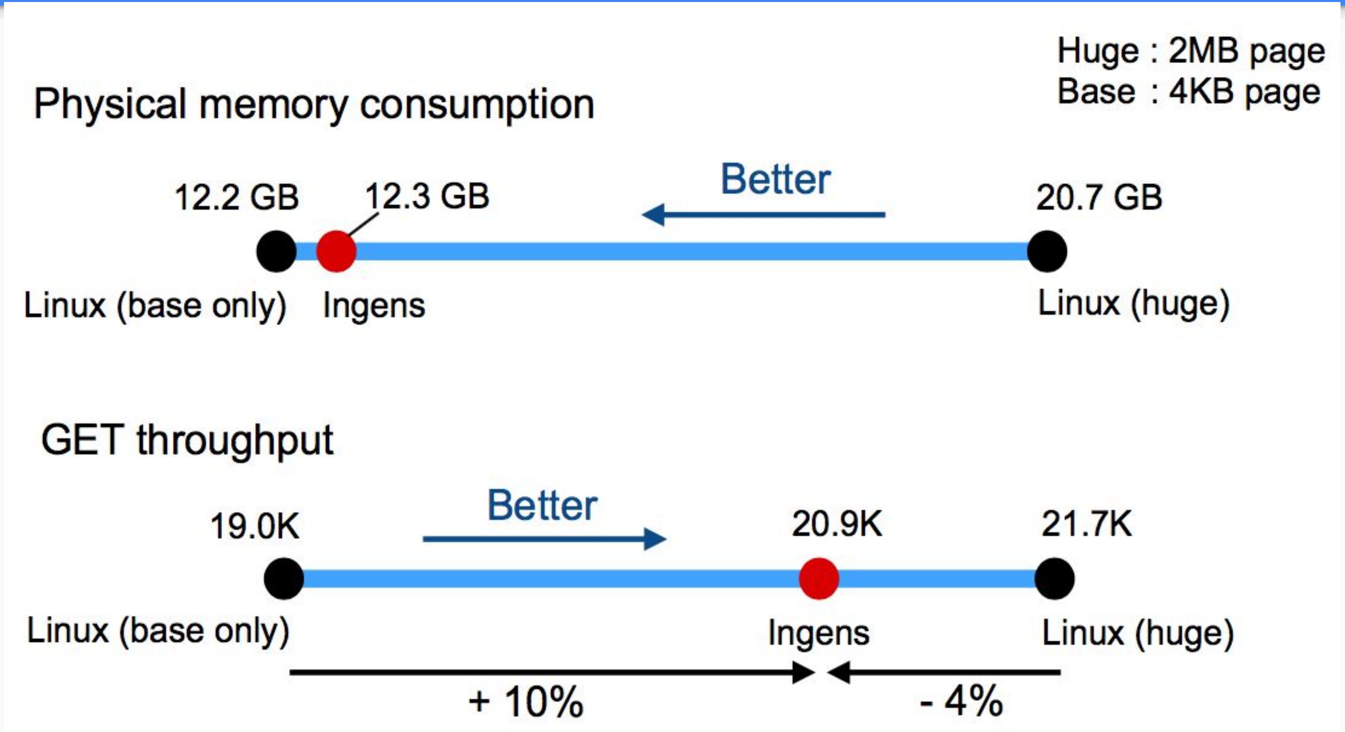
- Redis
  - Delete 70% objects after populating 8KB objects
- Mongo DB
  - 15 million GET requests for 1KB object with YCSB
- Bloating makes memory consumption unpredictable. Can't avoid swapping.

Physical memory consumption:



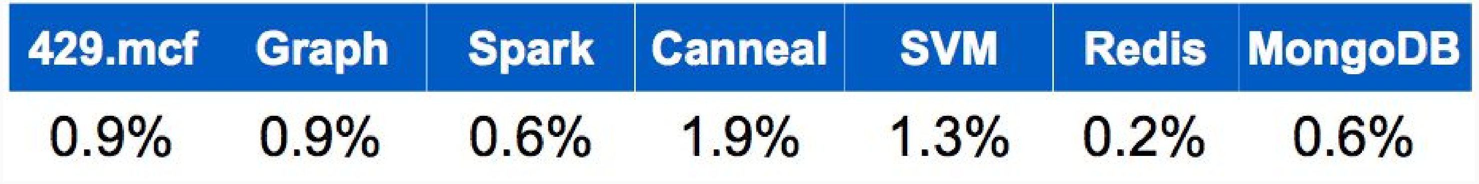
	Using huge page	Using only base page
Redis	20.7GB (+69%)	12.2GB
MongoDB	12.4GB (+23%)	10.1GB

Memory bloating experiment



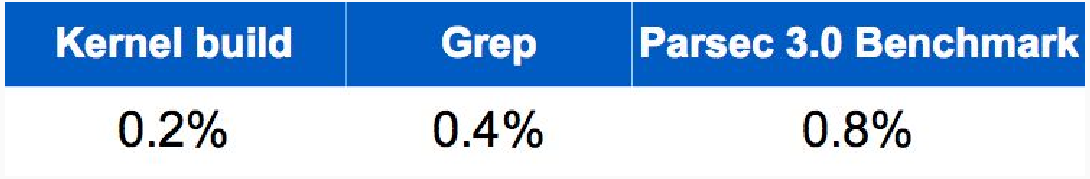
Ingens overhead is small

- Overhead for memory intensive applications:



429.mcf	Graph	Spark	Canneal	SVM	Redis	MongoDB
0.9%	0.9%	0.6%	1.9%	1.3%	0.2%	0.6%

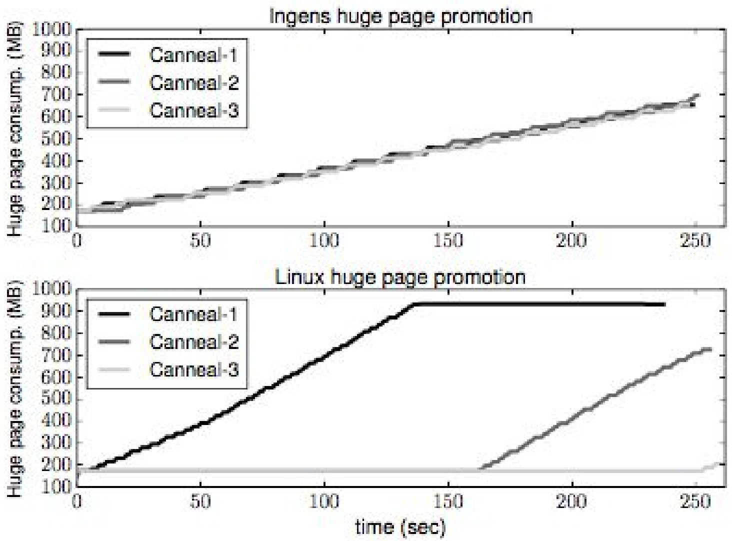
- Overhead for non-memory intensive applications:



Kernel build	Grep	Parsec 3.0 Benchmark
0.2%	0.4%	0.8%

Fair huge page promotion

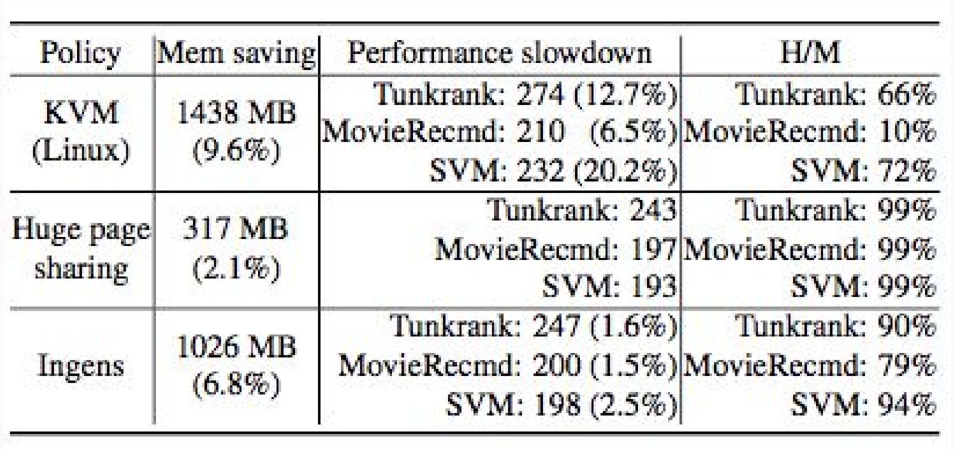
- Ingens promotes huge pages based on the fairness objective defined
- Fair distribution of huge pages translates to fair end-to-end execution time as well
- All applications finish at the same time



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

Trade off of memory saving and performance

- Three different OS configurations:
  - KVM with aggressive page sharing
  - KVM where pages of same type are shared
  - Ingens
- Infrequently used huge pages are demoted for sharing
- Ratio of huge pages to the total pages remains high in Ingens due to it's access frequency based approach



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%

Conclusion

- Hardware vendors are betting big on huge pages
- Ingens provides coordinated, transparent huge page support
- Ingens reduces tail latency and bloat while improving fairness and performance