Ingens: Huge Page Support for the OS and Hypervisor

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

Memory capacity and demand have grown hand in hand in recent years. However, overheads for memory virtualization, in particular for address translation, grow with memory capacity as well, motivating hardware manufacturers to provide TLBs with thousands of entries for larger pages, or *huge pages*. Current OSes and hypervisors support huge pages with a hodge-podge of best-effort algorithms and spot fixes that make less and less sense as architectural support for huge pages matures. The time has come for a more fundamental redesign.

Ingens is a framework for providing transparent huge page support in a coordinated way. Ingens manages contiguity as a first-class resource, and tracks utilization and access frequency of memory pages, enabling it to eliminate pathologies that plague current systems. Experiments with a Linux/KVM-based prototype show improved fairness and performance, and reduced tail latency and memory bloat for important applications such as Web services and Redis. We report early experiences with our in-progress port of Ingens to the ESX Hypervisor.

1 Introduction

Modern computing platforms can support terabytes of RAM and workloads able to use large memories are commonplace [47]. However, increased capacity is a significant performance challenge for memory virtualization. All modern processors use page tables for address translation and TLBs to cache virtual-to-physical mappings. TLB capacities cannot scale at the same rate as DRAM, so TLB misses and address translation can incur crippling performance penalties for large memory workloads [41, 49] using traditional 4KB pages. Hardware-supported address virtualization (e.g., AMD's nested page tables) increase average-case translation overheads because multi-dimensional page tables amplify translation costs as much as 6× [54]. 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 good huge page management from operating systems and hypervisors.

While huge pages have been commonly supported in hardware since the 90s [70, 71], 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.

OS memory management has generally supported huge page-capable hardware with best-effort algorithms and spot fixes, keeping core memory management algorithms focused on the 4KB page (or *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 [22, 9, 10, 28, 24, 29, 31, 34]. Other OSes suffer similar or even more severe problems supporting huge pages.

Ingens is a memory manager for the OS and hypervisor that replaces best-effort mechanisms 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:

• **Latency.** Huge pages expose applications to high latency variation and increased tail latency (§3.1). Ingens improves the Cloudstone benchmark [72] 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 (§3.4). Ingens makes huge page allocation fair (e.g., Figure 5).

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. Measurements of our Ingens prototype on realistic workloads validates the approach.

2 Background

Virtual memory decouples the address space used by programs from that exported by physical memory. 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 large virtual pages must be backed by large 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).

Good huge page support is a major challenge, but the potential performance benefits are compelling. Current trends in memory management hardware make it increasingly critical for system software to support huge pages efficiently and flexibly. Our work on Ingens is motivated by three important hardware trends: *DRAM growth*, hardware virtualization with *multi-dimensional page tables*, and *increased TLB reach*.

2.1 Hardware trends

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 (slated to increase to 5 in the near future)– in the worst case, the number of memory references required for a single address translation is equal to the number of levels. Multi-dimensional page tables (e.g. Intel EPT [54], AMD NPT [37]) require additional indirection for each level. During translation, guest physical addresses are treated as host virtual addresses, and each layer of lookup in the guest can require a multi-level translation in the host, amplifying the worst-case cost [54, 37], and increasing average latencies [62]. Recently, Intel has improved huge page support in hardware, increasing the number of L2 TLB entries for huge pages 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 is an opportunity for the OS and the hypervisor, but it puts stress on current memory management designs. In addition to managing the complexity of different page granularities, system software must generate and maintain significant memory contiguity to use larger page sizes.

2.2 System software support for huge pages

Early OS 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 use an explicit memory allocation API for huge pages [19] 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 [14]. Initial huge page support in Linux used a similar separate interface (`hugetlbfs`). Alternative APIs increase complexity for the developer, and render legacy applications unable to enjoy the benefits of huge pages [6, 33].

Transparent support. Transparent huge page support [75, 63] 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). 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.

The techniques used by Ingens are not specific to a particular OS or hypervisor but we base our original prototype on Linux/KVM [57] as it is very widely used [25, 15, 3]. Implementation of Ingens in the ESX hypervisor is currently under way. 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.

Name	Suite/Application	Description
429.mcf	SPEC CPU 2006 [30]	Single-threaded scientific computation
Canneal	PARSEC 3.0 [26]	Parallel scientific computation
SVM [59]	Liblinear [20]	Machine learning, Support vector machine
Tunkrank [8]	PowerGraph [50]	Large scale in-memory graph analytics
Nutch [17]	Hadoop [4]	Web search indexing using MapReduce
MovieRecmd [23]	Spark/MLlib [5]	Machine learning, Movie recommendation
Olio	Cloudstone [8]	Social-event Web service (ngnix/php/mysql)
Redis	Redis [27]	In-memory Key-value store
MongoDB	MongoDB [21]	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 (§3.4)	O	O
Memory sharing		O

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

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 memory contiguity as a valuable resource and explicitly manages it.

Hypervisor support for huge pages. 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 system. We describe a number of problems with huge pages in §3. Some apply only to the OS, some only to the hypervisor (summarized in Table 2).

2.3 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 [75] for both the application (in the guest) and the virtual machine (in the host). The hardware configuration is detailed in §5.

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 with huge pages (2 MB) 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 base and huge pages.

Table 3 shows the performance improvements gained with transparent huge page support for both the guest and the host OS. 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.

3 Huge page problems

This section quantifies limitations in performance and fairness for the state-of-the-art in transparent huge page management, whose variety and severity motivate the design of Ingens. The problems discussed here do not represent an exhaustive list. We refer the interested reader to our original work with Ingens [58]. All results described here use the experimental setup described in §2.3.

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. Base and huge pages share this code path. If an application faults on a base page, Linux will immediately try to upgrade the request and allocate a huge page.

This approach 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

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.

pages, and thus slower to clear. Second, huge page allocation requires 2 MB of physically contiguous memory. If memory is fragmented, the OS must compact memory to create that contiguity, and Linux will synchronously compact memory in the page fault handler. Because memory quickly fragments in multi-tenant cloud environments [38], this approach increases average and tail latency for applications.

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) [53], 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). 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: aggressive asynchronous promotion incurs unacceptably high CPU utilization, reducing or erasing the performance benefits of huge pages [16, 13, 12, 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.

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.

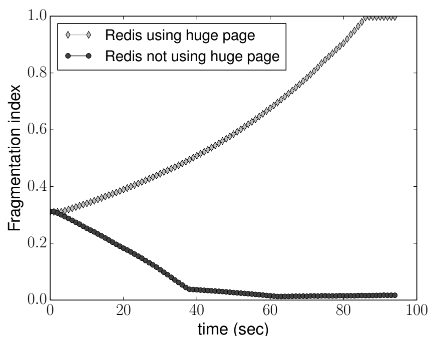
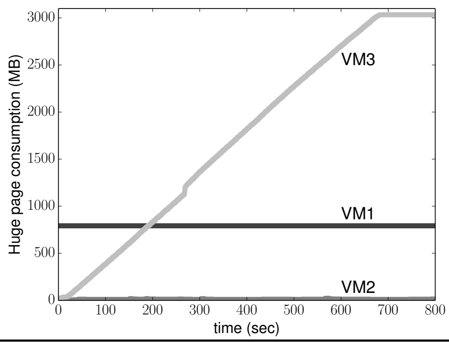


Figure 1: Fragmentation index in Linux when running a Redis server, with Linux using (and not using) huge pages. The System has 24 GB memory. Redis uses 13 GB, other processes use 5 GB, and system has 6 GB free memory.

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.

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. 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.



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.

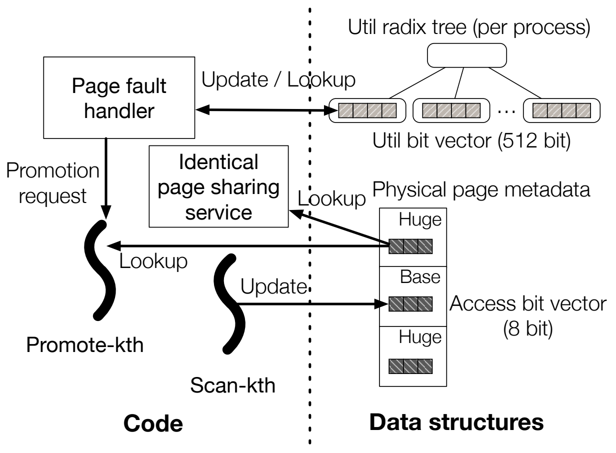


Figure 3: Important Ingens code and data structures.

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. 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 Unfair performance

All measurements presented in this paper are on VMs 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 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.

4 Design and Implementation

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. Figure 3 shows important data structures and code paths of Ingens. Ingens is implemented in Linux 4.3.0 with extensions to track page utilization and access frequency tracking.

4.1 Monitoring space and time

Ingens introduces 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 leverages it throughout the kernel to inform policy decisions. The information is stored as two bitvectors, called *util* and *access*.

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 an exponential moving average [11] from the bitvector with an empirically chosen weight parameter.

Ingens uses Linux's access bit tracking framework [65]. The framework 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 (default is every 2s) scan of application memory. Scan-kth clears the access bits at the beginning of the profiling period and queries the idle flag at the end.

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.

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.

Utilization-based demotion. A process can free a base page, usually by calling `free`. If that 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 [18], whose `free` implementation makes an `madvise` system call with the `MADV_DONTNEED` flag to release the memory[1]. Linux demotes the huge page that contains the freed base page.

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

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 controls 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 per compaction. Compaction moves pages, which necessitates TLB shootdowns. Ingens avoids moving frequently accessed pages.

4.5 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

[1]TCMalloc [32] also functions this way.

penalty [76]. 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 [76] to express our per-process memory promotion metric mathematically as follows.

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

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 ($0 \le f \le 1$) and τ, with $0 < \tau \le 1$, is a parameter to control the idleness penalty. Larger values of $\mathscr{M}$ receive higher priority for huge page promotion. 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.

4.6 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:

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

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 $\mathscr{O}$. We optimize $\mathscr{O}$, by iteratively selecting the process with the biggest $\mathscr{M}_i$, scanning its address space to promote huge pages, and updating $\mathscr{M}_i$ and $\mathscr{O}$.

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.

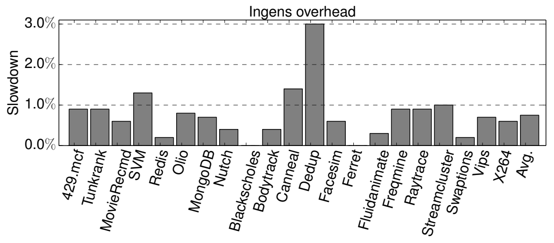


Figure 4: Performance slowdown of utilization-based promotion relative to Linux when memory is not fragmented.

5 Evaluation

We evaluate Ingens using the applications in Table 1, comparing against the performance of Linux THP. 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. Our experiments use only 4 KB and 2 MB huge pages. We set the number of vCPUs equal to the number of application threads.

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. We use a single configuration to evaluate Ingens which is consistent with our examples in Sections 4 and 4: 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 $F_t \ge 6$.

5.1 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. Additional overheads for proactive compaction depend on the frequency of compaction and the amount of data compacted: compacting 100MB every 2 seconds induces an additional 1.3% CPU utilization.

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

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 7: Redis memory use and performance.

Access tracking for MongoDB using 10.7 GB induces 11.4%.

5.2 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 [42] for load.

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 6 (a) shows the throughput attained by the benchmark running on Linux and Ingens. achieves a speedup of 1.18× over Linux. Table 6 (b) shows average and tail 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.

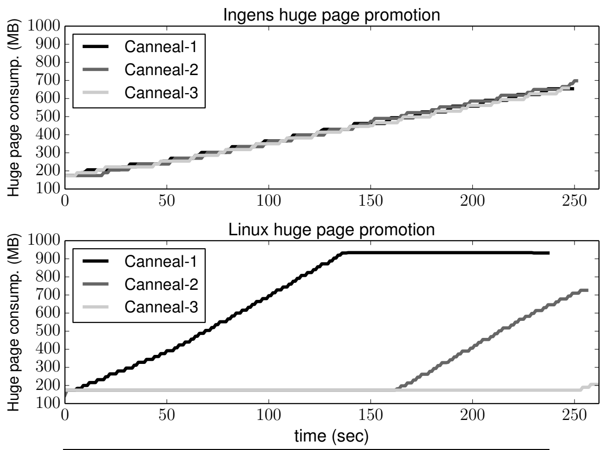
Ingens copes with this performance problem with an adaptive policy. When memory fragmentation is below 0.5 Ingens mimics Linux's aggressive huge page allocation. This policy restores Ingens's performance to Linux's levels. However, while bloat (§3.2) is not a problem for this workload, the adaptive policy 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.

5.3 Memory bloating evaluation

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 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 7 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.



	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.

5.4 Fair huge page promotion

Ingens guarantees a fair distribution of huge pages. If applications have the same share priority (§4.5), 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 promoted 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 4.6 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 Related work

Virtual memory is an active research area. Our evidence of performance degradation from address translation overheads is well-corroborated [41, 49, 44, 62].

Operating system support. Navarro et al. [63] implement OS support for multiple page sizes with contiguity-awareness and fragmentation reduction as primary concerns. Ingens's utilization-based promotion uses a util bitvector that is similar to the population map [63]. 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). Gorman et al. [51] 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 [52] proposes a software-exposed interface for applications to explicitly request huge pages like `libhugetlbfs` [60].

Hardware support. TLB miss overheads can be reduced by accelerating page table walks [39, 43] or reducing their frequency [48]; by reducing the number of TLB misses (e.g. through prefetching [45, 55, 69], prediction [64], or structural change to the TLB [74, 67, 66] or TLB hierarchy [44, 61, 73, 36, 35, 56, 41, 49]).

A number of related works propose hardware support to recover and expose contiguity. GLUE [68] groups contiguous, aligned small page translations under a single speculative huge page translation in the TLB. Speculative translations, (similar to SpecTLB [40]) can be verified by off-critical-path page-table walks, reducing effective page-table walk latency. GTSM [46] 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.

7 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.

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

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

[12] 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].

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

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

[15] 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].

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[36] J. Ahn, S. Jin, and J. Huh. Fast two-level address translation for virtualized systems. In *IEEE Transactions on Computers*, 2015.

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

[38] J. Araujo, R. Matos, P. Maciel, R. Matias, and I. Beicker. Experimental evaluation of software aging effects on the eucalyptus cloud computing infrastructure. In *Middleware Industry Track Workshop*, 2011.

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

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

[41] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift. Efficient virtual memory for big memory servers. In *International Symposium on Computer Architecture (ISCA)*, 2013.

[42] A. Beitch, B. Liu, T. Yung, R. Griffith, A. Fox, and D. Patterson. Rain: A workload generation toolkit for cloud computing applications. In *U.C. Berkeley Technical Publications (UCB/EECS-2010-14)*, 2010.

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

[44] A. Bhattacharjee, D. Lustig, and M. Martonosi. Shared last-level TLBs for chip multiprocessors. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2011.

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

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

[47] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. 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.

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

[49] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift. Efficient memory virtualization. In *International Symposium on Microarchitecture*, 2014.

[50] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. 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.

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

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

[53] M. Gorman and A. Whitcroft. The what, the why and the where to of anti-fragmentation. In *Linux Symposium*, 2005.

[54] 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`.

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

[56] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. nsal. Redundant memory mappings for fast access to large memories. In *International Symposium on Computer Architecture (ISCA)*, 2015.

[57] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: The linux virtual machine monitor. In *Linux Symposium*, 2007.

[58] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel. Coordinated and efficient huge page management with ingens. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 705–721, GA, 2016. USENIX Association.

[59] C.-P. Lee and C.-J. Lin. Large-scale linear RankSVM. *Neural Comput.*, 26(4):781–817, Apr. 2014.

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

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

[62] T. Merrifield and H. R. 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.

[63] J. Navarro, S. Iyer, P. Druschel, and A. Cox. Practical, transparent operating system support for superpages. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

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

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

[66] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh. Increasing TLB reach by exploiting clustering in page translations. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[67] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee. CoLT: Coalesced large-reach TLBs. In *International Symposium on Microarchitecture*, 2012.

[68] B. Pham, J. Vesely, G. Loh, and A. Bhattacharjee. Large pages and lightweight memory management in virtualized systems: Can you have it both ways? In *International Symposium on Microarchitecture*, 2015.

[69] A. Saulsbury, F. Dahlgren, and P. Stenström. Recency-based TLB preloading. In *International Symposium on Computer Architecture (ISCA)*, 2000.

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

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

[72] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, A. Klepchukov, S. Patil, O. Fox, and D. Patterson. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0, 2008.

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

[74] 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.

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

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