Coordinated and Efficient Huge Page Management With Ingens

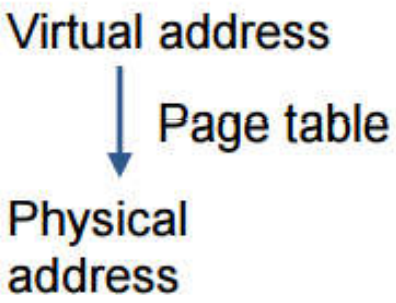
Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossabach, Emmett Witchel

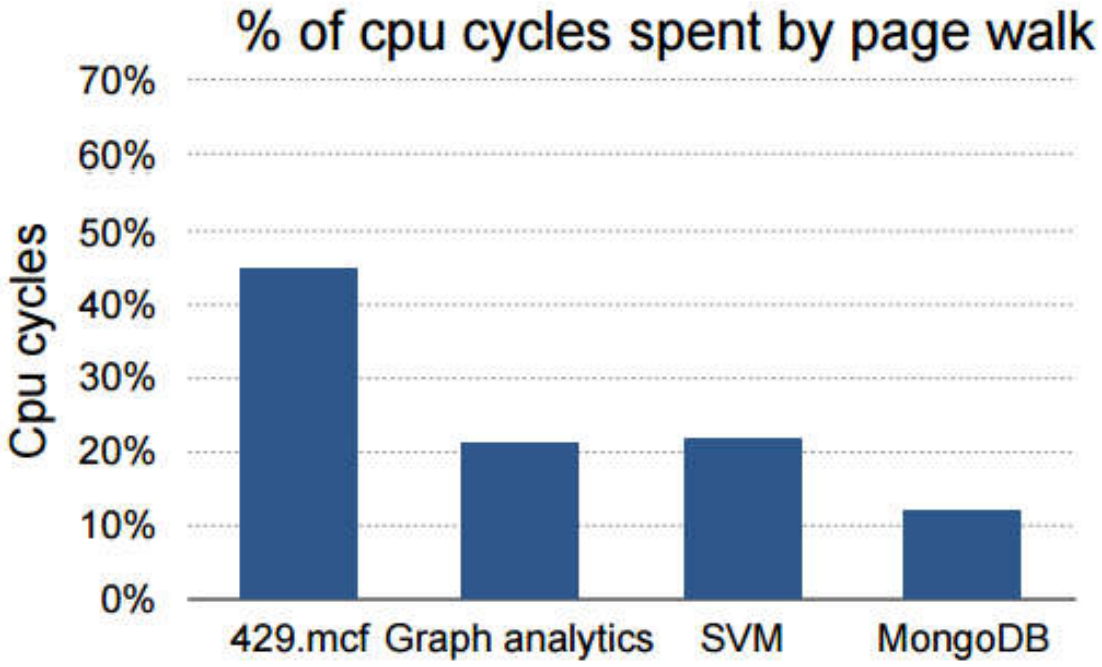
12th USENIX Symposium on Operating System Design and Implementation (OSDI ' 16)

Presented By: Arifa Anwar

- Virtual Address is a very useful abstraction for it requires address translation
- Address translation is not cheap as used to be
- Modern Applications use
  - Very large memory foot prints
  - Very low memory access locality
- TLB capacity does not scale with the DRAM size
  - Applications suffer from the **High Address Translation Cost**

Huge Address Translation Cost

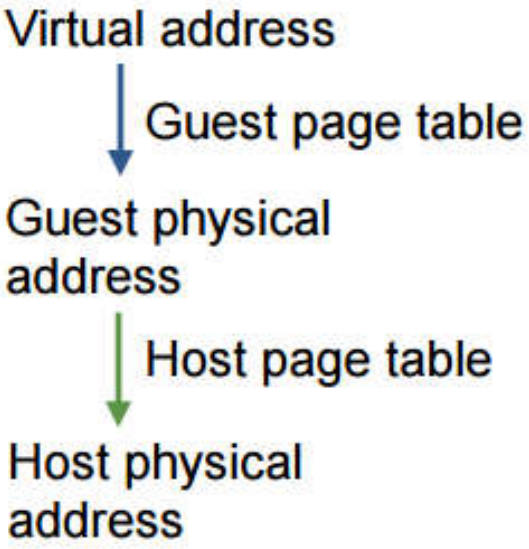


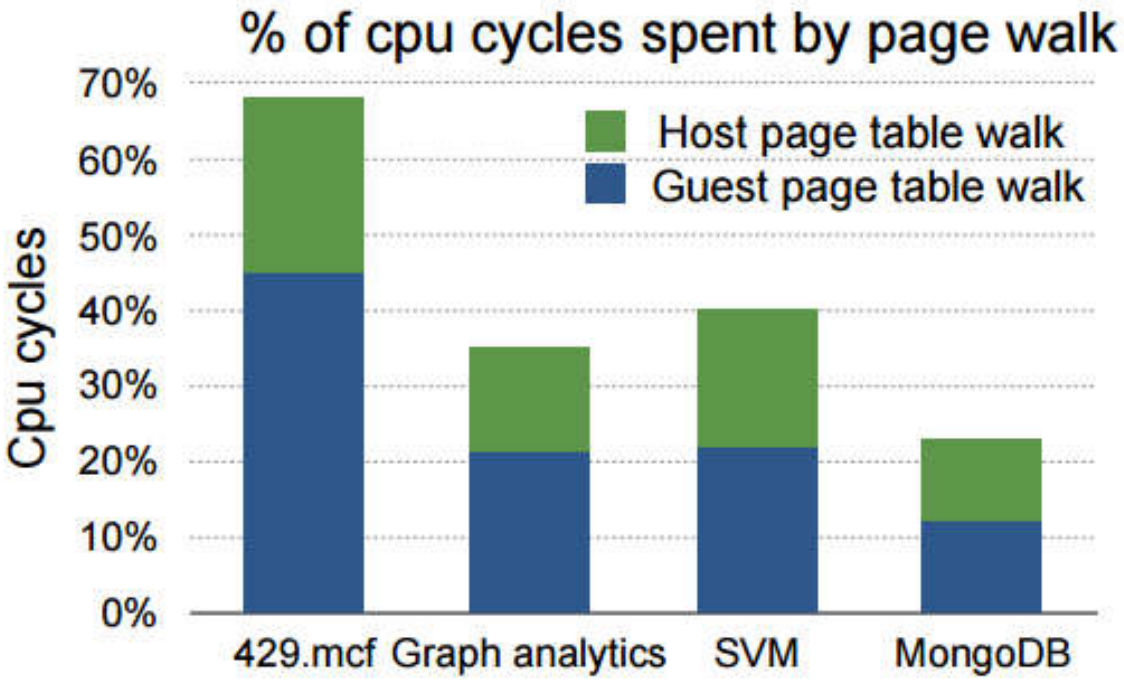


Courtesy Youngjin Kwon et al.

Huge Address Translation Cost

- Virtualization requires additional address translation layer
- TLB needs to cache these two layers

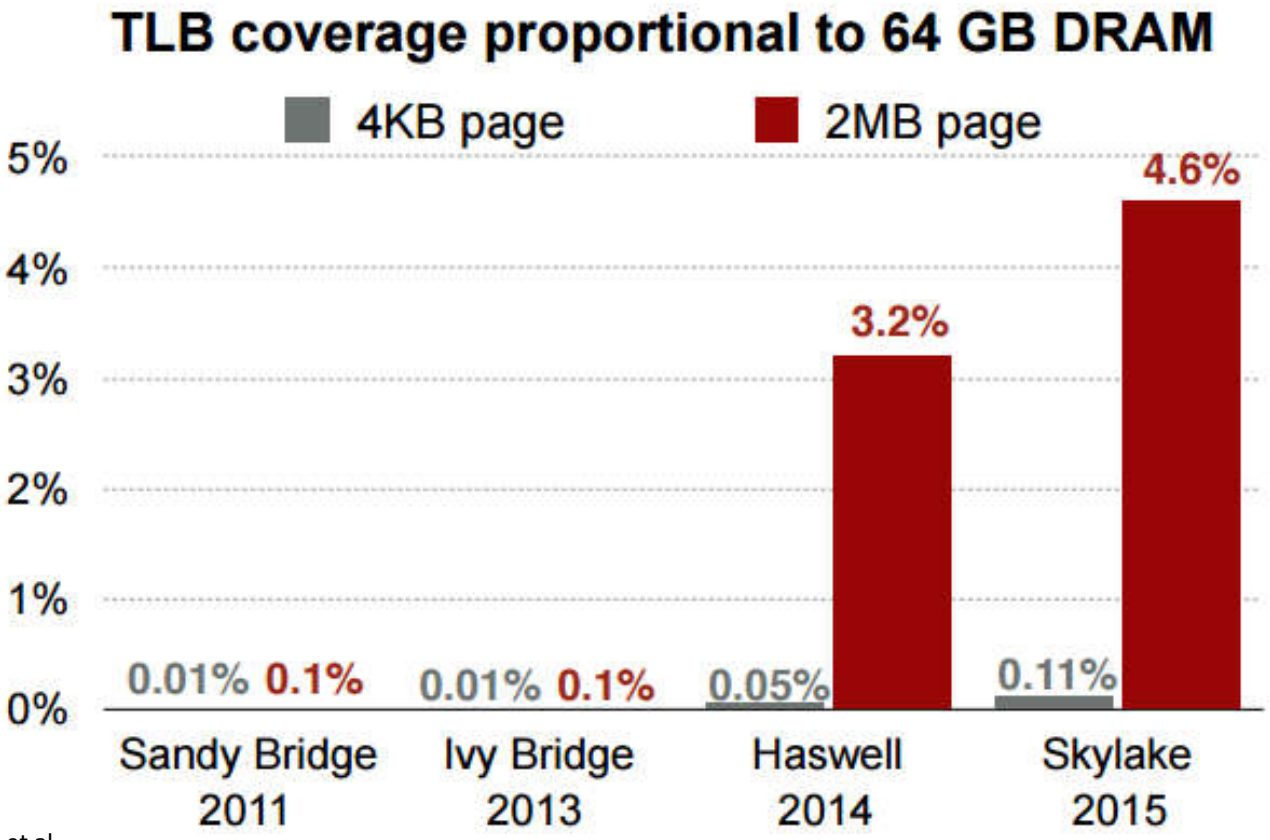




Courtesy Youngjin Kwon et al.

- To mitigate the problem, Hardware vendors provide a larger page size–Huge Page (e.g. 2MB page)
- Means, architecture support larger page size- drastically increase TLB capacity
- Hardware vendor response to count trend of applications, so they drastically increase TLB capacity for 2MB pages
- **Huge Pages improves TLB Coverage**

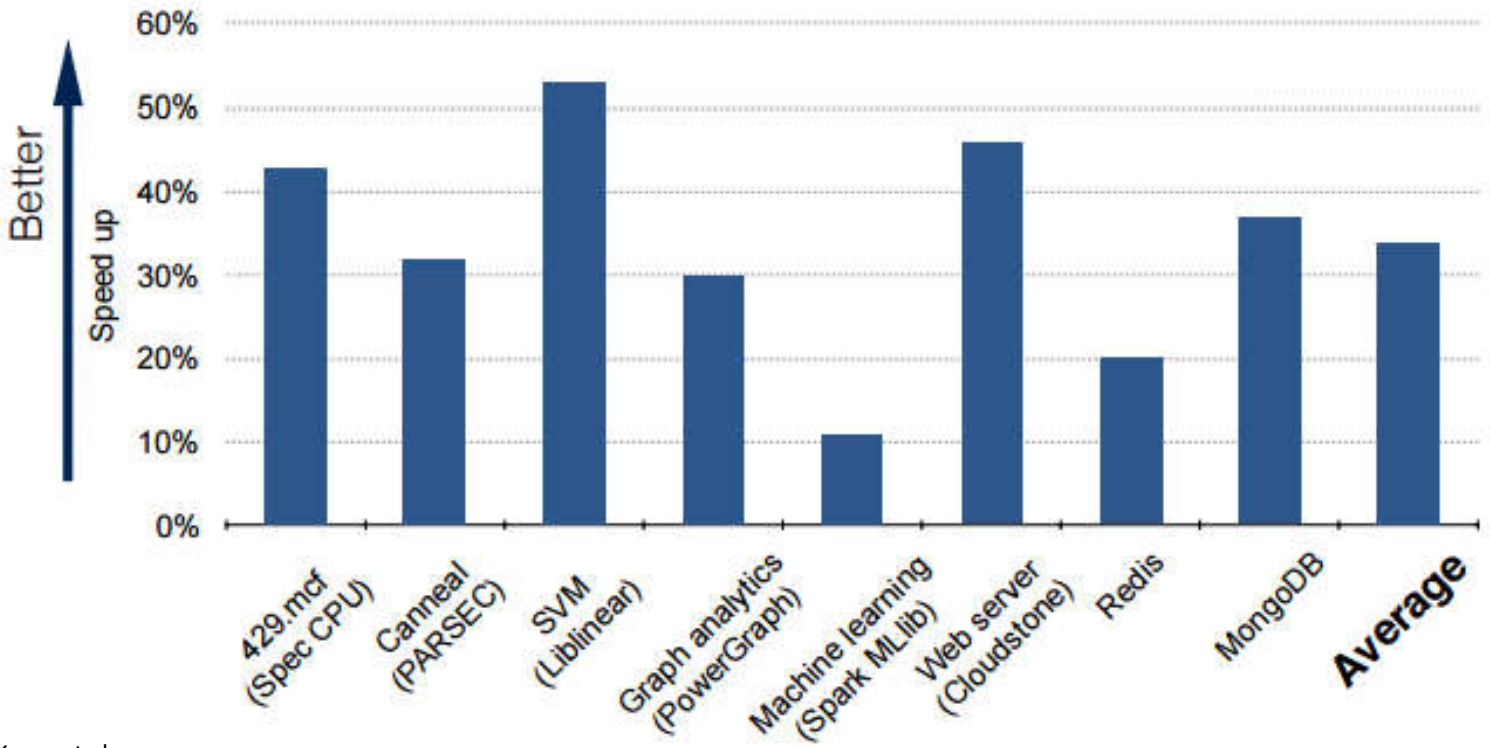
- Operating system has the burden of better huge page support



Courtesy Youngjin Kwon et al.

- **Operating System Support for huge pages**
- OS also provide huge page support transparently
  - Meaning, applications used a traditional API like malloc() and free()
- OS transparently allocates/ deallocates huge pages for both guest and host pages
- Super pages Integrated in FreeBSD and also Linux support transparent huge pages

- With hardware and OS support, measure actual **Performance benefits of Huge pages**
- Applications speed up over using base pages only



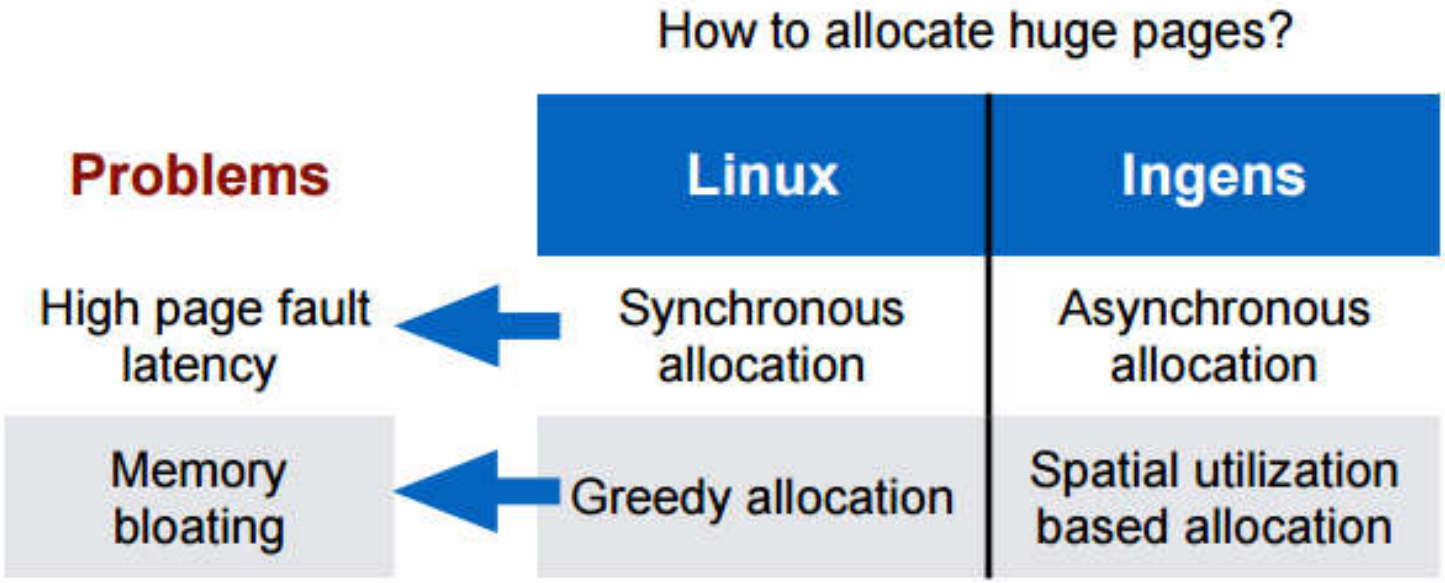
Courtesy Youngjin Kwon et al.

- Running Huge pages
  - On Redis server: “you are running huge pages, turn it off”
  - On MongoDB: “want to dissolve huge page support”
  - Many cloud vendors applications want to turn off these huge pages support
- Explicitly state

- Four main **Pathologies in using Huge page** management system
  - High page fault latency
  - Memory bloating
  - Unfair huge page allocation
  - Uncoordinated memory management

Ingens

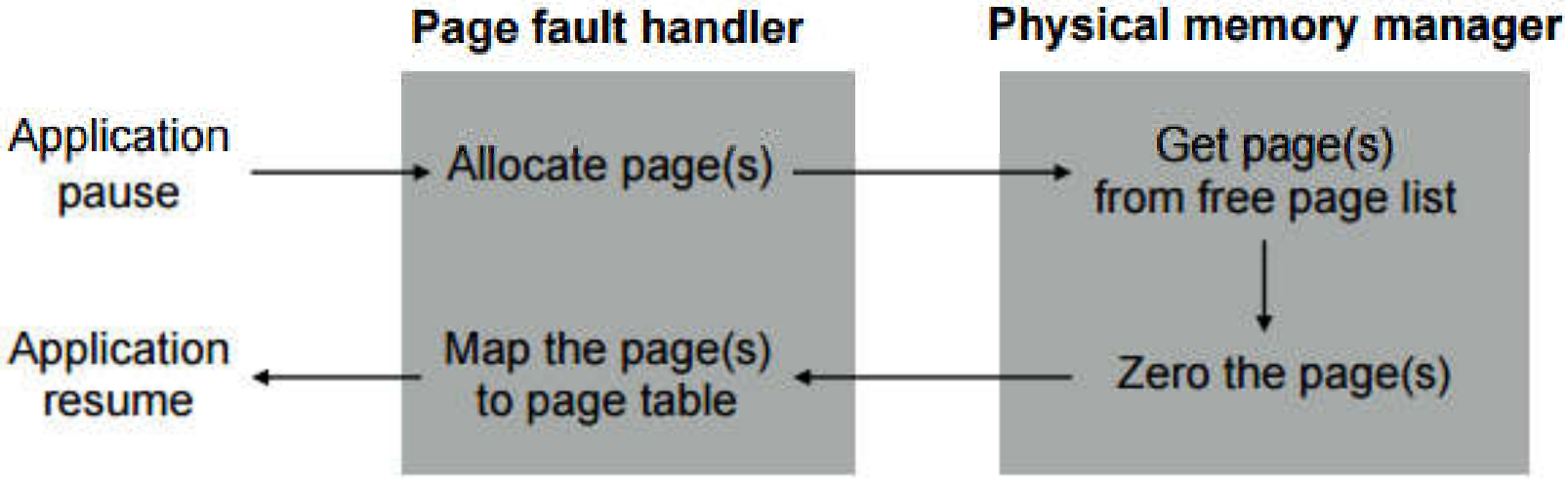
- Efficient huge page management system
- Address the pathologies



Courtesy Youngjin Kwon et al.

1st

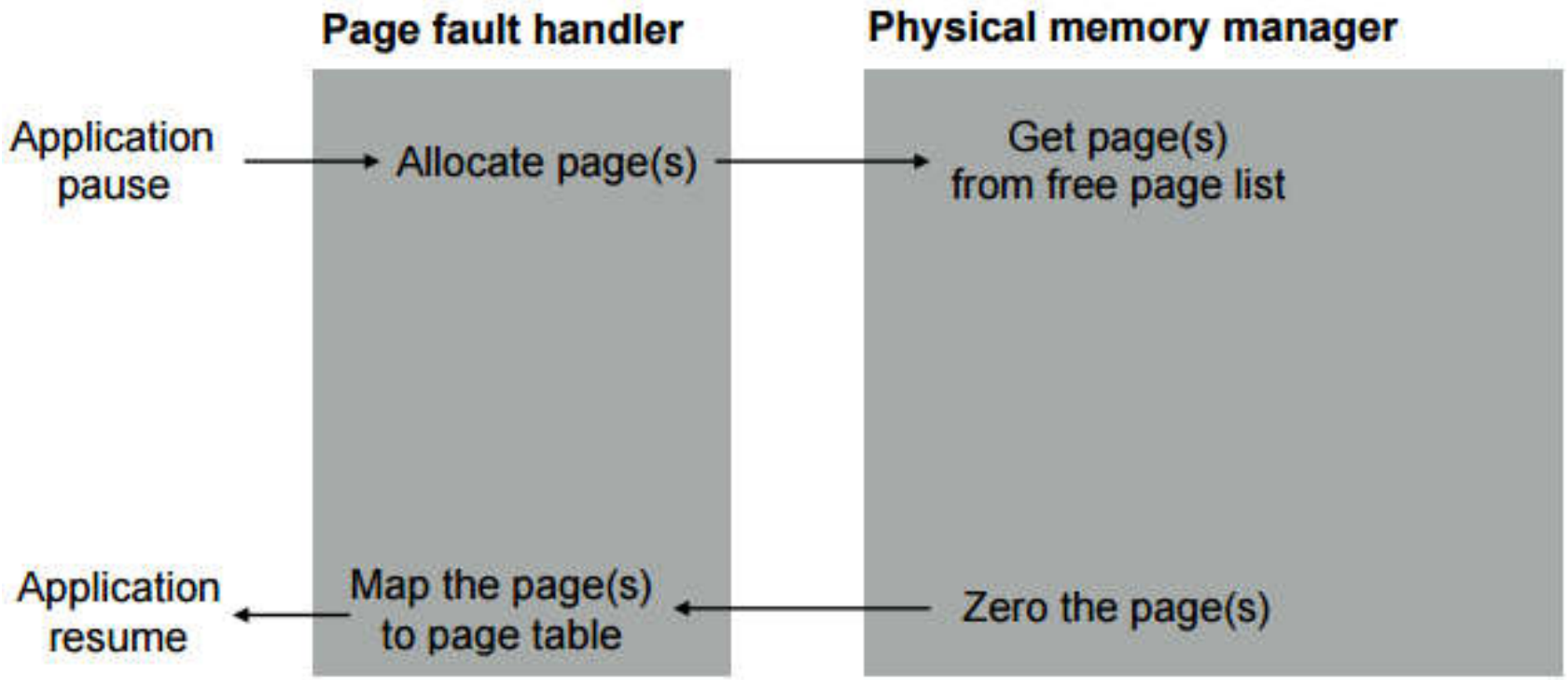
- **Huge page allocation increases page fault latency**
  - Page allocation path of both huge and base page



Page fault latency
- 4KB page : 3.6 us
- 2MB page : 378.0 us (mostly from page zeroing)
- Increases tail latency

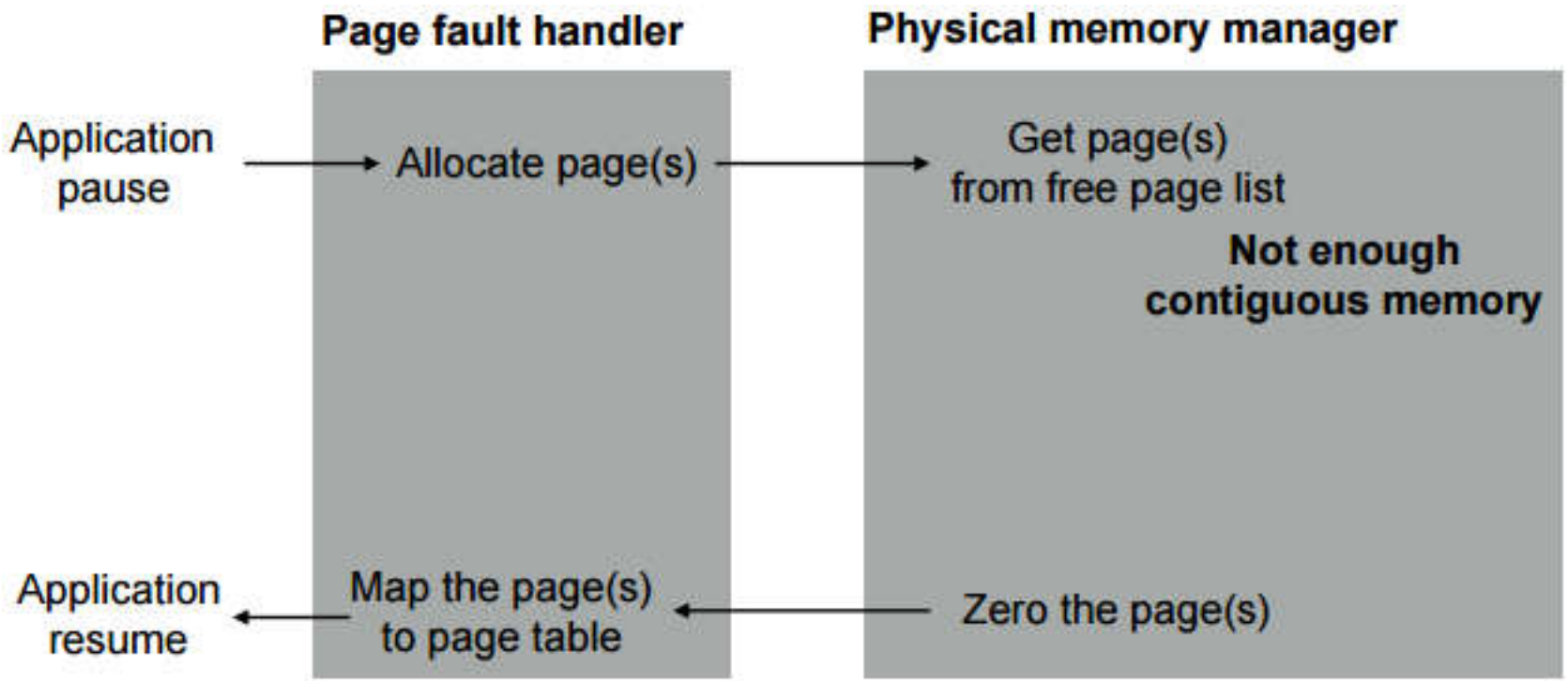
Courtesy Youngjin Kwon et al.

- **Huge page allocation might require extra memory copying**
  - Page allocation path of huge page



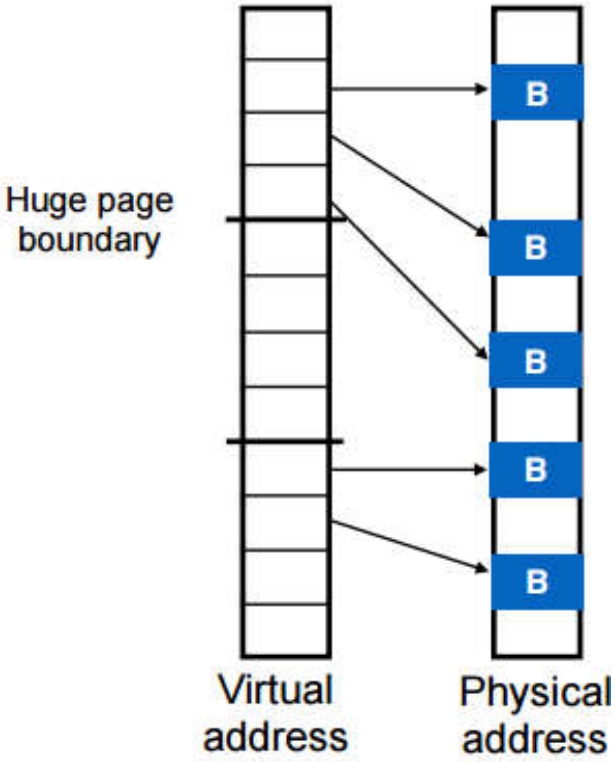
Courtesy Youngjin Kwon et al.

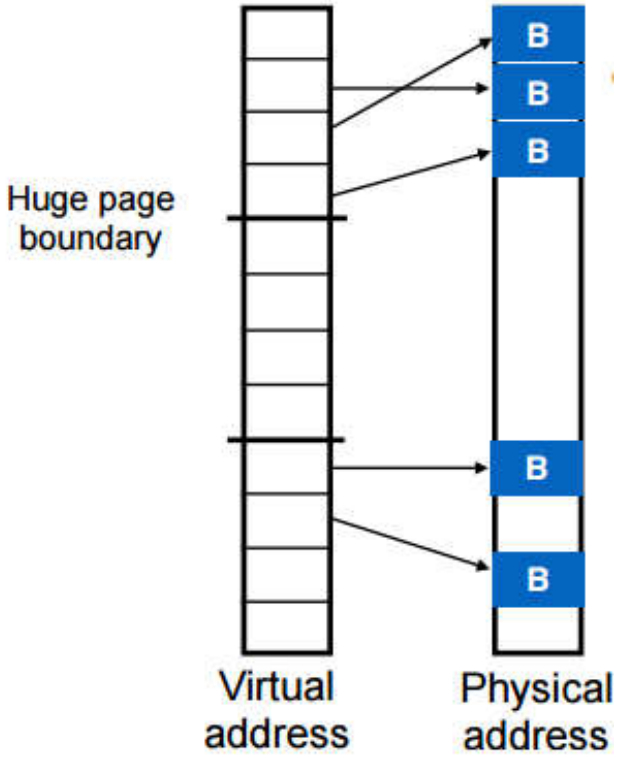
- **Huge page allocation might require extra memory copying**
  - Page allocation path of huge page

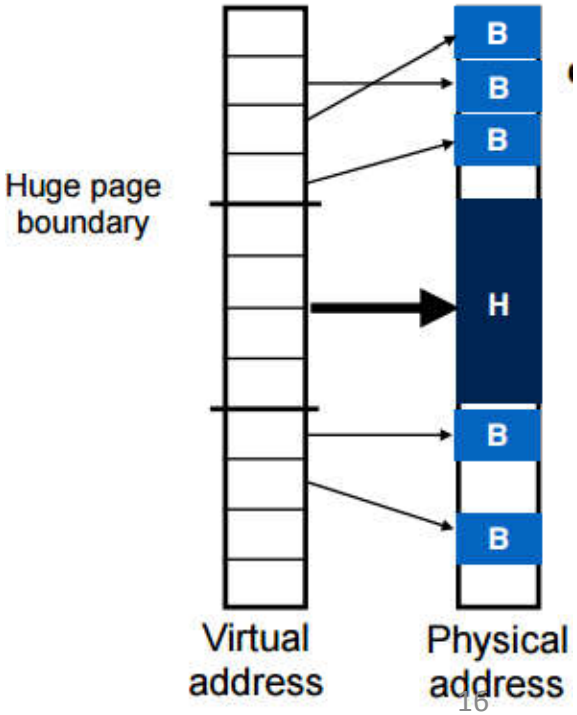


Courtesy Youngjin Kwon et al.

- **External Fragmentation**
  - Not enough contiguous memory
- As system ages, physical memory is gradually fragmented
  - 2 minutes to fragment 24GB of memory
- Memory fragmentation happens in any size of memory and TLB
- Linux compacts the physical memory to create contiguous memory





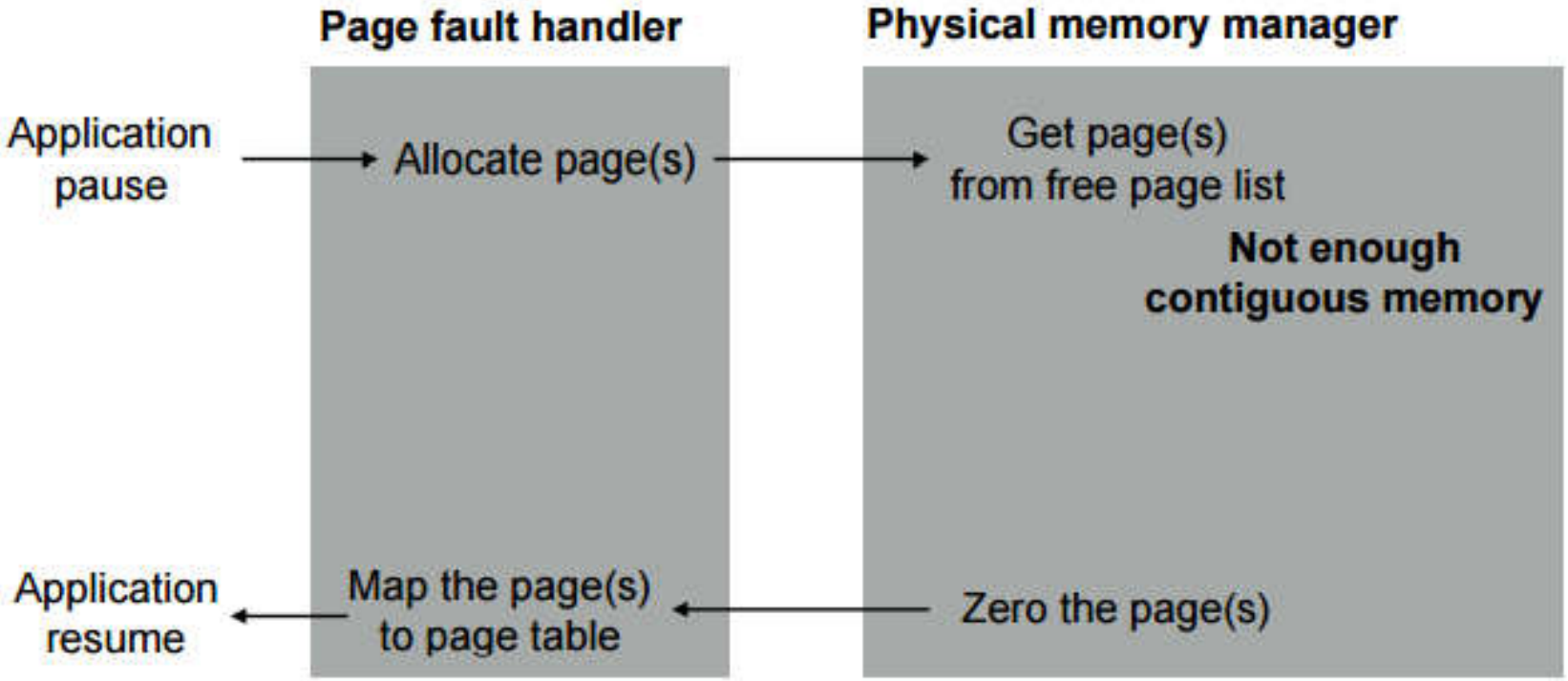


B Allocated Base page

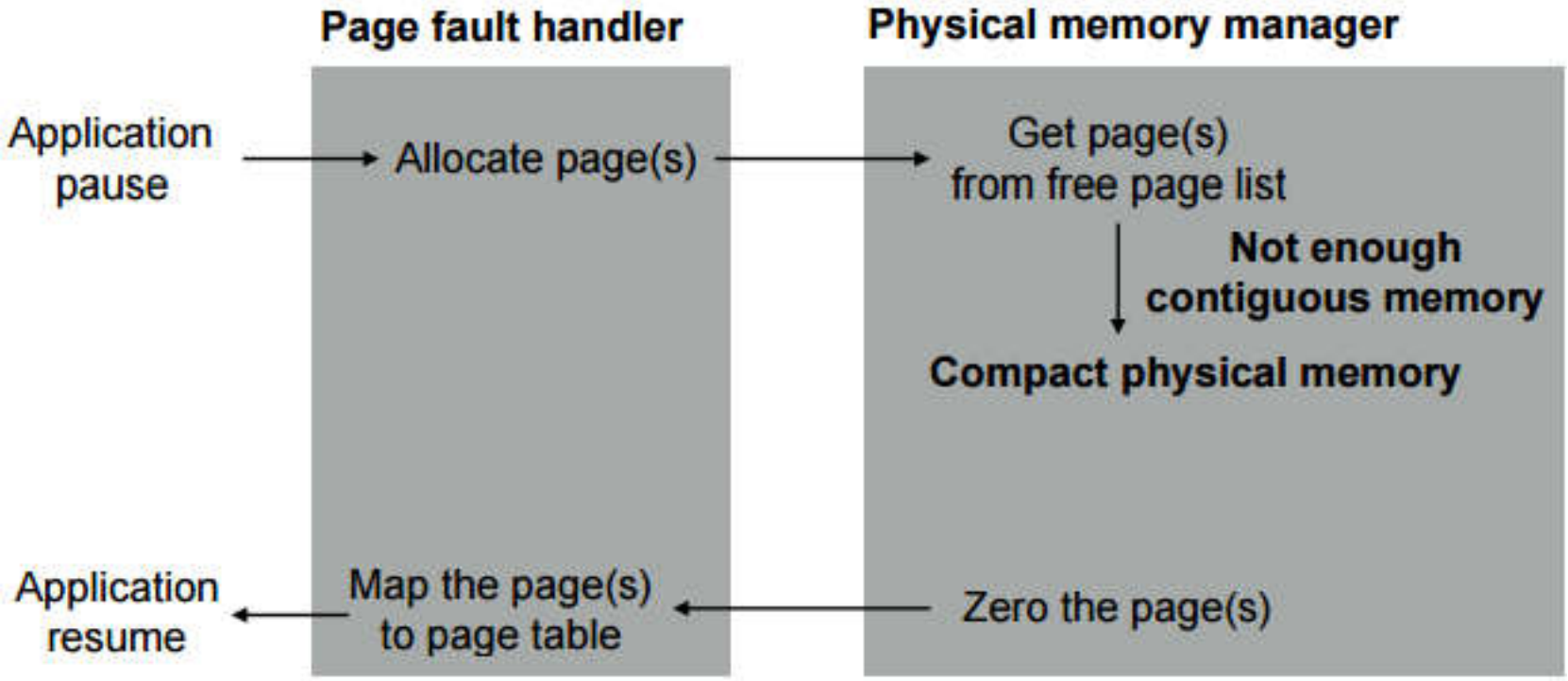
Courtesy Youngjin Kwon et al.

- **Huge page allocation might require extra memory copying**
- Although compaction is a very cool feature however should know that compaction happens page fault handlers
- When system know there is not enough contiguous memory, it compacts memory
- Compaction may or may not succeed, zero out the pages and return to the page fault handler
- Compaction adds unpredictable latency to the page fault handler

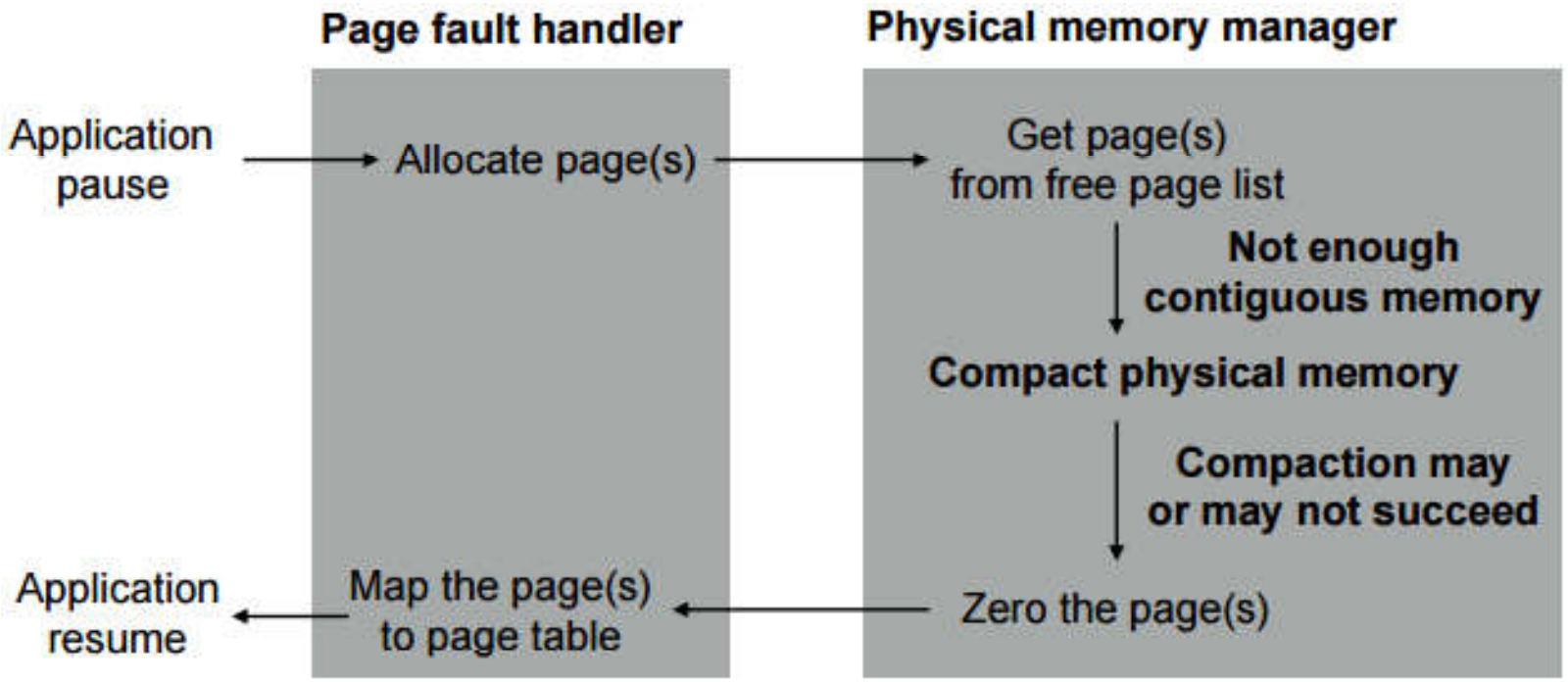
- Page allocation path of huge memory includes memory compaction



Courtesy Youngjin Kwon et al.

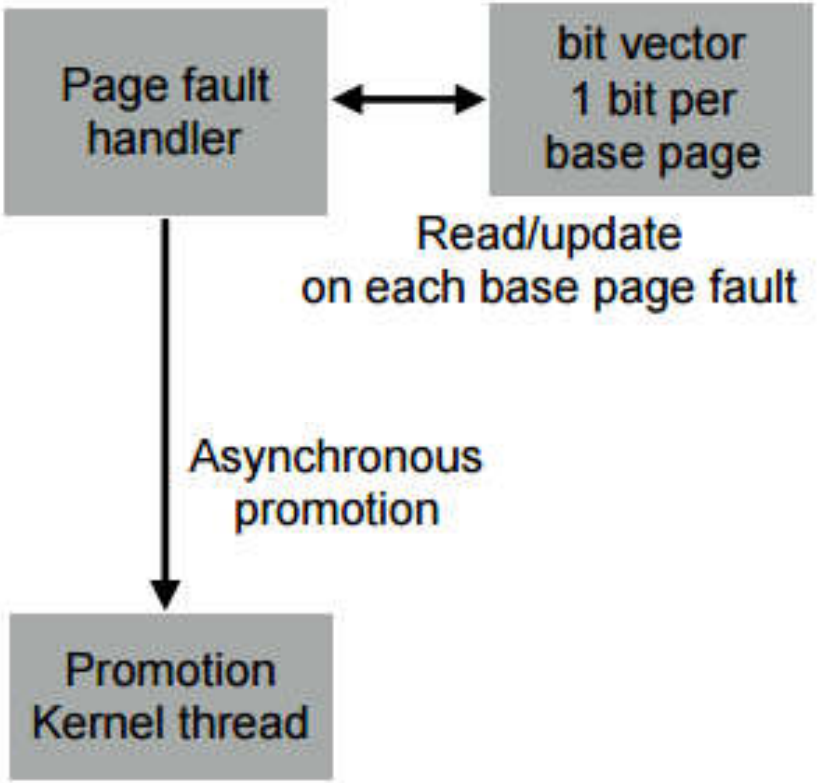


Courtesy Youngjin Kwon et al.



Courtesy Youngjin Kwon et al.

- Ingens solve this problem by **asynchronous allocation**
- Page fault handler allocates only base pages
- Use base page fault to track the allocation in the bit vector
- Lookup in the bit vector when need to promote huge pages
- Make asynchronous promotion request to kernel thread
- Kernel thread allocates
  - Huge pages in background
  - Memory compaction in background
- Ingens completely remove huge page allocation from the page fault handler
- Does not suffer from high page zeroing latency and memory compaction latency



Fast page fault handling

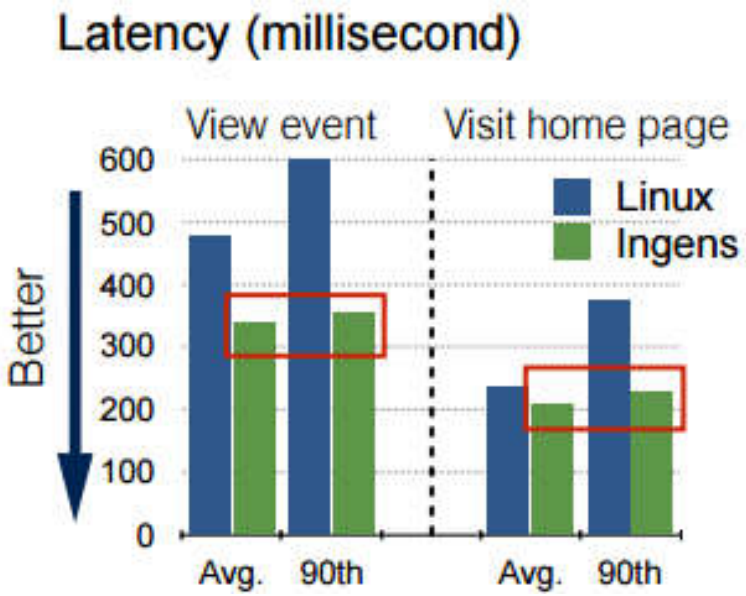
Courtesy Youngjin Kwon et al.

- **Experimental evaluation of asynchronous allocation**
- Machine specifications
  - Two intel Xeon E5-2640 2.60GHz CPUs
  - 64GB memory and two 250MB SSDs
- Cloudstone workload
  - Web service mimic in the social event planning
  - Work with nginx/PHP/MySQL
  - 85% read, 10% login and 5% write workload operations
- 2 of 7 web pages modified to use modern web page sizes (average webpage size is 2.1MB)

- Initially fragmented the memory
- For throughput, Ingens performance is 18% better than Linux
- For the latency, Ingens reduce 30% latency & 40% of tail latency

Throughput (requests/s)

Linux	Ingens
922.3	1091.9 (+18%)

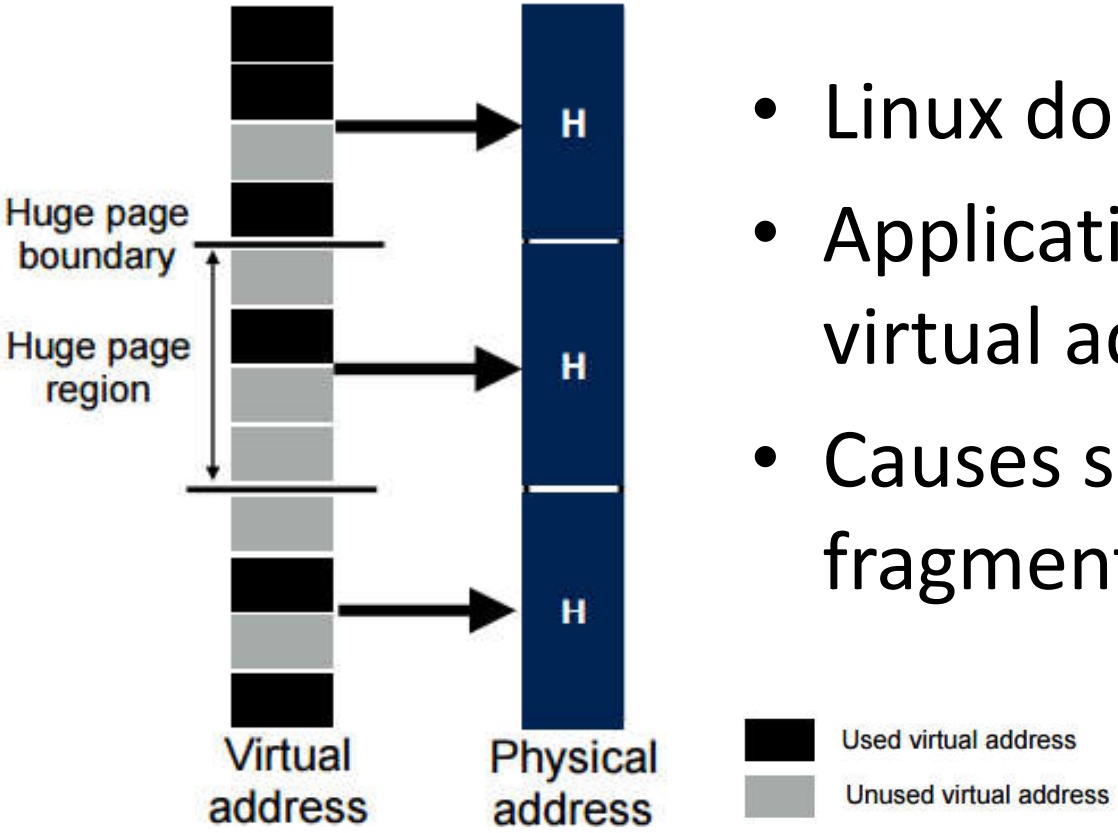


- Linux page fault handler performs huge amount of memory compaction but Ingens don't
- Therefore, gape between the average latency and tail latency is very narrow in Inges

Courtesy Youngjin Kwon et al.

2nd

- When applications occupy more memory than it uses causes **Memory Bloating**
- Happens because of Internal Fragmentation



- Linux do greedy allocation
- Applications sparsely use virtual address space
- Causes severe internal fragmentation

Courtesy Youngjin Kwon et al.

- To **check internal fragmentation in real application**, two experiments conducted
- Redis
  - Deleted 70% objects after populating 8KB objects
- MongoDB
  - Made 15million GET request for 1KB objects with the popular YCSB generator

- Redis
  - Used 70% or more memory using huge pages
- MongoD
  - Used 23% more memory using huge pages

Physical memory consumption

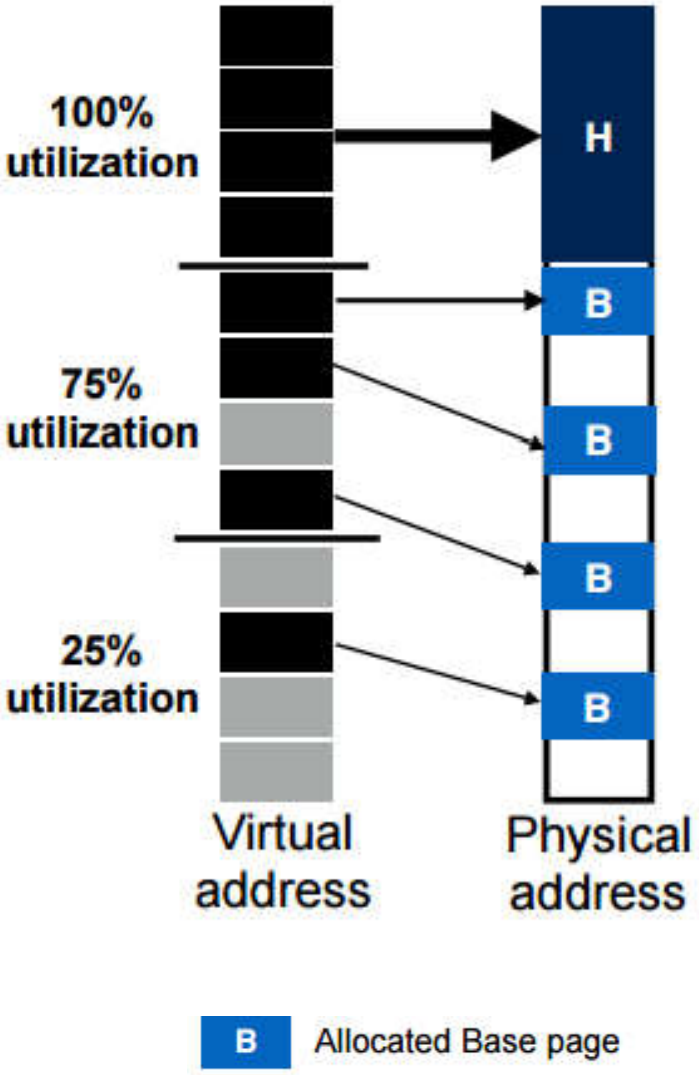
	Using huge page	Using only base page
Redis	20.7GB (+69%)	12.2GB
MongoDB	12.4GB (+23%)	10.1GB

- Bloating makes memory consumption unpredictable
- Memory intensive application can not provision to avoid swap

Courtesy Youngjin Kwon et al.

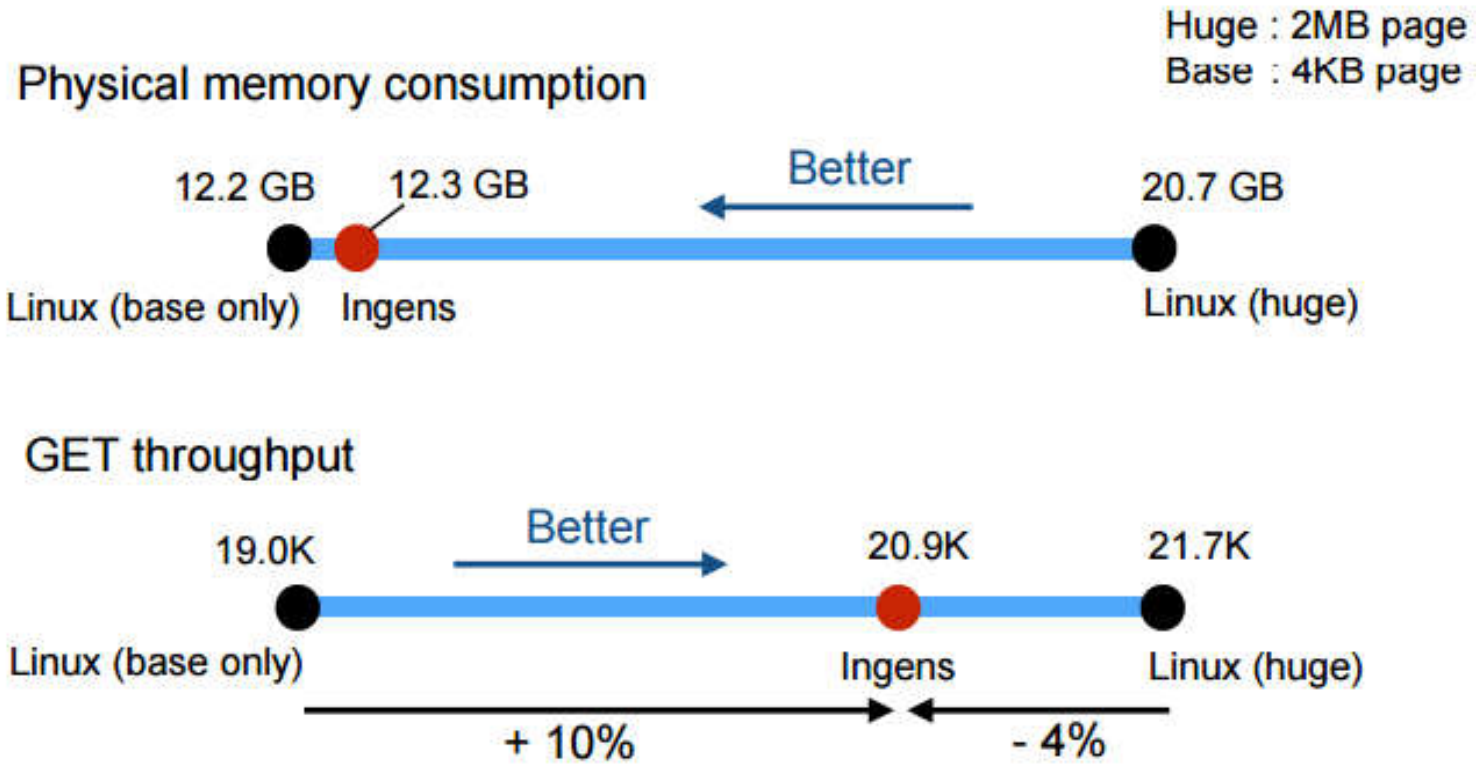
- Ingens do **Spatial utilization based allocations**
- Page fault handler track the utilization of each huge page region
- If utilization is beyond some threshold, makes asynchronous request to kernel thread
- Utilization based promotion provide bound size of internal fragmentation
  - If threshold is 90% than memory bloating size is of 10% at worst cases

- Spatial based allocation comes at the cost that it can lose some chance to allocate some huge pages, compared to greedy allocation
- There is a trade-off between memory consumption and performance



Courtesy Youngjin Kwon et al.

- Experiment to evaluate memory consumption and performance trade off



- Ingens provides good balancing between the memory consumption and performance

Courtesy Youngjin Kwon et al.

- Ingens **Overhead** comes from two facts
  - It has some slightly time delay due to the asynchronous allocation
  - Utilization based promotion, Ingens lose some chance to promote huge pages

Overhead for memory intensive application

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

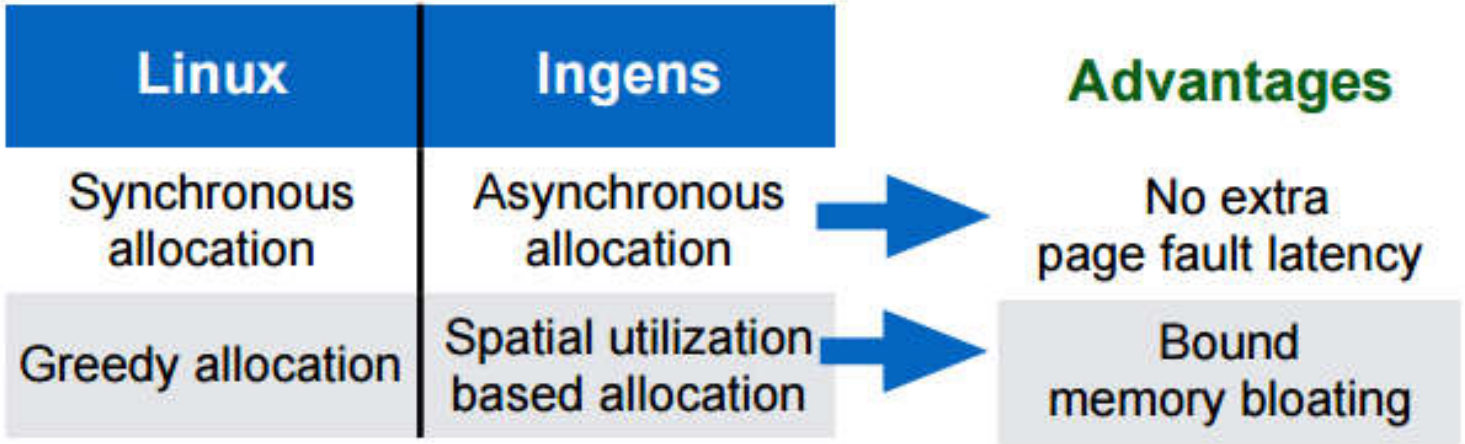
Overhead for non-memory intensive application

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

Ingens overhead is negligible

Courtesy Youngjin Kwon et al.

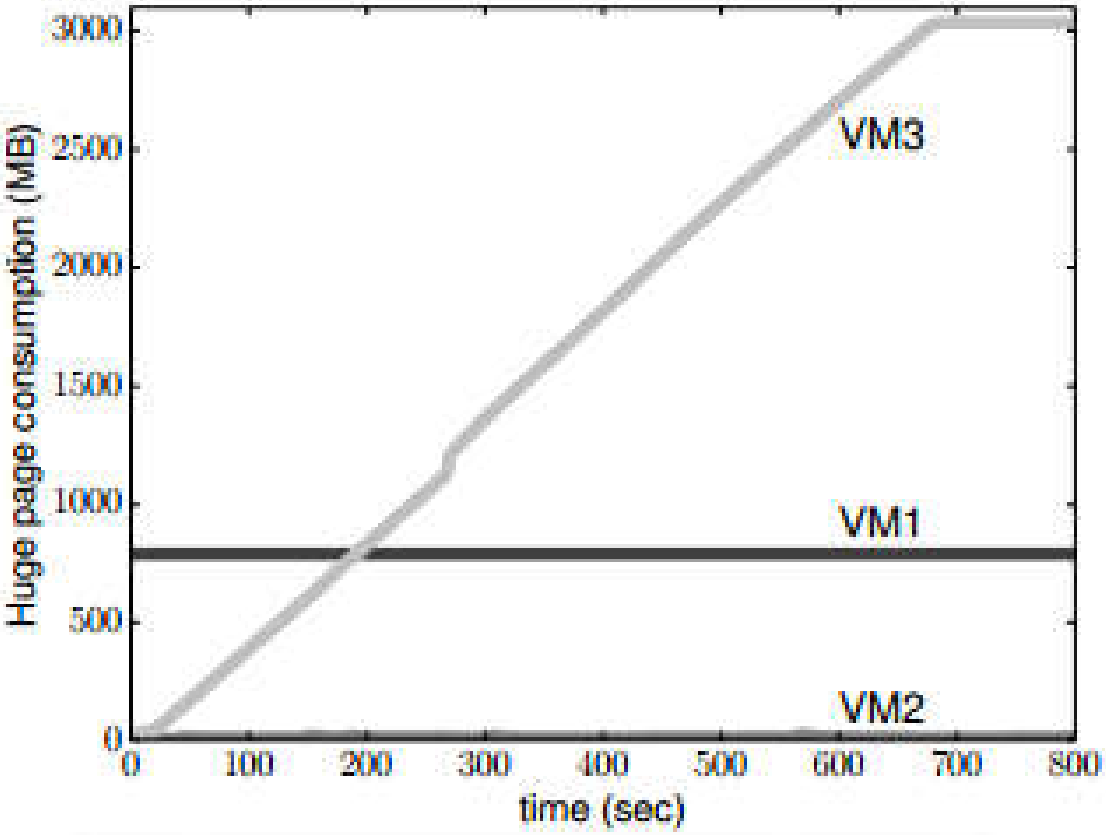
- Up to this point,
  - Ingens has a very simple goal
    - Make the huge pages widely used in practice
  - Provide a simple and practical solution



Courtesy Youngjin Kwon et al.

3rd

- **Unfair huge page promotion**
- Linux greedy allocation of huge pages is unfair
- Cause large and persistent performance variation across identical processes or VMs
- to demonstrate this problem
  - Ran four VMs in a setting where memory is initially fragmented
  - Each VM uses 8GB of memory
  - VM0 starts first and obtains all available huge pages(3GB)
  - Later VM1 starts and begin allocating memory, during which VM2 and VM3 starts
  - VM0 than terminates releasing 3GB of huge pages
  - Measured how Linux redistributes that contiguity to the remaining identical VMs

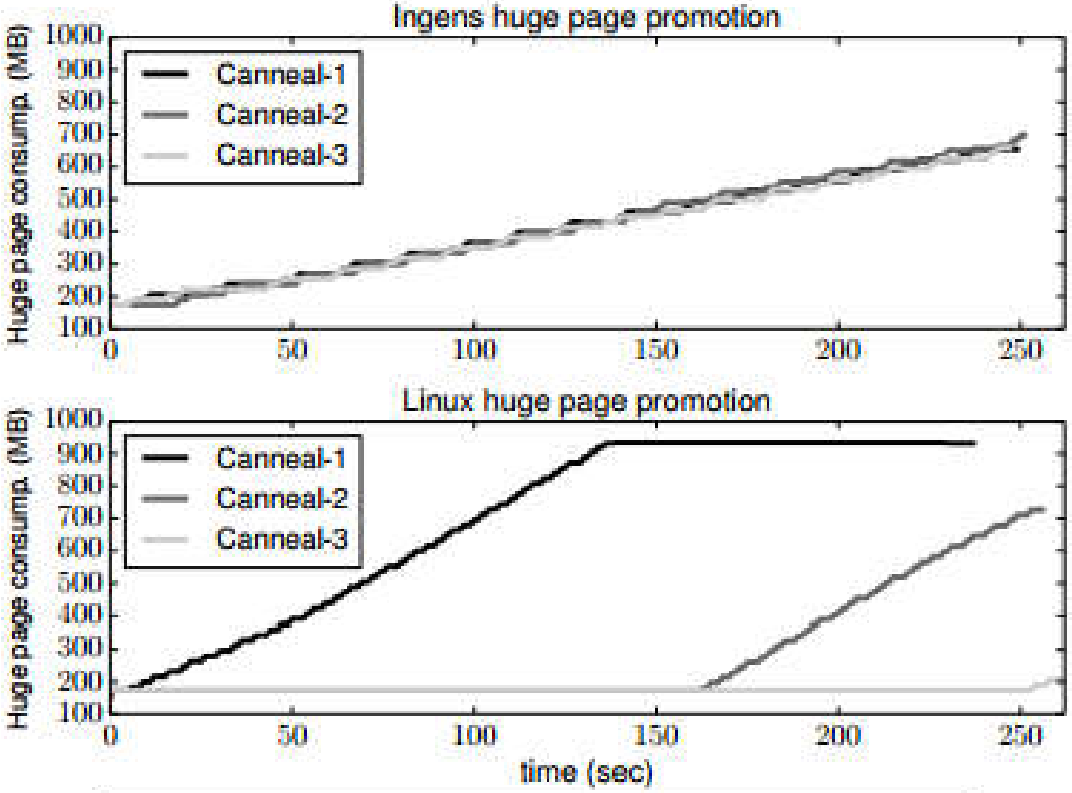


SVM	VM1	VM2	VM3
Exec. time (sec)	533 (1.12×)	589 (1.24×)	475

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.

- While **Ingens monitors and distributes memory contiguity fairly** among processes and VMs
- Employ techniques for proportional fair sharing of memory with idleness penalty
- Each process has a share priority for memory that begins at an arbitrary but standard value
- Ingens allocates huge pages in proportion to the share value
- Ingens counts frequently accessed pages as idle memory and improves a penalty for the idle memory
- An application that has received many huge pages but is not using them actively does not get more

- **Experimental evaluation to find fairness**
- Ran a set of three identical applications concurrently with the same share priority and idleness parameter
- Measure the amount of huge pages each one hold at any point in time



	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.

4th

- **High-performance memory savings**
- Services that reduce memory consumptions, kernel same page merging(KSM), can prevent a VM from using huge pages
- Modern hypervisors detect and share memory pages from different VMs where contents are identical
- The ability to share identical memory reduces the memory consumed by guest VMs
- Two policies are used

- In **KVM**
  - Identical page sharing in the host is done transparently in units of base pages
  - If, Contents of a base page are duplicated in a different VM-Duplicated page is contained with in a huge page
  - KVM split huge page into base page to enable sharing
  - Prioritizes reducing memory foot print over preservation of huge pages, penalizes performance
- **Huge page sharing policy** would not split huge pages
  - A base page is not allowed to share pages belonging to huge page but can share base pages
  - Huge pages are allowed to share only huge pages

- Implementation of huge page sharing to compare with KVM

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.

- Experiment to evaluate memory saving and performance trade off of identical page sharing
  - Ran a workload mix of three different applications, in there own VMs
  - Three different OS configurations
    - KVM with aggressive page sharing
    - KVM with huge page sharing
    - Ingens, where only infrequently used pages are demoted for page sharing

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.

Thank You

Questions?