Difference Engine: Harnessing Memory Redundancy in Virtual Machines

General

- What is the main hypothesis underlying the paper?
The main hypothesis is that significant additional gains can be achieved by leveraging both sub-page level sharing (through page patching) and in-core memory compression.

- Do the authors show that the hypothesis is true?
The authors prove their hypothesis by depicting the benefits of using sub-page sharing in Table 1. They show that an additional 50% of savings can be achieved over and above the savings obtained from full page sharing alone. Moreover additional pages can be saved by using page compression.

Architecture
- Content-based sharing:
Similar to VMWare approach. To find sharing candidates, it hashes contents of each page and uses hash collisions to identify potential duplicates. To guard against false collisions, it performs a byte by byte comparison before actually sharing the page.

- Patching
Patching is a technique that difference engine uses to reduce the memory required to store similar pages. A patch is defined as a difference of a page relative to a reference page. Using combination of patch and reference, the actual page can be reconstructed.

How much memory can a patch of size X save?
Assuming a page size of S, a patch of size X could save (S-X), where X is less than half of page size S.

- Study of sharing (Table 1): What does this table show us?
The table elucidates the performance benefits of sharing and patching by showing that 50% of savings can be obtained by full page sharing alone and a total of 77% savings can be achieved by full page sharing and patching.

- What is a HashSimilarityDetector(k,s)? What is k? What is s?
What does Figure 2 show us?
HashSimilarityDetector(k,s) is parameterized scheme to identify similar pages by hashing the contents of (k*s) 64 byte blocks at randomly chosen locations on the page and then grouping these hashes into k groups of s hashes.
'k' is the number of times a page will be indexed. Higher 'k' captures global similarity.
's' is the number of locations in the page that will be indexed. Higher 's' captures local similarity.
- How does compression work?

1) Identify candidate pages for compression using global clock algorithm like pages which haven't been accessed in the past few scans.
2) Compress pages using LZO/WKdm algorithms.

3) Invalidate the compressed pages and store in dynamically allocated storage area in machine memory.
4) Upon an access of a compressed page by a VM, uncompress and return to the VM.

- *What type of page needs to be identified in order to compress it?*
Pages that haven't been accessed for an extended period of time by the VMs needs to be identified.
- For compression to work effectively, what must be true about compression performance vs. speed of disks?
The compression performance should be smaller than the speed of disks. On the other hand if compression-uncompression latency is higher than disk speed, then storing in disk would save full space for the page while also being faster to access the page.

- *Overall, what is the main difference between a read to a full-page that is shared vs. a read to a patch or compressed page?*

Read to full-page that is shared	Read a patch or compressed page
Can directly give the page stored in machine memory to VM since the whole page is shared.	Can't directly give the machine memory page as whole page isn't stored. In the case of patch and compression, the required page is reconstructed and then given to VM.

- *Why do people use the word "Architecture" instead of "Design"?*
People say "*Architecture*" because it sounds fancier.

Implementation

What is ioemu in Xen?
ioemu is a user space I/O emulation process. It executes in Domain-0 and hence can map any page from guest OS in its address space. For simplicity, ioemu maps the entire memory of the guest into its address space.

Why this is a problem?
Since ioemu maps every VM's address space into Domain0, it prevents difference engine from saving any memory.

Solution?
Authors modify ioemu implementation to -
- Limit ioemu to map small, fixed number of pages from each VM at any given point of time.
- Since limiting pages can cause ioemu to constantly map VM pages into its address page, Dynamic aging mechanism is implemented in ioemu, which does not un-map the page immediately.

How is clock used?
- Global clock

- Not Recently Used (NRU) clock used to select candidate pages
- Referenced and Modified bits used to keep track of page usage
- In the presence of multiple VMs, small portion of memory from each VM is scanned to provide fairness

Page State - Classification and usage by Diff Engine -
C1 - Recently Modified - Not Considered for sharing or compression
C2 - Accessed but not modified pages - Considered for sharing and to be reference pages but cannot be patched or compressed themselves
C3 - Not recently accessed - Candidate pages for sharing or patching
C4 - Not accessed for extended period - Considered for everything, including compression and swapping

Memory limit in Xen
- Hash tables stored in Heap area of Xen, which is limited to 12 MB. The size is not sufficient to store hash table with entries for complete physical memory. Hence memory is split in 5 regions, with hash table storing entries for 1/5th of physical memory only. This means that a complete cycle of five passes covering all regions is required to identify all identical pages.

How many Hash tables?
Page Similarity Hash table - To store hash value for similar pages
Page Sharing Hash Table - To store hash value for identical pages

Page Compression
- Compressed pages cannot be used as reference pages. Hence compressing too many pages would reduce the possibility of patching.

Swapping Implementation
- Handled outside Xen, by a single swap daemon (swapd) process running in Domain0
- swapd takes swap-out and swap-in requests from Xen and other processes
- Treats swap-out requests as hints and not obligatory command, i.e, if for reasons swapping is not possible, swapd continues silently.
- Since a separate process in Domain0 handles swapping, race condition could occur when Xen initializes the swap-in process and the page requesting process access it before swapd could bring the page back.

Difference from ESX Server -
In ESX server, the VMM itself handles the page swapping process and hence the problems associated with swapd process are absent in its context.

Evaluation
- Table 2: What is shown? What would also be nice to see?
The table shows the mean execution times of commonly used functions like share_pages, cow_break, etc.. in Difference Engine. It didn't give comparison with any hypervisor which doesn't use sub-page level sharing or compression like ESX. It would be nice to see the overhead introduced by the novel features in Difference Engine.

- *Figure 2:*
The figure shows the effectiveness of the hash similarity detector(k,s) for different values of indices, index length and number of candidates. We get better savings for (2,1),1 and (4,1),1
Factors to consider in identifying similar pages.

a) Finding positions of similarities.
b) Overhead of finding similarities.

Difference Engine's hashsimilarity detector won't detect similarities with shifts

- *Figure 4: What is the time-between-refs for most pages?*
Lifetime – the time between when the page was patched/compressed and the time of first subsequent access(read or write).
From the figure, very few percentage of pages were accessed in the first 100 ms. Nearly 50% of pages weren't accessed for 10,000 ms (100 seconds). This is a great win for compression and patching.

- *Figure 5: What is being compared? What do we learn?*
In this figure, the memory savings during a workload with identical pages by different mechanisms like sharing, patching and compression are compared.
All 3 mechanisms exhibit similar gains for identical pages.

- *Figure 8: What technique matters the most here? Why?*
With homogeneous VMs, there will be lots of shared pages because of OS libraries. Hence sharing matters the most here.

- *Figures 9-11: What benefits does DiffEngine achieve?*
What kind of performance differences are seen between
ESX and DiffEngine?
Difference Engine is more aggressive, because it does full scan of memory, while ESX does only random page similarity.

- *What does Table 3 show? What doesn't it show?*
The table shows that the applications' performance under Difference Engine for heterogeneous workloads is within 7% of the baseline.
It didn't give head to head comparison with ESX.
An interesting thing to note that is that the performance overhead of Vim, Compile, lmbench is 13.22%, but a line at the end of Section 5.4.2 says "In all cases, performance overhead of Difference Engine is within 7% of the baseline".

- *Figure 12: What are the authors trying to show here?*
The authors are trying to prove that it is beneficial to add extra VMs from the memory saved by difference engine. In the graph, they show that additional VMs help in reducing the response time. However beyond a certain point, overhead of managing the extra VMs begins to offset the performance benefits.

- *Overall: Can aggressive swapping be used as an alternative to all of this machinery?*

It can be used as an alternative if the cost of swapping is considerably reduced as in SSDs.