Catalyst: GPU-assisted rapid memory deduplication in virtualization environments

Anshuj Garg, Debadatta Mishra and Purushottam Kulkarni
Department of Computer Science and Engineering
Indian Institute of Technology Bombay
{anshujgarg, deba, puru}@cse.iitb.ac.in

Abstract
Content based page sharing techniques improve memory efficiency in virtualized systems by identifying and merging identical pages. Kernel Same-page Merging (KSM), a Linux kernel utility for page sharing, sequentially scans memory pages of virtual machines to deduplicate pages. Sequential scanning of pages has several undesirable side effects—wasted CPU cycles when no sharing opportunities exist, and rate of discovery of sharing being dependent on the scanning rate and corresponding CPU availability. In this work, we exploit presence of GPUs on modern systems to enable rapid memory sharing through targeted scanning of pages. Our solution, Catalyst, works in two phases, the first where pages of virtual machines are processed by the GPU to identify likely pages for sharing and a second phase that performs page-level similarity checks on a targeted set of sharable pages. Opportunistic usage of the GPU to produce sharing hints enables rapid and low-overhead duplicate detection, and sharing of memory pages in virtualization environments. We evaluate Catalyst against various benchmarks and workloads to demonstrate that Catalyst can achieve higher memory sharing in lesser time compared to different scan rate configurations of KSM, at lower or comparable compute costs.

Keywords Memory Deduplication, Graphics Processing Units, Virtualization, KSM;

1. Introduction
Resource over-commitment is a sought-after strategy in virtualization-based infrastructure-as-a-service platforms [37, 38]. The basic premise of over-commitment being exploitation of non-overlapping peak resource demands and to exploit policy-driven performance vs. resource allocation trade-offs. Along the memory management dimension, the main techniques to enable over-commitment are—demand paging [37], ballooning [13, 37], deduplication [3, 4, 37] and hypervisor-based disk page caching [15, 18]. Each of these techniques have different cost-benefit tradeoffs.

Hypervisor-based demand paging [37] is non-intrusive, requires no cooperation from guest operating systems but is hampered by the lack of knowledge regarding utility of pages for the guest OS. Memory ballooning [13, 37] while addressing this semantic gap relies on guest cooperation. Hypervisor-based disk caching [15, 18] aims to explicitly manage disk caches and manage them centrally to increase memory efficiency by employing deduplication and compression across disk pages. This technique is also intrusive and is restricted to disk caches. System-wide memory deduplication—scan and merge same pages across VMs and the host [3, 17, 37] is non-intrusive and can be effectively combined with the ballooning and hypervisor caching techniques.

Content-based deduplication techniques can be broadly classified into two categories: in-band sharing and out-of-band sharing [25]. In-band techniques [17] intercept the I/O path and deduplicate identical pages in I/O buffers and page caches. In contrast, out-of-band techniques [3, 37] are instantiated as a separate execution entity that periodically scans memory allocated to virtual machines for deduplication opportunities. Out-of-band techniques are more effective than in-band techniques as they can consider all the pages of virtual machines for merging and not the pages only in the I/O path. As part of this work, we focus on improving the efficiency of the out-of-band deduplication process.

The sharing efficiency of out-of-band techniques largely depends upon the time interval between successive scans and number of pages scanned in each interval. Share-meter [25] demonstrated the effect of changing the scan interval and pages-to-scan (in each scan) parameters of the Kernel Samepage Merging (KSM) feature of the Linux kernel. KSM scans pages in sequential order over the virtual...
address space associated with a virtual machine (virtual machines are processes in the Linux+KVM virtualization solution). KSM’s procedure suffers from two potential problems. First, if there is no or very low sharing potential in the system, it takes a full scan of all virtual addresses across VMs for this discovery. Clearly, the CPU cycles spent are not proportionate to the sharing benefits. Second, since KSM scans virtual addresses sequentially, the latency to capture the sharing potential can be high if identical pages are scattered or lie closer to the end of the virtual address space. For memory-oriented workloads this can adversely affect sharing potential. The longer it takes to deduplicate a shareable page, sharing is short-lived or potential pages are missed.

Towards designing an efficient out-of-band sharing process, the wish-list of requirements is as follows,

(i) Not to waste CPU cycles when no or low sharing opportunity exists in the system.
(ii) Detect sharing potential as quickly as possible and maintain a high ratio of shared pages to shareable pages.
(iii) Reduce overhead of detecting opportunities for sharing and merging of identical pages.

Current out-of-band sharing techniques [3, 16, 37] do not simultaneously meet all these requirements. Our central idea is to shift compute requirements of the sharing process off the CPU. A possible approach to minimize CPU demand for the sharing related processes is to opportunistically use the CPU for aggressive scanning during low utilization periods. We push this idea to the extreme by offloading almost all the sharing related compute requirements independent of the CPU utilization levels. This is especially relevant for systems executing CPU-intensive workloads, where interference due to computation requirements for deduplication can be non-trivial [3].

Most modern systems have specialized co-processors—Graphics Processing Units (GPUs), Floating Point Units (FPUs) etc. These co-processors are designed to exclusively solve specific types of computing tasks in an efficient manner. Using GPUs for general purpose tasks which would typically be executed by the CPU have also been explored [33, 36, 39]. We aim to exploit the complementary computational capabilities of GPUs to offload compute requirements of the sharing process and simultaneously meet the three goals described above. With a GPU-assisted setup, the CPU can perform only the essential tasks of sharing (setting up the shared memory areas etc.) and the groundwork for deduplication (identifying redundant pages) can be handled by the GPU. Further, GPU-based computations will also benefit from the large set of cores for rapid identification of sharing potential.

As part of this work, we design and implement Catalyst, a GPU-assisted system for efficient memory deduplication. Catalyst enables rapid detection of sharing potential, and allows the CPU to perform targeted scan-and-merge of pages that are highly likely to be deduplicated. The main challenges of building such a system are to engineer a low-overhead mechanism for copying data from the kernel space to the GPU and integration with current deduplication processes. We demonstrate the efficacy of Catalyst by comparing it with different configurations of the vanilla KSM process.

2. Background and Motivation

2.1 CPU efficiency and KSM

Kernel Same-page Merging (KSM) [3] is a kernel feature that scans through virtual address ranges of memory allocated to processes and deduplicates identical pages. In the context of the Linux+KVM virtualization solution, where each virtual machine’s instantiation is essentially a process, KSM can be used without modifications to deduplicate pages of virtual machines. KSM uses a scan-and-merge approach—scan pages in a sequential manner and merge when identical. A hash for each page is computed and compared against other pages for similarity. If a match is found, KSM performs a byte-by-byte comparison of the pages to ascertain similarity. Similar pages are merged, both virtual addresses are mapped to the same physical page and the mappings to the physical page are marked copy-on-write (CoW). The CoW semantics is required to handle writes to shared pages.

The sequential scanning approach of KSM has to scan all pages, even those that are not shareable, to exploit the sharing potential across all virtual machines. In cases with low sharing potential, the computation cost is disproportionate relative to the sharing benefits.

Figure 1 shows the percentage of all CPU cycles used for scanning that resulted in no sharing benefits. For the
experiment, the setup consisted of a single VM which read a single file of size 1 GB. The contents of the file were configured to exhibit deterministic sharing potential—20%, 40%, 60% and 80%. On reading the file, we measured the number of CPU cycles KSM spent on scanning pages that were shared and not shared.

Referring to Figure 1, the x-axis represents the setups with different sharing levels and y-axis represents the number of cycles wasted on scanning non-sharable pages. As can be seen, with less sharing potential most of the CPU cycles are wasted in scanning non-sharable pages—72% CPU cycles yielded no sharing benefits. As the sharing potential goes up, percentage of CPU cycles wasted decreases. In an ideal situation the CPU cycles should only be used for scanning sharable pages and never be wasted, or at least not wasted to a large degree. As the sharing potential is not known a priori, achieving this goal requires additional inputs or pre-processing before the KSM process takes over.

Vanilla KSM not only wastes cycles in scanning non-sharable pages but also can take considerable amount of time to realize the maximum sharing potential. The time taken by KSM to realize the maximum sharing potential depends on the number of pages scanned per second and the distribution of identical pages in the set of all pages.

Figure 2 shows the result of an experiment that reports the cumulative distribution of the time taken to realize the sharing potential with different spatial distribution of identical pages. Distance 0 indicates all identical pages are contiguous, Distance 1 indicates a gap of one (non-identical) page between any two identical pages and so on. The experiment setup was identical to the previous experiment which estimated the wastage of CPU cycles, except the manner in which degree of sharing was instantiated in the file to be read. Figure 2 reports results from a file with 20% sharing potential and a KSM scan rate of 4000 pages per second. As can be seen from the Figure, when sharing is spatially closeby, the time to achieve 100% sharing potential is much quicker than when the spread of identical content is more. With Distance 3, KSM requires about 70 seconds to achieve the sharing potential, where as it requires only 30 seconds with Distance 0. Similar to the first experiment, since there is no a priori knowledge regarding the spatial distribution of the shareable pages, time required to reach the sharing potential can considerably vary for different distributions of shareable content. Further, the time to achieve that sharing potential is also inversely proportional to the scan rate — low rates results in large time intervals.

In our work we try to address the above problems with KSM and also explore the possibility of using the co-processors like GPUs to solve them. In absense of prior knowledge regarding the memory state, we aim to achieve a win-win situation of not using compute cycles more than that is required for sharing identical pages and achieve sharing potential rapidly.

2.2 Freeriding the GPU

The central idea of Catalyst is to exploit the presence of a GPU to enable rapid and efficient deduplication of memory pages across VMs. Here we present a very brief overview of GPU characteristics and its relevance for the problem at hand.

A GPU is a highly parallel microprocessor specifically designed for fast and efficient processing of graphics and general purpose applications [33, 36, 39]. GPUs are connected via the Peripheral Component Interconnect (PCI) interface of a computer. Code and data has be transferred to the GPU via the PCI bus for execution. The asynchronous data transfer to GPU device memory is DMA-based. Application programmers have a restrictive view of the GPU. GPU APIs provide abstraction of thread group hierarchy and shared memory hierarchy to the programmer. Programmers write functions called kernels for execution on a GPU and additionally specify the number of threads for simultaneous execution.

Programs of SIMD nature (single instruction multiple data) are well suited for exploiting the parallel execution model of GPUs. Page hashing operations of KSM are also SIMD operations—same hashing operation performed on different pages. Hence the hashing and hash comparison operations can be executed in parallel and can be offloaded to a GPU.

To summarize, there are two main advantages of offloading KSM’s main compute operations of page content hashing and hash comparison to a GPU.

1. Hashing and hash comparison takes CPU cycles and is performed periodically. CPU is a shared resource in virtual environments and several VMs contend for the CPU. Offloading these tasks to a GPU will save CPU cycles.
2. Page hashing and comparison is SIMD in nature, can be executed in parallel and be potentially achieved in orders of magnitude faster if performed on a GPU.

3. Catalyst Design and Architecture

3.1 Strategy
Our approach is to offload the page hashing and hash comparisons of the KSM process to the GPU. The main tasks here are to transfer the content from pages of virtual machines to the GPU, and compute hashes of the page contents and compare them on GPU itself. The pages with same hash values are considered potential candidates for sharing. The output of GPU processing is a set of hints: the set of pages whose hash values are identical to one or more pages in the system. Hints are then used as an input by KSM for targeted scanning. Vanilla KSM wastes a lot of cycles in scanning non-sharable pages. Hints enable KSM to perform targeted scanning of only those pages which have a high probability of being shared.

3.2 Challenges
Towards designing such a system there exist two main challenges,

(i) Kernel does not have direct access to the GPU
GPU libraries like CUDA [22], OpenCL [14] etc., provide user-space APIs for programming the GPUs. These user-level library calls are directly interfaced with proprietary drivers to provide the desired functionality. Operating systems do not have access to manipulate the GPU access interface directly. In order to work around this constraint, an efficient mechanism is required to expose the GPU interface to the kernel.

(ii) Efficient transfer of data to the GPU
The physical memory (RAM) is not directly accessible to a GPU. In order to execute a program on a GPU, the data required by a program needs to be first transferred to the GPU’s memory. GPU programs use API libraries to transfer data to the GPU and operating systems do not have direct access to the GPU API libraries. A first-cut solution is to transfer data from kernel space to a user-space process, which then invokes the GPU APIs to transfer data to the GPU. This involves two data transfers—one from the kernel space to the user space and other from the user space to the GPU. It is very likely that the data transfer overheads might surpass the benefits of offloading kernel tasks to GPUs and hence the data transfer component needs careful design.

3.3 Architecture
Figure 3 shows the architecture of Catalyst. The two main components of our system are the Catalyst kernel module and a user-space Catalyst daemon. Together these components solve the challenges associated with availing GPU services for efficient and rapid memory sharing. The Catalyst kernel module is responsible for transfer of kernel data and control information to the Catalyst daemon. The user-space daemon sets up execution of page sharing related tasks on the GPU. Details of the two components are as follows,

- **Catalyst kernel module**
The Catalyst kernel module acts as a communication channel between KSM and the Catalyst daemon. The kernel module is responsible for transfer of kernel data to the user-space daemon and vice-versa. It provides the Catalyst daemon with the data contained in virtual machines’ pages and transfers the output of Catalyst daemon (hints) to KSM. It also avoids the double copy problem discussed in Section 3.2.

- **Catalyst daemon**
The Catalyst daemon is a user space process that executes periodically. It is responsible for transferring kernel data to the GPU and setting up execution of KSM’s offloaded tasks on the GPU. The Catalyst daemon maps pages of virtual machines with coordination from the Catalyst kernel module. It then provides these pages to the GPU, and invokes appropriate GPU functions to compute and compare hash values of virtual machine pages, thus generating hints. The Catalyst daemon receives output of computations on the GPU and forwards the hints to Catalyst kernel module. The Catalyst daemon executes periodically and can be configured using the num_sleep_seconds parameter. num_sleep_seconds specifies the interval after which the daemon requests the kernel module for the next round of pages to estimate the set of hints.

4. Implementation Details
Catalyst is implemented for the KVM hypervisor on Linux hosts enabled with the KSM page deduplication service. Figure 4 shows the high level control flow among the different components of Catalyst. The Catalyst daemon requests the Catalyst kernel module to acquire mappings of pages associated with VMs. The acquire mapping operation maps physical pages of a VM on to the Catalyst daemon virtual address space (CVA). Subsequently, the Catalyst daemon
transfers content of the VM pages (mapped on to the Catalyst daemon address space) to the GPU. After transferring page content data to the GPU, the Catalyst daemon invokes the necessary GPU functions to compute hints—prospective sharable pages. After computing hints, the Catalyst daemon requests the Catalyst kernel module to release mappings, which unmaps pages of a virtual machine from the Catalyst daemon’s virtual address space. The Catalyst daemon collects hints generated by the GPU and delivers them to KSM via the Catalyst kernel module. KSM then performs scan-and-merge operations on the provided hints. As soon as KSM is done with scanning the hints, it requests the Catalyst kernel module to notify the Catalyst daemon to be in readiness to repeat the same process. This entire procedure is termed as one round of Catalyst operations.

There are three main operations in our implementation which are explained in the rest of this section.

4.1 GPU Catalyst memory exchange

To facilitate targeted scan and merge of memory pages by KSM, GPU’s parallel hash computation capabilities can be leveraged. This necessitates transfer of pages of virtual machines to the GPU and collection of processed deduplication hints to assist KSM. Designing efficient high volume data exchange from main memory to GPU memory is a non-trivial exercise. To meet the above requirements, following are the major operations involved in the exchange of data between GPU and Catalyst.

4.1.1 Getting a handle on VM memory

As already mentioned earlier (Section 3.2), it takes multiple intermediate memory copies to transfer data from the operating system to GPU. Physical memory pages used by virtual machines are accessible only from within the privileged kernel mode. However, the NVIDIA GPU programming model allows only user space communication with the GPU; which becomes a bottleneck to transmit data in pages of virtual machines. Catalyst’s acquire mapping operation avoids the overheads due to the copy of data from kernel to user space. The Catalyst daemon allocates a large virtual address region with no backing of physical pages, this virtual address space allocation is termed CVA (the Catalyst daemon virtual address space). When deduplication hints for a VM is to be generated, the Catalyst kernel module is requested to update mappings of the CVA to physical pages of a virtual machine. To update mappings of the CVA, the Catalyst kernel module translates page frames of VMs (which are part of the host OS virtual address space (HVA)) to machine frame numbers (MFNs). An unmapped Catalyst daemon’s virtual page (in the Catalyst virtual address space) is made to point to a physical page (MFN) of the virtual machine by modifying the Catalyst daemon process page tables (refer to Figure 5). When all the physical pages are mapped to CVAs, access to the CVAs from the user space implies accessing physical pages belonging to a virtual machine. In the first round of Catalyst, we consider all pages of a virtual machine for hashing and hash comparison. In subsequent rounds, Catalyst only considers unshared pages and changes the CVA mapping to point to the unshared pages, and transfers them to the GPU for hashing and hint generation. This is achieved by checking the map count of each VM page, the map count is a count of mappings for each physical page. A map count greater than one, indicates a shared page and hence the corresponding page is not considered. However, we store the hash
values of shared pages in GPU memory (referred to as *active hints*) to enable comparisons of the newly hashed pages with the hashes of already shared pages. When the Catalyst daemon is done with processing of virtual machine pages, it removes the mapping of CVAs to the virtual machine physical pages.

### 4.1.2 Data transfer to GPU

When the Catalyst kernel module completes mapping of all the VM physical pages onto the Catalyst daemon user space, it signals Catalyst daemon to start computation on the virtual machine data. The Catalyst daemon then uses GPU API calls to transfer the data stored in the mapped CVAs to the GPU and calls the appropriate GPU functions to compute *hints* for the transferred pages.

Figure 6 shows the data transfer and GPU computation sequence across multiple virtual machines. Catalyst employs a sequence of rounds for the offloading procedure. Each Catalyst round consists of two phases. In the first phase, data present in VM pages is transferred to GPU and hashed one VM at a time. In the second phase, the hashes of all virtual machines are compared against each other and with *active hints*, and new hints (for the current round) are generated. When a round begins, the Catalyst daemon requests the Catalyst kernel module to change virtual address mappings of the Catalyst daemon considering the first virtual machine. When CVA-to-HVA mappings of the Catalyst daemon are updated, page data of the virtual machine is transferred to the GPU. Along with page data, the virtual addresses of pages which need to be hashed and virtual addresses of pages (of VM under consideration) which are already shared (*shared hints*) are also transferred to the GPU. When data transfer for first virtual machine is complete, the GPU computes the hash values of each page, sorts the calculated hashes and updates the set of *active hints* (details in Section 4.2). On completion of these GPU operations, the Catalyst daemon requests the Catalyst kernel module to reset its virtual address mappings. The Catalyst daemon then transfers data of the next virtual machine, computes hashes, performs comparisons and builds per VM *active hints*.

### 4.2 Hints generation in GPU

Catalyst daemon transfers the virtual address of unshared VM pages, their corresponding page data and virtual address of already shared pages to GPU. When virtual machine’s data transfer to GPU is complete, Catalyst daemon executes the GPU functions to compute hash and hash comparison on GPU. To compute the hash of VM page data parallelly, we create the number of GPU threads equal to number of pages of virtual machines and each thread is given one page. These threads then parallelly apply the same hash function on their page data. GPU maintains a mapping of VM virtual page address (HVA) to its calculated hash value. After the hashing is done, for computing self sharing, we sort the page hashes in increasing order of hash values in GPU. Sorting clubs together VM pages with equal hash values. In case of a single VM, we can report self sharing by comparing hashes of the VM among each other and with previously shared

---

**Figure 6.** Procedure for transfer of data and computations at the GPU with multiple virtual machines.

**Figure 7.** An example of how hints are generated in case of single VM. Notations: A- *active hints*, D-virtual machine page data, V-virtual page number, SH- *shared hints*, H-hash values of pages, and B-bitmap.
pages (active hints). We extract these virtual pages with equal hash values and report them as hints. Since in each round, we transfer the data of only those virtual machine pages which are not shared, we have to store the hashes of hints generated in previous rounds. We refer to these hints as active hints and store them in the GPU device memory.

Figure 7 shows an example of how hints are generated for a single virtual machine. At the beginning of each round we have a set of active hints. Each active hint is composed of a hash value and the corresponding virtual page address. The active hints lists the set of shared pages of a VM and corresponding hash values from the previous round. The Catalyst daemon transfers contents of pages, virtual address of pages to be hashed and virtual address of already shared pages (shared hints). Shared hints represent the set of pages of a VM that are currently shared and whose sharing has not been broken. The set of pages in the shared hints is a subset of pages in the active hints. A page may be sharable at the end of round $k$ (and hence part of the active hints), is subsequently shared, then overwritten and hence not in the shared hints at the beginning of round $k + 1$. The GPU first computes hash values over virtual machine pages and then sorts them. When the hashes are sorted, we also update the order of the corresponding virtual page addresses (referred as virtual page number in Figure 7). Next, the active hints are updated using the shared hints data. There is a need to update active hints as it is quite possible that the pages that were shared at the end of the previous round are unshared before beginning of the current round. The shared hints are used to remove entries corresponding to the unshared pages from active hints list. Referring to Figure 7, virtual page address of active hint $(15, 3)$ is not present in the set of shared hints and hence is removed from the set of active hints. We maintain a per VM bitmap corresponding to virtual page addresses of each page of the VM. Size of the bitmap is equal to the number of pages of a VM. The newly computed hash values are compared with hash values of pages in the set of active hints. For a page whose hash value matches a value in the set of active hints, its corresponding bitmap location is set to 1. The set of pages whose bits are set to 1 are the set of new hints for sharing. Referring to Figure 7, the virtual page 3 has a hash value of 11, which is also present in the active hints list and as a result the bit corresponding to virtual page 3 in the bitmap is set to 1.

After comparing the hashes with active hints, we look for sharing within the newly hashed pages. Since the hash values and corresponding virtual pages are sorted, virtual pages with equal hash values are neighbors. To identify self sharing we compare the hash value of each virtual page with its neighbors. In Figure 7, virtual pages 2 and 94 have the same hash values of 13 and do so virtual pages 1 and 8. Bitmap locations of the corresponding virtual page addresses (bit locations 1, 2, 8 and 94) are set to 1. The bitmap eventually holds the set of pages which have at least one other page with equal hash value.

In case of multiple VMs, because of GPU memory capacity limits, it may not be possible to hash pages of all VMs in one go. In such cases, we compute and store hashes and update active hints on a per-VM basis. After each VM state is updated, the per-page hashes and active hints of all the VMs are compared against each other. To compare two VMs with each other, say VM1 and VM2, we create GPU threads equal to the number of pages of VM1. Each thread picks one hash value corresponding to a VM1 page and performs a binary search for this value in the set of hash values of VM2, active hints of VM1 and active hints of VM2. If a match is found, the bitmap location of virtual page address corresponding to the hash value is set (a hint of the current round). For example, as shown in Figure 8, virtual pages $V_2$ and $V_3$ are part of the hints list based on the bitmap state. In this manner we compare the hashes of all the VMs against each other. At the end of the comparison we have a hint bitmap for each VM. We extract the virtual page addresses for each VM whose bitmap locations are set and report them as hints. At the end of the GPU computation procedure, the set of hints represents the virtual page addresses per VM which are likely to be sharable.

4.3 Targeted sharing by KSM

The Catalyst daemon gets the per VM set of virtual page addresses (sharing hints) whose hash values are equal to values of one or more pages. These set of virtual page addresses are then transferred to KSM through the Catalyst kernel module. KSM is modified to incorporate non-sequential scan behavior which is essential to realize speedy sharing in an efficient manner. Instead of scanning all pages of virtual machines, KSM invokes scan and merge operations on the provided hints to share the pages quickly. KSM scans pages corresponding to the hints of one VM at a time. When hints of one VM are exhausted, KSM moves to the hints of the next VM and so on. When KSM consumes the hints belonging to all VMs, it sends an end-of-current-round notification to the Catalyst daemon via the Catalyst kernel module.
5. **Experimental Evaluation**

In this section, we present the evaluation of Catalyst against several benchmarks, workloads and demonstrate its efficacy in different types of scenarios. The base case for comparison is with vanilla KSM. The main metrics are number of pages shared, sharing rate and CPU cycles required for sharing. We also present results showing the effect of different parameters like `num_sleep_seconds` and Copy-on-Write (CoW) breaks. A comparative study of an adaptive rate limiting policy designed for Catalyst against KSM, and a use-case that maintains a user-specified sharing level demonstrates Catalyst’s real-world applicability.

5.1 **System setup and benchmarks**

All experiments were conducted on a machine with a 3.4 GHz Intel i7-3770 processor with 8 CPU cores and 32 GB physical memory. We used NVIDIA’s GTX 970 GPU with 1664 CUDA cores, each executing at 1.18GHz, and with 4 GB device memory. The operating system on the host machine and the virtual machines was a 64-bit Ubuntu 12.04.5 LTS operating system. The host OS was a modified Linux 3.14.65 kernel and KVM was used as a virtualization platform.

Following benchmarks were used for the experiments,

- **Synthetic benchmark**: This benchmark was used to create files of specified extent of duplicate content. When read, these files would provide the equivalent sharing potential. The benchmark was also capable of inducing copy-on-write (CoW) page breaks at a specified rate by modifying file contents (via the in-memory page cache) according to the specified CoW break rate.

- **Filebench Varmail [35]**: The Varmail workload (mail server workload) represents the behavior experienced by the `/var/mail` directory of a UNIX system. The set of operations it performed includes reading, writing, rereading, and deleting emails. This workload was configured to vary the number of threads, file size, number of files, running time etc. We ran this workload for 1000 files, files of size 16 KB file, 16 threads and for 600 seconds.

- **Filebench Fileserv [35]**: The Filebench Fileserv workload generated a load experienced by a real file server. The operations that the Fileserv performed on files included reading, writing, appending and deleting. Each workload executed a combination of these operations using the configuration parameters provided. We ran this workload with 25000 files, files of size 64 KB and 8 threads.

- **OLTP Twitter benchmark [5]**: This benchmark mimicked a web-based micro-blogging website. The main operations included get tweets (read operations) and insert tweets (write operations). The workload allowed configuration of the mix of read-write operations, rate of operation, number of users and total duration of each run.

5.2 **Comparison with KSM**

In this section we compare the performance of Catalyst and KSM. The metric used for performance includes number of
Table 1. CPU cycles required by different workloads with Catalyst and different scan rates of KSM. All values are reported as Gigacycles.

<table>
<thead>
<tr>
<th>Sharing technique</th>
<th>Synthetic benchmark</th>
<th>Varmail</th>
<th>Fileserver</th>
<th>Twitter</th>
<th>Redis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Catalyst (20 seconds)</td>
<td>2.36</td>
<td>35.80</td>
<td>81.70</td>
<td>84.41</td>
<td>50.79</td>
</tr>
<tr>
<td>KSM (1000 pps)</td>
<td>10.39</td>
<td>11.87</td>
<td>8.32</td>
<td>8.45</td>
<td>12.37</td>
</tr>
<tr>
<td>KSM (2000 pps)</td>
<td>10.28</td>
<td>22.33</td>
<td>27.13</td>
<td>18.10</td>
<td>24.11</td>
</tr>
<tr>
<td>KSM (3000 pps)</td>
<td>9.88</td>
<td>32.27</td>
<td>41.05</td>
<td>26.46</td>
<td>37.63</td>
</tr>
<tr>
<td>KSM (4000 pps)</td>
<td>10.37</td>
<td>49.80</td>
<td>47.72</td>
<td>35.99</td>
<td>46.56</td>
</tr>
<tr>
<td>KSM (5000 pps)</td>
<td>9.79</td>
<td>53.03</td>
<td>52.24</td>
<td>48.66</td>
<td>61.06</td>
</tr>
<tr>
<td>KSM (10000 pps)</td>
<td>-</td>
<td>-</td>
<td>81.48</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 11. Comparison of page sharing of Catalyst and KSM for OLTP Twitter benchmark.

The sharing technique, Synthetic benchmark, Varmail, Fileserver, Twitter, and Redis.
reshared in every round. Catalyst can share them quickly, but still needs to perform the GPU orchestration in each round to share the newly broken pages.

With the other two workloads, *Fileserver* and *Redis*, we observed behavior similar to the *Varmail* and *Twitter* workloads, respectively.

Table 2 shows the average resident set size (RSS), average pages shared and CPU cycles required to share each page. It can be observed that, as compared to KSM, Catalyst achieves higher levels of average sharing for all the workloads. Average pages shared is calculated as average of the value of number of pages shared sampled every second. The CPU cycles spent per shared page in case of Catalyst are also lower in almost all the cases as compared to KSM. The CPU cycles per shared page is obtained by dividing the total CPU cycles by average sharing. With the *Twitter* workload, the high CoW break rate lead to a higher CPU cycles per shared page value. However, the average sharing of Catalyst with the *Twitter* workload was higher as compared to KSM. Other than the *Twitter* workload, Catalyst required 15% to 35% fewer CPU cycles per shared page compared to those required by KSM.

To summarize, with stable and non-changing sharing potential the time required by Catalyst to achieve potential is an order of magnitude lesser than that required by KSM. For applications that update memory pages at a moderate rate, Catalyst is still able to perform better in terms of number of pages shared, time required to achieve sharing potential and the CPU cycles required. With a workload that performs aggressive memory updates, Catalyst performs better than KSM with comparable computation costs.

### 5.3 Impact of CoW breaks

In case of *Twitter* workload, we observed that there are short lived spikes in the graph of pages shared with time (Figure 11 and Table 1).

As reported in Figure 11, workloads which are characterized by frequent memory updates result in the achieved sharing potential by Catalyst to oscillate continuously. Due to a high memory update rate, as soon as pages are shared by Catalyst, they also start getting over-written resulting in CoW breaks and decrease in sharing. On the next hint generation cycle, a set of new pages are reshared quickly and the number of shared pages increases sharply. The extent of this oscillation, number of spikes, the extent of CoW breaks and quick sharing of pages (just after hints generation) is a function of the memory update behavior and the duration between two consecutive hint generation events.

To study the impact of CoW breaks, we varied the CoW break rate of the *Synthetic* benchmark and observed the value of number of pages shared at 1 second intervals. Note that we are not measuring the page sharing rate, but report the value of page sharing achieved at 1 second intervals. The *Synthetic* benchmark created a 1 GB file with 60% sharing potential (around 160000 pages). Next, it read and wrote to the file in such a way that caused CoW breaks at the specified rate and then restored the sharing potential. We observed the number of pages shared with time for different CoW break rates: 100 breaks/sec, 1000 breaks/sec and so on. Figure 12 shows the complementary cumulative distribution of the number of pages shared in 1 second bins over the duration of the experiment. With a CoW break rate of 100 CoW breaks/s, 90% of reported page sharing is very close to the maximum sharing potential (around 160000 pages). For a CoW break rate of 2000 CoW breaks/s, for 90% of the time, the lower extent of number of pages shared was 140000 pages. As we increase the CoW break rate, this extent further decreases to 80000 shared pages with a CoW break rate of 5000 page breaks/second.

Figure 13 shows the impact of CoW breaks on total CPU cycles consumed by Catalyst. The *Synthetic* benchmark not only performs CoW breaks but also makes the content of the overwritten pages to be identical. This restores the sharing potential. As a result, if the CoW break rate is high, the pages that will be scanned and shared after CoW break will also be high. This leads to increase in the time taken by Catalyst to scan hints as number of hints generated will be equal to the number CoW breaks. As shown in Figure 13, with increase in CoW break rate, the number of CPU cycles required for scanning hints also increases. This periodic load in Catalyst, which is designed to always greedily target all sharing potential results in increased CPU cycles consumption. This effect is also seen in comparisons with KSM (in Section 5.2). Note that in all cases, even if the CPU cycles consumed by Catalyst are more, it always achieves a higher sharing potential.

### 5.4 Effect of hint generation periodicity

Figure 14 shows the variation in total number of CPU cycles consumed by Catalyst with different sleep intervals between consecutive hint generation events for the *Varmail* workload.
As expected, the total CPU cycles required by Catalyst decreases with sleep interval. An aggressive scanning process leads to an increase in number of rounds when the Catalyst procedure is executed. Each Catalyst procedure has to change and restore mappings, transfer data to the GPU, set up execution on the GPU and collate results of the GPU process. This is an important parameter to balance the CPU consumption and the extent of sharing to be maintained; hence needs to be carefully selected as per resource availability and sharing related policy.

### 5.5 Sharing across multiple VMs

We tested our implementation in a multi-VM setup to demonstrate the correctness and applicability of Catalyst. We started three virtual machines, one executed the Fileserver benchmark, second the Varmail benchmark and third executed the Synthetic benchmark with CoW break rate of 5000 CoW breaks/s. We ran Catalyst with 20 seconds sleep interval and KSM with scan rates of 5000 pps, 10000 pps and 15000 pps.

Figure 15 shows the number of pages shared during the experiments. As can be seen, Catalyst achieved a much higher sharing levels than all the other cases. Catalyst achieved up to 1.5 times and 1.25 times more sharing compared to KSM with scan rates 5000 pps and 15000 pps, respectively. The CPU cycles required for the KSM with a scan rate of 5000 pps, 10000 pps and 15000 pps, and Catalyst are 76 Gigacycles, 125 Gigacycles, 146 Gigacycles, and 120 Gigacycles, respectively. Catalyst outperformed the two aggressive cases both in terms of performance and CPU cost.

### 5.6 Adaptive rate limiting

We implemented an adaptive rate limiting policy for both Catalyst and KSM. The policy takes as input a sharing threshold parameter from the user and adjusts the scan rate of KSM or sleep interval of Catalyst. Whenever the total memory shared decreases below the threshold, the scan rate is in-
creased, and conversely scan rate is decreased when amount of memory shared is above the threshold. The setup for this experiment is same as the experiment in Section 5.5

**Policy for Catalyst:** Catalyst starts execution with a default sleep interval value, a value of $n$ seconds. Every $n$ seconds, Catalyst checks whether the memory shared is below $sharing\_threshold$. If memory shared is less than the threshold, then Catalyst invokes the GPU hint generation functionality and proceeds with targeted scanning. Once sharing of that round is achieved, it sleeps for $n$ seconds and the procedure repeats. On the other hand, if shared memory at the start of a round is more than the $sharing\_threshold$ then Catalyst does nothing; it ends the round and sleeps for $n$ seconds.

**Policy for KSM:** KSM starts execution with a low scan rate of 100 pps. After every $n$ seconds KSM checks whether memory shared is less than the $sharing\_threshold$. If memory shared is less than the threshold, scan rate of KSM is doubled. If memory shared is more than the threshold, KSM’s scan rate is reset to 100 pps. We set an upper limit of 4000 pps of scan rate of KSM.

Figure 16 shows the number of pages shared during the duration of the experiment with adaptive rate limiting. The $sharing\_threshold$ is set to 500 MB and Catalyst’s sleep interval is set to 10 seconds. From the figure it can be observed that Catalyst is able to achieve higher sharing levels compared to KSM. Also, as the sharing falls below the $sharing\_threshold$, Catalyst is able to recover quickly and achieve higher level of sharing, whereas KSM takes relatively more time to recover. Figure 17 reports the total CPU cycles required by Catalyst and KSM. Catalyst is able to achieve better performance with lower CPU cycles.

### 5.7 Micro-benchmarking Catalyst operations

We performed micro-benchmarking experiments to determine the execution requirements of different operations of Catalyst. Table 3 shows the time profile of different operations. Despite avoiding the data copy from kernel space to user space, transferring data to GPU from user-space is not cheap—requires about 2500 CPU cycles per page. The other major contributor is the process that changes the memory mappings of the memory region used by the Catalyst daemon. The average cost of the mapping update operations is also approximately 2500 CPU cycles per page. In a multi-VM setup and with aggressive memory update behavior this requirement increases the CPU cycles overhead of Catalyst.

<table>
<thead>
<tr>
<th>Operation</th>
<th>CPU Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change Mapping</td>
<td>$\sim 2500$ Per Page</td>
</tr>
<tr>
<td>Copying Data to GPU</td>
<td>$\sim 2500$ Per Page</td>
</tr>
<tr>
<td>Reading Hints from GPU</td>
<td>$\sim 17$ Per Hint</td>
</tr>
<tr>
<td>Restore Mapping</td>
<td>$\sim 600$ Per Page</td>
</tr>
<tr>
<td>Sending Hints to KSM</td>
<td>$\sim 6$ Per Hint</td>
</tr>
</tbody>
</table>

**Table 3.** Per page CPU-Cycles taken by different Catalyst operations.
6. Related Work

6.1 Memory deduplication approaches

Disco [4] was one of the first systems to propose and implement transparent page sharing. Their approach identifies identical read-only pages across multiple virtual machines and merges them to save memory. Instead of periodic scanning of pages, pages originating from same disk block were used to identify shareable pages. Disco required several guest OS modifications. VMware ESX server [37] introduced the method of content based page sharing using periodic scanning of memory pages. It hashes the content of memory pages and compares them to identify the duplicate pages. Similar to VMware ESX server, KSM [3] also periodically scans the memory pages of virtual machines and uses hashing for page deduplication. Difference Engine [9] extends the content based page sharing to sub-page level granularity by employing techniques like deduplication, compression and patching. XLH [16] monitors the I/O accesses of guest virtual machines and generates hints for KSM to scan I/O pages. These hints are in the form of pages which have high probability to be shared. Hints are then scanned by KSM preferentially enabling quick identification of sharable pages. Catalyst takes a step further assisting the KSM to scan pages in a targeted manner. Singleton [28], an extension of KSM, identifies sharing opportunities in host and guest page caches. It merges identical pages in the host and guest page caches and maintain a single copy of duplicate pages. Catalyst is an orthogonal optimization to Singleton.

6.2 GPU optimizations

GPUs have been used to accelerate the application level tasks. A number of research works [33, 36, 39] have shown the utility of GPUs in accelerating applications like image processing, scientific computation etc. Apart from application level acceleration, GPUs have also been used to accelerate system software and networking related tasks. SNAP [31] offloads the packet forwarding task to a GPU and reports a speedup of 4x. SSLSHADER [12] uses GPUs to accelerate the SSL cryptographic operations and obtains speed up of 22x to 31x over the fastest CPU implementation. GNORT [34] is a high performance Network Intrusion Detection System (NIDS) based on GPUs. GPUStore [32] accelerates the file system encryption/decryption tasks using GPUs and reports significant improvement in disk throughput.

To the best of our knowledge, there is not much work related to use of GPUs for accelerating virtualized systems or virtualization management tasks. The only reported work is by Naoi and Yamada [20] which performs compression and decompression of VM pages before and after live migration of VMs. They report a speed up of 3 to 4 times in the live migration process when GPUs are used for compression and decompression. Catalyst is an attempt in the same direction, to enable usage of GPUs to assist or offload operating system level operations and optimization tasks on to the GPU.

6.3 GPU virtualization

GPUs computation capabilities have not just made them integral part of desktop computers but also for high performance computing systems and cloud computing setups. Several applications are being developed that rely on GPUs for accelerated computing. A GPU can be accessed by virtual machines using the PCI passthrough technique. PCI passthrough gives direct access of a GPU to a VM but it does not allow the GPU to be used by another VM. GPU virtualization resolves the multiplexing issue with PCI passthrough and facilitates sharing of a GPU among multiple virtual machines. rCUDA [6], vCUDA [29, 30], GViM [10], gVirtus [7, 19] and Cu2rcu [26] virtualize the GPU by intercepting the GPU API calls. They intercept the GPU API calls in the guest machines and redirect those calls to the GPU in the host machines. Vendor supported virtualization solutions focus on manufacturing GPUs that allows multiple VMs to run their applications simultaneously on GPUs. NVIDIA is the only GPU manufacturer that currently provides such a solution. NVIDIA with the launch of GPU grid [11] technology has paved the way for new kind of GPU virtualization solution. NVIDIA GRID K1 and K2 [21] are specifically designed to support GPU sharing in the virtualized environment. These new variants allow for multiplexing of the GPU in hardware and hence the GPU can be simultaneously used by different execution entities (with the multiplexing handled at the hardware-level).

6.4 Data movement with advanced GPU architectures

An important bottleneck with usage of GPU devices is the latency of data transfer to/from the GPU. GPU devices are connected to PCI slots and interface with the CPU via the PCI bus. In order to execute tasks on GPUs, the associated data needs be transferred from CPU memory to GPU memory through the PCI bus and vice-versa for copying data back to the CPU. Heterogeneous system architectures [1] (HSA) aim to bridge this gap by fusing the CPU and the GPU on the same processor die. With HSA, GPUs have direct access to CPU physical memory, hence there is no need to explicitly transfer data via a device bus. Another alternative on the same lines is a specialized high speed interconnect between the CPU and the GPU and between GPUs (NVIDIA’s NVLink [23] technology). Such a specialized link can transfer data at a speed 5 to 12 times [23] faster compared to the traditional PCI bus. An interesting extension of Catalyst would be study cost-benefit tradeoffs with such advanced architectures.

7. Discussion

7.1 Catalyst extensions

Techniques exploiting different granularities for deduplication can be candidates for GPU assistance. For example, Difference Engine [9] relies on finding similarities at the sub-page level. Further, Guo et.al [8] propose breaking of large
pages to improve deduplication potential. Catalyst can be extended to both these cases, (i) to perform sub-page level deduplication, and (ii) to rank the large pages in order of potential benefits.

Currently, Catalyst uses an adaptive policy to tune its scanning rate. The rate is determined by using a threshold on the minimum required sharing. Additionally, other system parameters can be used to tune the behavior of Catalyst. For example, with the Twitter benchmark we observed a high rate of CoW breaks which lead to a performance degradation. An adaptive policy can be proposed in which the scan rate is inversely proportional to the rate of CoW breaks. Similarly, Catalyst can be sensitive to the CPU availability of the system, i.e., be more aggressive during low CPU utilization periods.

7.2 Architecture advancements and implications
Heterogeneous system architectures have changed the GPU memory access paradigm and enabled unified memory addressing schemes for the CPU and the GPU. These architectures can potentially mitigate the data transfer overheads and improve the efficiency of Catalyst. It would be interesting to explore these alternative models of interfacing with the GPU for OS level services like deduplication, encryption, compression etc. The latest NVIDIA CUDA API provides a software-based unified memory access solution [24]. The API frees the programmer from the burden of explicitly transferring the data to/from the CPU and the GPU. This feature works for user-level memory allocations and is not directly applicable to transfer kernel-level data objects. Extensions to the CUDA APIs would be required for efficient transfer of kernel state to the GPUs to enable Catalyst-style solutions.

7.3 Implementation alternatives
In order to avoid the copy of data from kernel to user space we have changed the mapping of user-space process variables to point to virtual machine physical pages. Another technique to access physical pages of a virtual machine is through use of the /dev/mem device interface provided by Linux. The device file can be used along with the mmap system call to map all physical pages into user-space. While this approach can be useful, the whole physical address range has to be mapped, whereas with Catalyst only subset of the physical pages correspond to the virtual machines. A naive technique is to identify physical pages of virtual machines and selectively mmap them to the Catalyst user-space daemon. A possible optimization that uses /dev/mem with mmap and relies on low-overhead and dynamic identification of physical pages for each VM can be an alternative design option for Catalyst.

Data transfer from CPU memory to GPU memory requires the data to be first transferred from pageable memory to pinned memory. Data is transferred to GPU through DMA which requires data to be present in pinned memory areas. This leads to two data copies: one from pageable memory to pinned memory and then from the pinned memory to GPU memory. Current implementation of Catalyst relies on this mechanism as well. An advancement to this is a zero-copy mechanism [2] which allows programmers to allocate variables in pinned memory through CUDA API calls. Such variables can be directly copied to the GPU via DMA-based transfer. A possible design is to allocate a large pinned memory area and map to physical pages of the virtual machines. We tried to implement this design but were not able to successfully change the mappings of addresses in pinned regions. A successful implementation of such a design can further reduce the data transfer costs of Catalyst.

8. Conclusions
The challenge of memory deduplication techniques are in identifying and leveraging the sharing opportunities quickly and efficiently. An ideal memory consolidation goal should be to quickly achieve maximum sharing with minimum CPU utilization. Towards this objective, we presented a mechanism to enhance deduplication efficiency through the usage of GPUs. We proposed Catalyst, a novel and efficient technique, to offload the kernel level memory deduplication task of page hashing and hash comparison to GPU devices. Catalyst used GPU to rapidly identify the pages that have a high likelihood of getting shared. We evaluated our system against vanilla KSM with different realistic benchmarks. The empirical results demonstrated that Catalyst can achieve higher memory sharing in lesser time when compared to different scan rate configuration of KSM. Further, the CPU utilization costs of Catalyst was lesser or comparable with KSM with different scan rate settings. We also evaluated our system for multiple-VM scenario with different VMs running different workloads. In this case too Catalyst performed better than KSM in terms of sharing achieved and CPU cycles consumed. In the end, we proposed an illustrative adaptive rate limiting policy for both Catalyst and KSM, and compared the results. The results show that Catalyst could meet higher level sharing objectives in a cost effective manner.

References


