PageCmp: Bandwidth Efficient Page Deduplication through In-memory Page Comparison

Mehrnoosh Raoufi
Computer Science Department
University of Pittsburgh
Pittsburgh, PA, USA
mraoufi@cs.pitt.edu

Quan Deng
College of Computer
National University of Defense Technology
Changsha, China
dengquan12@nudt.edu.cn

Youtao Zhang
Computer Science Department
University of Pittsburgh
Pittsburgh, PA, USA
zhangyt@cs.pitt.edu

Jun Yang
Electrical and Computer Engineering Department
University of Pittsburgh
Pittsburgh, PA, USA
juy9@pitt.edu

***Abstract*—KSM-based page deduplication is an important Linux system service for reducing main memory consumption on cloud servers. However, it tends to incur large computation and memory bandwidth overheads. Recently proposed hardware-assisted KSM approaches, while effectively addressing the computation overhead, still need to consume a dramatic amount of off-chip memory bandwidth.**

In this paper, we propose PageCmp, a PIM (Processing-In-Memory) based page deduplication approach, to achieve bandwidth efficiency on cloud servers. PageCmp exploits the bitwise operation capability inside the DRAM cell array to enable fast page comparison. By integrating a lightweight local comparator inside the output buffer of DRAM modules, PageCmp sends only the page comparison result back to the processor. Our experimental results show that, comparing to the state-of-the-art, PageCmp achieves 4x memory bandwidth reduction while introducing less than 1% hardware overhead.

***Index Terms*—Processing-in-memory, Page deduplication, DRAM, Bandwidth efficient, In-memory comparison**

I. Introduction

Modern cloud servers widely adopt server consolidation [1], i.e., having multiple virtual machines (VMs) and applications hosted on one physical server, to achieve high resource utilization. The aggregated memory demands from these VMs and applications are often high, making memory one of the most precious system resources in the system. Studies have revealed that a large amount of memory pages from different VMs contain the same data, making it possible to keep only one physical copy of all pages that have the same data, this is referred to as page deduplication.

The KSM (Kernel Same page Merging) page deduplication has been proven to be an effective system service for reducing memory space consumption [2]. However, KSM loads the source and the target pages into the processor and performs intensive byte-to-byte comparison, which not only introduces large computation overhead, but also consumes a dramatic amount of off-chip memory bandwidth. When having KSM activated, the system may have to allocate a dedicated core to perform the page comparison and consume 10GB/s off-chip memory bandwidth [3]. To address the computation overhead, Skarlatos *et al.* proposed PageForge [3] to offload the page comparison to a specially designed comparison engine inside the memory controller. While PageForge effectively addresses the computation overhead, the memory bandwidth consumption remains a big challenge for the system design.

In this paper, we propose PageCmp, a novel PIM (Processing-In-Memory) design to address the bandwidth consumption problem in page deduplication. We summarize our contributions as follows.

- We exploit the capability of bulk bitwise operation in DRAM enabled by charge-sharing to enable fast in-memory page comparison. By returning only the comparison result to the processor, PageCmp effectively mitigates large off-chip bandwidth consumption caused by page deduplication.
- We propose PageCmp-Hybrid that exploits fine grained control to terminate the page comparison early so that page inequality comparison result can finish early. PageCmp-Hybrid targets at achieving performance and energy consumption improvements over the basic PageCmp design.
- We evaluate the proposed PageCmp schemes and compare them to the state-of-the-art. Our experimental results show that PageCmp achieves 4x memory bandwidth reduction while introduces less than 1% hardware overhead. PageCmp-Hybrid outperforms the state-of-the-art page deduplication by 31% and achieves up to 4x energy consumption reduction.

II. Background and Motivation

Modern cloud servers widely adopt server consolidation to run multiple VMs on the same server. Since these VMs often need the same libraries, packages, and drivers, they share a significant amount of the same pages in the memory. Page deduplication is a technique implemented in most of hypervisors such as VMWare ESX, Xen, and Linux KVM. The key idea is to keep only one physical copy of virtual pages whose content happen to be the same. Deduplication enables hypervisors to reduce the memory footprint of VMs.

Kernel Same-page Merging (KSM). Kernel Same-page Merging (KSM) is the page deduplication service integrated in Red Hat Linux [4]. Being the state-of-the-art open source software-based page deduplication approach, KSM is also utilized by the Linux KVM hypervisor to merge the same pages across different KVM VMs. KSM is a kernel service that, once enabled, runs continuously in the background. It scans all pages in the memory periodically, checks whether they are mergeable, merges identical pages, and marks them as Copy-on-Write (CoW). KSM repeats the full scan process at a specified frequency. Recent studies showed that KSM can reduce the memory footprint by about 50% [5].

Mergeable pages in KSM are those that have not been modified. Once a page is modified by an application or OS, it is not considered as a candidate for merging. To this end, KSM stores the hash values of pages and recomputes the hash at each scan to determine if a page has been modified.

KSM uses two red-black trees, i.e., the Stable tree, and the Unstable tree, to manage its scanned pages. The stable tree keeps the pages that have been successfully merged so far while the unstable tree keeps unmerged pages at each round. Each node of the stable tree represents one physical page and a list of virtual pages that share the physical page. The stable tree is persistent and is maintained as long as KSM is running. In contrast, the unstable tree is destroyed at the end of each full scan. The unstable tree keeps track of the scanned pages in the current round if no identical page is found for them so far. At the end of each full scan round, the unstable tree contains all unmerged pages.

For each full scan, KSM picks up a list of pages in the memory as candidate pages for merging. Given one candidate page, KSM computes its hash to determine if it has been modified. KSM searches for an identical page to the candidate in the stable tree. If it is found, it merges the candidate page to the stable tree and marks it as CoW. In case it is not found in the stable tree, KSM searches for an identical one in the unstable tree. If it is found, it merges the candidate page to that node, adds that node to the stable tree and removes the node from the unstable tree. In case it is not found even in the unstable tree, the candidate is added to the unstable tree as a new node.

All the pages in the stable tree are marked as CoW. Whenever a process initiates a write request to one of these pages, a copy of that page is created by the OS such that the write request is granted on the copy. This guarantees the consistency of the physical page that is being shared by different processes.

KSM performs an exhaustive byte-by-byte comparison between pages. The exhaustive comparison process keeps CPU busy with a simple repetitive task. In addition, it consumes a large amount of bandwidth. The CPU brings two pages into its cache and uses the kernel function `memcmp` to compare them byte-by-byte. `memcmp` can tell whether the two memory segments are equal and if they are not which one is bigger. In section IV, we conduct a comprehensive study on the `memcmp` function call in KSM; to measure KSM bandwidth usage, and to assess the benefit our design will achieve.

Hardware assisted deduplication. A couple of hardware designs have been proposed to assist memory deduplication. Tian *et al.* proposed [6] hardware support to merge the same cache lines (instead of pages), which improves cache utilization. HICAMP [7] is a hardware-based deduplication design that introduces a completely new memory structure for storing unique data. HICAMP is a complex design that, in addition to memory redesign, requires new programming models.

PageForge [3] is a recent work that introduced a hardware accelerator inside the memory controller (MC) for improving software-based page deduplication. It focuses on saving CPU cycles of KSM. While PageForege effectively reduces the computation overhead, it still demands the large bandwidth usage. As an example, KSM and PageForge consume 10 GB/s and 12GB/s, respectively, for a 2GB/s workload [3]. The large bandwidth overhead motivates our design in this paper.

Processing-in-memory (PIM). Processing-in-memory (PIM) has recently emerged as a promising solution to address the memory wall crisis in the era of big data. PIM moves a portion of computation to the memory, i.e., the location where data resides. PIM improves the overall system performance by mitigating the data movement overhead. Ambit [8] is a recent study that exposes the great potential of in-memory computation capability in DRAM. Ambit enables performing bitwise AND, OR, XOR, and NOT operations using charging sharing across multiple DRAM rows. In this paper, we exploit the bulk in-memory computation capability in DRAM to enable in-memory page comparison.

III. The PageCmp Design

In this section, we discuss the PageCmp design and elaborate the architecture details.

A. Overview

Figure 1 presents an overview of the PageCmp scheme. It works as follows. When KSM is activated, the system chooses a page P from the pool of pages to be scanned for deduplication, and compares it to a page Q that is fetched from the Stable tree or the Unstable tree. Given both trees are red-black trees, the page comparison produces one of three comparison results, i.e., '$>$', '$=$', or '$<$'. While the '$>$' and '$<$' results guide the search to the left and the right child trees, respectively, the '$=$' result terminates the search and enables the page merging.

PageCmp replaces the "memcmp(P,Q)" function in the KSM process. The system sends the addresses of P and Q to the memory module, which drives the two memory pages for bitwise comparison. The in-memory comparison can only generate the bitwise logic results, e.g., $P \oplus Q$. Given such results are distributed across different chips and lack the ability to search in the red-black tree (using '$>$', '$=$', or '$<$'), PageCmp drives them to the corresponding output buffers at each chip and generates a local comparison result on each chip. The local comparison results are then sent back to the processor to determine the final comparison result.

Given a row buffer is often of 8KB and a memory page is of 4KB, a page may occupy the first or the second half the row buffer. To avoid misalignment during in-memory comparison, PageCmp prepares a 8KB system buffer and copies the page P twice to the buffer, and then compares the contents in the buffer and Q. By default, a full KSM scan chooses 100 pages to scan while the two trees contains hundreds of thousands pages. As a result, the overhead of preparing the system buffer is amortized. In the following discussion, we still use P and Q to denote two pages to be compared and assume that they are aligned.

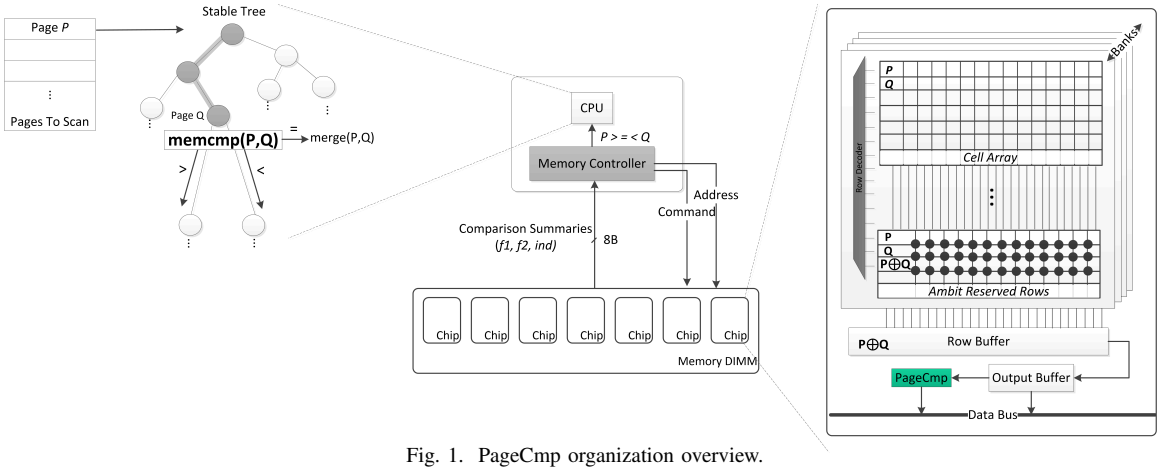

Fig. 1. PageCmp organization overview.

B. The Comparison Algorithm

Assume a memory rank consists of eight DRAM chips and the content bits of a memory page are distributed across all chips with the interleaving unit being 64b. That is, Chip 0 stores 4096 bits, i.e., all bits 512*i to 512*i+63 with 0≤i≤63. This is consistent with traditional memory mapping, e.g., a 64B (or 512b) cache line fetches data from all chips. The assumption is that a page is stored in one row of a bank. In some other configuration, a page may be interleaved over the banks [9] but in this paper, we stick to the conventional configuration. Note that there is one output buffer per chip and it can hold 32, 64 or 128 bits in modern DRAM architectures based on Micron data sheet [10]. In this paper, we assumed a configuration that has 64-bit output buffer.

When comparing pages P and Q, PageCmp works takes the following comparison steps.

- PageCmp first leverages the bitwise logic operation capability in DRAM cell array [8] to compute $P \oplus Q$. Each chip only computes the partial results for the bits that it stores.
- The partial results on one chip are then driven to the comparator inside the output buffer of the chip to determine an 8-bit comparison summary $(f1, f2, ind)$. The comparator compares 64 bits at a time and finishes the comparison in 64 rounds. For the j-th round of comparison of P_j and Q_j, each of P_j and Q_j is of 64 bits, the comparator sets $f1$ to 1 if $P_j \oplus Q_j$=0, and 0 otherwise. Assuming $f1 = 0$, the comparator finds the leftmost position, e.g., the m-th bit, of bit '1' of $P \oplus Q$. It then sets the $f2$ flag as the m-th bit value of P, and sets ind to j and Ri to m.
  The m-th bit value of P indicates whether $P_j > Q_j$ or $P_j < Q_j$. Given this is the first bit that P_j differs from Q_j, if $f2$=0, then the m-th bit of P_j is 0, indicating $P_j < Q_j$; otherwise, if $f2 = 1$, then $P_j > Q_j$.
- If $f1 = 1$, PageCmp proceeds to the next round and overwrites $(f1, f2, ind)$ with the summary of the new round. If $f1 = 0$, PageCmp sets the summary of the current round as the final result and skips the following rounds.
- With each chip sending back an 8-bit comparison summary, the processor receives 8B summary from which it determines the final comparison result, i.e., $P > Q$, $P = Q$, or $P < Q$.

For example, to compare P=01010101 and Q=01010100, we use two chips with the first chip saves '0101' for P and '0101' for Q; the second chip saves '0101' for P and '0100' for Q. Each chip takes two rounds and each round compares two bits. $P \oplus Q$ is computed as '0000' for the first chip and '0001' for the second chip.

We then need to compute the comparison summary for each chip. For the first round, both chips has '00' from $P \oplus Q$ such that $f1 = 1$ and the comparison proceeds to the second round. In the second round, the first chip has '00' from $P \oplus Q$ so that the final result is '1xxx xxxx' with 'x' indicates don't-care bit. The second chip has '01' from $P \oplus Q$ so $f1 = 0$. It then sets $f2 = 1$ and ind=1. The comparison summary from the second chip is '0100 0001'. After receiving the comparison results from both chips, the processor finds out the comparison depends on the second chip summary. For the first bit that P differs from Q, P stores bit '1' so that $P > Q$.

C. The PIM based Page Comparison

PageCmp leverages Ambit [8] to generate $P \oplus Q$ inside the cell array. To analyze the XOR result we equipped PageCmp with a small comparator inside the output buffer of each chip to provide the local comparison summary. This section explains the design of the comparator in details.

The comparator consists of several basic components to compute $f1$, $f2$, ind and Ri in 64 rounds. It works as follows. $f1$ is determined by a Zero Detector unit which has a simple design of 3-level hierarchical OR and NOR gates. To determine ind, there is a 6-bit counter inside the comparator that increments at each round. For computing Ri, we devised a fast and simple index generation scheme that figure 2 demonstrates.

Index generation scheme mainly requires 3 basic units; Leftmost One Detector, Index Mapping, Tri-state buffer. It also reuses some partial result of the Zero Detector unit. The 6-bit index generation works as follows. To accelerate the process, we generate the 3 most significant bits; Ri_h and the 3 least significant bits; Ri_l separately in parallel. For generating Ri_l, we break the 64 bits into eight chucks (8 bits per chunk). Then, we determine 3-bit index of the leftmost position of bit '1' in each chunk. Thus, we will have 8 candidates for Ri_l, we then need to select one of them. The 3-bit index generation

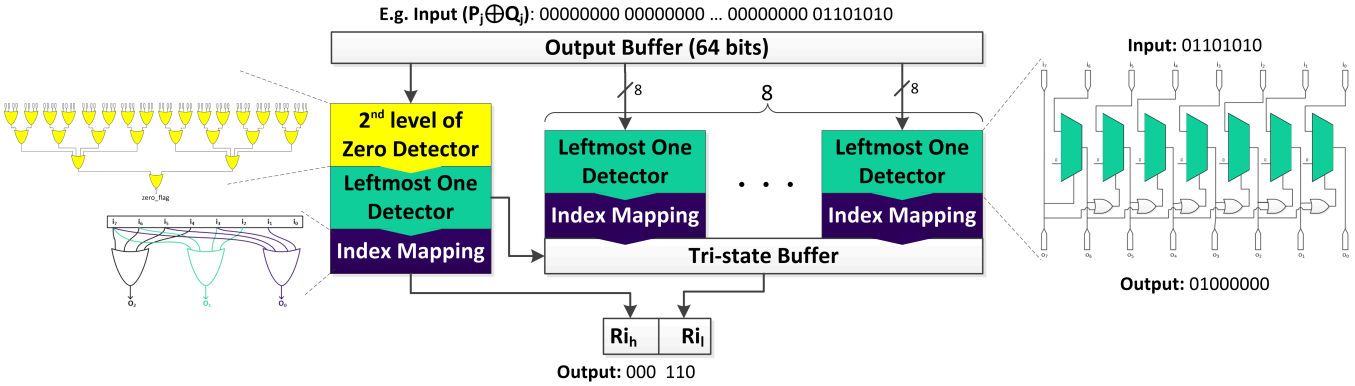

Fig. 2. PageCmp comparator design and index generation scheme.

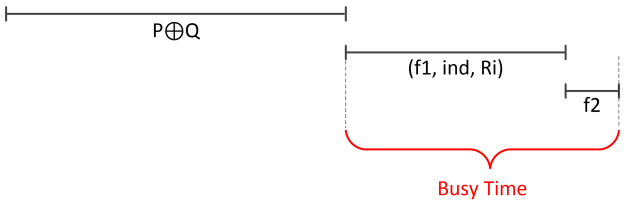

Fig. 3. PageCmp timeline.

of each chunk works as follows. We feed each of 8-bit chunk into a Leftmost One Detector unit that is shown in figure 2. This circuit gets 8 bits and zeros out all of its bits except the leftmost bit '1'. For instance, if it gets "01101010", it outputs "01000000". At last, this output is driven to an Index Mapping unit. It is a simple unit that consists of 3 OR gates. It works such that when it receives the "01000000", it outputs 110 that is the index of leftmost '1'.

To select among 8 candidates for Ri_l, we reuse partial result of Zero Detector unit, i.e., output of the second level as figure 2 indicates. It is an 8-bit result, let us call it Z. It is associated with aforementioned 8 chunks such that if the k-th chunk is non-zero the k-th bit of Z would be '1' otherwise it would be '0'. Then, we feed Z into a Leftmost One Detector unit, let the output be S. As previously discussed, only one bit of S would be '1'. If n-th bit of S is '1', it means the result of n-th chunk (i.e. the n-th Ri_l candidate) should be selected. In other words, n-th chunk is the leftmost chunk that is non-zero so it contains the leftmost bit '1' of the output buffer content. To generate Ri_h, all we need is to feed S to an Index Mapping unit. At last, Ri would be constructed by concatenating Ri_h and Ri_l as figure 2 depicts.

The computed $f1$, Ri and ind at each round may or may not be overwritten in the following rounds depending on value of $f1$. As soon as it comes to a round that $f1 = 0$, PageCmp sets $f1$, Ri and ind to their final result and skips the following rounds. Then, the flag $f2$. needs to be set. $f2$ is actually the m-th bit value of P where m is $\{ind,Ri\}$. At last, the chip will send back comparison summary $(f1, f2, ind)$ to the processor. Figure 3 represents the timeline of PageCmp.

D. Summarizing Local Comparison Result

Final result is going to be determined in the memory controller. It requires a small logic to aggregate local results. It works as follows. First, it investigates whether all $f1$ flags are zero in $\{LR_0, \dots LR_7\}$ with LR_j being the local result of the j-th chip. If all $f1$ flags are zero, it then sets the final result to $P = Q$. Otherwise, from $\{LR_0, \dots LR_7\}$ it picks those that have non-zero $f1$, then, among them picks the one that has the highest ind . Let us call it LR_{target}. The chip that LR_{target} belongs to, contains the overall leftmost '1' in the entire 4096×8 bits of $P \oplus Q$. Hence, if $f2$ of LR_{target} is '1', the final result is set to $P > Q$, otherwise it is set to $P < Q$.

E. PageCmp-Hybrid Design

We proposed PageCmp-Hybrid that is an alternative design to the original PageCmp to achieve a better performance and energy efficiency. The motivation arose from a key observation we made from our experiment. We exclusively studied data transfer of `memcmp` invoked by KSM and determined the distribution of page comparisons over the number of brought cache lines (Section IV). We observed that for those comparisons in the category that requires `memcmp` to bring only one cache line, `memcmp` outperforms the PageCmp in terms of execution time. However, for the rest of categories in the distribution, PageCmp performs faster. This happens because bringing and comparing one cache line is pretty fast in CPU and apparently it is not worth the overhead of entire page comparison in memory.

Thus, we introduce PageCmp-Hybrid in which `memcmp` is not entirely replaced. instead, PageCmp and `memcmp` are integrated. It works as follows. For each comparison, `memcmp` starts comparing byte-by-byte as usual. If it terminates after bringing one cache line (i.e. comparing 64 bytes), there will be no need to invoke in-memory comparison. Otherwise, CPU requires memory controller to invoke an in-memory page comparison which PageCmp will perform. The integration of `memcmp` and PageCmp requires only a subtle modification in KSM code. Note that despite the fact that most of comparisons belong to the first category of distribution (i.e. requiring one cache line comparison), the first category is neither the most energy-consuming nor the most time-consuming one.

IV. Experiment

We made an extensive study on KSM service to analyze its data movement overhead. Since KSM is a kernel service,

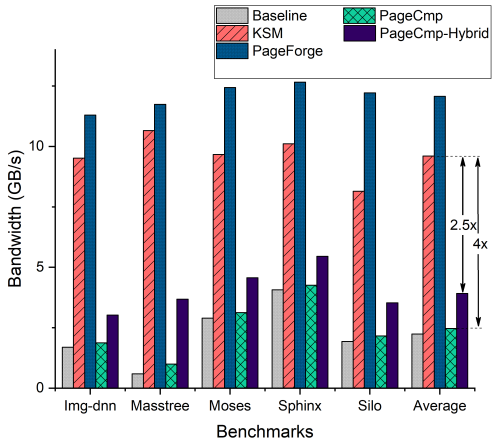

Fig. 4. Memory bandwidth consumption during page deduplication.

we profile it via modifying and recompiling the kernel source code. Basically, we set several timers and counters inside the kernel source code to profile KSM. To load KVM VM guests with some workload, we run 5 applications from Tailbench suite [11] on them; Img-Dnn (handwriting recognition application based on a deep neural network), Masstree (in-memory key-value store services), Moses (statistical machine translation system), Sphinx (speech recognition systems) and Silo (online transaction processing systems). Table I represents the system configuration of the experiments.

TABLE I
SYSTEM CONFIGURATION.

Hardware Specification	
Processor Cores	8
Frequency	3.4GHz
Main Memory	8GB
Hypervisor & Guests	
Host OS	Ubuntu 16.04
Guest OS	Ubuntu 16.04
Hypervisor	QEMU-KVM
#VMs	5
Memory/VM	512MB
KSM Settings	
sleep_millisecs	100
pages_to_scan	100
#Full Scans (up to profiling)	5

A. Memory Bandwidth Consumption

We specifically analyzed bandwidth consumption of `memcmp` function. Recall that `memcmp` performs a byte-by-byte comparison. However, each time that CPU accesses the memory it fills up one cache line which is 64B. It means, no matter `memcmp` triggers comparison of 1 byte or 64 bytes, the amount of data transferred would be 64B. Since each page is 4KB, at most 64 cache lines will be brought so there would be 64 categories of cache lines count. We then determined the distribution of comparisons over those categories. Note that in our calculation we assumed only one page is going to be brought from the main memory per comparison and the other page is already available in the cache since it is a candidate page. This assumption is realistic since in KSM there is always a candidate page being compared against other pages in KSM's trees so that candidate page is very likely to exist in the cache.

In contrast to `memcmp`, PageCmp incurs a fixed amount of 8 bytes data transfer per comparison. Thus, we calculated the proportion of data transfer of PageCmp with respect to KSM. Then, we borrowed the bandwidth numbers form PageForge [3], having these two, we ultimately estimated the bandwidth usage of our design as figure 4 illustrates. Baseline indicates when page deduplication is disabled. Note that the estimated reduction in data movement is only applied to the deduplication part so bandwidth usage of PageCmp and PageCmp-Hybrid is additive to the Baseline. As figure 4 illustrates, PageCmp and PageCmp-Hybrid can achieve up to 4x and 2.5x bandwidth saving compared to KSM, respectively. Comparing with PageForge, PageCmp and PageCmp-Hybrid can achieve up to 5x and 3x bandwidth saving, respectively.

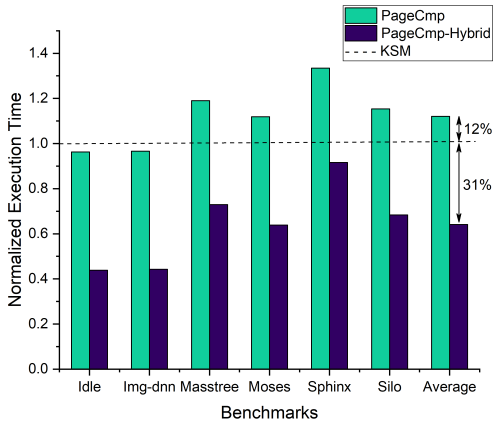

Fig. 5. PageCmp execution time normalized to the baseline.

B. Execution Time

To record the execution time of `memcmp`, we set up a timer that exclusively timed each `memcmp` function invocation. Then, to calculate the execution time of PageCmp, we did as follows. Mainly, there are two factors that affect the execution time: 1) required beforehand copies, and 2) in-memory comparison. For the first factor, we determined the proportion of comparisons that need inter-bank and intra-bank copies and we considered copy latency of RowClone [12] and LISA [13] for them respectively because they are state-of-the-art for in-memory copies. The copy time is comparable to bitwise operation time. Nevertheless, our experiment showed that a single search in KSM incurs on average 20 comparisons (which means 20 invocation of `memcmp` function). It means, if we copy a candidate page P into a bank, it is probable that in several consecutive upcoming comparisons we will be using the same page P.

For the second factor, we calculated the estimated latency of our in-memory compactor considering the fact that a logic circuit implemented inside the DRAM memory would be slower than expected. Based on a previous work on DRAM/logic technology [14] we applied 22% performance degradation to

our HSPICE simulation results for latency. In addition, we applied latency of XOR operation that Ambit [8] reported. We also further determined that `memcmp` execution time corresponds to 37% of KSM deduplication time. Provided that percentage, we compared execution time of our design versus the original KSM that uses `memcmp` function. In figure 5 y-axis is normalized to KSM execution time. As the figure demonstrates, PageCmp degrades performance of page deduplication by 12% whereas PageCmp-Hybrid improves it by 31%.

C. Performance, Energy & Area Overhead

There is a notion of busy time that figure 3 depicts. It is the time we keep the device busy and prevent it from serving regular memory requests. The busy time is incurred by; 1) beforehand copies, and 2) analyzing the result of XOR by the comparator near the output buffer. In total, PageCmp incurs 24% performance penalty for other application. However, this overhead is reduced by PageCmp-Hybrid to 3%. It is quite significant reduction that is achieved by avoiding in-memory comparisons for those comparisons that are both the most-frequent and the least time-consuming for CPU. PageCmp incurs less than 1% area overhead to DRAM layout. Note that the modification in layout is constrained to the output buffer for each chip so it does not affect DRAM subarrays.

TABLE II
ENERGY CONSUMPTION PER 4KB PAGE COMPARISON.

Mechanism	Energy Consumption (nJ)	Energy Reduction
memcmp	80 - 5120	1x
PageCmp	341	1.3x
PageCmp-Hybrid	115	4x

Energy consumption of each in-memory page comparison is mainly composed of three factors; 1) in-memory XOR operation, 2) inter-bank copy. 3) sending back 8B comparison summaries. We extracted energy consumption of each of these factors as follows. 1) XOR operation consumes 22 nJ per comparison. (Ambit paper reported energy consumption of 5.5 nJ/KB [8]). 2) Based on RowClone paper [12], inter-bank consumes 1.1 μJ. 3) Molka *et al.* showed that data transfer from RAM consumes 1250 pJ per byte [15], therefore, transferring 8B comparison summaries consumes 10 nJ. We estimated memory and channels energy consumption of our design based on the aforementioned three factors. Table II indicates energy consumption of the 4KB page comparison. As it is stated in the table, energy consumption of `memcmp` varies between 80 nJ to 5120 nJ. This is because number of cache line that `memcmp` brings varies form 1 to 64. Note that we only estimated energy consumption of `memcmp` by the amount of its data transfer so that its computational energy consumption is not included. Basic PageCmp consumes 1.3x less energy than the baseline (i.e. `memcmp`). However, PageCmp-Hybrid can achieve up to 4x energy reduction in comparison to the baseline. This is because PageCmp-Hybrid exploits `memcmp` for comparisons that it consumes low energy of 80 nJ.

V. CONCLUSION

We presented PageCmp, a novel bandwidth-efficient in-memory page comparator to mitigate intensive data transfer of page deduplication process. PageCmp achieves up to 4x bandwidth reduction at the expense of 12% increase in page deduplication execution time and less than 1% area overhead to the DRAM layout. We exploit the opportunity of bulk bitwise operation enabled by charge-sharing phenomenon in DRAM to devise a PIM based comparison scheme. We also proposed an alternative to our original design, PageCmp-Hybrid that achieves up to 2.5x bandwidth reduction in addition to improving execution time by 31%. PageCmp-Hybrid is 4x more energy efficient compared to the conventional existing approach for page comparison.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their constructive suggestions. The work is supported in part by NSF CCF# 1718080 and NSF CCF# 1617071.

REFERENCES

[1] R. W. Ahmad, A. Gani, S. H. A. Hamid, M. Shiraz, A. Yousafzai, and F. Xia, "A survey on virtual machine migration and server consolidation frameworks for cloud data centers," *J. Netw. Comput. Appl.*, vol. 52, pp. 11–25, June 2015.
[2] I. Red Hat, "Kernel same-page merging (ksm)," 2019.
[3] D. Skarlatos, N. S. Kim, and J. Torrellas, "Pageforge: A near-memory content-aware page-merging architecture," MICRO-50 '17, (New York, NY, USA), pp. 302–314, ACM, 2017.
[4] A. Arcangeli, I. Eidus, and C. Wright, "Increasing memory density by using ksm," in *In OLS*, 2009.
[5] C.-R. Chang, J.-J. Wu, and P. Liu, "An empirical study on memory sharing of virtual machines for server consolidation," ISPA '11, (Washington, DC, USA), pp. 244–249, IEEE Computer Society, 2011.
[6] Y. Tian, S. M. Khan, D. A. Jiménez, and G. H. Loh, "Last-level cache deduplication," ICS '14, (New York, NY, USA), pp. 53–62, ACM, 2014.
[7] D. Cheriton, A. Firoozshahian, A. Solomatnikov, J. P. Stevenson, and O. Azizi, "Hicamp: Architectural support for efficient concurrency-safe shared structured data access," ASPLOS XVII, (New York, NY, USA), pp. 287–300, ACM, 2012.
[8] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology," MICRO-50 '17, (New York, NY, USA), pp. 273–287, ACM, 2017.
[9] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for cross-cpu attacks," in *25th USENIX Security Symposium (USENIX Security 16)*, (Austin, TX), pp. 565–581, USENIX Association, 2016.
[10] I. Micron Technology, "8gb: x4, x8, x16 ddr4 sdram data sheet," 2018.
[11] H. Kasture and D. Sanchez, "Tailbench: a benchmark suite and evaluation methodology for latency-critical applications," pp. 1–10, Sept 2016.
[12] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Rowclone: Fast and energy-efficient in-dram bulk data copy and initialization," MICRO-46, (New York, NY, USA), pp. 185–197, ACM, 2013.
[13] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu, "Low-cost inter-linked subarrays (lisa): Enabling fast inter-subarray data movement in dram," *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 568–580, 2016.
[14] Y.-B. Kim and T. Chen, "Assessing merged dram/logic technology," in *1996 IEEE International Symposium on Circuits and Systems. Circuits and Systems Connecting the World. ISCAS 96*, vol. 4, pp. 133–136 vol.4, May 1996.
[15] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller, "Characterizing the energy consumption of data transfers and arithmetic operations on x86-64 processors," GREENCOMP '10, (Washington, DC, USA), pp. 123–133, IEEE Computer Society, 2010.