PageForge: A Near-Memory Content-Aware Page-Merging Architecture

Dimitrios Skarlatos, Nam Sung Kim, and Josep Torrellas
University of Illinois at Urbana-Champaign
{skarlat2,nskim,torrella}@illinois.edu

ABSTRACT

To reduce the memory requirements of virtualized environments, modern hypervisors are equipped with the capability to search the memory address space and merge identical pages — a process called page deduplication. This process uses a combination of data hashing and exhaustive comparison of pages, which consumes processor cycles and pollutes caches.

In this paper, we present a lightweight hardware mechanism that augments the memory controller and performs the page merging process with minimal hypervisor involvement. Our concept, called *PageForge*, is effective. It compares pages in the memory controller, and repurposes the Error Correction Codes (ECC) engine to generate accurate and inexpensive ECC-based hash keys. We evaluate PageForge with simulations of a 10-core processor with a virtual machine (VM) on each core, running a set of applications from the TailBench suite. When compared with RedHat's KSM, a state-of-the-art software implementation of page merging, PageForge attains identical savings in memory footprint while substantially reducing the overhead. Compared to a system without same-page merging, PageForge reduces the memory footprint by an average of 48%, enabling the deployment of twice as many VMs for the same physical memory. Importantly, it keeps the average latency overhead to 10%, and the 95th percentile tail latency to 11%. In contrast, in KSM, these latency overheads are 68% and 136%, respectively.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; • **Software and its engineering** → *Operating systems*; *Memory management*; Virtual machines;

KEYWORDS

Cloud computing, page merging, deduplication, memory management, near memory computing

ACM Reference format:
Dimitrios Skarlatos, Nam Sung Kim, and Josep Torrellas. 2017. PageForge: A Near-Memory Content-Aware Page-Merging Architecture. In *Proceedings of MICRO-50, Cambridge, MA, USA, October 14–18, 2017,* 13 pages.
https://doi.org/10.1145/3123939.3124540

1 INTRODUCTION

Cloud computing is based on virtualization technology. Public clouds such as Google's Compute Engine [19], Amazon's EC2 [18], IBM's Cloud [12], and Microsoft's Azure [2], as well as private ones based on OpenStack [42] and Mesos [27], rely heavily on virtualization to provide their services. Virtual Machines (VMs) enable server consolidation by allowing multiple service instances to run on a single physical machine, while providing strong failure isolation guarantees, independent configuration capabilities among instances, and protection against malicious attacks. In such environments, VMs can be easily multiplexed over CPU resources, by sharing processor time. However, each VM requires its own private memory resources, resulting in a large total memory footprint. This is especially concerning as more cores are integrated on chip and, as a result, main memory sizes are increasingly insufficient [23, 55].

Indeed, server consolidation and the emergence of memory intensive workloads in the datacenter has prompted cloud providers to equip machines with hundreds of GBs of memory. Such machines are expensive for several reasons. First, the cost of DRAM is more than an order of magnitude higher than flash memory, and more than two orders of magnitude higher than disk [5]. In addition, adding more memory chips to a machine is limited by the number of available slots in the motherboard, which often leads to the replacement of existing DRAM modules with denser ones, increasing cost. Finally, the additional memory consumes more power, which increases the cost of ownership. In spite of these higher costs, we expect users to continue to request more memory over time.

An effective way to decrease memory requirements in virtualized environments is same-page merging or page deduplication. The idea is to identify virtual pages from different VMs that have the same data contents, and map them to a single physical page. The result is a reduced main memory footprint. VMware adopts this technique with the ESX server [55], and attains memory footprint reductions of 10–40%. Similar approaches, like the Difference Engine [23], further extend page merging to include subpage-level sharing and memory compression, and attain over 65% memory footprint reductions. RedHat's implementation of page-merging, called Kernel Same-page Merging (KSM) [1], targets both virtualized and scientific environments. It has been integrated into the current Linux kernel and the KVM hypervisor. A recent study showed that KSM can reduce the memory footprint by about 50% [10].

Unfortunately, same-page merging is a high-overhead operation. Processors scan the memory space of the VMs and compare the contents of their pages exhaustively. When two identical pages are found, the hypervisor updates the page table mappings, and frees one physical page. To reduce the overhead of same-page merging, numerous optimizations have been adopted. One of them is the assignment of hash keys to pages, based on the contents of a portion of the page. For example, in KSM, a per-page hash key is generated based on 1KB of the page's contents, and is used to detect whether the page's contents change. Still, in modern server-grade systems with hundreds of GBs of memory, the performance overhead of same-page merging can be significant. It is especially harmful in user-facing services, where service-level-agreements (SLAs) mandate response times of a few milliseconds or even microseconds [14]. Unluckily, this effect will only become worse with the increase in memory size induced by emerging non-volatile memories.

In this paper, we eliminate the processor cycles and cache pollution induced by same-page merging by performing it with hardware near memory. To the best of our knowledge, this is the first solution for hardware-assisted same-page merging that is *general*, *effective*, and requires *modest hardware modifications and hypervisor involvement*. We call our proposal *PageForge*. It augments the memory controller with a small hardware module that efficiently performs page scanning and comparison semi-autonomously and transparently to the VMs. In addition, it repurposes the ECC engine in the memory controller to generate accurate and cheap ECC-based hash keys.

We evaluate PageForge with simulations of a 10-core processor with a VM on each core, running a set of applications from the TailBench suite. When compared with RedHat's KSM, a state-of-the-art software implementation of page merging, PageForge attains identical savings in memory footprint while substantially reducing the overhead. Compared to a system without same-page merging, PageForge reduces the memory footprint by an average of 48%, enabling the deployment of twice as many VMs for the same physical memory. Importantly, it keeps the average latency overhead to 10%, and the 95th percentile tail latency to 11%. In contrast, in KSM, these latency overheads are 68% and 136%, respectively.

2 BACKGROUND

Same-page merging consists of utilizing a single physical page for two or more distinct virtual pages that happen to contain the same data. The goal is to reduce the consumption of physical memory. This process is also called page deduplication or content-based sharing. In same-page merging, there are two main operations. First, each page is associated with a hash value obtained by hashing some of the page's data. Second, potentially identical pages are exhaustively compared; if two pages contain the same data, they are merged into a single physical page.

Same-page merging is highly successful in datacenter environments. In such environments, a significant amount of duplication occurs across VMs. The reason is that multiple VMs co-located in the same machine often run the same libraries, packages, drivers, kernels, or even datasets. This pattern can be exploited by the hypervisor to reduce the memory footprint of the VMs.This allows the deployment of additional VMs without additional hardware.

Figure 1(a) shows an example where two VMs have two pages of data each. Page 2 in VM-0 and Page 3 in VM-1 have the same contents. Same-page merging changes the Guest Physical Address to Host Physical Address mapping of Page 3 as shown in Figure 1(b), and frees up one physical page. This operation is performed by the hypervisor transparently to the VMs. From now on, both VMs will share the same physical page in read-only mode. If one of the VMs writes to the page, a new physical page is allocated, reverting the system back to Figure 1(a).

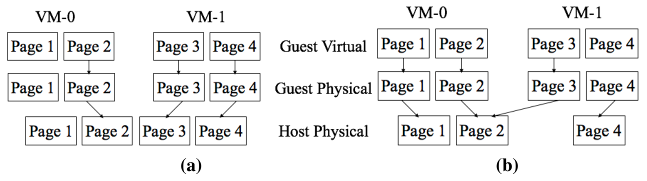

Figure 1: Example of page mapping without (a) and with (b) page merging. Pages 2 and 3 are identical.

There are software approaches to optimize same-page merging in virtualized environments [1, 3, 6, 8, 20, 23, 40, 54, 55]. However, software solutions can introduce substantial execution overhead, especially in latency-sensitive cloud applications. The overhead is due to processor cycles consumed by the page-merging process, and the resulting cache pollution. There have been some proposals for hardware to reduce data redundancy, but they either address only part of the problem (e.g., data redundancy in the caches [53]), or require a major redesign of the system [11]. All of these approaches are discussed in detail in Section 7.

2.1 RedHat's Kernel Same-page Merging

Kernel Same-page Merging (KSM) [1] is a state-of-the-art open-source software implementation of same-page merging by Red-Hat [44]. It is incorporated in the Linux kernel and targets RedHat's KVM-enabled VMs [37, 45]. Furthermore, KSM is one of the key components of Intel's recently proposed Clear Containers [30, 31]. When a VM is deployed, it provides a hint to KSM with the range of pages that should be considered for merging. In the current implementation, this is done with the *madvise* system call [38] and the *MADV_MERGEABLE* flag.

KSM continuously scans all the pages that have been marked as mergeable, discovers pages with identical content, and merges them. KSM places pages in two red-black binary trees: the *Stable* and the *Unstable* trees. The stable tree stores pages that have been successfully merged, and are marked as Copy-on-Write (CoW); the unstable tree tracks unmerged pages that have been scanned in the previous pass, and may have remained unchanged since then. Each tree is indexed by the contents of the page.

Algorithm 1 shows pseudo code for KSM. KSM runs all the time while there are mergeable pages (Line 3). It is organized in passes. In each pass, it goes over all the mergeable pages (Line 5), picking one page at a time (called *candidate page*), trying to merge it. During a pass, KSM accesses the stable tree and creates the unstable tree. At the end of each pass, it destroys the unstable tree.

After KSM picks a candidate page, it performs several operations. First, it uses the page to search the stable tree (Line 7). To search the tree, it starts at the root, comparing the candidate page to the root page byte-by-byte. If the data in the candidate page is smaller or larger than in the root page, KSM moves left or right in the tree, respectively. It then compares the candidate page to the page at that position in the tree, and moves left or right depending on the result. If KSM finds that the pages are identical, it merges the candidate page with the page at that position in the tree (Line 8). This involves

Algorithm 1: RedHat's Kernel Same-page Merging.

```
1  tree initialization
2  /* Running all the time */
3  while mergeable pages > 0 do
4      /* Go over all mergeable pages */
5      while pages for this pass > 0 do
6          candidate_pg = next page in pass
7          if search(stable_tree,candidate_pg) then
8              merge(stable_tree, candidate_pg)
9          else
10             /* Not found in stable tree */
11             new_hash = compute_hash(candidate_pg)
12             if new_hash == previous_hash(candidate_pg) then
13                 if search(unstable_tree, candidate_pg) then
14                     merged_pg=merge(unstable_tree,candidate_pg)
15                     cow_protect(merged_pg)
16                     remove(unstable_tree, merged_pg)
17                     insert(stable_tree, merged_pg)
18                 else
19                     /* Not found in unstable tree */
20                     insert(unstable_tree, merged_pg)
21                 end
22             end
23             /* Drop the page */
24         end
25     end
26     /* Throw away and regenerate */
27     reset(unstable_tree)
28 end
```

updating the mapping of the candidate page to point to that of the page in the tree, and reclaiming the memory of the candidate page.

Figure 2(a) shows an example of a red-black tree with pages as nodes. Assume that the candidate page contents are identical to Page 4. KSM starts by comparing the candidate page to Page 3. As soon as the data diverges, KSM moves to compare the candidate page to Page 5, and then to Page 4. After that, it merges it to Page 4.

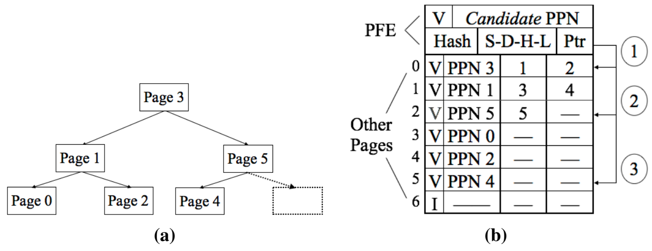

Figure 2: Example page tree (a) and corresponding PageForge Scan Table (b).

If a match is not found in the stable tree (line 10), KSM checks if the candidate page was modified since the last pass. To perform this check, it uses the data in the page to generate a hash key (Line 11). The hash key is generated with the jhash2 function [21] and 1KB of the page's data. KSM then compares the newly generated hash key to the hash key generated in the previous pass. If the keys are not the same, it means that the page has been written. In this case (or if this is the first time that this page is scanned), KSM does not consider this page any more (Line 22) and picks a new candidate page. Otherwise, KSM proceeds to compare the candidate page to those in the unstable tree (Line 13).

The search in the unstable tree proceeds as in the stable tree. There are two possible outcomes. If a match is not found, KSM inserts the candidate page in the unstable tree (Line 20). Note that the pages in the unstable tree are not write-protected and hence may change over time. If a match is found, KSM merges the two pages by updating the mappings of the candidate page (Line 14). Then, it protects the merged page with CoW (Line 15). Finally, it removes it from the unstable tree and inserts it into the stable tree (Lines 16-17). Note that, in practice, after a match is found, KSM immediately sets the two pages to CoW, and performs a second comparison of the pages. This is done to protect against racing writes during the first comparison.

Overall, the stable tree contains pages that are merged and are under CoW, while the unstable tree contains pages that may change without notification. If a write to an already merged page occurs, then the Operating System (OS) enforces the CoW policy by creating a copy of the page and providing it to the process that performed the write. The status and mapping of the other page(s) mapped to the original physical page remain intact.

Two parameters are used to tune the aggressiveness of the algorithm. First, *sleep_millisecs* is the amount of time the KSM process sleeps between work intervals, and is usually a few milliseconds. Second, *pages_to_scan* is the number of pages to be scanned at each work interval, and is usually a few hundred to a few thousand pages. In the current version of the Linux kernel, KSM utilizes a single worker thread that is scheduled as a background kernel task on any core in the system. For big servers with several VMs and hundreds of GBs of memory, a whole core can be dedicated to this process [1].

2.2 Memory Controller and ECC

Memory ECC provides single error correction and double error detection (*SECDED*), and is usually based on Hamming or similar codes [24, 28]. Previous work [15, 41, 49] has extensively studied and optimized ECC and its applicability in the datacenter. DRAM devices are commonly protected through 8-16 bits of ECC for every 64-128 data bits. In commercial architectures, an ECC encode/decode engine is placed at the memory controller.

Figure 3 shows a block diagram of a memory controller with ECC support. When a write request arrives at the memory controller, it is placed in the write request buffer, and the data block being written goes through the ECC encoder. Then, the generated ECC code and the data block are placed in the memory controller's write data buffer. The data block eventually gets written to the DRAM, and the ECC code is stored in a spare chip. Figure 4 shows one side of a DRAM DIMM with eight 8-bit data chips and one 8-bit ECC chip.

When a read request arrives at the memory controller, it is placed in the read request buffer. Eventually, the request is scheduled and the command generation engine issues the necessary sequence of commands to the memory interface. When the DRAM response arrives at the memory controller, the ECC code for the requested block arrives along with the data block. The ECC code is decoded, and the data block is analyzed for errors by checking the bit values

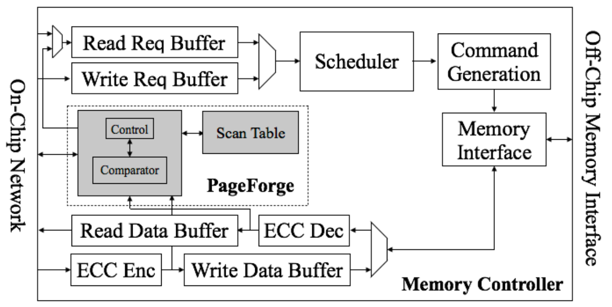

Figure 3: PageForge memory controller architecture.

in the data block and in the ECC code. If there is no error, the block is placed in the read data buffer to be delivered to the network. Otherwise, the ECC repairs the data block or notifies the software.

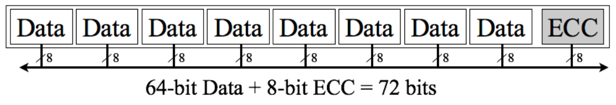

Figure 4: A single side of a DRAM DIMM architecture.

3 PAGEFORGE DESIGN

3.1 Main Idea

Existing proposals to support same-page merging in software [1, 3, 6, 8, 20, 23, 40, 54, 55] induce significant performance overhead — caused by both processor cycles required to perform the work, and the resulting cache pollution. This overhead is especially harmful in latency-sensitive cloud applications. There have been some proposals for hardware to reduce data redundancy, but they either address only part of the problem (e.g., data redundancy in the caches [53]), or require a major redesign of the system [11].

Our goal is to devise a solution for hardware-assisted same-page merging that is *general*, *effective*, and requires *modest hardware modifications and hypervisor involvement*. Our idea is to identify the fundamental operations in same-page merging, and implement them in hardware. Moreover, such hardware should be close to memory and not pollute caches. The remaining operations should be done in software, which can be changed based on the particular same-page merging algorithm implemented.

We identify three operations that should be implemented in hardware: (i) pairwise page comparison, (ii) generation of the hash key for a page, and (iii) ordered access to the set of pages that need to be compared to the candidate page. The software decides what is this set of pages that need to be compared. This hardware support is not tied to any particular algorithm.

Our design is called *PageForge*. It includes three components. The first one is a state machine that performs pairwise page comparison. The second one is a novel and inexpensive way to generate a hash key for a page by reusing the ECC of the page. The last one is a hardware table called *Scan Table*, where the software periodically uploads information on a candidate page and a set of pages to be compared with it. The hardware then accesses these pages and performs the comparisons. These three components are placed in the memory controller. With this design, accesses to memory are cheap and, because the processor is not involved, caches are not polluted.

Figure 3 shows a memory controller with the PageForge hardware shadowed. It has a Scan Table, a page comparator, and some control logic. Figure 5 shows a multicore with two memory controllers. One of them has the PageForge hardware. The L3 slices and the memory controllers connect to the interconnect.

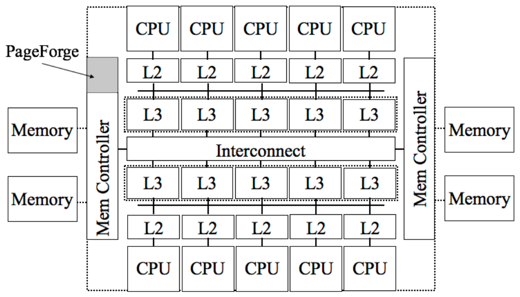

Figure 5: Multicore with PageForge in one of the memory controllers.

In the following, we discuss the process of page scanning and the Scan Table (Section 3.2), exploiting ECC codes for hash keys (Section 3.3), implementing the KSM algorithm (Section 3.4), interaction with the cache-coherence protocol (Section 3.5), and the software interface (Section 3.6). PageForge is not tied to any particular same-page merging algorithm.

3.2 Page Scanning and Scan Table

Figure 2(b) shows the structure of the Scan Table. The *PageForge (PFE)* entry contains information on the candidate page, while the *Other Pages* entries have information on a set of pages that should be compared with the candidate page. The PFE entry holds a Valid bit (V), the Physical Page Number (PPN) of the candidate page, the hash key of the page, a few control bits, and a pointer (Ptr) to one of the Other Pages in the Scan Table (i.e., the page which it is currently being compared with). The control bits are: Scanned (S), Duplicate (D), Hash Key Ready (H), and Last Refill (L). We will see their functionality later. Each entry in Other Pages contains a Valid bit (V), the PPN of the page, and two pointers to Other Pages in the Scan Table (called Less and More). Such pointers point to the next page to compare with the candidate page after the current-page comparison completes. If the current-page comparison finds that the candidate page's data is smaller than the current page's, the hardware sets Ptr to point where Less points; if it is higher, the hardware sets Ptr to point where More points. The pointed page is the next page to compare with the candidate page.

3.2.1 The Search for Identical Pages. The scanning process begins with the OS selecting a candidate page and a set of pages to compare with it. The OS inserts the candidate's information in the PFE entry, and the other pages' information in the Other Pages entries. The OS then sets the Less and More fields of each of the Other Pages entries based on the actual same-page merging algorithm that it wants to implement. It also sets the Ptr field in the PFE entry to point to the first entry in the Other Pages. Finally, it triggers the PageForge hardware.

The PageForge hardware initiates the scanning by comparing the data in the candidate page with that in the page pointed to by

Ptr. PageForge issues requests for the contents of the two pages. To generate a memory request, the hardware only needs to compute the offset within the page and concatenate it with the PPN of the page. Since a single data line from each page is compared at a time in lockstep, PageForge reuses the offset for the two pages.

The outcome of the page comparison determines the next steps. If the two pages are found to be identical after the exhaustive comparison of their contents, the Duplicate and Scanned bits in the PFE are set, and comparison stops. Otherwise, the hardware updates Ptr based on the result of the comparison of the last line. If the line of the candidate page was smaller, it sets Ptr to the Less pointer; if was larger, it sets Pts to the More pointer. The page comparison restarts.

If Ptr points to an invalid entry, PageForge completed the search without finding a match. In this case, only the Scanned bit is set. The OS periodically checks the progress of PageForge. If it finds the Scanned bit set and the Duplicate bit clear, it reloads the Scan Table with the next set of pages to compare with the candidate page. This process stops when the candidate page has been compared to all the desired pages. Then, a new candidate page is selected.

3.2.2 Interaction with the Memory System. During the scanning process, a required line can either be residing in the cache subsystem or in main memory. In a software-only implementation of same-page merging, all the operations are performed at a core and, hence, are guaranteed to use the most up-to-date data values thanks to cache coherence. However, two downsides of this approach are that it utilizes a core to perform the scanning process, and that it pollutes the cache hierarchy with unnecessary data.

The goal of PageForge is to alleviate both downsides. To achieve this, the control logic issues each request to the on-chip network first. If the request is serviced from the network, no other action is taken. Otherwise, it places the request in the memory controller's Read Request Buffer, and the request is eventually serviced from the DRAM. If, before the DRAM satisfies the request, another request for the same line arrives at the memory controller, then the incoming request is coalesced with the pending request issued from PageForge. Similarly, coalescing also occurs if a request was pending when PageForge issues a request to memory for the same line.

3.3 Exploiting ECC Codes for Hash Keys

An integral part of any same-page merging algorithm is the generation of hash keys for pages. Hash keys are used in different ways. For example, the KSM algorithm generates a new hash key for a page and then compares it to the previous key for the same page, to estimate if the page remains unchanged. Other algorithms compare the hash keys of two pages to estimate if the two pages contain the same data and can be merged.

Fundamentally, the key is used to avoid unnecessary work: if two keys are different, we know that the data in the two pages is different. However, false positives exist, where one claims that the pages have the same data but they do not. The reason is two-fold. First, a key hashes only a fraction of the page contents and, second, hashes have collisions. However, the probability of false positives does not affect the correctness of the algorithm. This is because, before PageForge merges two pages, it first compares them exhaustively.

3.3.1 Designing ECC-Based Hash Keys. PageForge introduces a novel and inexpensive approach to generate hash keys using memory ECC codes. PageForge generates the hash key of a page by concatenating the ECC codes of several, fixed-location lines within the page. This approach has two main advantages. First, the key is very simple to generate, as it simply requires reading ECC bits. Second, the PageForge hardware generates the key of a candidate page in the background, as it compares the page with a set of pages in the scanning algorithm. This is because, to compare the page, PageForge brings the lines of the page to the memory controller. If a line comes from the DRAM, the access also brings the line's ECC code; if the line comes from a cache, the circuitry in the memory controller quickly generates the line's ECC code.

Figure 6 shows a possible implementation. It assumes a 4 KB page with 64B lines and, for each line, an 8B ECC code. PageForge logically divides a 4KB page into four 1KB sections, and picks a different (but fixed) offset within each section. It then requests the lines at each of these four offsets, and picks the least-significant 8-bits of the ECC codes of these lines (called minikeys). These minikeys are concatenated together to form a 4B hash key. Hence, PageForge only brings 256B from memory to generate a hash key.

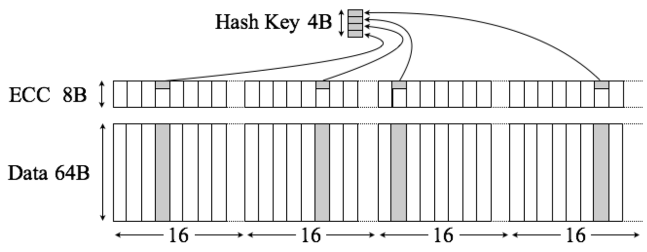

Figure 6: Example implementation of ECC-based hash keys.

Compare this to KSM. To generate a hash key, KSM requires 1KB of consecutive data from the page. In addition, its *jhash* hash function is serial, meaning that it traverses the data from the beginning to the end in order. If we were to implement a hash function similar to jhash in hardware, we would require to buffer up to 1KB of data. This is because some requests could be serviced from the caches, while other requests would go to main memory, and hence the responses could arrive at the memory controller out-of-order. PageForge reduces the memory footprint required for key generation by 75%. Moreover, it can read the lines to produce the hash key out-of-order. We evaluate the impact on accuracy in Section 6.2.

As indicated above, PageForge generates the hash key of the candidate page during the process of comparing the page to the other pages in the Scan table. As soon as the hash key is fully generated, the hardware stores the hash key in the PFE entry, and sets the Hash Key Ready (H) bit. It could be possible, however, that the hardware would be unable to complete the hash key by the time all the comparisons of the candidate page are done. To avoid this case, when the OS reloads the Scan table with the last set of pages to compare to the candidate page, it sets the Last Refill (L) bit. This forces the hardware to complete the generation of the hash key during this last processing of the entries in the Scan table.

3.3.2 Interaction with the Memory Controller. PageForge follows a series of steps to generate ECC-based hash keys. Consider again Figure 3, which shows PageForge's basic components and their connections to the ECC engine in the memory controller. The PageForge control logic generates memory requests as described in Section 3.2. A PageForge request can be serviced either from the on-chip network or from the DRAM, depending on where the requested line resides. In case it is serviced from the on-chip network, the response enters the memory controller and goes through the ECC engine. The control logic of PageForge snatches the generated ECC code. If it is serviced from the DRAM, PageForge grabs the ECC code from the ECC decoder.

3.4 Example: Implementing the KSM Algorithm

We now describe how PageForge supports the KSM algorithm. The Scan Table is used first for the stable tree, and then for the unstable tree. In both cases, the process is similar. As PageForge picks a candidate page, it loads the page information in the PFE entry. Then, it takes the root of the red-black tree currently being searched, and a few subsequent levels of the tree in breadth-first order, and loads their information in the Other Pages entries. The Less and More fields are set accordingly. Then, the OS triggers the hardware.

Figure 2(b) shows the Scan Table for the example described in Section 2.1 and shown in Figure 2(a). We have inserted 6 Other Pages entries in the Scan Table, corresponding to all the nodes in the tree. Recall that we assume that the candidate page contents are identical to Page 4. In step ①, Ptr points to the root of the tree, which is Entry 0 in the Table. After the comparison, the result shows that the candidate page is greater than Page 3. Hence, in step ②, PageForge updates Ptr to point to Entry 2, which is the right child of Page 3. PageForge then compares the candidate page to Page 5. This time the result is that the candidate page is smaller. Hence, in step ③, PageForge updates Ptr to point to Entry 5. After these steps, PageForge finds out that the candidate page and Page 4 are duplicates, and sets the Duplicate and Scanned bits of the PFE entry.

If, instead, a match was not found, the OS reloads the Scan table and triggers the PageForge hardware again to continue the search. This time, the pages loaded into the Other Pages entries of the Scan Table are those in the first few levels of the subtree on the right or on the left of Page 4 (depending on the outcome of the previous comparison to Page 4).

During the search of the stable tree, PageForge generates the hash key of the candidate page in the background, and stores it in the Hash field of the PFE entry. If a match in the stable tree was not found, the OS compares the newly-generated hash key of the candidate page with the page's old hash key (saved by the OS in advance). Based on the outcome of the comparison, PageForge either proceeds to search the unstable tree, or picks a new candidate page.

3.5 Interaction with the Cache Coherence

The PageForge module in the memory controller accesses and operates on data that may be cached in the processor's caches. Consequently, it has to participate in the cache coherence protocol to some extent. However, in our design, we have tried to keep the hardware as simple as possible.

When the memory controller issues a request on the on-chip network, it has to obtain the latest value of the requested cache line. Such request is equivalent to a request from a core. If the chip supports a snoopy protocol, all the caches are checked, and one may respond; if the chip supports a directory protocol, the request is routed to the directory, which will obtain the latest copy of the line and provide it.

However, the PageForge module does not have a cache and, therefore, does not participate as a supplier of coherent data. Instead, PageForge uses read and write data buffers in the memory controller to temporarily store its requests. If the chip supports a snoopy protocol, PageForge does not participate in the snooping process for incoming requests; if the chip supports a directory protocol, PageForge is not included in the bit vector of sharers.

This design may not produce optimal performance; however, it simplifies the hardware substantially. One could envision PageForge having a cache to cache the candidate page, or even multiple candidate pages at the same time. Caching would eliminate the need to re-read the candidate page into the PageForge module multiple times, as the candidate page is compared to multiple pages. This would reduce memory bandwidth and latency. However, this design would have to interact with the cache coherence protocol and would be substantially more complex.

It is possible that, while a candidate page is being compared to another page, either page is written to. In this case, PageForge, as well as the original software implementation of KSM, may be making decisions based on a line with a stale value. In practice, this does not affect the semantics of the merging process. The reason is that, before the actual page merging, a final comparison of the two pages is always performed under write protection, which guarantees that the page merging is safe.

3.6 Software Interface

PageForge provides a five-function interface for the OS to interact with the hardware. The interface is shown in Table 1. The main functions are *insert_PPN* and *insert_PFE*, which enable the OS to fill Other Pages and PFE entries, respectively, in the Scan table.

Function	Operands	Semantics
insert_PPN	Index, PPN, Less, More	Fill an Other Pages entry at the specified index of the Scan Table
insert_PFE	PPN, L, Ptr	Fill the PFE entry in the Scan Table
update_PFE	L, Ptr	Update the PFE entry in the Scan Table
get_PFE_info		Get the hash key, Ptr, and the S, D, and H bits from the Scan Table
update_ECC_offset	Page offsets	Update the offsets used to generate the ECC-based hash keys

Table 1: API used by the OS to access PageForge.

insert_PPN fills one of the Other Pages entries in the Scan table. It takes as operands the index of the entry to fill, the PPN of the page, and the Less and More indices. Recall that Less is the index of the next page to compare if the data in the candidate page is smaller than that in the current page; More is the index of the next page to compare if data in the candidate page is larger than that in the current page. The OS fills the whole Scan table by calling *insert_PPN* with all the indices, correctly setting the Less and More indices. The entry with index 0 is the one that will be processed first.

insert_PFE fills the PFE entry. The call takes as operands the PPN of the candidate page, the Last Refill (L) flag, and the Ptr pointer set to point to the first entry in the Other Pages array. Recall that L is set to 1 if the scanning will complete after the current batch of pages in the Scan table is processed; otherwise, it is set to 0.

As soon as the PageForge hardware is triggered, it starts the comparison of the candidate page to the pages in the Other Pages array. Once all the pages have been compared (or a duplicate page has been found and the Duplicate (D) bit has been set), PageForge sets the Scanned (S) bit, and idles.

Typically, the OS fills the Scan table multiple times, until all the relevant pages have been compared to the candidate one. Specifically, the OS periodically calls *get_PFE_info* to get S and D. If S is set and D reset, it refills the Scan table with another batch of *insert_PPN* calls, and then calls function *update_PFE*. The latter sets L to 0 or 1, and Ptr to point to the first entry in the Other Pages array. PageForge then restarts the comparison.

The last batch of comparisons is the one that either started with L set to 1, or terminates with D set because a duplicate page was found. Either of these conditions triggers PageForge to complete the generation of the hash key. Consequently, when the OS calls *get_PFE_info* after either of these two conditions, it sees that the Hash Key Ready (H) bit is set, and reads the new hash key. If D is set, the value of Ptr tells which entry matched.

The last function in Table 1 is *update_ECC_offset*. It defines the page offsets that should be used to generate the ECC-based hash key. Such offsets are rarely changed. They are set after profiling the workloads that typically run on the hardware platform. The goal is to attain a good hash key.

4 IMPLEMENTATION TRADE-OFFS

4.1 Discussion of Alternative Designs

State-of-the-art server architectures usually have 1–4 memory controllers, and interleave pages across memory controllers, channels, ranks, and banks to achieve higher memory-level parallelism. As a result, the decision of where to place PageForge on the chip and the total number of PageForge modules is not trivial. With respect to the placement of PageForge, we discuss the trade-offs between placing it inside the memory controller, and placing it outside the memory controller, directly connected to the on-chip interconnect. As for the number of PageForge modules, we discuss the trade-offs between having the PageForge module in one of the memory controllers, and having one PageForge module per memory controller.

The main benefit of placing PageForge outside of the memory controller, and directly attaching it to the on-chip network is that it leads to a more modular design. This is where accelerator modules are often placed in a processor chip. On the other hand, such approach would require all responses from the main memory to be placed on the on-chip interconnect, significantly increasing the on-chip traffic. By placing PageForge in the memory controller, we avoid the generation of unnecessary interconnect traffic when PageForge requests are serviced from the local memory module. Further, PageForge leverages the existing ECC engine for the hash key generation, which resides in the memory controller. Hence, this approach eliminates the hardware overhead of additional hash-key engines.

By increasing the number of PageForge modules in the system, we linearly increase the number of pages being scanned concurrently. As a result, the upside of having a PageForge module per memory controller is that we can improve the rate at which pages are scanned. However, this approach is accompanied by several drawbacks. First, the memory pressure increases linearly with the number of PageForge modules. Considering that the page-merging process is a very expensive background task, this approach would lead to increased memory access penalties to the workloads running on top of the VMs. In addition, while at a first glance we would expect that having one PageForge module in each memory controller will avoid cross memory controller communication, this is not the case. In practice, the page merging process searches pages across VM instances with multiple MB of memory allocated for each one. So, the common case is to compare pages that are spread out across the memory address space, and hence across memory controllers. Finally, with multiple PageForge modules, we would have to co-ordinate the scanning process among them, which would increase complexity.

We choose a simple and efficient design, namely a single PageForge module for the system that is placed in one of the memory controllers (Figure 5). Memory pressure due to page comparison remains low, since we are only comparing two pages at a time. Moreover, PageForge requests that can be serviced from the local memory module cause no on-chip interconnect traffic.

4.2 Generality of PageForge

The division between hardware and software in the PageForge design, as outlined in Sections 3.1 and 3.6, makes PageForge general and flexible. Given a candidate page, the software decides which pages should be compared to it (i.e., those that the software places in the Scan table), and in what order they should be compared (i.e., starting from the entry pointed to by Ptr, and following the Less and More fields that the software has set). The software also decides how to use the page hash key generated by the hardware.

In turn, the hardware efficiently supports three operations that are widely used in same-page merging algorithms: pairwise page comparison, generation of the hash key in the background, and in-order traversal of the set of pages to compare.

In Section 3.4, we discussed how this framework can be used to support the KSM algorithm in hardware. However, we can apply it to other same-page merging algorithms. For example, consider an algorithm that wants to compare the candidate page to an arbitrary set of pages. In this case, the OS uses *insert_PPN* to insert these pages in the Scan table (possibly in multiple batches). For each page, the OS sets *both* the Less and More fields *to the same value*: that of the *subsequent entry* in the Scan table. In this way, all the pages are selected for comparison.

Alternatively, PageForge can support an algorithm that traverses a *graph* of pages. In this case, the OS needs to put the correct pages in the Scan table with the correct More/Less pointers. Finally, PageForge can also support algorithms that use the hash key of a page in different ways.

4.3 In-Order Cores or Uncacheable Accesses

We consider two design alternatives to PageForge. The first one is to run the page-merging algorithm in software in a simple in-order core, potentially shared with other background tasks. The second one is to run the page-merging algorithm in software on a regular core, but use cache-bypassing accesses.

There are several drawbacks to running a software algorithm on an in-order core. First, the core is farther than the PageForge module from the main memory, and from the ECC-generating circuit. This means that main memory accesses and hash key generation are more costly. Second, a core consumes more power than PageForge. Section 6.4.2 estimates that, in 22nm, PageForge consumes only 0.037W on average, while an ARM-A9 core without an L2 cache consumes 0.37W on average. Third, the in-order core has to fully support the cache coherence protocol, while the PageForge module does not — which makes the system simpler. Finally, it is unrealistic to assume that the in-order core will be shared with other background tasks. Page deduplication is a sizable task that uses a large fraction of a core. In fact, it is typically pinned on an out-of-order core. Moreover, page deduplication will become heavier-weight as machines scale-up their memory size.

Running a software algorithm with cache-bypassing accesses can potentially reduce some of the performance overhead due to cache pollution. However, the CPU cycles required to run the software algorithm still remain. Further, non-cacheable requests occupy MSHR resources in the cache hierarchy, leading to resource pressure within the cache subsystem, and reducing the potential benefits of this approach. On the other hand, PageForge eliminates cache pollution, circumvents unnecessary MSHR pressure, and eliminates CPU cycles needed to run the page-deduplication algorithm.

5 EVALUATION METHODOLOGY

5.1 Modeled Architecture

We use cycle-level simulations to model a server architecture with a 10-core processor and 16GB of main memory. The architecture parameters are shown in Table 2. Each core is an out-of-order core with private L1 and L2 caches, and a shared L3 cache. A snoopy MESI protocol using a wide bus maintains coherence. We use Ubuntu Server 16.04 [50] with KVM [37] as the hypervisor, and create QEMU-KVM VM instances. Each VM is running a Ubuntu Cloud Image [29] based on the 16.04 distribution. During the experiments, each VM is pinned to a core. When the KSM process is enabled, all the cores of the system are included in its scheduling pool. Table 2 also shows the parameters of PageForge and KSM.

5.2 Modeling Infrastructure

We integrate the Simics [39] full-system simulator with the SST framework [47] and the DRAMSim2 [48] memory simulator. Additionally, we utilize Intel SAE [9] on top of Simics for OS instrumentation. Finally we use McPAT [36] for area and power estimations. We run some applications from the Tailbench suite [33]. More specifically, we deploy a total of ten VMs, one for every core in the system, each running the same application in the harness configuration provided by the TailBench suite.

We assess our architecture with a total of five applications from TailBench. *Img-dnn* is a handwriting recognition application based on a deep neural network auto-encoder which covers the general area of image recognition services. *Masstree* represents in-memory key-value store services, and is driven by a modified version of the Yahoo Cloud Serving Benchmarks [13] with 50% get and 50% put queries. *Moses* is a statistical machine translation system similar to services like Google Translate. *Silo* is an in-memory transactional database that represents online transaction processing systems (OLTP) and is driven by TPC-C. Finally, *Sphinx* represents speech recognition systems like Apple Siri and Google Now. Table 3 lists the applications and the Queries Per Second (QPS) of the runs.

Processor Parameters	
Multicore chip; Frequency	10 single-issue out-of-order cores; 2GHz
L1 cache	32KB, 8 way, WB, 2 cycles Round Trip (RT), 16 MSHRs, 64B line
L2 cache	256KB, 8 way, WB, 6 cycles RT, 16 MSHRs, 64B line
L3 cache	32MB, 20 way, WB, shared, 20 cycles RT, 24 MSHRs per slice, 64B line
Network; Coherence	512b bus; Snoopy MESI at L3
Main-Memory Parameters	
Capacity; Channels	16GB; 2
Ranks/Channel; Banks/Rank	8; 8
Frequency; Data rate	1GHz; DDR
Host and Guest Parameters	
Host OS	Ubuntu Server 16.04
Guest OS	Ubuntu Cloud 16.04
Hypervisor	QEMU-KVM
# VMs; Core/VM; Mem/VM	10; 1; 512MB
PageForge and KSM Parameters	
sleep_millisecs = 5ms; *pages_to_scan* = 400; # PageForge modules = 1	
# Scan table entries = 31 Other Pages + 1 PFE	
ECC hash key = 32bits; Scan table size ≈ 260B	

Table 2: Architectural parameters.

Application	QPS
Img_Dnn	500
Masstree	500
Moses	100
Silo	2000
Sphinx	1

Table 3: Applications executed.

5.3 Configurations Evaluated

We target a cloud scenario where we have 10 homogeneous VMs running the same application on 10 cores. Each VM is pinned to one core. This scenario effectively describes a widely-used environment that exploits replication of the applications to achieve better load balancing and fault tolerance guarantees. We compare three configurations: *Baseline*, *KSM*, and *PageForge*. Baseline is a system where same-page merging is disabled. KSM is a system running RedHat's KSM software algorithm. PageForge is a system with our proposed architecture, using the same tree search algorithm as KSM. KSM and PageForge have the same sleep interval and number of pages to scan per interval, as shown in Table 2.

We evaluate three characteristics of same-page merging: memory savings, behavior of ECC-based hash keys, and execution overhead. For the memory savings experiments, we run the application multiple times, until the same-page merging algorithm reaches steady state. At that point, we measure the memory savings attained. Because KSM and PageForge achieve the same results, we only compare two configurations: one with page merging and one without.

For the experiments on the behavior of ECC-based hash keys, we also measure the behavior when the same-page merging algorithm reaches steady state. We compare PageForge's ECC-based hash keys to KSM's jhash-based hash keys. We report the fraction of hash key matches and mismatches.

For the execution overhead experiments, we measure the time it takes to complete a request of the application under each of the three configurations. We report two measures. The first one is the average of all the requests, also called *mean sojourn latency* or mean waiting time. The second one is the latency of the 95^{th} percentile, also called *tail latency*. We report these latencies normalized to those of Baseline. The measurements are taken after a warm-up period of 1 billion instructions executed.

6 EVALUATION

6.1 Memory Savings

Figure 7 measures the savings in memory allocation attained by same-page merging. For each application, the figure shows the number of physical pages allocated without page merging (left) and with page merging (right). The bars are normalized to without page merging. Each bar is broken down into *Unmergeable*, *Mergeable Zero*, and *Mergeable Non-Zero* pages.

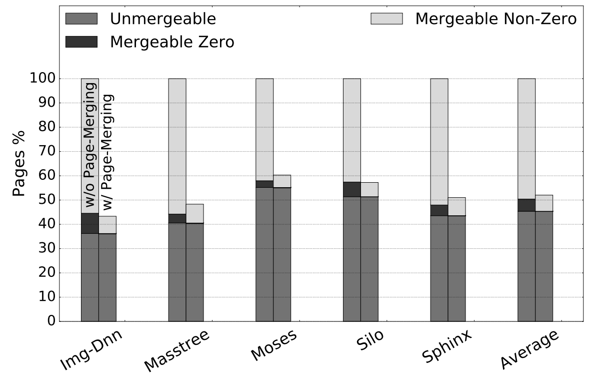

Figure 7: Memory allocation without and with page merging.

Unmergeable are the pages in the system that have unique values or whose data changes too frequently to be merged. As a result, they are not merged. On average, they account for 45% of the pages in the original system.

Mergeable Zero are pages whose data is zero. On average, they account for 5% of the pages in the original system. In current hypervisors, when the guest OS tries to allocate a page for the first time, a soft page-fault occurs, which invokes the hypervisor. The hypervisor picks a page, zeroes it out to avoid information leakage, and provides it to the guest OS. Most of the zero pages are eventually modified. However, at any time, a fraction of them remains, and is available for merging. When zero pages are merged, they are all merged into a single page (Figure 7).

Mergeable Non-Zero are non-zero pages that can be merged. On average, they account for 50% of the pages in the original system. The large majority of them are OS pages, as opposed to application pages. The figure shows that, on average, these pages are compressed to an equivalent of 6.6% of the pages in the original system.

Overall, page deduplication is very effective for these workloads. On average, it reduces the memory footprint by 48%. Intuitively, the memory savings attained imply that we can deploy about twice as many VMs as in a system without page deduplication, and use about the same amount of physical memory.

6.2 ECC Hash Key Characterization

We compare the effectiveness of PageForge's ECC-based hash keys to KSM's jhash-based hash keys. Recall that KSM's hash keys are much more expensive than PageForge's. Specifically, to generate a 32-bit key, KSM uses the jhash function on 1KB of the page contents. PageForge only needs to read four cache lines, which take 256B in our system. Hence, PageForge attains a 75% reduction in memory footprint to generate the hash key of a page.

Figure 8 compares the accuracy of the jhash-based and the ECC-based hash keys. For the latter, we use a SECDED encoding function based on the (72,64) Hamming code, which is a truncated version of the (127, 120) Hamming code with the addition of a parity bit. In the figure, we consider the hash key comparison performed by KSM and PageForge when the algorithm considers whether or not to search the unstable tree for a candidate page, and plot the fraction of key mismatches and of key matches.

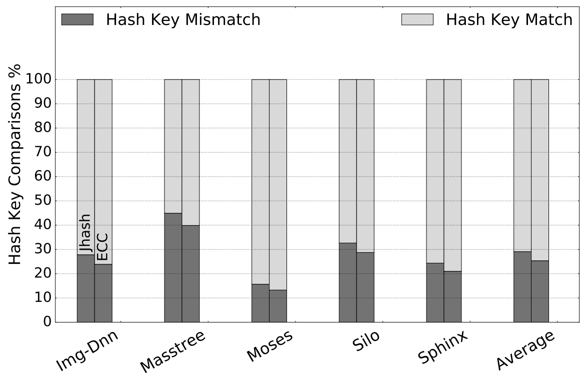

Figure 8: Outcome of hash key comparisons.

Recall that a mismatch of keys guarantees that the pages are different, while matches may be false positives (i.e., the pages may actually differ). The figure shows that ECC-based keys have slightly more matches than jhash-based ones, which correspond to false positives. On average, these additional false positives only account for 3.7% of the comparisons. This leads to the initiation of slightly more searches in the unstable tree with ECC-based keys. However, the benefits of ECC-based keys — a 75% reduction in memory footprint for key generation, the elimination of any dedicated hash engine, and the ability to overlap the search of the stable tree with the hash key generation — more than compensate for these extra searches.

6.3 Execution Overhead

Table 4 presents a detailed characterization of the KSM configuration. The second column shows the number of cycles taken by the execution of the KSM process, as a percentage of total cycles in a core. The column shows both the average across cores, and the maximum of all cores. On average, the KSM process utilizes the cores for 6.8% of the time. Recall that the Linux scheduler keeps migrating the KSM process across all the cores. However, the time that KSM spends in each core is different. As shown in the table, the core that runs the KSM process the most spends on average 33.4% of its cycles in it.

Applic.	KSM				Baseline
	Cycles (%)			L3	L3
	Avg, Max KSM Process / Total	Page Comp / KSM Process	Hash Key Gen / KSM Process	Miss Rate (%)	Miss Rate (%)
Img_Dnn	6.1, 27	54	13	47.2	44.2
Masstree	6.4, 34	50	13	39.8	26.7
Moses	7.4, 31	49	23	34.5	30.8
Silo	7.1, 39	49	12	31.7	26.5
Sphinx	7.0, 36	57	13	42.9	41.0
Average	6.8, 33.4	51.8	14.8	39.2	33.8

Table 4: Characterization of the KSM configuration.

The next two columns show the percentage of cycles in the KSM process taken by page comparison and by hash key generation, respectively. On average, 52% of the time is spent on page comparisons. Page comparisons occur during the search of the stable and unstable trees. An additional 15% of the time is spent on hash key generation. Finally, the last two columns show the local miss rate of the shared L3 for two configurations, namely, KSM and Baseline. In the KSM configuration, the shared L3 gets affected by the migrating KSM process. We can see that, on average, the L3 miss rate increases by over 5%, and goes from 34% to 39%.

To see the execution overhead of page deduplication, Figure 9 compares the mean sojourn latency in the Baseline, KSM, and PageForge configurations. For each application, the figure shows a bar for each configuration, normalized to Baseline. Recall that the sojourn latency is the overall time that a request stays in the system, and includes both queuing and service time. In an application, each bar shows the geometric mean across the ten VMs of the system.

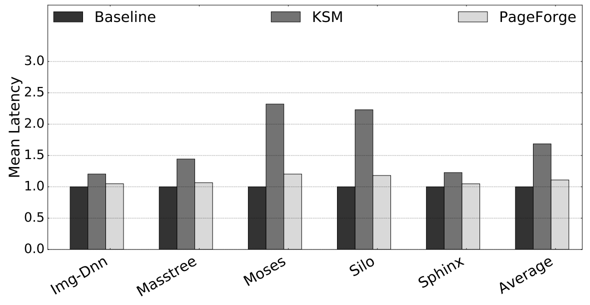

Figure 9: Mean sojourn latency normalized to Baseline.

Baseline does not perform page deduplication and, therefore, is the shortest bar in all cases. KSM performs page deduplication in software. On average, its mean sojourn latency is 1.68 times longer than Baseline's. This overhead varies across applications, depending on the load of the application (measured in queries per second (QPS)), and on the time granularity of each query. Specifically, for a given application, the overhead caused by the KSM process is higher for higher QPS loads. Furthermore, applications with short queries are more affected by the KSM process than those with long queries. The latter can more easily tolerate the queueing induced by the KSM process before it migrates away. For example, Sphinx queries have second-level granularity, while Moses queries have millisecond-level granularity. As we can see from Figure 9, their relative latency increase is very different.

PageForge performs page deduplication in hardware and, as a result, its mean sojourn latency is much lower than KSM for all the applications. On average, PageForge has a mean sojourn latency that is only 10% higher than Baseline. This overhead is tolerable. Unlike KSM, PageForge masks the cost of page comparison and hash key generation by performing them in the memory controller. It avoids taking processor cycles and polluting the cache hierarchy.

We further explore the execution overhead by comparing the latency of the 95th percentile (also called tail) latency of the three configurations. This is shown in Figure 10, which is organized as Figure 9. The tail latency better highlights the outliers in the system, when compared to the mean sojourn latency. For example, we see that Silo's tail latency in KSM is more than 5 times longer than in Baseline, while the mean sojourn latency is only twice longer. On average across applications, KSM increases the tail latency by 136% over Baseline, while PageForge increases it by only 11%.

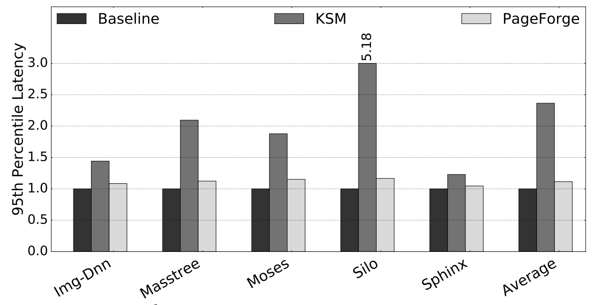

Figure 10: 95th percentile latency normalized to Baseline.

Overall, PageForge effectively eliminates the performance overhead of page deduplication. This is because it offloads the core execution with special hardware, and avoids cache pollution. In addition, it performs hash key generation with very low overhead.

6.4 PageForge Characterization

6.4.1 Memory Bandwidth Analysis. Figure 11 shows the memory bandwidth consumption of the KSM and PageForge configurations during the most memory-intensive phase of the page deduplication process in each application. The bandwidth consumption of the Baseline configuration during the same period of the application is also shown as a reference.

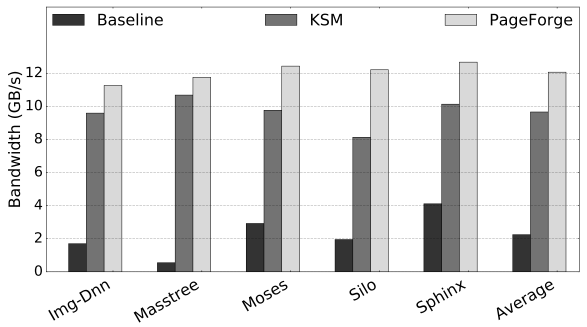

Figure 11: Memory bandwidth consumption in the most memory-intensive phase of page deduplication.

From the figure, we see that the bandwidth consumption of KSM and PageForge is higher than in Baseline. On average, Baseline consumes 2GB/s, while KSM and PageForge consume 10 and 12GB/s, respectively. The reason is that, during this active phase of page deduplication, the memory is heavily utilized with data being streamed for the page comparisons and hash key generation. Further, PageForge consumes more bandwidth than KSM. This is because deduplication in PageForge proceeds independently of the cores. Hence, its bandwidth consumption is additive to the one of the cores. Overall, however, even in these phases of the applications, the absolute memory bandwidth demands of PageForge are very tolerable.

6.4.2 Design Characteristics. Table 5 shows some design characteristics of PageForge. As indicated in Table 2, the Scan table stores a total of 31 Other Pages entries (which correspond to a root node plus four levels of the tree), and the PFE entry, for a total of 260B. We measure that, to process all the required entries in the table, PageForge takes on average 7,486 cycles. This time is mostly determined by the latency of the page comparison operations, which depend on the page contents. Hence, the table also shows the standard deviation across the applications, which is 1,296. The table also shows that the OS checks the Scan table every 12,000 cycles. Typically, the table has been fully processed by the time the OS checks.

Operation	Avg. Time (Cycles)	Applic. Standard Dev.
Processing the Scan table	7,486	1,296
OS checking	12,000	0

Unit	Area (mm^2)	Power (W)
Scan table	0.010	0.028
ALU	0.019	0.009
Total PageForge	0.029	0.037

Table 5: PageForge design characteristics.

The table also lists PageForge's area and power requirements. For the Scan table, we conservatively use a 512B cache-like structure. For the comparisons and other ALU operations, we use an ALU similar to those found in embedded processors. With 22 *nm* and high performance devices, our tools show that PageForge requires only 0.029 mm^2 and 0.037 W. In contrast, a server-grade architecture like the one presented in Table 2 requires a total of 138.6 mm^2 and has a TDP of 164 W. Compared to such architecture, PageForge adds negligible area and power overhead.

We also compare PageForge to a simple in-order core. Our tools show that a core similar to an ARM A9 with 32KB L1 data and instruction caches, and without an L2 cache, requires 0.77 mm^2 and has a TDP of 0.37 W, at 22nm and with low operating power devices. Compared to this very simple core, PageForge uses negligible area and requires an order of magnitude less power.

7 RELATED WORK

7.1 Hardware-Based Deduplication

There are two proposals for hardware-based deduplication. Both schemes perform merging of memory lines rather than pages as in PageForge. The first one by Tian et al. [53] merges cache lines that have the same contents in the Last Level Cache (LLC). The proposed cache design utilizes a hashing technique to detect identical cache lines, and merges them. The result is an increase of the LLC capacity and, hence, an improvement in application performance. However, deduplication does not propagate to the main memory and, therefore, the scheme does not increase the memory capacity of the system. Hence, this scheme is orthogonal to PageForge and can be used in conjunction with it.

The second proposal is HICAMP [11, 52], a complete redesign of the memory system so that each memory line stores unique data. In HICAMP, the memory is organized in a content addressable manner, where each line has immutable data over its lifetime. The memory is organized in segments, where each segment is a Directed Acyclic Graph (DAG). To see if the memory contains a certain value, the processor hashes the value and performs a memory lookup. The lookup can return multiple lines, which are then compared against the value to eliminate false positives. The authors introduce a programming model similar to an object oriented model.

HICAMP requires a complete redesign of the memory, introduces a complex memory access scheme, and needs a new programming model. PageForge focuses on identifying identical pages and merging them. PageForge's hardware can be easily integrated in current systems. Moreover, it requires little software changes, and no special programming model.

7.2 Software-Based Deduplication

Most same-page merging proposals and commercial products are software-based. One of the first implementations, Transparent Page Sharing (TPS), originated from the Disco research project [6]. A limitation of TPS is that it relies on modifications to the guest OS to enable the tracking of identical pages. This limitation was later addressed by VMware's ESX Server [55]. ESX enables the hypervisor to transparently track and merge pages. It generates a hash key for each page and, only if the keys of two pages are the same, it compares the pages. A similar approach is used by IBM's Active Memory Deduplication [8], which generates a signature for each physical page. If two signatures are different, there is no need to compare the pages.

The Difference Engine [23] is the first work that proposes sub-page level sharing. In this case, pages are broken down into smaller pieces in order to enable finer-grain page sharing. In addition, this work extends page sharing with page compression to achieve even greater memory savings. Memory Buddies [56] proposes the intelligent collocation of multiple VMs in datacenters in order to optimize the memory sharing opportunities. Furthermore, Despande et al. [16] present a deduplication-based approach to perform group migration of co-located VMs.

The Satori system [40] introduces a sharing technique that monitors the read operations from disk to identify identical regions. The authors argue that additional memory sharing opportunities exist within the system, but only last a few seconds. They conclude that the periodic scanning of the memory space is not sufficient to exploit such sharing opportunities. PageForge potentially enables the exploitation of such sharing opportunities since it can perform aggressive memory scanning at a fraction of the overhead of software approaches.

RedHat's open-source version of same-page merging is Kernel Same-page Merging (KSM) [1]. KSM is currently distributed along with the Linux kernel. It targets both KVM-based VMs and regular applications. We describe it in Section 2.1. A recent empirical study [10] shows that KSM can achieve memory savings of up to 50%. Since KSM is a state-of-the-art open-source algorithm, we compare PageForge to it. However, PageForge is not limited to supporting KSM.

UKSM [54] is a modification of KSM available as an independent kernel patch. In UKSM, the user defines the amount of CPU utilization that is assigned to same-page merging, while in KSM, the user defines the *sleep_millisecs* between runs of the same-page merging algorithm, and the *pages_to_scan* in each run. In addition, UKSM performs a whole-system memory scan, instead of leveraging the *madvise* system call [38] and *MADV_MERGEABLE* flag that KSM uses. While this approach enables UKSM to scan every anonymous page in the system, it eliminates the tuning opportunities provided through the *madvise* interface, and does not allow a cloud provider to choose which VMs should be prevented from performing same-page merging. UKSM also uses a different hash generation algorithm.

Overall, PageForge takes inspiration from an open-source state-of-the-art software algorithm (i.e., KSM), and implements a general and flexible hardware-based design. It is the first hardware-based design for same-page merging that is effective and can be easily integrated in current systems.

7.3 Other Related Work

Previous work in virtual memory management has focused on large pages [4, 17] and their implications on memory consolidation in virtualized environments [22, 43]. These approaches can be transparently integrated with PageForge to boost the page sharing opportunities in the presence of large pages.

DRAM devices are usually protected through 8/16-bits of ECC for every 64/128 data bits. Previous work [15, 24, 28, 41, 51] explores the trade-offs and methodologies required to efficiently provide single error correction and double error detection.

Previous work proposes modifications to the memory controller to attain better performance (e.g., [7, 25, 26]). Some work focuses on optimizing request scheduling policies [32, 34, 35, 46]. These optimizations are orthogonal to our goal, and can be employed together with PageForge.

8 CONCLUSION

This paper presented PageForge, a lightweight hardware mechanism to perform same-page merging in the memory controller. To the best of our knowledge, this is the first solution for hardware-assisted same-page merging that is general, effective, and requires modest hardware modifications and hypervisor involvement. We evaluated PageForge with simulations of a 10-core processor and a VM on each core, running a set of applications from the TailBench suite. When compared to RedHat's KSM, a state-of-the-art software implementation of page merging, PageForge achieved identical memory savings while substantially reducing the overhead. Compared to a system without same-page merging, PageForge reduced the memory footprint by an average of 48%, enabling the deployment of twice as many VMs for the same physical memory. Importantly, it kept the average latency overhead to 10%, and the 95^{th} percentile tail latency to 11%. In contrast, in KSM, these overheads were 68% and 136%, respectively.

9 ACKNOWLEDGMENT

This work was supported in part by NSF grants CCF-1536795, CNS-1557244, and CCF-1649432, and a Samsung Electronics grant in lieu of a subcontract from Yonsei University.

REFERENCES

[1] Andrea Arcangeli, Izik Eidus, and Chris Wright. 2009. Increasing memory density by using KSM. In *Proceedings of the Linux Symposium*. 19–28.
[2] Microsoft Azure. https://azure.microsoft.com.
[3] Sean Barker, Timothy Wood, Prashant Shenoy, and Ramesh Sitaraman. 2012. An Empirical Study of Memory Sharing in Virtual Machines. In *USENIX Annual Technical Conference*. USENIX, Boston, MA, 273–284.
[4] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2011. SpecTLB: A Mechanism for Speculative Address Translation. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA '11)*. ACM, New York, NY, USA, 307–318. https://doi.org/10.1145/2000064.2000101
[5] Luiz Andre Barroso and Urs Hoelzle. 2009. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines* (1st ed.). Morgan and Claypool Publishers.
[6] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. 1997. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *ACM Trans. Comput. Syst.* 15, 4 (Nov. 1997), 412–447. https://doi.org/10.1145/265924.265930
[7] John B. Carter, Wilson C. Hsieh, Leigh Stoller, Mark R. Swanson, Lixin Zhang, Erik Brunvand, Al Davis, Chen-Chi Kuo, Ravindra Kuramkote, Michael A. Parker, Lambert Schaelicke, and Terry Tateyama. 1999. Impulse: Building a Smarter Memory Controller. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*.
[8] Rodrigo Ceron, Rafael Folco, Breno Leitao, and Humberto Tsubamoto. Power Systems Memory Deduplication. http://www.redbooks.ibm.com/abstracts/redp4827.html.
[9] Nadav Chachmon, Daniel Richins, Robert Cohn, Magnus Christensson, Wenzhi Cui, and Vijay Janapa Reddi. 2016. Simulation and Analysis Engine for Scale-Out Workloads. In *Proceedings of the 2016 International Conference on Supercomputing (ICS '16)*. ACM, New York, NY, USA, Article 22, 13 pages. https://doi.org/10.1145/2925426.2926293
[10] C. R. Chang, J. J. Wu, and P. Liu. 2011. An Empirical Study on Memory Sharing of Virtual Machines for Server Consolidation. In *2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications*. 244–249. https://doi.org/10.1109/ISPA.2011.31
[11] David Cheriton, Amin Firoozshahian, Alex Solomatnikov, John P. Stevenson, and Omid Azizi. 2012. HICAMP: Architectural Support for Efficient Concurrency-safe Shared Structured Data Access. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, New York, NY, USA, 287–300. https://doi.org/10.1145/2150976.2151007
[12] IBM Cloud. https://www.ibm.com/cloud-computing.
[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. ACM, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152
[14] Jeffrey Dean and Luiz Andre Barroso. 2013. The Tail at Scale. *Communications of the ACM,* 56 (2013), 74–80. http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext
[15] Timothy J. Dell. 1997. A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory. In *IBM Executive Overview*.
[16] Umesh Deshpande, Xiaoshuang Wang, and Kartik Gopalan. 2011. Live Gang Migration of Virtual Machines. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing (HPDC '11)*. ACM, New York, NY, USA, 135–146. https://doi.org/10.1145/1996130.1996151
[17] Y. Du, M. Zhou, B. R. Childers, D. Mosse, and R. Melhem. 2015. Supporting superpages in non-contiguous physical memory. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 223–234. https://doi.org/10.1109/HPCA.2015.7056035
[18] Amazon EC2. https://aws.amazon.com/ec2.
[19] Google Compute Engine. https://cloud.google.com/compute.
[20] Mikinori Eto and Hidenori Umeno. 2008. Design and implementation of content based page sharing method in Xen. In *International Conference on Control, Automation and Systems (ICCAS)*. 2919–2922. https://doi.org/10.1109/ICCAS.2008.4694255
[21] Linux Kernel JHash Header File. http://lxr.free-electrons.com/source/include/linux/jhash.h.
[22] Fei Guo, Seongbeom Kim, Yury Baskakov, and Ishan Banerjee. 2015. Proactively Breaking Large Pages to Improve Memory Overcommitment Performance in VMware ESXi. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International*

Conference on Virtual Execution Environments (VEE '15). ACM, New York, NY, USA, 39–51. https://doi.org/10.1145/2731186.2731187

[23] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. 2008. Difference Engine: Harnessing Memory Redundancy in Virtual Machines. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 309–322. http://dl.acm.org/citation.cfm?id=1855741.1855763

[24] R. W. Hamming. 1950. Error detecting and error correcting codes. *The Bell System Technical Journal* 29, 2 (April 1950), 147–160. https://doi.org/10.1002/j.1538-7305.1950.tb00463.x

[25] Milad Hashemi, Khubaib, Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt. 2016. Accelerating Dependent Cache Misses with an Enhanced Memory Controller. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 444–455. https://doi.org/10.1109/ISCA.2016.46

[26] H. Hassan, G. Pekhimenko, N. Vijaykumar, V. Seshadri, D. Lee, O. Ergin, and O. Mutlu. 2016. ChargeCache: Reducing DRAM latency by exploiting row access locality. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 581–593. https://doi.org/10.1109/HPCA.2016.7446096

[27] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*. USENIX Association, Berkeley, CA, USA, 295–308. http://dl.acm.org/citation.cfm?id=1972457.1972488

[28] M. Y. Hsiao. 1970. A Class of Optimal Minimum Odd-weight-column SEC-DED Codes. *IBM Journal of Research and Development* 14, 4 (July 1970), 395–401. https://doi.org/10.1147/rd.144.0395

[29] Ubuntu Cloud Images. https://cloud-images.ubuntu.com.

[30] Intel. Intel Clear Containers. https://clearlinux.org/features/intel-clear-containers

[31] Intel. Intel Clear Containers: A Breakthrough Combination of Speed and Workload Isolation. https://clearlinux.org/sites/default/files/vmscontainers_wp_v5.pdf

[32] Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana. 2008. Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. In *2008 International Symposium on Computer Architecture*.

[33] H. Kasture and D. Sanchez. 2016. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. 1–10. https://doi.org/10.1109/IISWC.2016.7581261

[34] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. 2010. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 1–12. https://doi.org/10.1109/HPCA.2010.5416658

[35] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. 2010. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In *43rd Annual IEEE/ACM International Symposium on Microarchitecture*. 65–76. https://doi.org/10.1109/MICRO.2010.51

[36] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*. ACM, New York, NY, USA, 469–480. https://doi.org/10.1145/1669112.1669172

[37] Kernel Virtual Machine. https://www.linux-kvm.org.

[38] Linux Programmer's Manual MADVISE(2). http://man7.org/linux/man-pages/man2/madvise.2.html.

[39] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. 2002. Simics: A Full System Simulation Platform. *Computer* 35, 2 (Feb. 2002), 50–58. https://doi.org/10.1109/2.982916

[40] Grzegorz Miłós, Derek G. Murray, Steven Hand, and Michael A. Fetterman. 2009. Satori: Enlightened Page Sharing. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference (USENIX'09)*. USENIX Association, Berkeley, CA, USA, 1–1. http://dl.acm.org/citation.cfm?id=1855807.1855808

[41] Panagiota Nikolaou, Yiannakis Sazeides, Lorena Ndreu, and Marios Kleanthous. 2015. Modeling the Implications of DRAM Failures and Protection Techniques on Datacenter TCO. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 572–584. https://doi.org/10.1145/2830772.2830804

[42] Openstack. https://www.openstack.org.

[43] Binh Pham, Ján Veselý, Gabriel H. Loh, and Abhishek Bhattacharjee. 2015. Large Pages and Lightweight Memory Management in Virtualized Environments: Can You Have It Both Ways?. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2830772.2830773

[44] RedHat. https://www.redhat.com.

[45] RedHat. https://www.redhat.com/en/resources/kvm-kernel-based-virtual-machine.

[46] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. 2000. Memory Access Scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00)*. ACM, New York, NY, USA, 128–138.

[47] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balls, and B. Jacob. 2011. The Structural Simulation Toolkit. *SIGMETRICS Perform. Eval. Rev.* 38, 4 (March 2011), 37–42. https://doi.org/10.1145/1964218.1964225

[48] P. Rosenfeld, E. Cooper-Balls, and B. Jacob. 2011. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architecture Letters* 10, 1 (Jan 2011), 16–19. https://doi.org/10.1109/L-CA.2011.4

[49] Yiannakis Sazeides, Emre Özer, Danny Kershaw, Panagiota Nikolaou, Marios Kleanthous, and Jaume Abella. 2013. Implicit-storing and Redundant-encoding-of-attribute Information in Error-correction-codes. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 160–171. https://doi.org/10.1145/2540708.2540723

[50] Ubuntu Server. https://www.ubuntu.com/server.

[51] Jaewoong Sim, Gabriel H. Loh, Vilas Sridharan, and Mike O'Connor. 2013. Resilient Die-stacked DRAM Caches. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 416–427. https://doi.org/10.1145/2485922.2485958

[52] J.P. Stevenson. Fine-grain In-memory Deduplication for Large-scale Workloads, PhD Thesis, Stanford University, Department of Electrical Engineering. https://books.google.com/books?id=tsjdnQAACAAJ

[53] Yingying Tian, Samira M. Khan, Daniel A. Jiménez, and Gabriel H. Loh. 2014. Last-level Cache Deduplication. In *Proceedings of the 28th ACM International Conference on Supercomputing (ICS '14)*. ACM, New York, NY, USA, 53–62. https://doi.org/10.1145/2597652.2597655

[54] UKSM. http://kerneldedup.org/.

[55] Carl A. Waldspurger. 2002. Memory Resource Management in VMware ESX Server. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2002), 181–194. https://doi.org/10.1145/844128.844146

[56] Timothy Wood, Gabriel Tarasuk-Levin, Prashant Shenoy, Peter Desnoyers, Emmanuel Cecchet, and Mark D. Corner. 2009. Memory Buddies: Exploiting Page Sharing for Smart Colocation in Virtualized Data Centers. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '09)*. ACM, New York, NY, USA.