SVAGC: Garbage Collection with a Scalable Virtual Address Swapping Technique

Ismail Ataie
Department of Computer Science
Florida State University
Tallahassee, USA
ataie@cs.fsu.edu

Weikuan Yu
Department of Computer Science
Florida State University
Tallahassee, USA
yuw@cs.fsu.edu

2022 IEEE International Conference on Cluster Computing (CLUSTER) | 978-1-6654-9856-2/22/$31.00 ©2022 IEEE | DOI: 10.1109/CLUSTER51413.2022.00047

***Abstract*—Managed programming languages including Java and Scala are very popular for data analytics and mobile applications. However, they often face challenging issues due to the overhead caused by the automatic memory management to detect and reclaim free available memory. It has been observed that during their Garbage Collection (GC), excessively long pauses can account for up to 40% of the total execution time. Therefore, mitigating the GC overhead has been an active research topic to satisfy today's application requirements. This paper proposes a new technique called SwapVA to improve data copying in the copying/moving phases of GCs and reduce the GC pause time, thereby mitigating the issue of GC overhead. Our contribution is twofold. First, a SwapVA system call is introduced as a zero-copy technique to accelerate the GC copying/moving phase. Second, for the demonstration of its effectiveness, we have integrated SwapVA into SVAGC as an implementation of scalable Full GC on multi-core systems. Based on our results, the proposed solutions can dramatically reduce the GC pause in applications with large objects by as much as 70.9% and 97%, respectively, in the *Sparse.large/4* (one quarter of the default input size) and *Sigverify* benchmarks.**

***Index Terms*—GC optimization, virtual address swapping, zero-copying, large objects, Full GC, Java garbage collection**

I. Introduction

Cluster computing is progressing rapidly with the use of Big Data frameworks such as Apache Spark, Hadoop, and Flink, which are based on managed languages, such as Java and Scala [1], [2], and virtual machines (VM) for data analytics and machine learning.

VMs can utilize the Full GC algorithms that collect all heap memory, not just part of it. GC algorithms including Full GCs, in general, operate in aperiodic cycles such that VMs initiate the GC cycle while objects, either live or dead, use up all available space, and the system cannot respond to any new memory allocation request. The Full GC process uses the Stop The World (STW) mechanism to stop all mutators (application threads) to maintain memory consistency. The VM can then proceed with a Full GC cycle shortly after STW is completed.

However, two cost issues arise from VM Garbage Collection: 1) To detect and reclaim the wasted memory allocation at run-time, VM garbage collection (GC) can significantly cause overheads on system throughput [3]. 2) When some Full GC cycles are running, excessively long pauses can be imposed upon the applications.

For the first issue, GC throughput is a critical performance factor in large-scale data analytics applications. Therefore, several research studies were conducted to improve throughput through the timely collection of garbage in predefined regions [4]–[7]. Performance can be improved by optimizing Full GC algorithms to maximize resource utilization. Researchers have investigated how to allocate resources fairly among GC threads or how to increase the parallelism of GC threads [8]–[10]. However, Gog et al. [11] have enabled off-heap memory management to provide more flexibility and eliminate some of GC costs. The downside to this is that managed languages lose the advantages of automatic memory management.

Regarding the second issue (GC pauses), real-time and interactive systems can be adversely affected by STW pauses during Minor or Full GCs. Accordingly, many modern GC algorithms have concentrated on reducing the pause time, including G1 [12], C4 [13], ZGC [14] and Shenandoah [15] through concurrently running some parts of GC workloads with applications, which sacrifices VM throughput for the GC pause time. Concurrently running the moving/copying phase [14], [15], a.k.a evacuation, with application threads can reduce the pause time significantly. GCs that trace live objects in memory heaps can be categorized into two groups; moving and non-moving. In

This work is supported in part by the National Science Foundation awards 1744336 and 1763547. This research has used the NoleLand facility that is funded by the U.S. National Science Foundation award CNS-1822737. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

2168-9253/22/$31.00 ©2022 IEEE
DOI 10.1109/CLUSTER51413.2022.00047

moving GCs, all live objects are copied contiguously into available free space. The non-moving GC, in contrast, does not move live objects and may result in memory fragmentation [16]. But even non-moving GCs have to copy live objects when memory exhausted or fragmented. Immix [17], as a non-moving collector, attempts to reduce the fragmentation by performing partial defragmentation on smaller regions, a coarse-grained memory allocation unit. However, the allocation of large objects (i.e., objects larger than one memory page in size) in non-copying Large Object Spaces (LOSs) to avoid copying costs results in the fragmentation of these allocations, as well as increased maintenance costs and eventual compactions [16], [17].

The purpose of this paper is to investigate the cost of memory operations in moving GCs, particularly for large objects in Full GC compaction, and to propose a solution that could be extended and applied to other phases of GC algorithms, whether it is a parallel or a state-of-the-art concurrent GC, even non-moving GCs that need heap compaction, either fully or partially. Thus, our system call, SwapVA, allows the copying or moving of large objects by swapping their mappings of virtual memory pages instead of moving the bytes verbatim. It can optimize Full/Major GC algorithms and improve performance by reducing memory bandwidth requirements and the time required for moving objects in the compaction phase. Additionally, it makes large objects behave as small objects for GC and prevent the need to store them in LOS. On multi-core systems, GC activity from running multiple Java applications can dramatically increase memory bandwidth requirements. Reducing the GC pause time via increasing the concurrency of GC threads [8], [10] still suffers from memory bandwidth [18]. Consequently, we have further optimized SwapVA to make it more scalable on multi-core systems. Due to fewer memory operations, it can greatly reduce the required memory bandwidth for JVMs running on multicore processors. On top of the proposed system call, we have developed a novel GC implementation called SVAGC to take advantage of its performance benefits. Since data copying/moving is a common characteristic of all GC algorithms, our SwapVA technique can also be applied to other algorithms such as concurrent GCs.

Our contribution is twofold. First, we introduce a new system call, SwapVA, that provides a zero-copy technique to support the moving/copying of large objects through the swapping of their page table entries (*PTE*s). The technique can replace and speed up memory content moving/swapping operations during the compaction and evacuation phases of GCs. Second, we propose a scalable Full GC prototype in multi-core systems by utilizing the SwapVA system call, which shows the potentiality of our approach in designing GCs.

The rest of this paper is organized as follows. Section II provides an overview of GC in Java. Section III describes the design of SwapVA system call. Section IV presents the integration of SwapVA into a prototype Full GC algorithm called SVAGC. Section V provides our experimental evaluation. Section VI reviews related studies. Finally, we conclude the paper in section VII.

II. Background of Java Garbage Collection

Generally, a full Java Garbage Collection (GC) detects live objects and compact (defragment) heap memory. A regular Full GC, based on LISP2 algorithm [19], performs in four phases, including I) marking, II) forwarding address calculation, III) adjusting pointers, and IV) compaction.

I. Marking. In this phase, all heap's live objects are identified, and their addresses are marked in some related data structures such as bitmaps.

II. Forwarding Address Calculation. The phase calculates and determines the final location of live objects. Live objects are located on the first empty spaces of the heap from beginning to end, respectively.

III. Adjusting Pointers. After determining the new address of objects, references must be updated. The same as the previous phase, this one starts with the beginning of the heap and updates each referencing field of objects to a new address. In actuality, the updated address is available at a particular location of the referenced object that was calculated during the previous phase.

IV. Compaction. Afterward, some live objects would be moved into new memory locations, whose addresses are stored, for example, in the object headers. The cost of this phase varies with the size of the working set.

The following section will discuss two concerns related to GCs: compaction costs and scalability.

Compaction (Copying) Cost in GC: In applications with large working sets, including a high number of large objects, the compaction can significantly impact the GC time and VM performance. We examine the impact by running *FFT.large* and *Sparse.large*, with an average object size of 64KB and 50KB [20], respectively, from the SPECjvm2008 suite. The applications are run on top of OpenJDK15 VM using an adapted LISP2 GC prototype. This is done to measure the exact weight of each phase and to determine the effect of compaction on the overall Full GC time. As shown in Fig. 1, the most of the GC time, about 79.33%-84.76% for *Sparse.large* and *FFT.large* respectively, is spent in the compaction phase.

GC Scalability Issue: In today's prevalent multi-core systems, GC scalability is another key factor for supporting applications with intensive parallelism [21]. In fact, applications with high memory bandwidth requirements will make scaling much more difficult. In order to examine GC scalability, we run multiple instances of a single-threaded

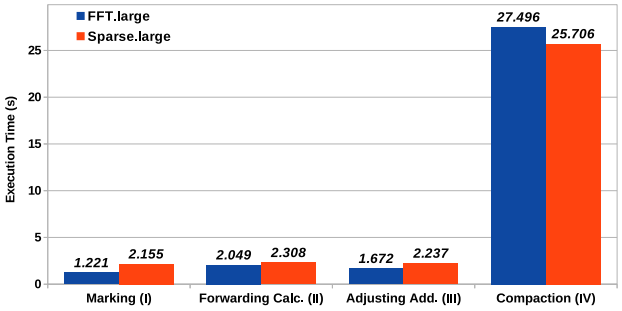

Fig. 1. Execution time of the full GC phases (Tested on Intel® Core™ i5-7600 @3.50GHz with 24GB DRAM, DDR4 @2400 MT/s).

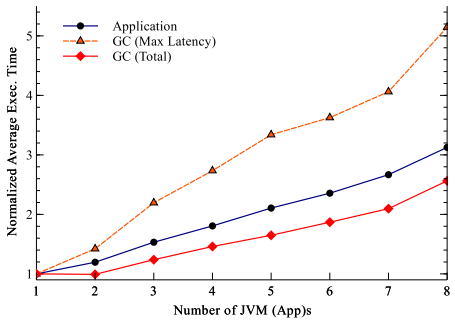

Fig. 2. Scalability issue in LRUCache benchmark (run on 32 cores dual Intel® Xeon Gold 6130 CPU @2.10GHz with 192GB DRAM, DDR4 @2666MHz). To mitigate context switching overhead on GC performance, each GC tuned to use 4 Cores by setting the GCThreadsCount parameter to 4.

memory-intensive application simultaneously, working as a LRU cache, and then benchmark the impact of multi-JVMs on each of the application and GC time. In this test, applications are run on the OpenJDK15 VM by using its ParallelGC algorithm. According to Fig. 2, by increasing the number of running applications, which is equivalent to the number of JVMs, both GC latency (maximum and total) and application execution time increase significantly.

Considering these two issues, we deem that it is desirable to optimize the compaction phase to improve both GC and application execution time.

III. Design of SwapVA

In this section, we first present the design of SwapVA for swapping the virtual memory address of objects, and then describe a few performance and scalability optimizations.

A. Overview of SwapVA

As mentioned, regardless of heap organization and applied algorithms, GC performance can be significantly reduced by delay in object moving and copying operations during a GC compaction or copying phase. By devising some strategies to decrease operations' delay, GC pause times could be effectively moderated. To this end, a number of GC algorithms have been designed to avoid these excessive copy operations within big data applications [5]. Instead, we focus on reducing the operation's cost, which

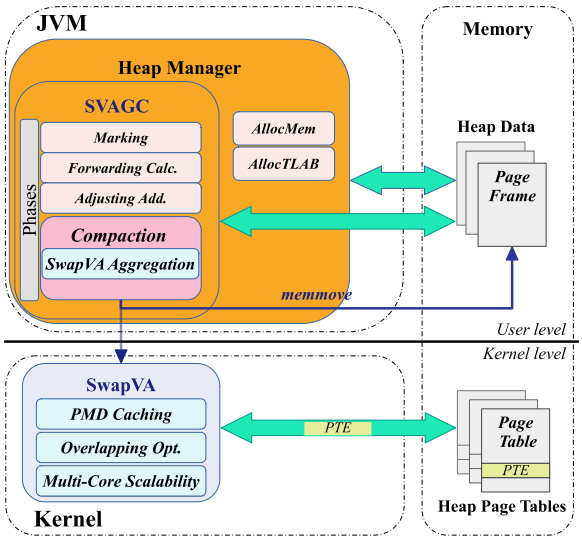

Fig. 3. Software architecture of SVAGC.

is the copying time, rather than controlling the frequency of copy operations. By directly improving the copying, the method can be applied to all GCs, regardless of their heap organization and algorithm.

Fig. 3 presents an overview of the SwapVA system call and its integration into our prototype GC, SVAGC. SwapVA provides a technique for swapping virtual addresses, i.e., exchanging two memory areas with minimal memory operations. When the values of the source operand are not in demand after the operation, it can also be used as a move operation. Within SwapVA, two internal optimizations are provided: *PMD caching* and *Overlapping*. There are also some related mechanisms for scaling SwapVA in multicore systems, which will be discussed in section IV.

B. SwapVA and Internal Optimizations

Modern operating systems and CPU architectures both provide a memory management unit (MMU) that maps virtual/logical addresses into physical/real ones. The virtual memory addresses are divided into smaller subgroups to serve as indexes for mapping tables. The last part, typically the least significant bits (12 bits for 4KB pages), is used to determine the offset in the page's physical address. The mappings are stored in the memory as hierarchically organized directories, which are called page tables. To improve performance and avoid the MMU recalculations, the Translation Lookaside Buffer (TLB) could cache each virtual-to-physical address mapping.

The design of this architecture ensures that applications can only use virtual addresses and not physical addresses. When an application needs to swap two memory areas, it must exchange them word by word. This practice generates significant overhead for applications when copying/moving large chunks of memory.

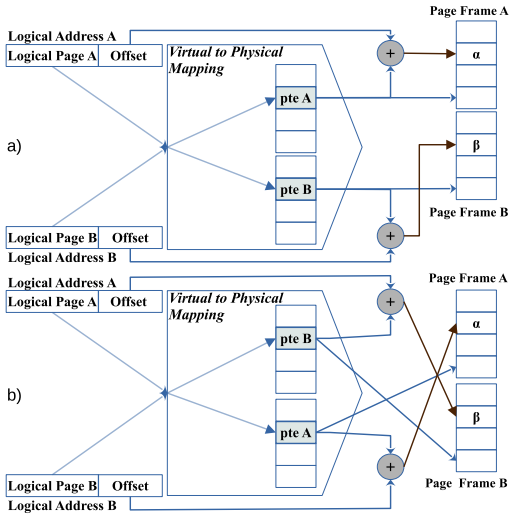

Fig. 4. Swapping virtual addresses.

To resolve the aforementioned issue, we propose a mechanism to swap memory areas based on exchanging the last level of virtual-to-physical address mappings, i.e., *PTE*s, as shown in Fig. 4. Conceptually, by using our technique to swap memory regions, we can reduce a large part of the costs associated with naive copy operations. For example, to swap the contents of two virtual pages, A and B, the cost consists only of exchanging their *PTE*s. There is however an obstacle as *PTE*s are protected in kernel space, so manipulations must be performed in privileged mode. In this regard, some OS extension mechanisms could be leveraged, such as system calls or loadable kernel modules (*LKM*), to lift the restriction.

Algorithm 1 illustrates simple steps for swapping two memory areas identified by two sets of continuous virtual addresses. In each step, through looking at the page tables, it determines which pair of PTEs should be swapped and then locks them. Once they are swapped, the locks are released. Finally, it invalidates all *TLB*s of calling process by $flush\_tlb\_local$ function.

SwapVA Aggregation: For a single-core architecture, the execution time of SwapVA might be calculated as follows:

$$T_{SwapVA} = T_{SystemCall} + T_{flush\_tlb} + T_{FunctionBody} \quad (1)$$

Practically, swapping a large number of pages is highly efficient; however, if there are few pages, then the cost of system call invocation, $T_{SystemCall}$, can make up a large portion of the overall execution time. We addressed the issue by *aggregating multiple swapping operations* into a single SwapVA invocation. In turn, context switching overhead can be reduced across multiple system calls, leading to improved GC performance for frequent data object copying. Fig. 5 illustrates how to aggregate multiple small-sized swapping operations into one system call invocation (see Fig. 5 (b)). According to Fig. 6, reducing overall call

Algorithm 1 SwapVA on Intel x86-64 Architecture

procedure SWAPVA($vAdd1, vAdd2, pages$)
  function GETPTE($vAdd, ptlp$)
    $pgd \leftarrow pgd\_offset(vAdd)$
    $p4d \leftarrow p4d\_offset(pgd, vAdd)$
    $pud \leftarrow pud\_offset(p4d, vAdd)$
    $pmd \leftarrow pmd\_offset(pud, vAdd)$
    $pte \leftarrow pte\_offset\_map\_lock(pmd, vAdd, ptlp)$
    return pte
  end function
  $spinlock\_t *\ ptlp1,\ ptlp2$
  $pid \leftarrow getPID()$ ▷ find the current process id
  for $i \leftarrow 0, ..., pages - 1$ **do**
    $j \leftarrow i \ll PAGE\_SHIFT$
    $pte1 \leftarrow$ GETPTE$(vAdd1 + j, ptlp1)$
    $pte2 \leftarrow$ GETPTE$(vAdd2 + j, ptlp2)$
    $temp \leftarrow pte2;\quad pte2 \leftarrow pte1;\quad pte1 \leftarrow temp$
    $pte\_unmap\_unlock(pte1, ptlp1)$
    $pte\_unmap\_unlock(pte2, ptlp2)$
  $flush\_tlb\_local(pid)$ ▷ flush the process's TLBs
end procedure

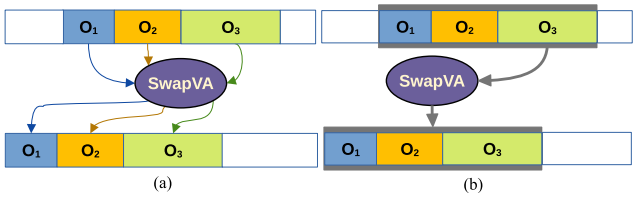

Fig. 5. Aggregation of SwapVA requests (O stands for object/data). a) Separated SwapVA calls b) Aggregated SwapVA calls.

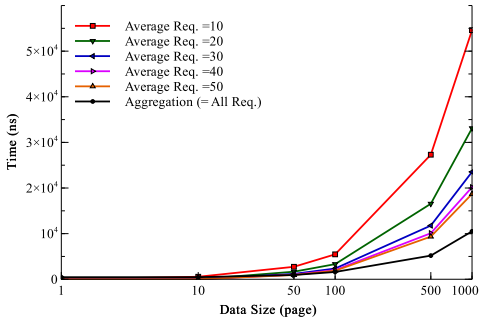

Fig. 6. Aggregated vs Separated SwapVA Calls (Tested on Intel® Core™ i5-7600 @3.50GHz with 24GB DRAM, DDR4 @2400 MT/s).

costs can be achieved by increasing the average input size of SwapVA.

PMD Caching: As an additional enhancement to SwapVA, we introduce *PMD* (Page Mid-Level Directory) *caching* for swapping memory areas with a large number of page frames. Fig. 7 illustrates how PMD caching works. In large-scale swapping operations, since addresses are accessed sequentially, it is most likely that prefixes are the same, so a calculated pmd value can be reused. Therefore, caching the last *pmd* to the base address of current page table entries can shorten numerous memory accessing op-

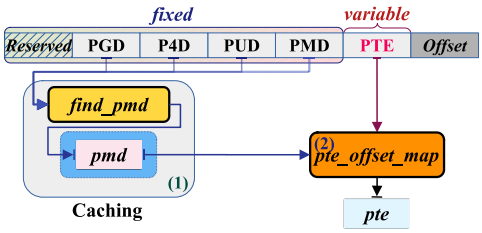

Fig. 7. SwapVA optimization by PMD caching.

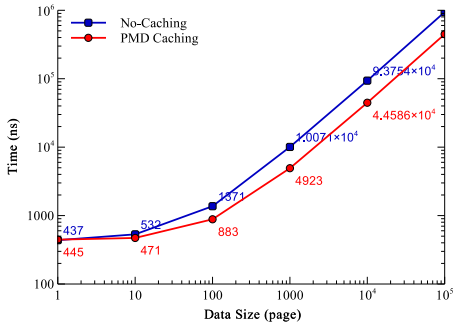

Fig. 8. Benefits of PMD caching (Tested on Intel® Core™i5-7600 @3.50GHz with 24GB DRAM, DDR4 @2400 MT/s).

erations to locate the table of *PTE*s. For example, locating the *pte* of a virtual address needs to find *pmd* first, Fig. 7(1), and then use it to obtain the *pte*, Fig. 7(2). But, the result of step 1 could be kept in some *pmd* variable for the following iterations. Accordingly, in subsequent iterations, only step 2 is required, and all step 1 sequences could be eliminated.

The effectiveness of *PMD caching* is shown in Fig. 8. This optimization improves the performance of SwapVA by up to 52.48% and an average of 36.73% for multi-page copying operations.

Overlapping Areas: In some cases, two areas can share some page table entries due to overlapping address ranges. We can detect such overlaps and optimize the swapping for their *PTE*s, thus it could reduce the cost of data copying even further. For example, by applying this optimization, the cost of swapping $2n$ pages reduces from $O(2n)$ to $O(n+\delta)$; δ is the distance between the swapping areas, $0 < \delta \leq n$.

Practically, Algorithm 2 presents a unique design to swap two overlapping areas using a few, around four, temporary memory words to track all *PTE*s. It also uses a *gcd* function to control the iteration. The low memory requirement of SwapVA allows it to perform at higher efficiency. The inner loop of Algorithm 2 works as a cycle. The cycle starts from a *PTE A* to find its target *PTE B*. It copies the target *PTE B* into a temporary variable and then updates the target by *PTE A*. Now this routine is repeated for the *PTE B*. These replacements are continued until a chain of *PTE*s are replaced by each others. Similarly, the outer loop of Algorithm 2 controls and starts a new cycle of such replacements. Note that getPTE can be replaced with its Algorithm 1's version for synced accesses.

Algorithm 2 Swapping Overlapping Areas

```
1:  procedure SwapOverlap(vAdd1, vAdd2, pages)
2:      function findSwapPlace(i, j, pages)
3:          if i<j then return i + pages
4:          return i − j
5:      end function
6:      addIdx2 ← ⌊(vAdd2 − vAdd1)/|PAGE|⌋
7:      upCurIdx ← gcd(addIdx2, pages)
8:      for curIdx ← 0 ... upCurIdx do
9:          vAdd1cur ← vAdd1 + curIdx · |PAGE|
10:         pteCur ← getPTE(vAdd1cur)
11:         pteTemp ← pteCur
12:         k ← findSwapPlace(curIdx, addIdx2, pages)
13:         while k ≠ curIdx do
14:             vAdd1K ← vAdd1 + k · |PAGE|
15:             pteKTemp ← getPTE(vAdd1K)
16:             pteK ← pteTemp
17:             flush_tlb_page(vAdd1K)
18:             pteTemp ← pteKTemp
19:             k ← findSwapPlace(k, addIdx2, pages)
20:         pteCur ← pteTemp
21:         flush_tlb_page(vAdd1cur)
22: end procedure
```

In terms of data security, since all swapping operations take place within the same address space of an application, page permissions and access controls should not be a concern. If there is a need to prevent data breaches between threads, the system call can be extended to clean up memory after each swapping in multi-threaded applications. In a summary, due to the significant drop in memory access frequency, SwapVA could provide a zero-copying mechanism for GC operations in data-intensive applications.

IV. SVAGC: Integration of SwapVA for GC

In this section, we describe the integration of SwapVA into a GC to reduce its pause time and improve its performance. In order to accomplish this, we can replace the large sequential memory copy operations in the GC with the virtual address swapping calls, SwapVA. Algorithm 3 (see *MoveObject* procedure) shows how to utilize the SwapVA call in the primary copy operation of GCs, *MoveObject*, for data objects that are bigger than a configurable threshold.

There are a few prerequisites and adaptations to consider prior to applying SwapVA as a bulky copier. First, the candidate object and the next immediate one must be located in page-aligned addresses. At allocation time, therefore, an object larger than the threshold, which may be one or more physical pages, must be placed on the first free page in memory (see Algorithm 3 Line 16).

Second, to keep the consistency of memory layout after each SwapVA call, the system allocates the next object on

Algorithm 3 MoveObject, Memory Allocation, and Forwarding Address Calc.

```
1: procedure MoveObject(source, dest, length)
2:     pages ← ⌈length/|PAGE|⌉
3:     if pages ≥ Threshold_Swapping then
4:         SwapVA(source, dest, pages)
5:     else memmove(source, dest, length)
6: end procedure
7: function IFSwapAlign(object, address)
8:     if |object| ≥ Threshold_Swapping · |PAGE| then
9:         return align(address, |PAGE|)
10:    return address
11: end function
12: procedure allocMem(object, heap)
13:    newTop ← IFSwapAlign(object, heap.top)
14:    if newTop + |object| ≥ heap.end then
15:        GC()                          ▷ Call GC to free memory
16:    heap.top ← IFSwapAlign(object, heap.top)
17:    object.address ← heap.top
18:    heap.top += |object|
19:    heap.top ← IFSwapAlign(object, heap.top)
20: end procedure
21: procedure CalcNewAdd(object, compPnt)
22:    compPnt ← IFSwapAlign(object, compPnt)
23:    object.address ← compPnt
24:    compPnt += |object|
25:    compPnt ← IFSwapAlign(object, compPnt)
26: end procedure
```

the first free page adjacent to the current object position (see Algorithm 3 Line 19). It protects the next objects from side effects of applying the call. Finally, during the compaction phase, relocating objects to the new addresses, large objects need to be defined in a paged-aligned format (see Algorithm 3 Line 22 and 25).

Memory Fragmentation Issue: Memory address requirements of SwapVA could raise some issues related to memory fragmentation. Page-aligned objects may create some external fragmentation between a new allocated large object and previous small object. In heap memory allocation, objects are usually allocated from the thread's TLAB (Thread Local Allocation Buffer) to reduce synchronization issues between different threads at allocation time. If TLAB is exhausted or object is too big to be placed in TLAB, allocation will be failed and object will be allocated from a new TLAB or shared space of heap. Accordingly, an effective solution is to allocate large page-aligned objects from the end to the beginning of the TLAB and also smaller ones from the beginning to the end. As a result, all large and small objects are placed separately. When objects are too large and need to be allocated from the shared space of the heap, the vicinity of large objects is also guaranteed. This can remove the external fragmentation. Allocating new large objects with proper swapping thresholds can also help control the rate of fragmentation. For example, in our evaluations, by setting the threshold value to ten pages we could reduce possible internal fragmentation about less than 5% of heap size, statistically up to half a memory page could be wasted for every ten pages or more.

Multi-Core Scalability of SwapVA: Flushing TLB entries can increase memory access time on single-core systems since evicted virtual address mappings must be re-translated and re-cached. However, this cost per page is relatively low. For example, for each page, 4KiB of memory in *x86-64*, MMU wastes roughly a five-fold memory access time to walk through the page tables, find the physical address, and finally cache it in the core's TLB for further references. In this scenario, after updating a PTE, the corresponding TLB entry must be invalidated or flushed. The flushing costs may be manageable in a single-core system, but in multi-core environments they would rise dramatically since all TLBs of a process on all cores must be flushed. Besides, multi-core OSs utilize a mechanism called inter-processor interrupt (IPI) to execute a piece of code remotely on a specific core. OSs use IPI to flush some TLB entries of other cores. Nevertheless, IPI calls can be costly operations, affecting the system's performance and scalability. Therefore, to precisely calculate the cost of flushing in multi-core systems, we should combine the cost of all local core TLB flushes with their peer IPI calls. For instance, each invocation of the swapping system call on a typical eight-core CPU triggers seven subsequent IPIs that force other cores to clean their TLBs. As a result, it requires an overall eight local TLB flushing operations. For this purpose, the kernel would send IPIs to all online processors to clean their own TLBs. Thus, this would result in a huge impact on total execution time.

Moreover, there are some complications in the implementation of SwapVA to avoid races. For instance, its caller needs to be pinned during SwapVA execution on the current core and invoke kernel function, $flush\_tlb\_others$, to flush all other cores TLBs at the end of the system call. These constraints increase the cost of SwapVA adversely.

SwapVA should be used with caution in applications such as GCs. The copying process might update some PTEs and, in the meantime, be migrated to other cores. So, it could result in an inconsistent view of virtual address mappings cached in the TLBs on different cores. To ensure that applications, such as VMs, see the most recent updates of address mappings (PTEs) on each core, all cores' TLBs must be flushed after each change. This functionality can also be integrated into SwapVA implementation.

The following two complementary techniques can improve SwapVA's scalability. These techniques help SwapVA to reduce both costly IPIs and TLB flush requests.

The first technique, to be integrated in SwapVA, is achieved by sending many IPIs to flush all TLB entries that belong to the current process on other cores. Afterwards, the relevant TLBs are flushed locally without doing any global flushes. This approach will assist in reducing the cost of broadcasting requests after each TLB invalidation. It also removes redundant TLB flushes on other cores.

The second technique is similar to the first, but it is carried out on a much larger scale. This technique pins the entire compaction process in the same core throughout each GC cycle. At the beginning of the compaction phase, the process only broadcasts a flushing request toward all cores to invalidate its TLB entries (see Algorithm 4 Line 5). This assures us that other cores' processes refer to the updated PTEs. Next, during invocation of the swapping system calls and updating PTEs, only the current core TLBs are flushed.

Algorithm 4 Optimized Compaction Phase

```
1:  procedure CompactionOpt(liveObjects)
2:      pid ← getPID()                    ▷ find the process id
3:      cpuid ← getCPU()                  ▷ find the CPU id
4:      pin(cpuid, pid)                   ▷ pin the process
5:      flush_tlb_all_cores(pid)          ▷ flush TLB (all cores)
6:      while liveObjects.hasNext() do
7:          object ← liveObjects.next()
8:          objAdd ← object.currentAddress()
9:          newAdd ← object.newAddress()
10:         MoveObject(objAdd, newAdd, |object|)
11:     unpin(pid)                        ▷ unpin the process
12: end procedure
```

Therefore, by utilizing the optimized version of the compaction phase, as shown in Algorithm 4, the number of IPIs, and their subsequent flushes will be reduced from $\bar{l} \cdot c$ to c, c and $\bar{l}$ denote the core count and the average number of live swappable objects, respectively. So the gain is calculated as:

$$gain = \frac{\#IPI_{Unoptimized}}{\#IPI_{Optimized}} = \frac{\sum_{n=1}^{\bar{l}} c}{c} = \frac{\bar{l} \cdot c}{c} = \bar{l} \quad (2)$$

Fig. 9 shows the cost of running non-optimized SwapVA in a multi-core system. By limiting the broadcasting of both TLB flushes and IPI calls, the results also display the advantage of a scalable SwapVA implementation over a non-optimized one. Note that the number of live swappable objects is set to 100 in our evaluation.

TLB Shootdown Approaches: As a brief overview, some studies introduced techniques such as [22] to improve TLB shootdown by running them in parallel or [23] to track page accesses to prevent unnecessary issuance of TLB flushes. Even [24] introduced a timer-based self-flushing technique to reduce the cost. However, due to the lower cost

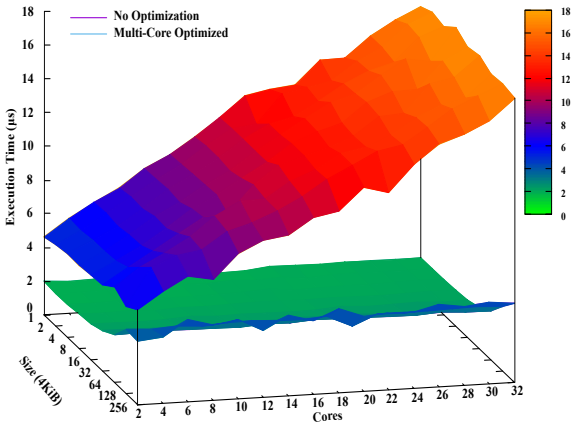

Fig. 9. Benefits of multi-Core optimizations to SwapVA (Tested on Intel® Xeon Gold 6130 CPU @2.10GHz with 192GB DRAM, DDR4 @2666MHz).

of implementation and lower complexity, we have adapted the pinning technique as tool to mitigate and control the cost of broadcasting TLB shootdowns and corresponding TLB flushes. We will demonstrate the versatility of this approach in our evaluation.

A. *The Applicability of SwapVA and its Optimizations*

In Table I, we summarize the applicability of SwapVA and its optimizations to different GC cycles/phases. The basic SwapVA call could be applied to all GC cycles/phases with various heap organizations, both generational and non-generational heaps. However, some other optimizations have their respective limitations. SwapVA aggregation is ideal in the compaction phase of a Full/Major GC since more data copying requests increase the chance of grouping SwapVA calls. However, since each copying operation is independent during the evacuation phase of concurrent GCs, the aggregation technique is not effective. Finally, the overlapping optimization is not applicable for the copying and evacuation phases of Minor or Concurrent GC, whereas source and destination have no shared addressable area. Note that since SVAGC has Full GC cycles, it utilizes all proposed optimizations.

TABLE I
THE APPLICABILITY OF SWAPVA AND OPTIMIZATIONS

GC (Phase) \ Optimization	SwapVA	Aggregation	PMD Caching	Overlapping
Full & Major (Compact, Moving)	✓✓	✓✓	✓✓	✓
Minor (Copying)	✓✓	✓	✓✓	–
Concurrent (Evacuation, Reloc.)	✓✓	–	✓✓	–

V. Evaluation

SwapVA, as a system call, has been implemented in Linux 4.17.0-rc2. We choose OpenJDK 15 and *Epsilon* memory allocator to integrate with SwapVA. Considering *Epsilon* works mainly as a simple memory allocator wrapped by a standard GC interface, it has been extended with a parallel version of *LISP2* algorithm (see section II) that is optimized by parallelized phases, same as ParallelGC. The swapping threshold in $MoveObject$ algorithm is set to ten pages as a break-even point that makes SwapVA more affordable than $memmove$. This threshold also determines the minimum size of large objects which is ten pages in our evaluations. As shown by Fig. 10, CPU performance and memory bandwidth can impact on threshold value and define it.

We have evaluated the performance of multiple garbage collection algorithms based on the SPECjvm2008 benchmarks designed for large (Big) objects, on average 1KB+ in size [20]. Some of these benchmarks are useful in the core of ML algorithms including *FFT* [25]–[28], *Sparse* (*SpMV*) [29]–[32], *SOR* [33], and *LU* (matrix factorization). To cover more cases, we also add some extra benchmarks from other suites such as JOlden, OpenJDK, and Spark-bench that have large objects. The benchmarks run in a multi-threaded paradigm to set up a sufficient workload for the evaluation. We run benchmarks by setting the heap size to 1.2× and 2× of minimum required size. We have also modified the default workloads of Sigverify, which have 1MiB objects, to include larger objects of 10MiB and 100MiB. The heap size and the number of threads are listed in Table II. All evaluations are completed on a machine with 32-Core Dual Intel® Xeon Gold 6130 CPU @2.10GHz with 192GB DRAM, DDR4 @2666MHz. To study the impact of object size on the performance of GCs, we have tuned the benchmarks with different object sizes. For example, $FFT.large$ is customized with two other different variants, including 1/8, 1/16 of the default input size. Similarly, we have made some customization with $Sparse.large$ and $SOR.large$. So, again we have created a version of $SOR.large$, ten times as large as its default input size. Moreover, the input size of *Bisort* and *Parallelsort* is set to 2M entries, and for *LRUCache* the cachable object size is selected from [1, 2M] bytes with 2K entries. *PR* benchmark also works on random graphs with 78K nodes and 780K edges.

A. GC Throughput and Pause Time

The first measured parameter is the throughput of the system based on the pause time. As shown by Fig. 11, the throughput improvement ranges from 3.44× ($Sparse.large/4$) to 33.3× ($Sigverify$). This can be explained by the fact that applications with both fewer and larger objects can benefit more from our implementation.

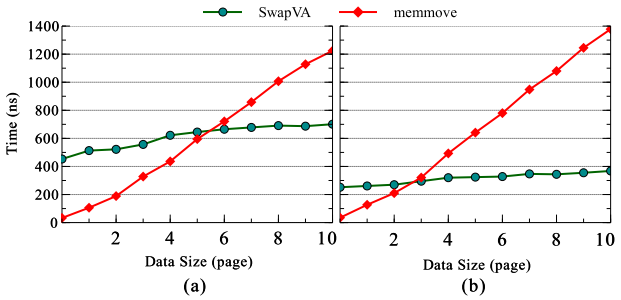

Fig. 10. Threshold value for SwapVA in different CPU/memory configurations (Tested by a single-threaded application). a) Intel® Xeon Gold 6130 CPU @2.10GHz with 192GB DRAM, DDR4 @2666MHz. b) Intel® Xeon Gold 6240 CPU @2.60GHz with 192GB DRAM, DDR4 @2933MHz.

TABLE II
Benchmarks Configuration

Benchmark	Suite	Threads	Heap (GiB)
FFT.large	SPECjvm2008	576	19.2 - 40
Sparse.large	SPECjvm2008	576	5 - 8.5
SOR.large	SPECjvm2008	32	51.5 - 85.8
LU.large	SPECjvm2008	224	3 - 5
Compress	SPECjvm2008	640	19 - 32
Sigverify	SPECjvm2008	256	28 - 56.7
CryptoAES	SPECjvm2008	96	5.2 - 8.67
PageRank (PR)	Spark	288	4 - 6.5
Bisort	JOlden	896	8 - 19.2
Parallelsort	OpenJDK	896	16 - 50
LRUCache	_	1	4.5

In these applications, the most expensive compaction and moving phases can take advantage of SwapVA calls. Accordingly, applications such as FFT with a small number of larger objects have an advantage over others, including $Sparse.large/4$, with numerous smaller ones.

The SVAGC throughput has also been evaluated against several state-of-the-art GC algorithms, including Shenandoah as a region-based concurrent and parallel pause-oriented collector [15], and ParallelGC as a matured generational throughput-oriented GC. Furthermore, since our focus is on evaluating Full GC performance, in particular

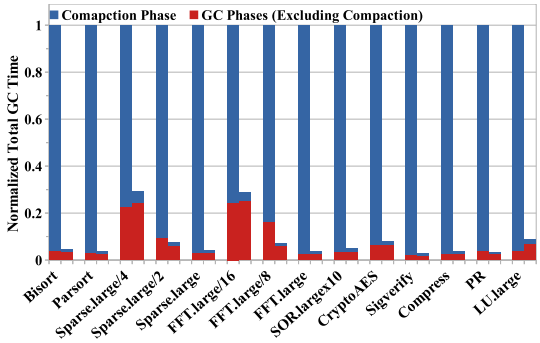

Fig. 11. Evaluation of GC time -/+ SwapVA on SVAGC (at 1.2× minimum heap); In each group left bar is the total GC time with just memmove, and the right one is the GC with our optimized moving/compaction phase including both SwapVA and memmove. Each bar is broken down into compaction phase (in blue), and all GC phases except compaction phase (in red).

on Large Objects (LO), and since G1GC does not regularly trigger the Full GC in our benchmarks, we did not include it in our analysis. As results in Fig. 12(a) show, the average latency of Full GC has been reduced 3.82× and 16.05× compared to ParallelGC and Shenandoah, respectively. As shown in Fig. 12(b), by increasing heap size the average latency has been reduced 2.74× and 13.62×, respectively, compared to ParallelGC and Shenandoah.

Finally, Fig. 13 (a) shows SVAGC has more improvement for maximum GC latency that is more important for pause sensitive applications. Accordingly, in SVAGC, the maximum pause time of Full GC has been reduced 4.49× and 18.25× in comparison to ParallelGC and Shenandoah, respectively. Of course, by increasing heap size to 2× minimum size, as shown by Fig. 13(b), these gains are slightly decreased to 3.60× and 12.24×, respectively. Therefore, larger heaps do not help ParallelGC and Shenandoah considerably to improve their performance against SVAGC. SVAGC does indeed maintain its performance gains since the hefty cost of moving (copying) LOs is the most influential factor in the compaction/moving phase. Furthermore, Shenandoah's copying phase is worst since it does not utilize the work-stealing mechanism and parallelism in its compaction (copying) phase.

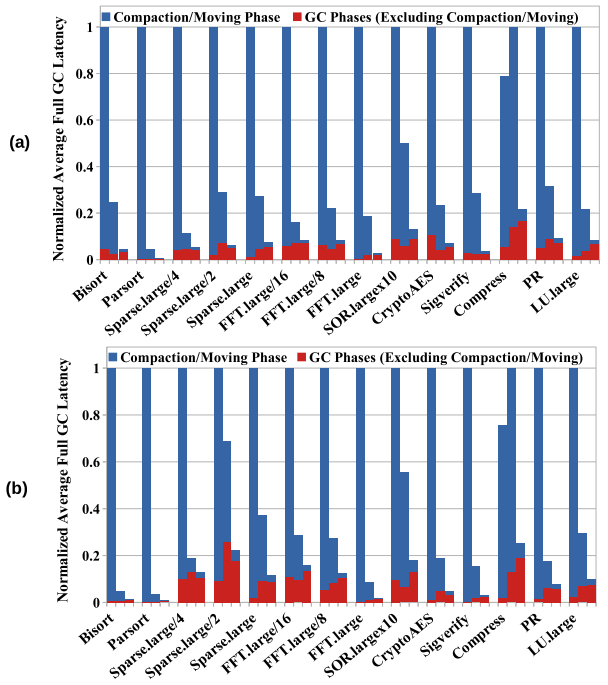

Fig. 12. Average latency of SVAGC vs. Shenandoah/ParallelGC at a) 1.2× minimum heap. b) 2× minimum heap. The first, second, and third bar of each group shows the average GC time of Shenandoah, ParallelGC, and SVAGC, respectively. Each bar is broken down into moving/compaction phase (in blue), and all other GC phases except moving/compaction phase (in red).

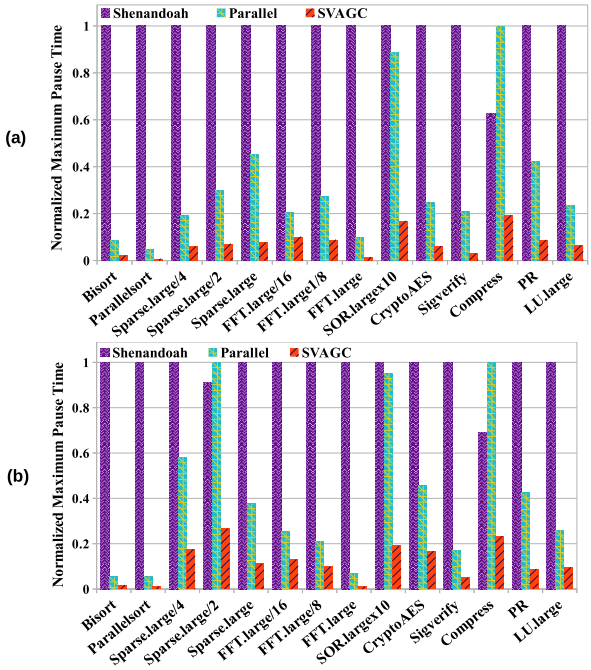

Fig. 13. SVAGC maximum latency vs. Shenandoah/ParallelGC at a) 1.2× minimum heap size. b) 2× minimum heap size.

B. Scalability

We have evaluated the scalability of SwapVA and SVAGC on multi-core systems. The tests include a synthesized single-threaded benchmark, a *LRU caching* as memory-bound application, to create and access a range of small to large objects randomly.

Since the optimized version of the compaction phase reduces the number of IPIs, it would minimize the overhead of running multiple VMs on multi-core systems concurrently. Moreover, as each IPI might trigger a TLB flush on another core, limiting the IPIs in the pinned version could further reduce the delays of calls. So more GC time gains through IPI reductions are expected. In addition, very small bandwidth requirements of SwapVA for copying objects during GC cycles, decrease the pressure on memory subsystem and scales better in comparison to naive memory copy operations. As shown by Fig. 14, although the application execution time surges by 327.5%, the GC time gradually increases by 52%.

C. Cache Performance and Application Time

Flushing TLBs could have a potential side effect on system performance. To evaluate the impact of SwapVA on cache and TLB performance, we run benchmarks using the Linux *perf* tool, to sample the cache and DTLB misses. Table III, shows some advantages of SwapVA over memmove. The results on cache misses indicates that SwapVA leads to less cache pollution than memmove. It results in a better cache performance and fewer cache misses. DTLB performance is also improved by SwapVA

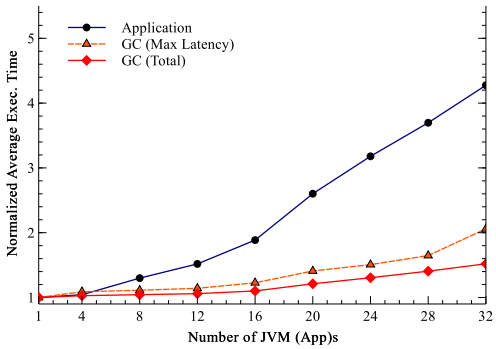

Fig. 14. Scalability of SVAGC in single/multi-JVM setting (run on the 32-Core configuration).

TABLE III
CACHE & DTLB MISSES EVALUATION AT 1.2× (2×) MINIMUM HEAP

Benchmark	Cache Misses (%)		DTLB Misses (‰)	
	memmove	SwapVA	memmove	SwapVA
Bisort	91.4(89.8)	89.4(89.8)	0.12(0.05)	0.05(0.04)
Parsort	89.94(83.5)	82.1(81.7)	1.25(0.72)	0.45(0.57)
Sparse.large/4	74.47(73.83)	71.83(71.79)	0.43(0.37)	0.29(0.28)
Sparse.large/2	87.40(86.16)	85.1(85.2)	0.95(0.54)	0.49(0.33)
Sparse.large	95.09(94.30)	92.32(93.22)	4.6(1.6)	0.68(0.4)
FFT.large/16	13.82(12.19)	10.70(10.66)	0.38(0.3)	0.29(0.26)
FFT.large/8	18.70(17.59)	16.58(16.84)	1.12(0.99)	0.89(0.9)
FFT.large	78.94(78.40)	78.27(78.18)	136.9(182.1)	139.9(183.4)
SOR.largex10	96.19(96.32)	96.44(96.52)	3.17(1.41)	1.21(1.1)
LU.large	96.62(96.90)	95.26(96.43)	6.51(3.01)	1.53(1.22)
CryptoAES	88.32(87.17)	85.90(86.57)	0.08(0.05)	0.03(0.03)
Sigverify	96.87(95.94)	94.97(95.47)	1.41(0.32)	0.15(0.08)
Compress	79.30 (75.25)	72.24(72.65)	0.504(0.346)	0.328(0.291)
PR	91.49 (91.37)	89.87(90.60)	1.68(1.05)	0.44(0.34)
min	13.82(12.19)	10.70(10.66)	0.08(0.05)	0.03(0.03)
max	96.87(96.32)	96.44(96.52)	136.9(182.1)	139.9(183.4)
geomean	69.32(67.48)	65.71(65.99)	1.28(0.74)	0.52(0.44)

since virtual copies do not pollute the DTLB excessively due to fewer memory accesses and VA-to-PA translations.

Finally, we have measured the benefits of SVAGC to the overall application time. Admittedly, the improvement to the application execution time reflects the benefits of SVAGC to the GC time. Note that the overall benefits from SwapVA are achieved despite the cost of using TLB flushing, which incurs an extra cost on application threads after GC cycles.

The results in Fig. 15 indicate, applying SwapVA and its optimizations lead to the system throughput improvement, ranging from 15.2% (*CryptoAES*) to 86.9% (*Sparse.large*). Furthermore, the results in Fig. 16(a) show SVAGC, in mentioned benchmarks, could outperform the throughput of ParallelGC and Shenandoah by an average of 30.95% and 37.27%, at 1.2× minimum heap configuration, respectively. As Fig. 16(b) shows at the 2× minimum heap the results are decreased to 15.26% and 16.79%, respectively. The reason is that the larger the heap size, the lower the frequency of costly Full GCs. Briefly, the variation in application throughput improvement among benchmarks

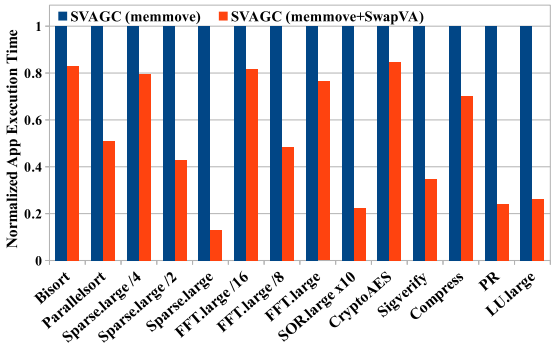

Fig. 15. Application throughput of SVAGC at 1.2× minimum heap size.

can be attributed to the frequency and time at which GC phases are invoked. In other words, if the ratio of memory (de)allocation to application computation increases, then GC cycles are more likely to be run. Therefore, in more memory-intensive applications such as SOR, $Sparse$, and $Sigverify$, our application throughput improvement would be higher than more compute-intensive ones such as $CryptoAES$ and FFT.

VI. Related Works

GC Optimization Efforts. Numerous applications and frameworks running on top of VMs depend greatly on GC performance. Researchers have tried to analyze and improve GC algorithms in different cases. Bruno et al. [4]–[6] proposed a pre-tenuring mechanism to reduce excessive object copying in generational GCs. Nguyen et al. [7] optimized a GC for big data application using the temporal

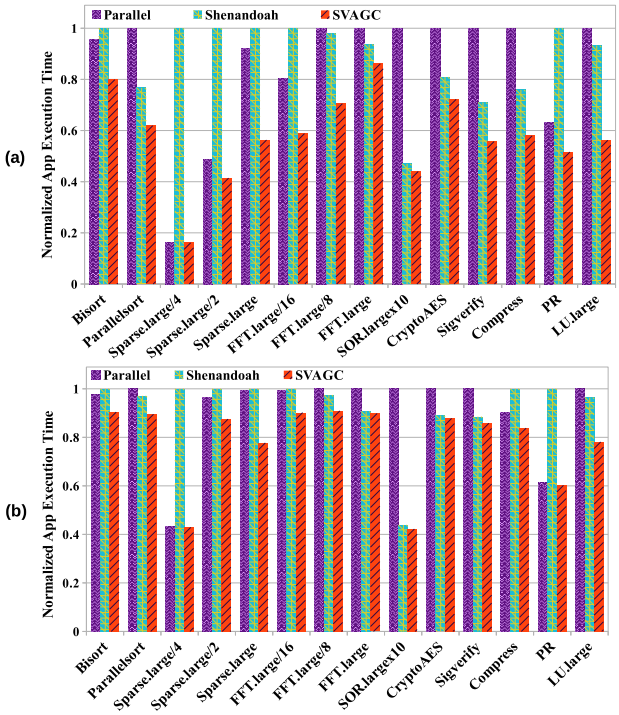

Fig. 16. Throughput of SVAGC vs. Shenandoah/ParallelGC at a) 1.2× minimum heap size. b) 2× minimum heap size.

locality and life-span of objects in analytical applications to develop a region-based GC called Yak. Morikawa et al. [34] presented a adaptive technique, heap and bitmap scanning, for reducing the time of marking live objects and GC time in LISP2 GC algorithms. Su et al. [8] recognized and addressed some issues on PS to utilize all multi-threading capacity of the system. Li et al. [9], [10] pointed out some deficiency in the compaction phase of Full GC and introduced a memory cache to resolve dependency between copying threads. To reduce the GC pause time, some algorithms support multi-threading and concurrent GC phases, such as marking and/or copying phases. A few examples of initiatives designed to reduce the application pause time are Flood et al. in Shenandoah [15], Detlefs et al. in G1 [12], and Tene et al. [13] in C4. Meantime, several researchers tried to remove GCs, including Broom [35], from VMs by using run-time facilities. There were also many attempts to create a run-time support such as Deca, by Shi et al. [36], to optimize GC for some frameworks such as Apache Spark. Wegiel et al. [37] proposed a non-moving approach using VM remapping, however, it has some synchronization issues in multi-threaded applications, and it requires STW compactions at VM address exhaustion.

Our approach uniquely and directly addresses the cost of copying/moving operations, which could be expensive in most GC algorithms. We use the virtual address swapping technique to perform a zero-copying operation toward replacing naive copy/move operations. The approach is orthogonal to any GC algorithm, including parallel and concurrent ones, and any heap organization, including (non)generational. Therefore, it can be adopted in GC compaction/evacuation phases to make them more affordable. Column-based databases, analytics frameworks with large buffers, and scientific computing applications working with large matrices are all examples of applications with large objects. Our technique allows large objects to be returned to conventional heap spaces, preventing LOS issues. Moreover, a recent trend of combining DRAM and Non-Volatile Memory, as a hybrid memory, results in novel heap designs and GC algorithms [38], [39]. Therefore, hybrid heaps and GCs could use our improvement to reduce frequency of writing cycles and mitigate wear-out issues. Also, GC implementations may increase their performance by replacing costly write operations of NVMs with our zero-copying ones on collecting large objects.

Zero-Copying Efforts. To reduce the cost of both copying and sharing data, previous studies such as *COW* [40]–[42], have focused on the copy by value semantic. In addition, some mechanisms such as mmap and pmap [40] have introduced as a by-reference sharing semantic between processes. These tools can be employed as an inter-process facility to achieve a zero-copying ambition. However, our work is focused on single-process address space in either single or multi-threaded applications. As well as, all previous efforts generally are either directed toward sharing physical address space by virtual page remapping [43] to communicate messages or addressed transferring data between processes and kernel by sharing buffers [44]. Though, in our effort, we have focused on an intra-process zero-copying mechanism based on data swapping operations in a single process address space. So, as an advantage, there is no need to get involved in data protection and security issues. Besides, it eliminates the need to invalidate CPU caches which is necessary for inter-process virtual address remapping implementations.

VII. Conclusion

GC compaction and evacuation phases could be the most costly operations in garbage collectors, especially in running applications with large working sets. Therefore, applying any optimization on them would reduce the GC's overhead, regardless of heap organization and their designated algorithms. Our results show that employing the proposed mechanism to swap virtual addresses could reduce the copying phase's operation cost, mainly in applications containing large objects. It also promises some specific improvements for the scalability of GCs in multi-core/multi-VM settings.

References

[1] C. Flood, D. Detlefs, N. Shavit, and X. Zhang, "Parallel garbage collection for shared memory multiprocessors," in *Symposium on Java Virtual Machine Research and Technology Symposium*, 2001.

[2] B. Hayes, "Using key object opportunism to collect old objects," in *Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 33 – 46, 1991.

[3] Y. Bu, V. R. Borkar, G. H. Xu, and M. J. Carey, "A bloat-aware design for big data application," in *ISMM'13*, 2013.

[4] R. Bruno and P. Ferreira, "POLM2: Automatic profling for object lifetime-aware memory management for hotspot big data applications," in *the 18th ACM/IFIP/USENIX Middleware Conference*, pp. 147 – 160, 2017.

[5] R. Bruno, L. P. Oliveira, and P. Ferreira, "NG2C: Pretenuring garbage collection with dynamic generations for hotspot big data applications," in *ACM SIGPLAN Notices*, 2017.

[6] R. Bruno, D. Patricio, J. Simão, L. Veiga, and P. Ferreira, "Runtime object lifetime profiler for latency sensitive big data applications," in *EuroSys'19*, 2019.

[7] K. Nguyen, L. Fang, G. Xu, B. Demsky, S. Lu, S. Alamian, and O. Mutlu, "Yak: A high-performance big-data-friendly garbage collector," in *the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, 2016.

[8] K. Suo, J. Rao, H. Jiang, and W. Srisa-an, "Characterizing and optimizing hotspot parallel garbage collection on multicore systems," in *EuroSys'18*, 2018.

[9] H. Li, M. Wu, and H. Chen, "Analysis and optimizations of java full garbage collection," in *APSys'18, Article No.: 18*, 2018.

[10] H. Li, M. Wu, B. Zang, and H. Chen, "ScissorGC: Scalable and efficient compaction for java full garbage collection," in *the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pp. 108 – 121, 2019.

[11] I. Gog, J. Giceva, Schwarzkopf.M., K. Vaswani, D. Vytiniotis, G. Ramalingan, M. Costa, D. Murray, S. Hand, and M. Isard, "Broom: Sweeping out garbage collection from big data systems," in *the 15th USENIX conference on Hot Topics in Operating Systems (HOTOS'15)*, 2015.

[12] D. Detlefs, C. Flood, S. Heller, and T. Printezis, "Garbage-first garbage collection," in *the 4th International Symposium on Memory Management (ACM)*, pp. 37 – 48, 2004.

[13] T. Gil, I. Balaji, and W. Michael, "C4: The continuously concurrent compacting collector," in *the International Symposium on Memory Management*, 2011.

[14] P. Liden and S. Karlsson, "JEP 333: ZGC: A scalable low-latency garbage collector (experimental)." http://http://openjdk.java.net/jeps/333, 2018. Accessed: 2021-03-19.

[15] C. H. Flood, R. Kennke, A. Dinn, A. Haley, and R. Westrelin, "Shenandoah: An open-source concurrent compacting garbage collector for openjdk," in *PPPJ'16*, 2016.

[16] M. Hicks, L. Hornof, J. T. Moore, and S. M. Nettles, "A study of large object spaces," in *Proceedings of the 1st International Symposium on Memory Management*, ISMM'98, (New York, NY, USA), pp. 138 – 145, Association for Computing Machinery, 1998.

[17] S. M. Blackburn and K. S. McKinley, "Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'08, (New York, NY, USA), pp. 22 – 32, Association for Computing Machinery, 2008.

[18] D. Chen and K. Ken, "The memory bandwidth bottleneck and its amelioration by a compiler," in *the International Symposium on Parallel and Distributed Processing*, pp. 181 – 189, 2000.

[19] R. Jones and R. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley, 1st ed., 1996.

[20] P. Lengauer, V. Bitto, H. Mössenböck, and M. Weninger, "A comprehensive java benchmark study on memory and garbage collection behavior of dacapo, dacapo scala, and specjvm2008," ICPE'17, (New York, NY, USA), pp. 3 – 14, Association for Computing Machinery, 2017.

[21] J. Singer, G. Kovoor, G. Brown, and M. Luján, "Garbage collection auto-tuning for java mapreduce on multi-cores," in *Proceedings of the International Symposium on Memory Management*, ISMM '11, (New York, NY, USA), pp. 109 – 118, Association for Computing Machinery, 2011.

[22] N. Amit, A. Tai, and M. Wei, "Don't shoot down tlb shootdowns!," in *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys'20, 2020.

[23] N. Amit, "Optimizing the tlb shootdown algorithm with page access tracking," in *the 2017 USENIX Annual Technical Conference (USENIX ATC'17)*, 2017.

[24] A. Awad, A. Basu, S. Blagodurov, Y. Solihin, and G. Loh, "Avoiding tlb shootdowns through self-invalidating tlb entries," pp. 273 – 287, 09 2017.

[25] K. Chitsaz, M. Hajabdollahi, N. Karimi, S. Samavi, and S. Shirani, "Acceleration of convolutional neural network using fft-based split convolutions," *arXiv preprint arXiv:2003.12621*, 2020.

[26] J. Lee-Thorp, J. Ainslie, I. Eckstein, and S. Ontanon, "Fnet: Mixing tokens with fourier transforms," *arXiv preprint arXiv:2105.03824*, 2021.

[27] S. Li, K. Xue, B. Zhu, C. Ding, X. Gao, D. Wei, and T. Wan, "Falcon: A fourier transform based approach for fast and secure convolutional neural network predictions," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 8705 – 8714, 2020.

[28] S. Lin, N. Liu, M. Nazemi, H. Li, C. Ding, Y. Wang, and M. Pedram, "Fft-based deep learning deployment in embedded systems," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1045 – 1050, IEEE, 2018.

[29] E. Nurvitadhi, A. Mishra, and D. Marr, "A sparse matrix vector multiply accelerator for support vector machine," in *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pp. 109–116, IEEE, 2015.

[30] W. Xiao, J. Xue, Y. Miao, Z. Li, C. Chen, M. Wu, W. Li, and L. Zhou, "Tux2: Distributed graph computation for machine learning," in *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, (USA), pp. 669 – 682, USENIX Association, 2017.

[31] S. Narang, E. Elsen, G. Diamos, and S. Sengupta, "Exploring sparsity in recurrent neural networks," *arXiv preprint arXiv:1704.05119*, 2017.

[32] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, (Cambridge, MA, USA), pp. 1135 – 1143, MIT Press, 2015.

[33] O. L. Mangasarian and D. R. Musicant, "Successive overrelaxation for support vector machines," *IEEE Transactions on Neural Networks*, vol. 10, no. 5, pp. 1032–1037, 1999.

[34] K. Morikawa, T. Ugawa, and H. Iwasaki, "Adaptive scanning reduces sweep time for the lisp2 mark-compact garbage collector," pp. 15 – 26, 06 2013.

[35] M. Maas, K. Asanović, T. Harris, and J. Kubiatowicz, "Taurus: A holistic language runtime system for coordinating distributed managed-language applications," in *ASPLOS'16*, 2016.

[36] X. Shi, Z. Ke, Y. Zhou, H. Jin, L. LU, X. Zhang, L. He, Z. Hu, and F. Wang, "Deca: A garbage collection optimizer for in-memory data processing," in *ACM Transactions on Computer Systems, Vol. 36, No. 1*, 2019.

[37] M. Wegiel and C. Krintz, "The mapping collector: Virtual memory support for generational, parallel, and concurrent compaction," *SIGPLAN Not.*, vol. 43, pp. 91 – 102, mar 2008.

[38] S. Akram, J. B. Sartor, K. S. McKinley, and L. Eeckhout, "Write-rationing garbage collection for hybrid memories," *SIGPLAN Not.*, vol. 53, pp. 62 – 77, jun 2018.

[39] A. M. Yang, E. Österlund, J. Wilhelmsson, H. Nyblom, and T. Wrigstad, "Thingc: Complete isolation with marginal overhead," in *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2020, (New York, NY, USA), pp. 74 – 86, Association for Computing Machinery, 2020.

[40] D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson, "Tenex, a paged time sharing system for the PDP-10," *Commun. ACM*, vol. 15, pp. 135 – 143, Mar. 1972.

[41] R. Fitzgerald and R. F. Rashid, "The integration of virtual memory management and interprocess communication in accent," *ACM Trans. Comput. Syst.*, vol. 4, no. 2, pp. 147 – 177, May 1986.

[42] D. Murphy, "Origins and development of TOPS-20." https://opost.com/tenex/hbook.html, 1989,1996. Accessed: 2021-05-13.

[43] P. Druschel and L. L. Peterson, "Fbufs: A high-bandwidth cross-domain transfer facility," in *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pp. 189 – 202, Dec 1993.

[44] Y. A. Khalidi and M. N. Thadani, *An Efficient Zero-Copy I/O Framework for UNIX*. Sun Microsystems, Inc., 1995.