Charon: Specialized Near-Memory Processing Architecture for Clearing Dead Objects in Memory

Jaeyoung Jang[†] Jun Heo[‡] Yejin Lee[‡] Jaeyeon Won[‡] Seonghak Kim[‡] Sung Jun Jung[‡]

Hakbeom Jang[†] Tae Jun Ham[‡] Jae W. Lee[‡]

[†]Sungkyunkwan University, Suwon, Korea [‡]Seoul National University, Seoul, Korea
{jaey86, hakbeom}@skku.edu
{j.heo, yejinlee, jerrywon, ksh1102, miguel92, taejunham, jaewlee}@snu.ac.kr

ABSTRACT

Garbage collection (GC) is a standard feature for high productivity programming, saving a programmer from many nasty memory-related bugs. However, these productivity benefits come with a cost in terms of application throughput, worst-case latency, and energy consumption. Since the first introduction of GC by the Lisp programming language in the 1950s, a myriad of hardware and software techniques have been proposed to reduce this cost. While the idea of accelerating GC in hardware is appealing, its impact has been very limited due to narrow coverage, lack of flexibility, intrusive system changes, and significant hardware cost. Even with specialized hardware GC performance is eventually limited by memory bandwidth bottleneck. Fortunately, emerging 3D stacked DRAM technologies shed new light on this decades-old problem by enabling efficient near-memory processing with ample memory bandwidth. Thus, we propose Charon[1], the *first* 3D stacked memory-based GC accelerator. Through a detailed performance analysis of HotSpot JVM, we derive a set of key algorithmic primitives based on their GC time coverage and implementation complexity in hardware. Then we devise a specialized processing unit to substantially improve their memory-level parallelism and throughput with a low hardware cost. Our evaluation of Charon with the full-production HotSpot JVM running two big data analytics frameworks, Spark and GraphChi, demonstrates a 3.29× geomean speedup and 60.7% energy savings for GC over the baseline 8-core out-of-order processor.

CCS CONCEPTS

• **Computer systems organization → heterogeneous (hybrid) systems**; **Special Purpose Systems**.

[1]Charon, in Greek mythology, is the ferryman who carries souls of the dead.

MICRO-52, October 12–16, 2019, Columbus, OH, USA

ACM ISBN 978-1-4503-6938-1/19/10…$15.00
https://doi.org/10.1145/3352460.3358297

KEYWORDS

Garbage collection, Near-memory processing, Domain-specific architecture, Memory management, Java Virtual Machine

ACM Reference Format:
Jaeyoung Jang, Jun Heo, Yejin Lee, Jaeyeon Won, Seonghak Kim, Sung Jun Jung, Hakbeom Jang, Tae Jun Ham, and Jae W. Lee. 2019. Charon: Specialized Near-Memory Processing Architecture for Clearing Dead Objects in Memory. In *the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3352460.3358297

1 INTRODUCTION

Garbage collection (GC) is a form of automatic memory management technique which is widely utilized in many programming languages like Java, C#, JavaScript, and Python. With GC, programmers do not need to explicitly deallocate an object after its use. Instead, a runtime garbage collector automatically identifies data objects that cannot be accessed in the future (i.e., garbage) and automatically performs deallocation to reclaim memory spaces used by such objects. With its ability to automatically deallocate memory, GC naturally improves the productivity of programmers and can completely eliminate or substantially reduce the memory-related bugs (e.g., memory leaks, dangling pointer, etc.).

Unfortunately, the advantages of GC often come with noticeable performance and power/energy cost. Even worse, GC is prohibitively expensive in big data analytics manipulating a large number of objects scattered across a large memory region. Multiple sources report that GC can account for some 50% of execution time in memory-intensive big data analytics [6, 14, 18, 49, 50]. Moreover, latency-sensitive applications suffer GC-induced long tail-latency [13, 37], significantly degrading quality of service and management cost in distributed cloud. Note that the amount of GC overhead is closely related to the application's working set size and the available heap size. Considering the recent trend of slowdown in DRAM technology scaling and rapid increase in dataset size and compute parallelism, the cost GC is expected to continuously increase in the future.

To counter the unavoidable, expected increase in the GC overhead, it is critical to improve the throughput of GC. Unfortunately, GC — a process of identifying live objects through graph traversals and migrating them to contiguous memory region — is a very memory-intensive workload that involves many ill-suited operations for general-purpose processors. For example, general-purpose

processors achieve limited memory-level parallelism (MLP) due to their limited instruction window and load/store queue size. At the same time, they have limited bandwidth to off-chip memory which can often become a bottleneck. In fact, we observed that the average IPC of modern Intel Xeon core running garbage collection is below 0.5 on various big-data analytics workloads, indicating that the modern CPU is not very effective for GC. Thus, our research focuses on exploiting the opportunities in specialization and near-memory computation to overcome challenges of GC in general-purpose processors.

The idea of accelerating GC in hardware has been explored for a long time since the introduction of GC by the Lisp programming language in the 1950s. However, they have had limited impact for various reasons. For example, some proposals target a specific language (e.g., Lisp [45], Smalltalk [64]), a specific hardware (e.g., FPGA [5], specialized memory [61, 67]), or a class of algorithms (e.g., reference counting [27, 61, 66]), to have narrow coverage. Others attempt to implement GC fully in hardware [38–41, 60, 62, 66] lack flexibility. For example, a state-of-the-art GC accelerator by Maas et al. [38] hard-wires a relatively simple mark-sweep algorithm, and hence cannot fully offload the popular generational GC, which requires a copying collector between semispaces (generations). Also, full offloading is invasive as it often requires major changes to the processor, thus incurring a high cost for hardware design and verification. Our analysis shows that a small number of tiny *primitives* dominate the total GC time, and we argue for offloading those primitives, not the entire GC. This approach is more *future-proof* as those primitives likely outlive continuously evolving GC algorithms. Furthermore, GC algorithms are often bottlenecked by memory bandwidth. Therefore, without addressing this bandwidth problem, building specialized hardware is only a half solution. Fortunately, emerging 3D stacked DRAM technologies like HBM [25] and HMC [9] uncover intriguing opportunities for this decades-old problem by enabling efficient processing on near-memory logic and providing ample memory bandwidth.

Thus, we architect, design, and evaluate Charon, a near-memory accelerator specialized for specific operations in GC. By unlocking massive memory-level parallelism through specialized hardware designs and exploiting the abundant memory bandwidth available at the near-memory logic layer, Charon accelerates key primitives of GC derived from HotSpot JVM [24], the most popular production JVM today, and enables general-purpose processors to offload GC very efficiently with minimal changes to the processor. In summary, this paper makes the following contributions:

- We perform a detailed analysis of the GC behavior in big data processing frameworks using the production-grade HotSpot JVM.
- We identify key algorithmic primitives of GC and present specialized hardware processing units to substantially improve memory-level parallelism and throughput of such primitives.
- We prototype Charon on HotSpot JVM to demonstrate its effectiveness with a production-grade GC algorithm.
- We evaluate Charon with two large-scale data analytics frameworks, Spark and GraphChi, using a detailed cycle-level simulator (executing over 770 billion instructions in regions of interest), and demonstrate that Charon achieves a 3.29× speedup and 60.7% energy savings over the baseline 8-core out-of-order processor.

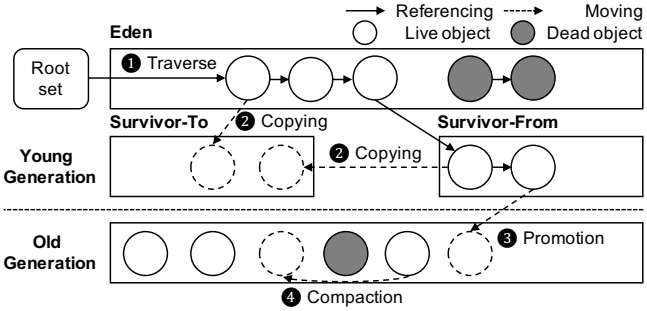

Figure 1: Overview of ParallelScavenge GC in HotSpot

2 BACKGROUND

Garbage Collection Algorithms. Garbage is heap allocated objects that will not be used in the future. These objects are not reachable by any chain of pointers from the *root set*, which consists of objects that are accessible from outside the heap (e.g., stack variables/pointers, global variables). A garbage collection (GC) is a process of reclaiming these garbage objects that are no longer in use by the program. In a language like C/C++, this process is handled manually by programmers; on the other hand, in many languages like Java, GC automatically handles this process.

There exist different types of GC algorithms with different goals. For example, Java offers Parallel Compacting Collector [43], a throughput-oriented collector, which focuses on the GC throughput. Such a collector incurs a complete stop of all application threads (*mutator* threads in a GC term) so that cores can devote their resources to GC threads. In contrast, there are concurrent garbage collectors (e.g., Concurrent-Mark-Sweep (CMS) in HotSpot JVM), which allow application threads to continue in parallel with GC threads. Such a concurrent garbage collector typically incurs less pause time compared to the throughput-oriented garbage collector. However, the concurrent garbage collector incurs overheads due to i) synchronization to maintain a consistent memory state with the application threads and ii) interference coming from sharing hardware resources with concurrent threads of different natures (i.e., GC and original application) and achieves lower overall throughput than the throughput-oriented one.

Generational Garbage Collection. Generational GC is a standard algorithm that uses multiple generations to take advantages of the weak generational hypothesis [63], which assumes that most objects in heap have a short lifetime while there exist a relatively small set of objects that remain live for a long time. Typically, generational collectors divide heap into a Young and an Old generation. The idea is to minimize the number of full garbage collection operation on entire heap by performing lightweight garbage collection on a relatively small space (i.e., Young generation) where most objects with short lifetime are collected.

Figure 1 presents the operations of the ParallelScavenge, which is a popular generational, throughput-oriented collector in HotSpot JVM. Objects are initially allocated to Eden space reserved for new objects. When this Eden space is filled up and fails to allocate a new object, a minor GC (MinorGC) is triggered. Starting from the root set, the collector ❶ traverses the object graph consisting of live objects. Specifically, MinorGC traverses live objects in the Eden and

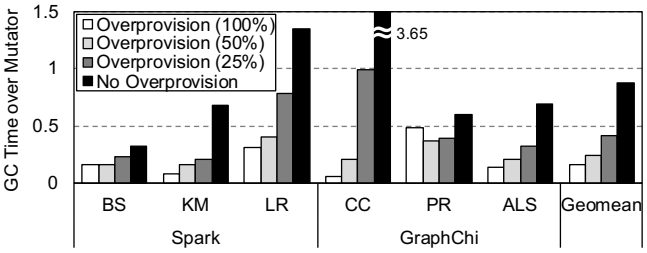

Figure 2: GC overhead normalized to mutator time (for useful work) in big data processing workloads over varying heap size

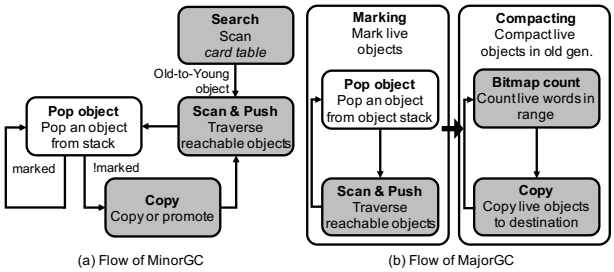

Figure 3: Simplified execution flow of GC

one of the two Survivor spaces (From) space, and ❷ copies them to the other empty Survivor (To) space, or ❸ promotes them to Old generation if the objects survive a certain number of MinorGCs in Young generation. Then, MinorGC cleans up Eden and Survivor (From) space, and designates the current From space as a To space (and vice versa). Eventually, the Old generation will be filled with promoted objects and this will trigger a major GC (MajorGC) event. While MajorGC traverses live objects in a way that is similar to that of MinorGC, it also triggers ❹ compaction to reduce heap fragmentation.

3 ANALYSIS OF GC INEFFICIENCY

3.1 GC Overheads in Big Data Applications

Today's big data processing applications with large heap often suffer noticeable performance and power/energy cost due to GC. Unlike traditional Java applications, big data processing applications often exhaust the heap due to massive amount of objects and large volumes of data manipulation such as join or shuffle [6]. Even worse, these objects cannot be reclaimed until all operations are completely done, leaving a large number of long-lived objects to be traversed during GC. While it is common to use most of the physical memory space in big data processing applications, traversing a large number of live objects becomes the main bottleneck to spend many CPU cycles for GC rather than useful computation.

Figure 2 shows the performance impact of GC over actual computation time on a modern Intel i7 processor [23]. We first find the minimum heap size that enables an application to finish without an out-of-memory (OOM) error, which results from the insufficient heap size. Then, we overprovision the heap by 25%, 50%, and 100% to observe the GC overhead across varying degree of memory over-provisioning. Even with a substantial overprovisioning (e.g., allocating 2× memory than what is actually required), GC slows down the application by 15%. And this overhead easily explodes as the memory size approaches towards the minimum, actually required heap size. In fact, the time spent on GC can exceed 365% of the actual application runtime (i.e., mutator runtime). Furthermore, the trend of rapidly increasing dataset size and the compute parallelism is likely to increase the minimum required memory size faster than the increase of physical memory size; this indicates that the GC is likely to take an even larger portion of the runtime in the coming future.

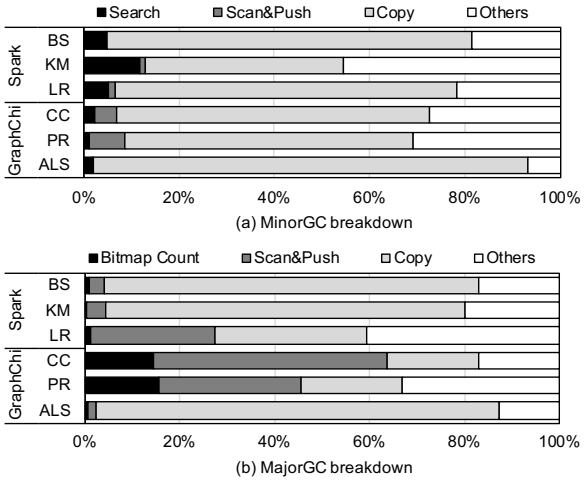

Figure 4: Runtime breakdown of GC

3.2 GC Execution Time Breakdown

This section presents runtime breakdowns for MinorGC and MajorGC algorithms, which enable us to identify few key operations that accounts for the large portion of the total GC time. For the experiments here, we use an 8-core Intel i7-4790 Processor [23] 3.60 GHz and profile the multithreaded portion of parallel, throughput-oriented GC collector (i.e., ParallelScavenge) on OpenJDK 7 HotSpot JVM [24].

Minor Garbage Collection (MinorGC). Figure 3(a) shows a simplified operation flow of MinorGC. As the MinorGC starts, the collector pushes the objects in the root set (e.g., local stack variables, global variables, etc.) into the *object stack*, where objects to be traversed reside. After this step, the collector pops an object from the stack (*Pop object*) and checks if this object is already processed. If not, the collector marks it as live and copies to the other survivor space or promotes it to the Old generation space if it is aged enough (*Copy*). Finally, the collector checks if this object references other objects (*Scan*) and pushes those referenced objects to the stack (*Push*). This process is repeated until the stack is completely drained.

One complication here is that there may be live objects residing in the Young generation, which are only referenced by an object(s) in Old generation. Such objects are not connected to the default root set for the MinorGC in the object graph and thus these objects need to be tracked with a separate metadata (called *card table* in HotSpot JVM). In addition to the root set mentioned in the above

paragraph, the card table is scanned (*Search*) at the beginning of MinorGC and objects referenced by other objects in Old Generation are pushed to the stack (*Push*) as well.

MinorGC Runtime Breakdown. Figure 4(a) shows how much time is spent on each operation in MinorGC. Both Spark and GraphC hi applications spend most of GC execution time on a small number of operations such as *Search*, *Scan&Push*, and *Copy*. These operations account for 71.42% (up to 81.48%) and 78.23% (up to 93.10%) of total MinorGC time on Spark and GraphChi, respectively. Specifically, for Spark, *Copy* and *Search* accounts for the most of MinorGC time. Similarly, the GC on GraphChi spends most time on *Copy* and *Scan&Push*. The fact that *Copy* takes most time on MinorGC is natural since big data analytics applications often manipulate data in large-chunk partitions (i.e., RDDs in Spark and shards in GraphChi). Since each object in such applications is large, relatively less time is spent on traversal itself and much larger time is spent on *Copy*.

Major Garbage Collection (MajorGC). Figure 3(b) shows a simplified MajorGC operation flow. MajorGC in HotSpot JVM mainly consists of two distinct phases: marking, and compaction[2]. As in MinorGC, the MajorGC starts with the root set objects in the object stack. In the marking phase, the collector pops an object from the object stack (*Pop object*) and checks if it is already processed. If not, the collector marks it as live, and scans and pushes all references (*Scan&Push*) like in MinorGC, except that no copy operations happen here.

In the compacting phase, the collector copies the live objects scattered across heaps to sequential, contiguous memory space. For each live object, the collector needs to calculate the destination location it should be moved. This is basically done by summing up the sizes of live objects that will be copied to the left of the current object (e.g., objects that are currently located at the left of the current object) when we view the heap space as a single large linear space. For this computation, two bitmap data structures, called begin and end bitmaps (*Bitmap Count*), are utilized in HotSpot JVM. In these bitmap structures, a single bit represent the 64-bit heap space (e.g., each bitmap is 256MB in size to cover the entire 16GB heap space). A set bit on the begin bitmap indicates that the corresponding heap space for the bit is the starting address of a live object. Similarly, a set bit on the end bitmap indicates that the corresponding heap space for the bit is the end address of the live object. By counting the distance between those two bits, the size of that live object can be obtained as well. Utilizing this structure, the collector identifies the destination address for an object and then the object is copied to the address (*Copy*). This process is repeated until all live objects (that needs to be moved) are copied. After the compaction, the heap is densely packed on the left side, while leaving a large, empty block on the right side.

MajorGC Runtime Breakdown. Figure 4(b) shows how much time each application spends on MajorGC operations. We also observe that both Spark and GraphChi applications spend most GC execution time on a few key operations just like in MinorGC. For both workloads, *Copy* takes a significant portion of the total MajorGC time. These operations account for 74.13% (up to 82.93%) and 79.06% (up to 87.21%) of the total MajorGC time for Spark and GraphChi, respectively.

Spark and GraphChi demonstrate different application-level behaviors in that *Scan&Push* takes a large portion in GraphChi. Spark tends to allocate large objects to memory with few references, while GraphChi allocates many long-lived objects with many references. Thus, Spark spends a lot more time on copying the objects, while GraphChi spends more time on other primitives such as *Scan* and *Bitmap Count*. ALS in GraphChi is an exception as the algorithm is different from the other two in that it takes a very large matrix data as a single object, which results in a huge copy.

3.3 Key GC Primitives for Offloading

The runtime breakdown in Figure 4(a) and 4(b) suggests that the total GC time is dominated by a handful of key *primitives*. This section takes a closer look at these primitives from a near-memory processing perspective.

Figure 4(a) shows three small primitives (i.e., *Search*, *Copy*, *Scan& Push*) dominate the MinorGC time in HotSpot JVM. Similarly, an overlapping set of three primitives (i.e., *Scan&Push*, *Bitmap Count*, *Copy*) accounts for over 75% of the MajorGC time. Most of these primitives perform memory operations without much computation. Unfortunately, a general-purpose processor is not well-suited for this type of workloads for its limited MLP due to the limited instruction window and load/store queue size. For example, *Scan&Push* and *Bitmap Count* primitives traverse the object graph sequentially with limited parallelism. Thus, GC algorithms often utilize multithreading to improve throughput.

Even in a case where a general-purpose processor can achieve high MLP, its performance is often limited by the off-chip memory bandwidth. Even worse, the key primitives of GC often do not have temporal or spatial locality. For example, a *Copy* primitive often touches a large memory region without any temporal reuse. A *Scan&Push* primitive — which traverses the list of referenced objects from a given object and pushes them to the stack — involves indirect memory accesses, which lead to poor locality (i.e., most of the fetched cache lines remain unused). With such data accesses with low data locality, a cache hierarchy of a general purpose processor cannot effectively alleviate memory latency and bandwidth bottleneck.

This motivates us to offload those key primitives to the near-memory logic layer of a 3D stacked DRAM. The stacked memory uncovers intriguing opportunities for GC offloading with abundant memory bandwidth that can be fully utilized by specialized processing units to maximize MLP and/or minimize operation latency. However, not all operations can benefit from offloading. For example, we found that offloading other operations in GC like *traverse linked list* gives relatively small benefits because of limited parallelism and latency-bound characteristics of the linked list traversal. Also, other operations like *allocate* and *check mark* in MajorGC are essentially single atomic instructions whose potential benefits from offloading are outweighed by the overheads due to their small offloading granularities.

Excluding those operations, we propose to offload three key primitives for each of MinorGC (*Copy*, *Search*, and *Scan&Push*) and MajorGC (*Copy*, *Bitmap Count* and *Scan&Push*). The next section

[2] Technically, there is a summary phase between the marking and the compaction phase but this phase takes very little time (less than 0.03% of the MajorGC time) and thus not a target for our accelerator.

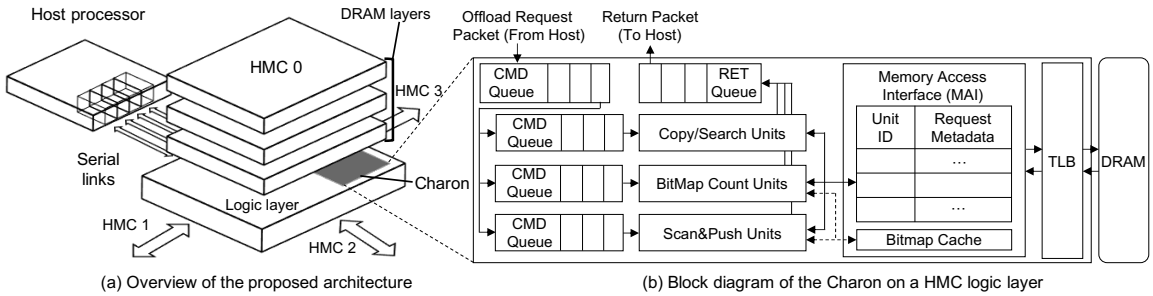

Figure 5: Charon overview

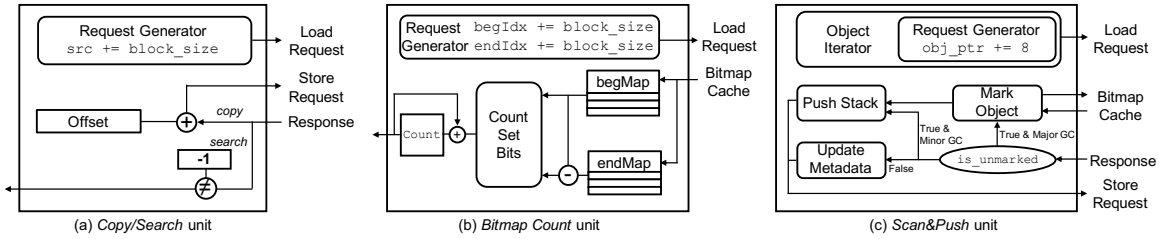

Figure 6: Hardware block diagram of each processing unit

provides descriptions of these primitives and how we optimize them for specialized processing units implemented on a near-memory processing logic layer.

4 CHARON ARCHITECTURE

4.1 Overview

Figure 5 is an overview of Charon architecture. We assume Hybrid Memory Cube (HMC) [9] as the baseline platform for its high internal bandwidth and energy efficiency, although non-HMC platforms can be flexibly supported by placing Charon elsewhere (e.g., memory controller, buffer-on-board [10]). In our setup, the host processor is directly connected to a single HMC cube, and other HMC cubes are connected to each other with a certain topology so that cubes can communicate each other without going through the host. While we use four HMC cubes connected in star topology, multiple HMC cubes are connected around the central cube as shown in Figure 5(a) throughout the paper, note that our architecture is not necessarily tied to a specific topology. For each HMC cube in the system, there is a logic layer beneath the stack of DRAMs. We implement Charon in the logic layer of each cube as shown in Figure 5(b). We base our design on ParallelScavenge in HotSpot JVM, but the primitives can be employed to accelerate other GC algorithms.

Host-Charon Interface. Charon provides two intrinsics for the host to interface. The first one is `initialize()`, which is called once at program launch. It sets constants and addresses of globally accessed data structures, such as start addresses of the heap and bitmap. Those configuration values are passed to memory mapped registers for each processing unit. The second one generates an offloading request and takes the following form:

val `offload`(**val** type, **addr** src, **addr** dst, **val** arg)

Once invoked, the HMC controller generates a packet, which is forwarded to the destination cube based on the type and address of the request. The offload request packet is 48B in size and consists of i) standard HMC header/tail (16B) including a destination cube *id*, ii) type of offloaded primitive (4 bits), and iii) two addresses (16B), and iv) extra operands (up to 124 bits). Once the offload request packet reaches the destination cube through the existing inter-HMC routing logic, it is first buffered in the command queue and then forwarded to the appropriate per-primitive command queue as in Figure 5(b). Unless all processing units are busy, the offloading request packet is moved to the available processing unit, and the unit will start execution. During the execution of an offloaded primitive in Charon, the host thread remains blocked until the offloaded primitive returns a packet back. This return packet is 32B in size when the response contains a return value; if not, it is 16B.

Memory Accesses from Processing Units. Processing units located at the logic layer access both their local and remote stacks. When a processing unit needs to access memory, it passes the address of the memory location to the *memory access interface* (MAI) (of the cube where the processing unit resides) along with its unit *id*, and an optional request metadata, which a processing unit wants to buffer in the MAI until the response returns. MAI finds an empty space in its request buffer and stores the unit *id* and the request metadata. MAI then issues an access request — whose request tag is the index of the request buffer — to memory, which will be routed to the appropriate destination cube depending on the request address. Once the request is finished, the response packet is handled by MAI, which retrieves the requested unit *id* and metadata, and forwards the response packet along with the request metadata to the requester. The role of MAI is similar to what MSHR (Miss Status Handling Register) does in host cores. Note that all the memory accesses from MAI go through virtual-to-physical address translation via an accelerator-side TLB (detailed in Section 4.6). Also,

```
1 void copy(ByteAddr *src, ByteAddr *dst, int size)
2   for(i=0; i<size; i++)
3     *(dst+i) = *(src+i);
4 bool search(ByteAddr *start, ByteAddr *end)
5   for(i=start; i<end; i+= block_size)
6     if(*i != -1)
7       return true;
8   return false;
```

Figure 7: Pseudocode of *Copy* and *Search* primitives

processing units send clflush to the host cache hierarchy (for both reads and writes) to avoid leaving a stale copy of the data in the cache or retrieving a stale copy of the data from memory. However, not all memory accesses trigger a host cache probe. For example, no clflush is necessary while executing *Bitmap Count* because i) all memory accesses are reads from the bitmap and ii) it is never updated by the host-side GC code.

4.2 Copy/Search Unit

Primitive. Figure 7 shows a pseudocode for both *Copy* and *Search* primitives sharing the same processing unit. A *Copy* primitive is used to move objects from one space of the heap to another. Specifically, in MinorGC, a *Copy* primitive moves objects from Eden and one Survivor space (From) to the other Survivor space (To) or Old generation. In MajorGC, a *Copy* primitive is used to compact regions. In addition to a *Copy* primitive, the unit can also perform a *Search* operation, which is used during MinorGC to check the existence of Old-to-Young objects (described in Section 3.2) within the specified range (Line 5). This unit receives two addresses (source and destination) and an integer (size) for *Copy* and two addresses (start and end of the range) for *Search*. When the host processor offloads either a *Copy* or *Search* primitive, it is scheduled to a cube that houses the source (*Copy*) or the start address (*Search*). This is to exploit an abundant internal bandwidth of HMC.

Implementation and Optimization. Figure 6(a) shows a block diagram of the *Copy*/*Search* unit. To achieve the best performance it is crucial to maximize MLP (i.e., the number of in-flight memory requests). Since the *Copy* primitive (Line 2) and *Search* are embarrassingly parallel, a whole memory copy/search operation can be executed in parallel.

Exploiting this abundant parallelism, the unit starts to send read requests at a 256B granularity (maximum granularity supported by HMC) every cycle as soon as it receives an offloading command packet from the host. This continues as long as i) the MAI can accept the requests and ii) end condition is not triggered. When a load response returns, it either issues a store request for *Copy* or performs the comparison for *Search*. Note that increasing MLP does not always result in an increase in throughput due to the limited memory bandwidth on a conventional memory system. In contrast, Charon exploits the huge internal bandwidth of the stacked DRAM to achieve greater speedups.

4.3 Bitmap Count Unit

Primitive. This primitive is heavily used in the compacting phase of MajorGC. During compaction, the collector relocates live objects to a new location. *Bitmap Count* is a primitive used to find the new

```
1 int live_words_in_range(ByteAddr *range_start, ByteAddr *range_end)
2   BitAddr *begMap = range_start;
3   BitAddr *endMap = range_start + OFFSET;
4   int num_bits = (range_end - range_start) * 8;
5   int begIdx = 0, endIdx = 0, count = 0;
6   while (begIdx < num_bits)
7     if(begMap[begIdx] == 1)
8       endIdx = begIdx;
9       endIdx++;
10      while (endIdx < num_bits)
11        if(endMap[endIdx] == 1)
12          count += endIdx - begIdx + 1;
13          begIdx = endIdx;
14          break;
15        endIdx++;
16      if(endIdx == num_bits)
17        begIdx = num_bits;
18    begIdx++;
19  return count;
```

Figure 8: Pseudocode of *Bitmap Count* primitive

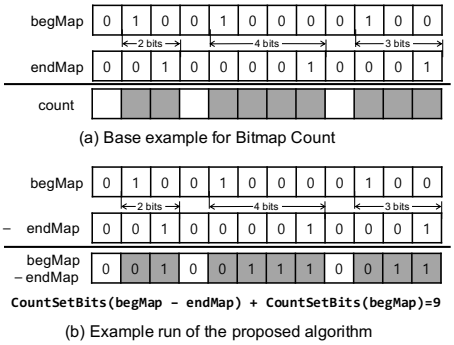

Figure 9: *Bitmap Count* primitive example

location by summing up the size of live objects within a certain memory range (`live_words_in_range` in HotSpot JVM (Figure 8)). This primitive reads two bitmaps: `begMap` and `endMap` (Line 2 and 3). A set bit in the `begMap` and `endMap` represents the start and end location of an object, respectively. As shown in Figure 9(a), the number of bits between a pair of set bits in `begMap` and `endMap` represent the size of a live object (in 8-byte words). What this primitive computes is the sum of those words occupied by the live objects within the specified range.

The unit receives two addresses marking the start and end of the range. These addresses are used for `begMap`, and the corresponding addresses in `endMap` can be derived by adding a constant `OFFSET` to it (Line 3). This constant is configured at program launch as static. This primitive is scheduled to the cube on which the bitmap address falls to exploit an abundant internal bandwidth of HMC like the *Copy* primitive.

Implementation and Optimization. The original software version simply iterates over `begMap` and `endMap` at a bit granularity (shown in Figure 8), which is very slow.

Charon optimizes this primitive, and Figure 6(b) shows our implementation. We first modified the algorithm (explained in the next paragraph) to be much more efficient. Besides, the processing unit identifies the exact amount of data that it needs to read at the beginning and issues memory requests as soon as the unit starts.

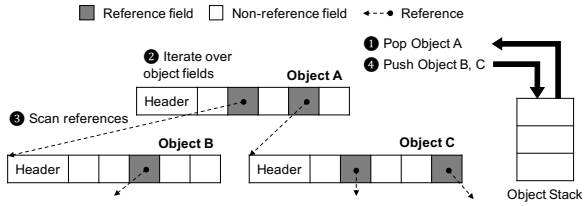

Figure 10: Example of object traversal

The processing unit further improves performance by employing a bitmap cache (detailed in Section 4.5) for the accesses to a small range of the bitmap frequently with ample temporal locality. According to our evaluation, the bitmap cache has a hit rate of about 90%. By doing so, the unit not only hides the long memory latency but also significantly reduces the latency of the primitive itself.

Optimized Algorithm. The processing unit utilizes an optimized algorithm based on the following expression.

```
CountSetBits(begMap-endMap) + CountSetBits(begMap)
```

It first subtracts `endMap` from `begMap`, where both are interpreted as binary numbers. Then, we count the number of set bits in the resulting binary number, which is equal to `CountSetBits(begMap-endMap)`. We also count the number of set bits in the `begMap` and add it to the previous result to obtain the final outcome. Figure 9(b) illustrates this with an example. Here, we assume a simple case, where both `begMap` and `endMap` have 3 set bits. Intuitively, subtracting `endMap` from `begMap` yields all 1's between paired set bits in `begMap` and `endMap`, except for the position of the 1 in `begMap`. In this example, counting the total set bits of the outcome gives us a bit count that is 3 less than the outcome of the original algorithm in Figure 8 as the set bits in `begMap` are not accounted for. To compensate for this, we add one for every pair of set bits. This is equal to the `CountSetbits(begMap)`. Note that our implementation does handle corner cases (i.e., where the number of 1's differ between `begMap` and `endMap`), but the descriptions of how to handle these cases are omitted due to limited space.

4.4 Scan&Push Unit

Primitive. This primitive performs object graph traversal, which is one of the most common operations in GC. As shown in Figure 10, this primitive scans an object's fields and pushes non-static references (i.e., reachable objects) to the object stack. By keeping track of object graph recursively, the collector identifies all the live objects in the heap. This primitive is utilized in both MinorGC and MajorGC.

In MinorGC, this unit covers `push_contents` in HotSpot JVM as shown in Figure 11. When MinorGC starts, the collector ❶ pops an object from the object stack (`minor_stack`). Then the collector marks the object as live and copies it to one of the two Survivor spaces (i.e., To). Then this unit ❷ iterates inside the object and ❸ scans the references one by one. If the loaded object is not traversed (marked) yet (i.e., `is_unmarked` is `True`), the unit ❹ pushes the object to the `minor_stack` to process it later (Line 11). Otherwise, the unit only updates the object metadata (i.e., *card table* for Old-to-Young object) and skips the push (Line 13). In this way, the collector recursively processes the `minor_stack` until it is empty,

```
1  // each type has a distinct iterate strategy
2  void iterate_object(ObjectPtr[] references)
3    for(objptr: references)
4      if(MinorGC)
5        push_contents(objptr);
6      else // MajorGC
7        follow_contents(objptr);
8  void push_contents(Object *objptr)
9    Object obj = *objptr;
10     if(is_unmarked(obj))
11       minor_stack.push(obj);
12     else
13       update_metadata(obj);
14 void follow_contents(Object *objptr)
15   Object obj = *objptr;
16   if(is_unmarked(obj))
17     mark_obj(obj); // atomic Read-modify-write
18     major_stack.push(obj);
```

Figure 11: Pseudocode of *Scan&Push* primitive

and eventually, all live objects will be marked as live and copied over to the Survivor space.

In MajorGC, this unit covers `follow_contents` in HotSpot JVM used in a marking phase as shown in Figure 11 again. The collector ❶ pops an object from the object stack (`major_stack`). Then the unit ❷ traverses object's fields and ❸ scans all reachable objects in it. If they are not marked yet (i.e., `is_unmarked` is `True`), the unit sets the corresponding bit in the bitmap (i.e., `mark_obj`) (Line 17) and ❹ puts the newly marked object to `major_stack` (Line 18). In this way, the collector recursively processes `major_stack` until it is totally drained, and eventually, all live objects will be marked.

Since there are 15 different class metadata types in HotSpot JVM (e.g., instanceKlass, objArrayKlass, etc.) which has distinct class metadata layout, scanning inside the objects requires different iteration strategies for each type. For the simplicity of the design, our design focus on handling a few dominant types (i.e., data class types). The host processor only needs to provide a type of the corresponding object and `start`/`end` addresses of the object's metadata region. Then the unit chooses the right iteration strategy based on the type.

Charon always schedules this primitive to the central cube (i.e., HMC 0 of Figure 5). It is because this unit has random data access patterns (e.g., reading the contents of objects referenced by the input object) and thus processing it at the central location often minimizes the overall delay and the bandwidth usage.

Implementation and Optimization. As shown in Figure 11, an object iterator generates a stream of sequential load requests (Line 3) from a set of references of inside the object layout in question. In the original control flow, the unit waits until the memory load request for a referenced object arrives. Then, it performs other memory operations (Line 11 or 13, Line 17 and 18) on arrival of the request. This type of indirect memory access sequences often result in very poor performance on a conventional CPU with the limited instruction window size. Specifically, the dependent instructions to the initial load easily clog the instruction window if it misses at a cache, which often leads to a core stall during the cache miss. In contrast, Charon amortizes the latency of the initial loads by exploiting MLP. The processing unit knows how

many memory load requests there will be to fetch referenced objects (Line 3) based on `start` and `end` addresses. Thus, once started, it generates a batch of memory load requests— every one cycle. When the first response comes back, it performs appropriate actions based on the response, performing either `minor_stack.push(obj)` (Line 11) or `update_metadata(obj)` (Line 13) for MinorGC and `is_unmarked(obj)` (Line 16) for MajorGC. Finally, in MajorGC, if a response from `is_unmarked(obj)` comes back, it performs `mark_obj(obj)` and `major_stack.push(obj)` (Line 17 and 18).

4.5 Bitmap Cache

Reading a specified range of the bitmap from memory is an essential part of *Bitmap Count*. There are benefits of caching the entire bitmap for two reasons. First, *Bitmap Count* is often called in a loop of multiple bitmap ranges or objects scanning iterations, performed for all objects in the Old generation. In this case bitmap accesses in *Bitmap Count* frequently happen during MajorGC to demonstrate good temporal locality. Second, a specified range of *Bitmap Count* typically overlaps with the previous range and is small enough to fit in the cache.

Besides, `mark_obj` operations (Line 17 in Figure 11) can also benefit from a bitmap cache. It performs an atomic read-modify-write (RMW) on a single 8B block in the bitmap. Although the processing unit only requires a single block from the bitmap, it always has to fetch 16B as the minimum memory access granularity to cause an overfetching problem without a cache.

Thus, we add a small writeback cache (8KB, 8-way, 32B block size) dedicated to the bitmap accesses, used by both *Bitmap Count* and *Scan&Push* units. Since *Bitmap Count* happens during the compacting phase in MajorGC (only reads), and *Scan&Push* happens during the marking phase in MajorGC, there is no chance of both accessing the bitmap cache simultaneously. Also, we flush this cache right after completing either of the two primitives in MajorGC for coherence.

4.6 System-level Issues and Discussion

Applicability. We base our analysis and design on the throughput oriented GC (ParallelScavenge) in HotSpot JVM. However, many of these primitives represent fundamental operations of GC and thus they are also commonly utilized in other collectors. For example, Table 1 shows the applicability of Charon primitives to popular collectors in HotSpot JVM. As shown in the table, primitives like *Copy* and *Scan&Push* are key operations of most GC algorithms and thus applicable to latency-oriented Concurrent Mark Sweep (CMS) GC or latency/throughput co-optimized Garbage-First (G1) GC. The processing unit for *Bitmap Count* can also be used in G1 GC scheme with slight modifications to the G1 code, where it scans the bitmap to identify the state of the entire heap.

While the Charon primitives are readily applicable, concurrent GCs pose additional challenges to maintain a consistent memory view between mutators and the collector. To this end, a short sequence of code (called *barrier* in JVM terminology) is executed at every read or write in the mutator, which incurs a significant runtime overhead (e.g., 15% for ZGC [70]). While orthogonal to Charon, this issue should be also carefully addressed for high efficiency [38, 42].

	Copy/ Search	Scan& Push	Bitmap Count	Remarks
ParallelScavenge	✓✓	✓✓	✓✓	High throughput
G1	✓✓	✓✓	✓	Low latency
CMS	✓✓	✓✓	×	No compaction

✓✓: *applicable as is*, ✓: *applicable with minor fix*, ×: *not applicable*

Table 1: Applicability of Charon Primitives

Programmer Effort. Offloading the Charon primitives requires minimal programmer effort, which is a major advantage over full offloading. It takes only 37 lines of modifications from the original HotSpot JVM code to replace the three primitives in Figure 7, 8, and 11 with the Charon intrinsic calls (plus initialization). This makes it much easier to port, verify, and deploy Charon for various GC algorithms.

Virtual Memory and Multi-Process Support. The processing units in Charon require an efficient mechanism for virtual-to-physic al address translation. For this purpose, we utilize huge pages, NUMA support and memory locking, which are well supported by the mainstream Intel/AMD architectures.

At application launch, JVM allocates 1GB huge pages for the entire heap space and pins down those pages using `mlock()` system call. To facilitate this, HotSpot JVM supports configurable options like `-XX:+UseLargePage` and `-XX:+AlwaysPretouch`. The huge pages are interleaved over different cubes using `numa_alloc_onnode()`, which was originally used for allocating memory at a specific NUMA node. The remaining memory space (e.g., code, off-heap) uses 4KB pages with the conventional demand paging. Note that nearly all API functions for controlling memory allocation are already introduced to Linux for NUMA support. Then, Charon can leverage the virtual memory system for efficient address translation and protection by maintaining the just enough number of duplicate TLB entries on the DRAM side to cover those pinned-down huge pages. Note that it is a common practice to pre-allocate a large heap within the physical memory size (with no oversubscription) in a managed runtime like JVM [21, 53, 68], which otherwise would significantly increase the cost of demand paging and GC. Thus, pinned-down pages are maintained throughout the execution of the program with no TLB misses or page faults.

To support multiple JVM processes Charon counts on the standard protection mechanism from virtual memory. The mainstream x86 architecture already supports distinct process identifiers (PCID [4, 11]) in TLB, thus it is a straightforward extension. Note that Charon currently does not allow oversubscription of physical memory as an attempt to pin down a huge page would fail beyond the capacity of physical memory, which effectively serves as an admission control mechanism for Charon.

Scaling Capacity of 3D Stacked DRAM. While we evaluate the proposed architecture with a single processor system paired with four HMCs, nothing prevents us from applying the proposed architecture for multi-processor, multi-stacked DRAM systems when higher memory capacity is desired. Recent studies [31, 51, 54] demonstrate that HMC-like stacked DRAM can provide terabytes of capacity by interconnecting multiple processors and DRAM modules, and the latency penalty and/or bandwidth contention can be effectively alleviated with efficient topologies and data placement for big data applications [71]. For a less scalable technology like

HBM the memory capacity can be scaled by utilizing conventional DDRx DIMM as backing storage. In such a system Charon can be integrated as the CPU-side accelerator sitting near the memory controller. Section 5.2 demonstrates the effectiveness of Charon as a CPU-side accelerator.

Scalability of Charon. An increase in HMC count naturally allows us to put more processing units so that the GC throughput increases even further as the memory size increases. Potential bottlenecks among Charon structures are a bitmap cache and TLBs. Currently, Charon employs a 4-cube star topology and thus we utilize a single bitmap cache at the center cube. However, it is possible to scale this structure by employing well-known proposals such as owner cache [15], where each cube has a cache slice to hold its local data only. Similarly, a TLB slice at each cube can cache only those mappings associated with its local pages. Note that a memory request can be sent to the right cube just with the virtual address (VA) as OS maps a VA region to a specific cube with `numa_alloc_onnode()`. We demonstrate the scalability of Charon with an increasing number of primitive units in Section 5.2.

Effect on Host Cache. Our design flushes the cache at the beginning of a GC so that our near-memory processing units can obtain its data from memory, rather than a host cache. This can technically cause a degradation of the application thread's performance after GC; however, at least in our target workloads, the amount of memory regions touched during the GC substantially exceeds the cache size (e.g., for running *Copy*), and thus it is very unlikely for a useful cache line (for application threads) to remain in the cache after GC finishes. Note that such a bulk flush can fully utilize HMC bandwidth and thus incur a relatively small overhead. For example, flushing 24MB LLC takes only 300μs with 80GB/sec HMC bandwidth while the average GC duration in our experiments exceeds hundreds of milliseconds.

5 EVALUATION

5.1 Methodology

Evaluation Model. We extend zsim [58] to model the performance impact of our proposal. Table 2 shows the configuration of the host processor and the HMC main memory in zsim. We use memory channel interleaving [row:col:bank:rank:ch] for DDR4 and [row:cube[31:30]:row:col:bank:rank:vault] for HMC to use 1GB huge pages effectively. For the power and energy evaluation of the host processor, we integrate McPAT [34] to the zsim. For the power, energy, and area evaluation of Charon hardware structures, we implement such structures using Chisel3 [7], which were functionally verified using realistic test inputs, and synthesize them using Synopsys Design Compiler with TSMC 40nm standard cell library. Lastly, we use CACTI [65] under 45nm technology to estimate the power/area cost of some buffer/queue structures (e.g., command queue, bitmap cache, etc.) in Charon.

Workloads. We run Spark 2.1.0 [69] and GraphChi 0.2.2 [33] (with HotSpot JVM from OpenJDK 1.7.0 [52]) on our proposed framework. We carefully pick a different set of applications from each framework with very different object characteristics and hence very different GC behaviors. Specifically, we run three machine-learning workloads with Spark (i.e., naive-bayes, k-means clustering, and logistic regression), two graph algorithms and one machine learning algorithm based workload with GraphChi (i.e., connected components, PageRank, and alternating least square). For evaluation, we specifically focus on how Charon improves GC performance and energy consumption. Thus, we set a region of interest (ROI) for the GC events only that occur during the run. Our experiments use default HotSpot JVM heap sizing policy (Young:Old = 1:2) and set the max heap size of each application to 1.25-2× of the minimum heap size with which the application can reliably run without encountering out-of-memory (OOM) error. Table 3 summarizes workloads with the corresponding input dataset and heap size.

Host Processor	
Core	8 × 2.67 GHz Westmere OoO core
	36-entry IW/ 128-entry ROB / 4-way issue
TLB	L1I/D 64-entry per core
	Shared L2 1024-entry
L1I/D Cache	32KB, 4-way, 3-cycle / 32KB, 8-way, 4-cycle
L2 Cache	256KB, 8-way, 12-cycle
L3 Cache	8MB, 16-way, 28-cycle
DDR4 Main Memory System	
Organization	32GB, 2 channels, 4 ranks per channel, 4Gb 8 banks per rank
Timing	tCK=0.937ns, tRAS=35ns, tRCD=13.50ns tCAS=13.50ns , tWR=15ns, tRP=13.50ns
Bandwidth/Energy	34GB/s (17GB/s per channel) / 35pJ/bit [35]
HMC Main Memory System	
Organization	32GB, 4 cubes, 32 vaults per cube
Timing	tCK = 1.6ns, tRAS = 22.4ns, tRCD = 11.2ns tCAS = 11.2ns, tWR = 14.4ns, tRP = 11.2ns
Bandwidth/Energy	320GB/s per cube / 21pJ/bit [59]
Serial Links	Total 80GB/s per link, 3ns latency
Charon Configuration	
Copy/Search	8 units (2 units per cube)
Bitmap Count	8 units (2 units per cube)
Scan&Push	8 units (8 units on a single cube)
Bitmap Cache	8KB, 8-way, 32B block size
MAI / TLB	8KB, 32B block size / 32 entries per cube

Table 2: Architectural parameters for evaluation

5.2 Performance Results

Overall Speedups. Figure 12 compares the throughput of the GC across four different platforms: the host processor with the conventional DDR4 memory system (DDR4), the host processor with the hybrid memory cube (HMC), the host processor paired with Charon in the logic layer of the hybrid memory cube (Charon), and the imaginary, ideal scenario where the host processor is paired with an ideal offloading device which can execute the offloaded primitives in zero cycle (Ideal). As shown in the figure, the host processor can achieve a speedup of 1.21× simply by replacing its DDR4-based memory system to HMC-based memory system which offers more off-chip memory bandwidth. However, this does not mean that the host processor is fully exploiting benefits of the HMC. Even though the HMC memory system offers more bandwidth, the host system has limited memory level parallelism (despite having 8 cores) and thus cannot fully utilize the available bandwidth. Even further, although the HMC memory system offers higher off-chip bandwidth from its high-speed serial links, the host processor alone cannot utilize the abundant internal bandwidth (i.e., bandwidth

	Workload	Input	Heap
Spark	Bayesian Classifier (BS)	KDD 2010 [28, 29]	10GB
	k-means Clustering (KM)	KDD 2010 [28, 29]	8GB
	Logistic Regression (LR)	URL Reputation [36]	12GB
Graphchi	Connected Components (CC)	R-MAT Scale 22 [44]	4GB
	PageRank (PR)	R-MAT Scale 22 [44]	4GB
	Alternating Least Squares (ALS)	Matrix Market Format (15000x15000) [55]	4GB

Table 3: Workloads

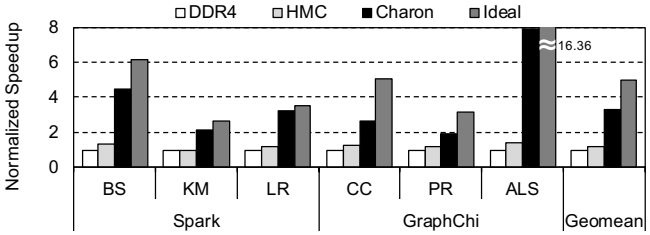

Figure 12: Normalized GC performance of Charon compared with the host CPU-only execution

between the logic layer and its DRAM stacks) an HMC can provide. As shown in the third bar, Charon overcomes this limitation and fully exploits these benefits to achieve 3.29× average speedup over the host system with DDR4 DRAM or 2.70× average speedup over the host system with HMC DRAM. Comparing Charon and the ideal scenario performance shows that Charon indeed handles the offloaded primitives in a very efficient way and takes very little time to process them. Note that Charon's speedup varies across different workloads as well as the type of GCs. This is because the portion of offloaded primitives can be substantially different for different workloads as demonstrated in Section 3. For example, Charon benefits best on ALS since the *Copy* primitive accounts for the large portion of the runtime which Charon benefits the most.

Bandwidth Analysis. Figure 13 (bar graph) shows the bandwidth usage of GC across different platforms. Without Charon, the only way for the host system to access memory is to utilize the off-chip links whether they are paired with the conventional DDR4 memory system or the HMC memory system. On the other hand, with Charon, each processing unit in the logic layer can utilize high-bandwidth TSV (Through-Silicon via) to access data in memory with higher internal bandwidth. In addition, Charon processing units also access remote cube through serial links connecting cubes. While this traffic does not exploit the internal high-bandwidth provided by TSV links, such accesses do not have to spend off-chip bandwidth between the host and the hybrid memory cubes and thus can be beneficial. Figure 13 shows that Charon effectively utilizes much higher bandwidth than the available off-chip memory bandwidth (i.e., 80GB/s on HMC) and other baselines. The figure also shows that over 70% of memory requests are serviced locally for most cases, to reserve sizable headroom for external link bandwidth. LR and CC are exceptions to have about a half of the memory requests are directed to a remote note; however, both are not as bandwidth-intensive as the other applications, hence alleviating the problem.

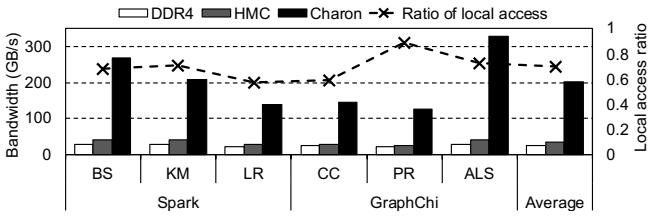

Figure 13: Utilized bandwidth during GC and ratio of local accesses

Per-Primitives Analysis. Figure 14 shows how much average speedup over the CPU with DDR4 memory system Charon achieved on each key primitive. As shown here, Charon achieves up to 26.15× maximum speedup (10.17× on average) on *Copy* (for both MinorGC and MajorGC) and up to 4.09× (2.90× on average) on *Search* primitive by exploiting the abundant internal bandwidth of HMC with maximized memory-level parallelism and large-granularity memory accesses. In addition to *Copy* and *Search* primitives, *Scan&Push* primitive achieves a maximum speedup of 1.86× (1.20× in average) with its additional memory-level parallelism as well. On the other hand, speedups of *Bitmap Count* primitive come from a combination of our novel algorithm and specialized hardware design for the particular primitive. With these changes, Charon improves the throughput of *Bitmap Count* primitive by up to 6.11× (5.63× on average) over the DDR4 memory systems. Since Charon offloads the minimal, relatively simple primitives to avoid requiring invasive changes to the software and the hardware, Charon does not handle the whole GC process. However, with only a handful of offloading primitives, Charon achieves a substantial speedup.

Scan&Push primitive shows relatively low speedup (or even degrades) in some applications like BS, KM, LR, and ALS. Those applications are based on machine learning algorithms that allocate a small number of large size objects which i) have very few references within them (e.g., large matrix, key-value pairs in a large table as objects) and ii) have short lifetime since such objects are usually discarded once they are used. In this case, where each object has very few references, the amount of parallelism is very low and thus Charon achieves low throughput.

However, Charon obtains more benefit from other primitives (e.g., *Copy*) for those applications because those applications manipulate large objects that generate a large number of memory accesses, which can benefit from an excessive MLP of Charon. Modest speedups of CC and PR are also explainable in a similar manner. CC and PR are based on graph algorithms and traverse a large number of nodes (i.e., objects) through edges (i.e., references). Thus, those objects have a long life cycle with many references, which can utilize and benefit from sufficient MLP in *Scan&Push* primitives.

GC Scalability. Multiple HMCs can be chained to scale capacity, allowing us to put more Charon primitives to their logic layers. To evaluate GC scalability, we scale the number of corresponding Charon primitive units as we increase the number of GC threads. We compare the scalability of both the unified design (i.e., a single bitmap cache and TLB on the center cube shared by all cubes) and distributed design (i.e., slices of the bitmap cache and TLB distributed across all cubes) as we discuss in Section 4.6.

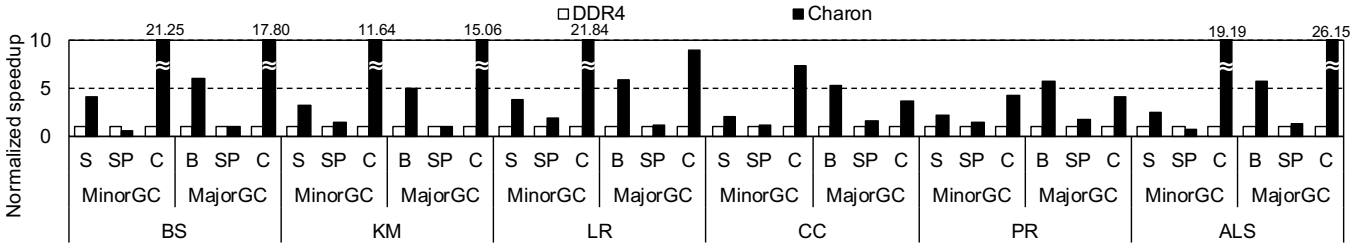

Figure 14: Per-primitive speedup analysis (S: *Search*, SP: *Scan&Push*, C: *Copy*, BC: *Bitmap Count*)

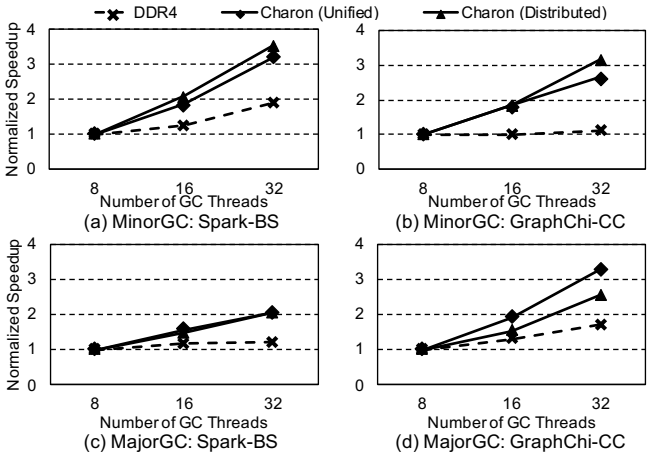

Figure 15: GC throughput scalability

Figure 15 shows performance scalability with an increasing number of GC threads. First, Charon scales significantly better than DDR4 based system by utilizing plenty of internal bandwidth, while the DDR4 system hardly scales due to limited memory bandwidth (max 34GB/s). Second, Charon with distributed structures generally scales better than the unified design as contentions at the center cube are alleviated. MajorGC in GraphChi-CC is an exception as memory pressure is relatively low (thus, contentions are less of an issue), while penalizing the distributed design for remote TLB accesses. Finally, performance scalability can be further improved by adopting bandwidth-scalable HMC topology [71] and locality-improving page interleaving policy to reduce remote traffic [17].

Charon as CPU-side Accelerator. Charon primitives are flexible enough and thus can also be utilized on a system other than the HMC-based ones. For example, Charon can be used in the HBM-like system as a CPU-side accelerator. Figure 16 compares the throughput of three different configurations: a CPU with the DDR4 memory system, Charon located on the CPU side paired with the HMC memory system, and Charon located on the logic die of the HMC memory system. Charon on the CPU side achieves better performance than baseline processor from its aggressive use of MLP and optimized bitmap operations. However, Charon on the CPU-side also misses out the abundant internal DRAM bandwidth at the HMC logic layer and thus its throughput is about 37% less than the Charon as a near-memory accelerator. While this bandwidth bottleneck might be partly alleviated using high-bandwidth DRAM technologies like HBM, placing Charon on the DRAM side still has some advantages for energy efficiency, scalability (of the logic area), and less bandwidth contentions at the memory controller.

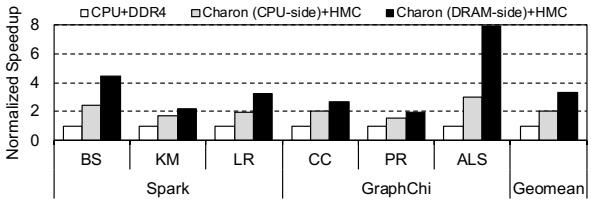

Figure 16: Memory-side implementation speedup over CPU-side on-chip implementation

5.3 Energy/Area and Thermal Analysis

Figure 17 shows the normalized energy consumption of Charon during GCs compared with the host-only execution case. While Charon shows a substantial improvement of throughput, it only increases moderate power consumption leading to a substantial reduction in energy consumption. As shown in Figure 17, Charon achieves 60.7% of energy reduction over the host system with DDR4 main memory on average and 51.6% of energy reduction over the host system with the HMC baseline across all workloads. Note that energy consumption of general components (i.e., queues, metadata arrays, TLB, and bitmap cache) is negligible compared to the total energy consumption of Charon (maximum 3.18% for ALS).

Table 4 shows the area usage of each component and the total area of Charon. The total area of Charon is only about 1.95mm^2 and thus the average area per cube is around 0.49mm^2. Assuming that the area of Hybrid Memory Cube logic layer as 100mm^2 [22], Charon takes only 0.49% of the total logic layer area, indicating that Charon on the HMC logic layer incurs little area overhead.

Regarding thermal issues, the primitive units of Charon do very simple computation with lots of memory accesses, thus its impact on thermal constraint is negligible. Our evaluation indicates the average power consumption is 2.98W for all workloads (maximum 4.51W for ALS). This is much lower than previous proposals to place computation logic in the HMC stacks [1, 16, 56]. Therefore, the maximum power density of the logic die is 45.1mW/mm^2, which is much lower than the maximum allowable power density of a low-end passive heat sink [12].

6 RELATED WORK

GC Optimizations Using Accelerators. The idea of accelerating GC using specialized hardware has been proposed for decades to overcome GC overhead in conventional processors. [27, 38, 61,

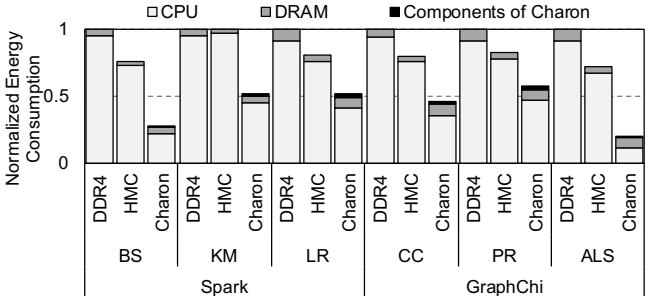

Figure 17: Energy consumption of Charon on GC compared with the host CPU-only execution

Component	Per-unit Area (mm^2)	# of Units	Total Component Area (mm^2)
General Components			
Command Queue	0.0049	4	0.0196
Request Queue(R)	0.0015	4	0.0060
Request Queue(W)	0.0162	4	0.0648
Metadata Array	0.0805	4	0.3220
Bitmap Cache	0.1562	1	0.1562
TLB	0.0706	4	0.2824
Processing Units			
Copy/Search	0.0223	8	0.1784
Bitmap Count	0.0427	8	0.3416
Scan&Push	0.0720	8	0.5760
Total Area: 1.9470mm^2 / Average Area per Cube: 0.4868mm^2			

Table 4: Total area usage of Charon for whole cubes

66, 67]. Active Memory Processor (AMP) [61] integrates a bitmap-based processor with a standard 2D DRAM array to improve the performance and predictability of dynamic memory management functions including allocation, reference counting, and GC. Joao et al. [27] proposed hardware GC accelerator based on a reference counting algorithm. Their hardware collector coexists with conventional GC algorithms which complement the limitations of reference counting GC such as cyclic dependencies. However, their accelerator design is tightly coupled with the host processor, which requires invasive changes to the host processor. As the most similar work to Charon, Maas et al. [38] propose a hardware GC technique that fully offloads the mark-sweep GC of JikesRVM to on-chip accelerator located near memory. While this work also utilizes abundant MLP using specialized hardware for GC, their GC algorithm is hard-wired at design time to have limited applicability. In particular, their design cannot fully offload the generational GC, which is widely used in production (including HotSpot), as it requires a copying collector among semispaces (generations). Their design and Charon demonstrate tradeoffs between specialization and flexibility. Our primitive-based approach likely has a bit of efficiency loss for Jikes RVM which their design is targeting, but has much broader applicability. Moreover, Charon is the first proposal to leverage 3D stacked memory for higher internal memory bandwidth and addresses design issues for DRAM-side offloading.

Near-Memory Processing on 3D stacked DRAM. With an introduction of 3D stacked DRAM technologies, many prior researches have proposed various near-memory processing architectures or techniques [3, 19, 32, 47, 48]. Several studies focus on hardware architecture to accelerate a wide variety of applications such as graph processing [1, 2, 46], vector operations [22, 26], deep learning [16, 30], MapReduce computation [20, 57]. They aim to leverage ample internal DRAM bandwidth, massive parallelism, and reduce data movement overheads in the conventional CPU memory hierarchy. Our work also shares these goals in that utilizing high levels of memory parallelism for key algorithmic primitives with energy efficient hardware, but is the first to present a concrete proposal for offloading GC to the stacked DRAM.

GC Optimizations for Emerging Applications. There has been a lot of interest in optimizing GC for emerging applications like big data analytics which has a large memory footprint in a distributed environment. Facade [50] and Yak [49] re-structure the heap space and optimize the GC algorithm for the epochal behavior of modern big data analytics frameworks by utilizing separate spaces for control and data objects. NumaGiC [17] presents a distributed design for improving object locality on a cache coherent NUMA machine by ensuring a GC thread to only processes objects located on its own memory space. Taurus [37] advocates a holistic approach to coordinate GCs (and JITs) among all runtimes to improve performance. Choi et al. [8] propose a biased reference counting algorithm (BRC), which reduces the execution time of non-deferred reference counting (RC) by allowing RC operations partially non-atomically, thus achieving low-latency RC. Our proposal complements these software-based optimizations by providing fine-grained offloading primitives which can potentially work together with some of these proposals.

7 CONCLUSION

This paper presents Charon, a novel near-memory accelerator for offloading GC. Although adoption of hardware-based GC has been scarce due to various limitations, we believe its time has finally come with the arrival of 3D stacked memory. To support various GC algorithms, we first perform performance analysis of the full-production HotSpot JVM in detail. Through this analysis, we identify three key algorithmic primitives, memory *Copy/Search*, *Bitmap Count*, and *Scan&Push*, which are commonly used in tracing GC algorithms. Then we design a specialized processing unit placed in the logic layer of 3D stacked memory, which executes these primitives efficiently with a low area/energy cost. Our evaluation using the full-production HotSpot JVM with two big data analytics frameworks, Apache Spark and GraphChi, demonstrates substantial performance and energy efficiency gains over the general-purpose CPU, without requiring intrusive software changes.

ACKNOWLEDGMENTS

We thank Martin Maas for his feedback on a draft of this paper and Jungwoo Ha for helping us understand the internals of HotSpot JVM. This work was partly supported by Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-IT1501-07 and Institute for Information & communications Technology Promotion (IITP) grant funded by Korea government (MSIT) (2014-0-00035, Research on High Performance and Scalable Manycore Operating System). Jae W. Lee is the corresponding author.

REFERENCES

[1] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A Scalable Processing-in-memory Accelerator for Parallel Graph Processing. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 105–117. https://doi.org/10.1145/2749469.2750386

[2] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-enabled Instructions: A Low-overhead, Locality-aware Processing-in-memory Architecture. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*. ACM, New York, NY, USA, 336–348. https://doi.org/10.1145/2749469.2750385

[3] Berkin Akin, Franz Franchetti, and James C. Hoe. 2015. Data Reorganization in Memory Using 3D-stacked DRAM. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 131–143. https://doi.org/10.1145/2749469.2750397

[4] Nadav Amit. 2017. Optimizing the TLB Shootdown Algorithm with Page Access Tracking. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 27–39. https://www.usenix.org/conference/atc17/technical-sessions/presentation/amit

[5] David F. Bacon, Perry Cheng, and Sunil Shukla. 2013. And then There Were None: A Stall-free Real-time Garbage Collector for Reconfigurable Hardware. *Commun. ACM* 56, 12 (Dec. 2013), 101–109. https://doi.org/10.1145/2534706.2534726

[6] Yingyi Bu, Vinayak Borkar, Guoqing Xu, and Michael J. Carey. 2013. A Bloat-aware Design for Big Data Applications. In *Proceedings of the 2013 International Symposium on Memory Management (ISMM '13)*. ACM, New York, NY, USA, 119–130. https://doi.org/10.1145/2491894.2466485

[7] Chisel3. https://github.com/freechipsproject/chisel3.

[8] Jiho Choi, Thomas Shull, and Josep Torrellas. 2018. Biased Reference Counting: Minimizing Atomic Operations in Garbage Collection. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT '18)*. ACM, New York, NY, USA, Article 35, 12 pages. https://doi.org/10.1145/3243176.3243195

[9] Hybrid Memory Cube Consortium. Hybrid Memory Cube Specification 2.1. http://hybridmemorycube.org/files/SiteDownloads/HMC-30G-VSR_HMCC_Specification_Rev2.1_20151105.pdf.

[10] Elliott Cooper-Balis, Paul Rosenfeld, and Bruce Jacob. 2012. Buffer-on-board Memory Systems. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*. IEEE Computer Society, Washington, DC, USA, 392–403. http://dl.acm.org/citation.cfm?id=2337159.2337204

[11] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual. Reference number: 325462-057US, 2015. https://software.intel.com/en-us/articles/intel-sdm.

[12] Yasuko Eckert, Nuwan Jayasena, and Gabriel H Loh. 2014. Thermal feasibility of die-stacked processing in memory. In *2nd Workshop on Near-Data Processing (WoNDP '14)*.

[13] Hua Fan, Aditya Ramaraju, Marlon McKenzie, Wojciech Golab, and Bernard Wong. 2015. Understanding the causes of consistency anomalies in Apache Cassandra. *Proceedings of the VLDB Endowment* 8, 7 (2015), 810–813.

[14] Lu Fang, Khanh Nguyen, Guoqing Xu, Brian Demsky, and Shan Lu. 2015. Interruptible Tasks: Treating Memory Pressure As Interrupts for Highly Scalable Data-parallel Programs. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 394–409. https://doi.org/10.1145/2815400.2815407

[15] Mingyu Gao, Grant Ayers, and Christos Kozyrakis. 2015. Practical Near-Data Processing for In-Memory Analytics Frameworks. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT) (PACT '15)*. IEEE Computer Society, Washington, DC, USA, 113–124. https://doi.org/10.1109/PACT.2015.22

[16] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. 2017. TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 751–764. https://doi.org/10.1145/3037697.3037702

[17] Lokesh Gidra, Gaël Thomas, Julien Sopena, Marc Shapiro, and Nhan Nguyen. 2015. NumaGiC: A Garbage Collector for Big Data on Big NUMA Machines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 661–673. https://doi.org/10.1145/2694344.2694361

[18] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingam, Manuel Costa, Derek G Murray, Steven Hand, and Michael Isard. 2015. Broom: Sweeping out garbage collection from big data systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*.

[19] Ramyad Hadidi, Lifeng Nai, Hyojong Kim, and Hyesoon Kim. 2017. CAIRO: A Compiler-Assisted Technique for Enabling Instruction-Level Offloading of Processing-In-Memory. *ACM Transactions on Architecture and Code Optimization* 14 (12 2017), 1–25. https://doi.org/10.1145/3155287

[20] Syed Minhaj Hassan, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. 2015. Near Data Processing: Impact and Optimization of 3D Memory System Architecture on the Uncore. In *Proceedings of the International Symposium on Memory Systems (MEMSYS)*. ACM, New York, NY, USA, 11–21. https://doi.org/10.1145/2818950.2818952

[21] Matthew Hertz, Yi Feng, and Emery D. Berger. 2005. Garbage Collection Without Paging. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 143–153.

[22] Byungchul Hong, Gwangsun Kim, Jung Ho Ahn, Yongkee Kwon, Hongsik Kim, and John Kim. 2016. Accelerating Linked-list Traversal Through Near-Data Processing. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT '16)*. ACM, New York, NY, USA, 113–124. https://doi.org/10.1145/2967938.2967958

[23] Intel. Intel i7-4790 Processor v4. https://ark.intel.com/ko/products/80806/Intel-Core-i7-4790-Processor-8M-Cache-up-to-4-00-GHz-.

[24] Java HotSpot Virtual Machine. http://openjdk.java.net/groups/hotspot.

[25] JEDEC. 2015. *JEDEC Standard JESD235A: High Bandwidth Memory (HBM) DRAM*. JEDEC Solid State Technology Association, Virginia, USA.

[26] Dong-Ik Jeon, Kyeong-Bin Park, and Ki-Seok Chung. 2018. HMC-MAC: Processing-in Memory Architecture for Multiply-Accumulate Operations with Hybrid Memory Cube. *IEEE Comput. Archit. Lett.* 17, 1 (Jan. 2018), 5–8. https://doi.org/10.1109/LCA.2017.2700298

[27] José A. Joao, Onur Mutlu, and Yale N. Patt. 2009. Flexible Reference-counting-based Hardware Acceleration for Garbage Collection. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. ACM, New York, NY, USA, 418–428. https://doi.org/10.1145/1555754.1555806

[28] KDD Cup 2010 Dataset. https://pslcdatashop.web.cmu.edu/KDDCup/downloads.jsp.

[29] KDD Cup 2010 transformed Dataset. https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html.

[30] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. 2016. Neurocube: A Programmable Digital Neuromorphic Architecture with High-density 3D Memory. In *Proceedings of the International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 380–392. https://doi.org/10.1109/ISCA.2016.41

[31] Gwangsun Kim, John Kim, Jung Ho Ahn, and Jaeha Kim. 2013. Memory-centric System Interconnect Design with Hybrid Memory Cubes. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE Press, Piscataway, NJ, USA, 145–156. http://dl.acm.org/citation.cfm?id=2523721.2523744

[32] Hyojong Kim, Ramyad Hadidi, Lifeng Nai, Hyesoon Kim, Nuwan Jayasena, Yasuko Eckert, Onur Kayiran, and Gabriel Loh. 2018. CODA: Enabling Co-location of Computation and Data for Multiple GPU Systems. *ACM Trans. Archit. Code Optim.* 15, 3, Article 32 (Sept. 2018), 23 pages. https://doi.org/10.1145/3232521

[33] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale Graph Computation on Just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Berkeley, CA, USA, 31–46. http://dl.acm.org/citation.cfm?id=2387880.2387884

[34] 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 *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, New York, NY, USA, 469–480. https://doi.org/10.1145/1669112.1669172

[35] S. Li, D. H. Yoon, K. Chen, J. Zhao, J. H. Ahn, J. B. Brockman, Y. Xie, and N. P. Jouppi. 2012. MAGE: Adaptive Granularity and ECC for resilient and power efficient memory systems. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–11.

[36] Justin Ma, Lawrence K. Saul, Stefan Savage, and Geoffrey M. Voelker. 2009. Identifying Suspicious URLs: An Application of Large-scale Online Learning. In *Proceedings of the 26th Annual International Conference on Machine Learning (ICML)*. ACM, New York, NY, USA, 681–688. https://doi.org/10.1145/1553374.1553462

[37] Martin Maas, Krste Asanović, Tim Harris, and John Kubiatowicz. 2016. Taurus: A Holistic Language Runtime System for Coordinating Distributed Managed-Language Applications. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 457–471. https://doi.org/10.1145/2872362.2872386

[38] Martin Maas, Krste Asanović, and John Kubiatowicz. 2018. A Hardware Accelerator for Tracing Garbage Collection. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA '18)*. IEEE Press, Piscataway, NJ, USA, 138–151. https://doi.org/10.1109/ISCA.2018.00022

[39] M. Meyer. 2004. A novel processor architecture with exact tag-free pointers. *IEEE Micro* 24, 3 (May 2004), 46–55. https://doi.org/10.1109/MM.2004.2

[40] Matthias Meyer. 2005. An On-Chip Garbage Collection Coprocessor for Embedded Real-Time Systems. In *Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE

Computer Society, Washington, DC, USA, 517–524. https://doi.org/10.1109/RTCSA.2005.25
[41] Matthias Meyer. 2006. A True Hardware Read Barrier. In *Proceedings of the International Symposium on Memory Management (ISMM)*. ACM, New York, NY, USA, 3–16. https://doi.org/10.1145/1133956.1133959
[42] Matthias Meyer. 2006. A True Hardware Read Barrier. In *Proceedings of the 5th International Symposium on Memory Management (ISMM '06)*. ACM, New York, NY, USA, 3–16.
[43] SUN Microsystems. Memory Management in the Java HotSpot™ Virtual Machine.
[44] MIT. GraphChallenge Dataset. http://www.graphchallenge.mit.edu.
[45] David A. Moon. 1984. Garbage Collection in a Large Lisp System. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming (LFP '84)*. ACM, New York, NY, USA, 235–246. https://doi.org/10.1145/800055.802040
[46] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim. 2017. GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 457–468. https://doi.org/10.1109/HPCA.2017.54
[47] Lifeng Nai, Ramyad Hadidi, He Xiao, Hyojong Kim, Jaewoong Sim, and Hyesoon Kim. 2018. CoolPIM: Thermal-Aware Source Throttling for Efficient PIM Instruction Offloading *(IPDPS)*. 680–689. https://doi.org/10.1109/IPDPS.2018.00077
[48] R. Nair, S. F. Antao, C. Bertolli, P. Bose, J. R. Brunheroto, T. Chen, C. . Cher, C. H. A. Costa, J. Doi, C. Evangelinos, B. M. Fleischer, T. W. Fox, D. S. Gallo, L. Grinberg, J. A. Gunnels, A. C. Jacob, P. Jacob, H. M. Jacobson, T. Karkhanis, C. Kim, J. H. Moreno, J. K. O'Brien, M. Ohmacht, Y. Park, D. A. Prener, B. S. Rosenburg, K. D. Ryu, O. Sallenave, M. J. Serrano, P. D. M. Siegl, K. Sugavanam, and Z. Sura. 2015. Active Memory Cube: A processing-in-memory architecture for exascale systems. *IBM Journal of Research and Development* 59, 2/3 (March 2015), 17:1–17:14. https://doi.org/10.1147/JRD.2015.2409732
[49] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. 2016. Yak: A High-performance Big-data-friendly Garbage Collector. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 349–365. http://dl.acm.org/citation.cfm?id=3026877.3026905
[50] Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, Jianfei Hu, and Guoqing Xu. 2015. FACADE: A Compiler and Runtime for (Almost) Object-Bounded Big Data Applications. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 675–690. https://doi.org/10.1145/2694344.2694345
[51] M. Ogleari, Y. Yu, C. Qian, E. Miller, and J. Zhao. 2019. String Figure: A Scalable and Elastic Memory Network Architecture. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 647–660. https://doi.org/10.1109/HPCA.2019.00016
[52] OpenJDK7. http://openjdk.java.net/projects/jdk7/.
[53] Oracle. Java Tuning White Paper. https://www.oracle.com/technetwork/java/tuning-139912.html.
[54] Matthew Poremba, Itir Akgun, Jieming Yin, Onur Kayiran, Yuan Xie, and Gabriel H. Loh. 2017. There and Back Again: Optimizing the Interconnect in Networks of Memory Cubes. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 678–690. https://doi.org/10.1145/3079856.3080251
[55] PHASE project of the Japanese National Institute of Advanced Industrial Science and Technology. Matrix Market. https://math.nist.gov/MatrixMarket/.
[56] Seth H Pugsley, Jeffrey Jestes, Huihui Zhang, Rajeev Balasubramonian, Vijayalakshmi Srinivasan, Alper Buyuktosunoglu, Al Davis, and Feifei Li. 2014. NDC: Analyzing the impact of 3D-stacked memory+ logic devices on MapReduce workloads. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 190–200.
[57] Seth H. Pugsley, Jeffrey Jestes, Huihui Zhang, Rajeev Balasubramonian, Vijayalakshmi Srinivasan, Alper Buyuktosunoglu, Al Davis, and Feifei Li. 2014. NDC: Analyzing the impact of 3D-stacked memory+logic devices on MapReduce workloads.. In *ISPASS*. IEEE Computer Society, 190–200. http://dblp.uni-trier.de/db/conf/ispass/ispass2014.html#PugsleyJZBSBDL14
[58] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*. ACM, New York, NY, USA, 475–486. https://doi.org/10.1145/2485922.2485963
[59] Juri Schmidt, Holger Fröning, and Ulrich Brüning. 2016. Exploring time and energy for complex accesses to a hybrid memory cube. In *Proceedings of the Second International Symposium on Memory Systems (MEMSYS)*. ACM, New York, NY, USA, 142–150. https://doi.org/10.1145/2989081.2989099
[60] William J. Schmidt and Kelvin D. Nilsen. 1994. Performance of a Hardware-assisted Real-time Garbage Collector. *SIGOPS Oper. Syst. Rev.* 28, 5 (Nov. 1994), 76–85. https://doi.org/10.1145/381792.195504
[61] Witawas Srisa-an, Chia-Tien Dan Lo, and Ji-en Morris Chang. 2003. Active Memory Processor: A Hardware Garbage Collector for Real-Time Java Embedded Devices. *IEEE Transactions on Mobile Computing* 2, 2 (April 2003), 89–101. https://doi.org/10.1109/TMC.2003.1217230
[62] Sylvain Stanchina and Matthias Meyer. 2007. Mark-sweep or Copying?: A "Best of Both Worlds" Algorithm and a Hardware-supported Real-time Implementation. In *Proceedings of the International Symposium on Memory Management (ISMM)*. ACM, New York, NY, USA, 173–182. https://doi.org/10.1145/1296907.1296928
[63] David Ungar. 1984. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE 1)*. ACM, New York, NY, USA, 157–167.
[64] David Michael Ungar. 1986. *The Design and Evaluation of a High Performance Smalltalk System*. Ph.D. Dissertation. University of California at Berkeley, Berkeley, CA, USA. UMI order no. GAX86-24972.
[65] Steven J. E. Wilton and Norman P. Jouppi. 1996. CACTI: An Enhanced Cache Access and Cycle Time Model. *IEEE Journal of Solid-State Circuits* 31 (1996), 677–688.
[66] David S. Wise, Brian Heck, Caleb Hess, Willie Hunt, and Eric Ost. 1997. Research Demonstration of a Hardware Reference-Counting Heap. *Lisp Symb. Comput.* 10, 2 (July 1997), 159–181. https://doi.org/10.1023/A:1007715101339
[67] Greg Wright, Matthew L. Seidl, and Mario Wolczko. 2006. An Object-aware Memory Architecture. *Sci. Comput. Program.* 62, 2 (Oct. 2006), 145–163. https://doi.org/10.1016/j.scico.2006.02.007
[68] Ting Yang, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. 2006. CRAMM: Virtual Memory Support for Garbage-collected Applications. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*. USENIX Association, Berkeley, CA, USA, 103–116.
[69] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*. USENIX Association, Berkeley, CA, USA, 10–10. http://dl.acm.org/citation.cfm?id=1863103.1863113
[70] ZGC: The Z Garbage Collector. https://openjdk.java.net/projects/zgc/.
[71] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian. 2018. GraphP: Reducing Communication for PIM-Based Graph Processing with Efficient Data Partition. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 544–557. https://doi.org/10.1109/HPCA.2018.00053