Polar: A Managed Runtime with Hotness-Segregated Heap for Far Memory

Dat Nguyen
Texas A&M University
tiendat.ng.cs@tamu.edu

Khanh Nguyen
Texas A&M University
khanhtn@tamu.edu

Abstract

Thanks to recent advances in high-bandwidth, low-latency interconnects, running a data-intensive application with a remote memory pool is now a feasibility. When developing a data-intensive application, a managed language such as Java is often the developer's choice due to convenience of the runtime such as automatic memory management. However, the memory management cost increases significantly in far memory due to remote memory accesses.

Our insight is that *data hotness* (i.e., access frequency of objects) is the key to reducing the memory management cost and improving efficiency in far memory. In this paper, we present an ongoing work designing Polar, an enhanced runtime system that is hotness-aware, and optimized for far memory. In Polar, the garbage collector is augmented to identify cold (infrequently accessed) objects and relocate them to remote memory pools. By placing objects at memory locations based on their access frequency, Polar minimizes the number of remote accesses, ensures low access latency for the application, and thus improves overall performance.

CCS Concepts

• **Information systems** → **Data management systems**; • **Software and its engineering** → **Garbage collection**; **Runtime environments**; *Allocation / deallocation strategies*.

Keywords

Far memory, managed runtime, garbage collection

ACM Reference Format:
Dat Nguyen and Khanh Nguyen. 2024. Polar: A Managed Runtime with Hotness-Segregated Heap for Far Memory. In *15th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '24), September 4–5, 2024, Kyoto, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3678015.3680490

APSys '24, September 4–5, 2024, Kyoto, Japan

ACM ISBN 979-8-4007-1105-3/24/09.
https://doi.org/10.1145/3678015.3680490

1 Introduction

Modern computing has a high demand for processing a huge amount of data, fueled by the increase in memory-hungry workloads such as AI/ML and graph processing in recent years. Managed languages (e.g., Java and Scala) which are known for their simple usage, easy memory management, and large community support are often developers' choice to implement large-scale frameworks. These languages simplify development efforts and help reduce a large number of memory-related bugs thanks to the garbage collector (GC[1]). Thanks to rapid technological advances in network controllers [22, 28, 44], it now becomes practical to utilize a remote memory pool to overcome memory capacity wall [10], giving rise to the far-memory architecture. In far memory, an application is deployed at a local server that is connected to a network of remote servers offering memory resources. The application, in addition to the local memory, can now use memory from remote servers without prohibitive overheads.

The GC provides automatic memory management, freeing up developers' time, hence the broad adoption of managed runtime such as Java Virtual Machine (JVM). GC is triggered whenever the heap occupancy reaches a threshold (e.g., 80%). As illustrated in Figure 1, GC scans the heap by following object references from a set of GC roots (e.g., global variables), moves reachable objects to a different memory space, and discards objects that are no longer in use. In a far-memory environment, however, GC cost is exacerbated due to the latency of remote accesses. Even though recent network advances have achieved lower latency than decades ago, one remote access takes an average of 40-80μs, which is still orders of magnitude slower than *ns*-latency local memory access. Our experience with a dozen representative data-intensive applications and evidence have shown that, running an application as-is, the GC cost is magnified, reducing end-to-end performance by up to 10× [25, 47, 48].

[1]As commonly in the literature, 'GC' also stands for 'garbage collection'

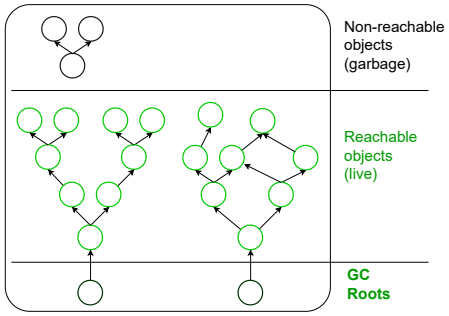

Figure 1: How GC works in Java. Essentially, GC is a transitive closure computation starting from a set of roots. Reachable objects are moved to another space.

Our insight and proposal. One root cause for the costly runtime system is that the GC is oblivious to program semantics — in this work, we focus specifically on data hotness (i.e., access frequency of objects). The golden rule for performance in a far-memory environment is that objects that are "hot" (frequently accessed) should be placed in local memory that is close to the CPU running the application, while "cold" objects are placed in the remote memory.

In this work, we propose Polar as an enhanced, far-memory-friendly runtime for data-intensive applications. Polar, to the best of our knowledge, is the first runtime that is hotness-aware at object granularity for far memory. Polar-heap consists of a local heap in the local server (hot) and a remote heap at remote servers (cold). Objects are initially allocated in the local heap. When the local heap runs out of space, GC is triggered and segregates the live heap based on hotness: hot objects are kept in the local heap, whereas cold objects are relocated to the remote heap. Traditional GC will run and manage the local heap while the remote heap will be managed by a Polar-agent.

Figure 2 shows the effect of Polar on the local heap usage. When the local heap is under pressure, GC identifies and evicts cold objects to the remote memory, reclaiming local heap space to avoid a potential out-of-memory crash.

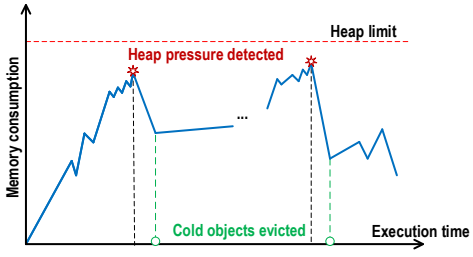

Figure 2: In Polar, upon memory pressure, the GC identifies and evicts cold objects to remote memory, avoiding a potential out-of-memory crash.

While Polar and the generational GC share commonalities, they are fundamentally different from each other. In Polar, the GC is restricted to the local heap, i.e., not following references that lead to the remote heap. Compared to some of the recent work such as Semeru [47], Mako [25], and MemLiner [48], Polar offers a finer granularity at the object level instead of a coarse grain such as (virtual) memory page or memory regions which has been shown to suffer from data amplification problem [36, 41]. We envision Polar will unlock additional benefits such as improved locality and more efficient data fetches.

Why JVM. Even though a number of far-memory techniques have been developed, almost all of them use a cache-and-swap mechanism to enhance the OS, treating the remote memory pool as an extended swap system [1–4, 17, 20, 27]. The application uses local memory as a data cache only. Once a page that does not reside in the local memory is accessed, a page fault is triggered and the page is fetched from a remote server into the local memory. Because they manage memory at the coarse granularity of pages, they are suboptimal for the average case where a page contains objects with different hotness. Moreover, replacing the OS kernel is intrusive. Polar runs atop an unmodified OS kernel with a memory management scheme that operates at object level, and is complementary to ongoing work in OS kernels.

Challenges and Solutions. While the idea of matching objects to memory locations based on their access frequency appears simple, there are several challenges in how to implement it efficiently and correctly. We discuss below the challenges in carrying out this high-level idea and our solutions.

The first challenge in designing Polar is *how to classify hot and cold objects* without incurring much overhead. A simple approach of using access count is impractical because there is no one-size-fits-all threshold value for all applications. More importantly, access count fails to capture temporal hotness: an object may be hot at one point but stay dormant for the rest of its lifetime. Orthogonal to this is the second challenge of *when to move cold objects to remote memory* to maximize space benefits given to the application while minimizing costs which include data transfer over the network costs. Finally, the third challenge is *how to guarantee memory safety* as the heap now spans multiple servers.

To overcome these challenges, we design a collaborative mechanism between GC and the application to compute a dynamic threshold ω to segregate the live heap. An object is a candidate for remote memory relocation if the application has not accessed it within a past window of ω GC runs. In Polar, the GC is enhanced to evict cold objects to the remote heap. By tying cold object eviction directly to a GC run, Polar helps maximize the effectiveness of cold object

eviction, giving back to the application the much-needed heap space for new allocations. Access to evicted data is still safe and guaranteed. Polar uses an indirection structure called *trampoline* as a proxy to access data across server boundaries. All cross-server accesses are forwarded to the trampoline which provides one-level indirection to resolve the actual address.

In the remainder of the paper, we describe internal design of Polar in §2. Because Polar is still under development, we include, as a preliminary result, in §3 a study validating our assumption of data hotness, hence demonstrating the potential of Polar.

2 Polar

In this Section, we describe the design of Polar. The key characteristic of Polar is that it splits the heap into a local (hot) heap and a remote (cold) heap, which correspond to 2 levels of data hotness. Each heap is subject to a different style of management. The GC is restricted to managing the local heap in normal ways and is *not* allowed to follow object references into the remote heap. The remote heap is assumed to be abundant, contains objects that are rarely needed by the application threads, and thus can be managed in a more relaxed manner (e.g., less frequent scanning).

2.1 Polar heap

Figure 3 illustrates Polar heap which is a partitioned global address space (PGAS), spanning across local and remote servers. During JVM's bootstrapping, when the heap is created, it reserves three disjoint ranges of virtual addresses, namely the nursery, the hot zone, and the cold zone. The nursery and hot zone are backed by the local server (the local heap) while the cold zone is backed by the remote servers (the remote heap).

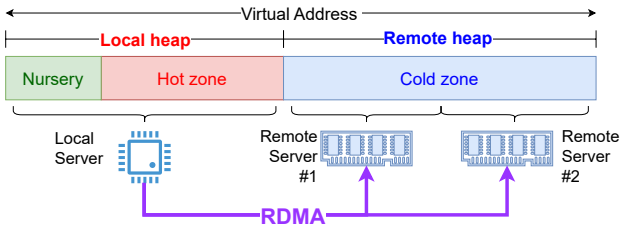

Figure 3: Polar's heap structure.

Allocation. All object allocations take place in the nursery. In order to be a lock-free operation, each application thread has a thread-local allocation buffer (TLAB) given by the nursery. Upon an allocation request, each thread uses a bump-pointer algorithm to accommodate a new object in its buffer. When running out of buffer space, the thread requests a new buffer from the nursery. Similar to conventional design, very large objects will be allocated in a special centralized heap area called *humongous area* (not shown in Figure 3).

2.2 Data hotness measurement

Data hotness metric. A naïve measure of hotness is the access count: an object is hot if it has been read and/or written by the application threads (a.k.a. the mutator) above a static threshold. This measure is impractical because there is no one-size-fits-all value for all applications. More importantly, access count fails to capture *temporal hotness*: an object may be hot at one point but stay dormant for the rest of its lifetime. We find it more efficient to focus on subjects for eviction. A (live) object is called cold and subject to eviction to the remote heap *iff* it has not been accessed by the mutator within a past window of ω GC runs. Defining an object's data coldness λ as the length of time since its last access by the mutator, we evict the object if $\lambda > \omega$. This is a better solution as it ties cold object eviction directly to the memory consumption behavior of the application. A memory-hungry application will trigger GC often, and thus will gain benefits from evicting cold objects out of the local heap, saving space for newly-allocated objects instead of wasting memory space to hold cold objects while struggling to accommodate new allocations.

Record λ transparently. Computing data coldness λ is a collaborative effort between the mutator and the GC. Polar piggybacks on the existing read/write (R/W) barriers to record λ. A R/W barrier contains code that is executed together with every heap load/store instruction, and is an indispensable component of modern GCs. For example, G1 GC [14] uses them to track inter-generation references; ZGC [31] and Shenandoah GC [16] use them to ensure heap integrity. Using this basic barrier, we add an additional Polar-logic for each R/W access of instance fields and array elements. For instance, with a heap write $a.f = b$, the coldness value λ of the object O_a is (re)set to 0. This additional step is light and thus will have negligible overhead. As the GC scans the heap and marks live objects, it will increase λ by 1. To store λ, as done in existing works [25, 30, 47, 48], we enlarge the object's header, adding 2 bytes.

2.3 Coldness threshold ω

Determining a concrete value of ω to classify objects as hot and cold is challenging. ω should be the sweet spot to (1) minimize the amount of cold data to be evicted due to an associated cost with each eviction, and (2) avoid unnecessary GC (in the local heap) to find space for new allocations. Effectively we are optimizing the local heap usage using two conflicting criteria. On the one hand, we want to fully utilize the local heap to save on remote memory access. If ω is low, we might evict too aggressively and cause unnecessary

roundtrips to retrieve the recently-evicted objects. On the other hand, if ω is high, the local heap will be too full and the runtime will struggle to find space for new allocations. It is critical for efficiency to strike a balance.

Using profiling techniques to determine ω is problematic due to their lack of precision (i.e., requiring representative input) and cost (i.e., profiling often is expensive). To address this issue, Polar's solution is to leverage GC runs to determine ω *dynamically*. As GC scans the local heap, it computes ω as a size-weighted average coldness of all live objects O. As confirmed in our study (see §3), this is a good proxy to capture the temporal property of the heap and can react well to the application's dynamic behavior.

2.4 Local heap management

Polar uses lazy memory expansion, i.e., it is unnecessary to involve remote memory when the local heap is enough to contain the application's working set. Accordingly, two types of GC exist: GC-and-Eviction (GCE), which is only triggered when the application is suffering from memory pressure, and thus beneficial for the application to evict cold objects to the remote memory, freeing up space for new allocations. Otherwise, Local-Only (LO) GC is triggered. The difference is that LO does not perform object eviction.

Steps of GCE. Because LO is a slim-down version of GCE where it does not need to perform cold object classification and eviction, we will focus the discussion on GCE. For the ease of exposition of Polar, we will adapt G1 GC [14], which is the default GC since OpenJDK 9. G1 is a generational, stop-the-world GC that provides the best balance between latency and throughput. We modify the generational algorithm to restrict its scope. If the GC reaches a reference whose target is in the remote heap, we ignore the reference. The frequency of GC is relative to the usage of the local heap only, disregarding the remote heap (which is managed by Polar-agent, discuss in §2.5). GCE has two phases: (1) *Marking phase* whose goal is to identify live, cold objects, and (2) *Evacuation phase* segregates the live heap and evicts cold objects.

① *Marking phase.* Algorithm 1 shows our marking phase as the first step of the GC cycle. The algorithm takes as input the nursery and a number of memory regions belonging to the hot zone using the same heuristic of G1. We first identify a set of objects to serve as tracing roots (Line 1). They include objects referred to by global variables and stack variables. Next, for each tracing root O_r, we perform a BFS traversal to compute a transitive closure (Line 4 – 16). *gray* is a set containing a closure of transitively reachable objects from O_r. For each object's visit, we increase λ in its header (Line 8) and gather size of the object as well as λ to collectively calculate the coldness threshold ω (Line 9) to be used in the evacuation phase. Formally, ω is calculated as

Algorithm 1: Marking phase

```
1  Set roots ← TracingRoot()
2  foreach Object O_r ∈ roots do
3      Queue gray = {O_r}
4      while gray is NOT empty do
5          Object O ← Dequeue(gray)
6          if O.visited = false then
7              O.visited = true // mark O live
8              O.λ ++
9              ComputeOmega(O.size, O.λ)
10             foreach Outgoing reference e of O do
11                 Object O' ← Target(e)
12                 if Addr(O') ∈ RemoteHeap then
13                     // ignore cross-zone references
14                     continue
15                 else
16                     Enqueue(O', gray)
```

$\sum_{\text{Live Objects } O}(O.size \times O.\lambda)/\sum_{\text{Live Objects } O}(O.size)$. As mentioned earlier, all references leading to the remote heap are ignored (Line 14). For efficiency, we launch multiple parallel marking tasks which will synchronize once finished.

② *Evacuation phase.* After calculating ω, we are ready to evacuate cold objects, relieving the program from memory pressure. As in G1, this phase is stop-the-word: all mutator threads are paused during evacuation. Each live object will be moved to the corresponding hot/cold zones based on its λ value. After being copied, a forwarding reference is left at the original location so that their pointers can update correctly. The application is resumed after the evacuation phase.

Maintaining integrity and efficiency. There are several potential problems if naïvely relocating objects to remote servers. First, recall that the GC runs in the local heap and not the remote heap. The lack of a global view of the entire heap can cause dangling pointers – a severe problem. Figure 4a illustrates a simple example. When a cold object $C1$ that is sandwiched between two hot objects is relocated, the reference $C1 \rightarrow H2$ is not followed by the GC (in Alg. 1), resulting in $H2$ and $H3$ unreachable and will be (mistakenly) deallocated. To ensure memory safety, Polar captures all incoming references to the local heap (e.g., $C1 \rightarrow H2$) and includes the pointees (e.g., $H2$) in the set of GC roots (i.e., Alg. 1, Line 1). These captured references should also be monitored. Figure 4b is a snapshot after Figure 4a. Here, reference $H1 \rightarrow C2$ is destroyed, thus $C1$ is a dead object, causing $C1 \rightarrow H2$ to also be dead. If the GC is unaware of such a dead reference, $H2$ is still included as a GC root and

thus, $H2$ and $H3$ will not be collected and become memory leaks.

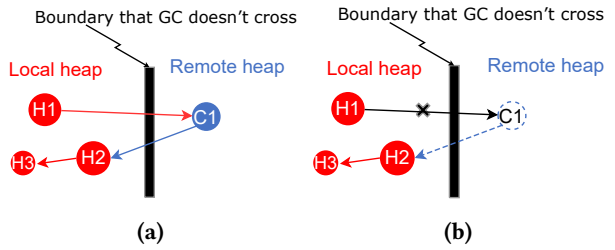

Figure 4: (a) Ignoring reference C1→ H2 will cause H2 & H3 to be mistakenly collected. (b) Unaware of the dead reference C1→H2 will cause H2 & H3 to become memory leaks.

Second, when an object is moved, the standard practice is to leave the new address at the original location. While it works for any intra-local heap relocation, it is problematic if there is a cold-to-hot reference. Using Figure 4a as an example where a cold object $C1$ has a reference to a hot object $H2$. If $H2$ is moved by the GC (to another location in the local heap, due to heap compaction), $C1$ has no way of knowing the new location of $H2$ to update the reference. Without updating, if we access $C1$ again, the reference will result in invalid data. A synchronization between the local heap and the remote heap is required but will be expensive.

In Polar, we develop an efficient solution: for each zone, it has an indirection structure called *trampoline* as a proxy for cross-zone access. Formally, a zone's trampoline records cross-zone, incoming references. A trampoline can be implemented as an array of entries, each recording pointee's address, and stored in an off-heap area. When an object is evicted to the cold zone, we are not leaving the remote address; instead, the address of the trampoline entry will be left as the forwarding reference.

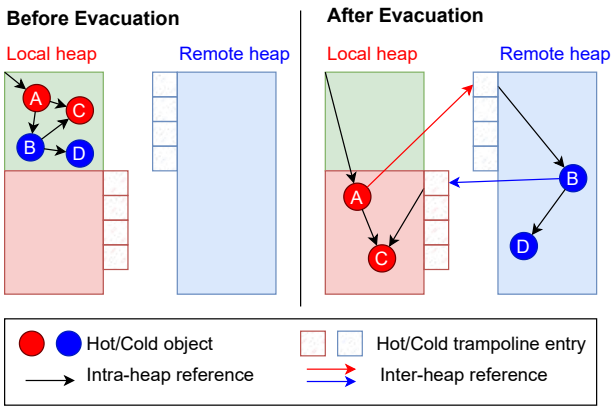

Figure 5: Heap snapshot before and after evacuation.

Figure 5 illustrates an example of a heap snapshot before (left) and after (right) evacuation. Consider the hot-to-cold reference $A \rightarrow B$ (stored in field f of A), after B is evicted to the remote heap, at original B's location, the address of an entry, says e of the cold trampoline is left. Field f is then updated with the address of e, which stores B's remote address. When we access field f of A, there is one-level indirection to be resolved to retrieve B's remote address. A similar treatment happens to the cold-to-hot reference $B \rightarrow C$. We do not have to worry about intra-zone references such as $A \rightarrow C$ and $B \rightarrow D$ because both the pointer and pointee are not crossing server boundaries.

Accessing cold objects. Cold objects are rarely needed by the mutator. As such, an obvious treatment is to perform, after resolving the indirection through the trampoline, RDMA operations on-demand. That is, for every heap read $b = a.f$ or heap write $a.f = b$, the runtime first checks if object O_a is in the remote heap. If so, the system redirects this access to a new RDMA-based path. Otherwise, the usual (local heap access) path is taken. While issuing singular RDMA operation for each access seems expensive, as our assumptions stand, such access to cold objects should be infrequent, making this handling acceptable.

2.5 Remote heap management

The remote heap could be managed using a spectrum of options. One option is that it can be treated akin to fast storage. In other words, cold objects are assumed to be immortal for the whole execution. As such, the remote heap is deallocated as a whole at the end of the execution. The treatment is clearly over-conservative and is not an efficient use of resources. For efficiency, a Polar-agent occasionally scans, reclaims dead objects, and compacts the remote heap. Due to space constraints, details of this agent are omitted.

3 Data hotness study

Polar is ongoing. In this Section, to establish the feasibility of Polar, we present a study examining data hotness of several real-world applications.

Setup & Methodology. We modified the codebase of OpenJDK v. 11u (a popular open-source version of the JVM) at various modules such as the GC, the memory allocator, the interpreter, and the multi-tiered compiler (C1 and C2) to enlarge the object's header and instrument all memory R/W to track data coldness λ as described in §2. The modified runtime system is then used to execute programs of Apache Spark v. 3.1.2 [8], a widely adopted distributed system for big data analytics. We used a cluster of three servers, each of which has 2 Intel(R) Xeon(R) Silver 4214R processors, 180GB of memory, and 500GB of SSD, connected via a Mellanox

ConnectX-6 card. Spark is run with Hadoop v. 3.3.1 [6]. Additionally, we run two programs from the DaCapo benchmark suite v. 9.12 [12] on one of the servers. Table 1 lists programs used in this study and their input datasets. These programs cover multiple categories of workload, including a typical Map-Reduce (WordCount), graph processing (PageRank, TriangleCount), and machine learning (KMeans, LinearRegression, and DecisionTree). From the DaCapo benchmark suite, H2 is a database program, and Tradesoap is a trading application based on Apache DayTrader J2EE. For each program, the heap size is configured to be at least three times the working set's size.

Programs	Datasets	Heap
Spark:		
WordCount	StackOverflow comments (8GB) [40]	10GB
PageRank	Wikipedia English (6GB) [43]	20GB
TriangleCount	Synthetic 1K nodes, 4K edges	5GB
KMeans	KDD2012 (5GB) [35]	20GB
LinearRegression	E2006-Unigram frequencies (2GB) [35]	12GB
DecisionTree	E2006-Unigram frequencies (2GB) [35]	12GB
DaCapo:		
H2	DaCapo Huge	2GB
Tradesoap	DaCapo Huge	2GB

Table 1: Programs, their input datasets and heap size used in our study.

In this study, no object eviction is done. At the end of each GC run, we used the calculated ω to compute the percentage of the heap that was hot vs. cold. Figure 6 shows the percentage of the heap that is hot (red) and cold (blue) for each program over time.

Results & Key Takeaways. Except for LinearRegression (with 8.13% of the heap is cold, on average), the *amount of cold data is significant* in the programs, ranging from 31.06% (DecisionTree) to 62.14% (TriangleCount) of the live heap, on average. More importantly, this result suggests that ω is a good proxy to capture the temporal property of the heap and reacts well to the application's dynamic behavior.

4 Related Work

Garbage Collection for Data-Instensive Workloads. There have been several works adapting GC algorithms for data-intensive workloads such as Yak [30], Taurus [26], Espresso [51], Panthera [46], Jade [50], among others [19, 23, 42, 52, 55]. Polar is similar in spirit to Semeru [47], Mako [25], and MemLiner [48], the GCs for memory-disaggregated environments that offload memory management tasks onto remote servers. However, they require a modified OS kernel to support the cache-and-swap mechanism while Polar does not. HCSGC [53] augments ZGC

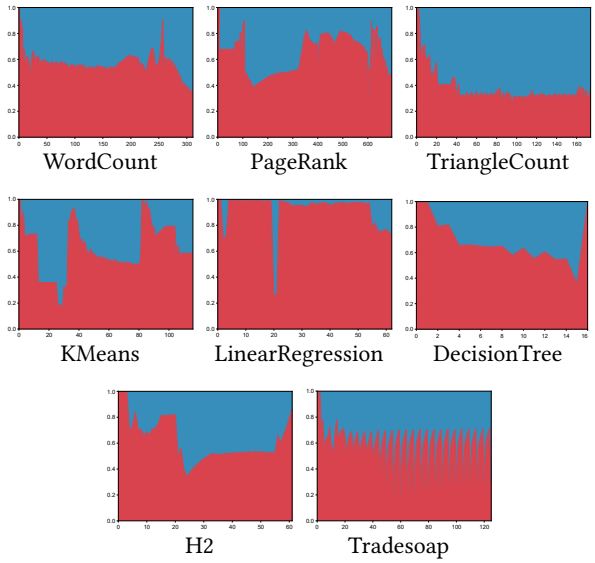

Figure 6: Heatmap of the heap of programs.

and segregates the live heap based on a simpler definition of hotness than Polar.

Resource Disaggregation. The has been a proliferation of remote-memory systems in the past few years. Remote memory is part of a general trend of resource disaggregation in datacenters [3, 5, 11, 18, 24]. Many optimizations and systems such as LegoOS [37], FaRM [15], Kona [13] and others [1, 4, 9, 20, 21, 32–34, 36, 38, 39, 41, 49, 56] have been developed to reduce remote latency. However, they all focus on low-level system stacks and do not consider the run-time characteristics of programs. They do not work well for managed applications such as Spark [54] and Hadoop [6] as well as [7, 29, 45]. Polar optimizes the runtime focusing on the far memory and does not require co-redesign support from the OS.

5 Conclusion

Far memory is an attractive solution to increase the scalability of modern workloads. This paper presents the design of Polar, a redesigned runtime for efficiently running managed applications in far memory. In Polar, the runtime automatically places objects at appropriate memory locations based on their access frequency to minimize remote accesses overheads, thereby improving end-to-end execution.

Acknowledgments

We thank the anonymous reviewers for their feedback in improving this paper. This work is supported by NSF grant CNS-2107010.

References

[1] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2018. Remote Regions: A Simple Abstraction for Remote Memory. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '18)*. 775–787.

[2] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2017. Remote Memory in the Age of Fast Networks. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*. 121–127.

[3] Marcos K. Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. 2019. Designing Far Memory Data Structures: Think Outside the Box. In *HotOS*. 120–126.

[4] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can Far Memory Improve Job Throughput?. In *EuroSys*. Article 14.

[5] Sebastian Angel, Mihir Nanavati, and Siddhartha Sen. 2020. Disaggregation and the Application. In *HotCloud*.

[6] Apache. 2005. Hadoop: Open-source implementation of MapReduce. http://hadoop.apache.org.

[7] Apache. 2024. Apache Flink. http://flink.apache.org/.

[8] Apache. 2024. Unified engine for large-scale data analytics. https://spark.apache.org/.

[9] Krste Asanović. 2014. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers. In *Keynote talk at the 12th USENIX Conference on File and Storage Technologies (FAST' 14)*.

[10] Krste Asanović, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. 2006. *The Landscape of Parallel Computing Research: A View from Berkeley*. Technical Report UCB/EECS-2006-183. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[11] Luiz Andre Barroso. 2011. Warehouse-Scale Computing: Entering the Teenage Decade. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA '11)*.

[12] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA*. 169–190.

[13] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. 2021. Rethinking Software Runtimes for Disaggregated Memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021)*. 79–92.

[14] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-first Garbage Collection. In *ISMM*. 37–48.

[15] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*. 401–414.

[16] Christine H. Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. 2016. Shenandoah: An Open-source Concurrent Compacting Garbage Collector for OpenJDK. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '16)*. Article 13, 9 pages.

[17] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*.

[18] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network Requirements for Resource Disaggregation. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. 249–264.

[19] 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* (Istanbul, Turkey) *(ASPLOS '15)*. ACM, New York, NY, USA, 661–673. https://doi.org/10.1145/2694344.2694361

[20] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with INFINISWAP. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (NSDI '17)*. 649–667.

[21] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. 2022. Clio: A Hardware-Software Co-Designed Disaggregated Memory System. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 417–433. https://doi.org/10.1145/3503222.3507762

[22] Intel. 2019. Intel High Performance Computing Fabrics. https://www.intel.com/content/www/us/en/high-performance-computing-fabrics.

[23] Haim Kermany and Erez Petrank. 2006. The Compressor: Concurrent, Incremental, and Parallel Compaction. In *PLDI*. 354–363.

[24] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. 2012. System-level implications of disaggregated memory. In *IEEE International Symposium on High-Performance Comp Architecture*. 1–12.

[25] Haoran Ma, Shi Liu, Chenxi Wang, Yifan Qiao, Michael D. Bond, Stephen M. Blackburn, Miryung Kim, and Guoqing Harry Xu. 2022. Mako: a low-pause, high-throughput evacuating collector for memory-disaggregated datacenters. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. 92–107. https://doi.org/10.1145/3519939.3523441

[26] Martin Maas, Tim Harris, Krste Asanović, and John Kubiatowicz. 2016. Taurus: A Holistic Language Runtime System for Coordinating Distributed Managed-Language Applications. In *ASPLOS*. 457–471.

[27] Hasan Al Maruf and Mosharaf Chowdhury. 2020. Effectively Prefetching Remote Memory with Leap. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*. 843–857.

[28] Mellanox. 2019. ConnectX-6 Single/Dual-Port Adapter supporting 200Gb/s with VPI. http://www.mellanox.com/page/products_dyn?product_family=265&mtag=connectx_6_vpi_card.

[29] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. 2013. Naiad: A Timely Dataflow System. In *SOSP*. 439–455.

[30] 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 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, 349–365.

[31] Oracle. 2019. The Z Garbage Collector. https://wiki.openjdk.java.net/display/zgc/Main.

[32] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 361–378. https://www.usenix.org/conference/nsdi19/presentation/ousterhout

[33] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. 2015. The RAMCloud Storage System. *ACM Trans. Comput. Syst.* 33, 3, Article 7 (Aug. 2015), 55 pages. https://doi.org/10.1145/2806887

[34] Yifan Qiao, Chenxi Wang, Zhenyuan Ruan, Adam Belay, Qingda Lu, Yiying Zhang, Miryung Kim, and Guoqing Harry Xu. 2023. Hermit: Low-Latency, High-Throughput, and Transparent Remote Memory via Feedback-Directed Asynchrony. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA, 181–198. https://www.usenix.org/conference/nsdi23/presentation/qiao

[35] Rong-En Fan. 2011. LIBSVM Data: Classification, Regression, and Multi-label. https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/.

[36] Zhenyuan Ruan, Malte Schwarzkopf, Marcos Aguilera, and Adam Belay. 2020. AIFM: High-Performance, Application-Integrated Far Memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*.

[37] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*. 69–87.

[38] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. 2019. Shoal: A Network Architecture for Disaggregated Racks. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation (NSDI '19)*. 255–270.

[39] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. 2020. StRoM: Smart Remote Memory. In *EuroSys*. Article 29.

[40] StackExchange. 2022. StackOverflow Comments. https://archive.org/download/stackexchange.

[41] Brian R. Tauro, Brian Suchy, Simone Campanoni, Peter Dinda, and Kyle C. Hale. 2024. TrackFM: Far-out Compiler Support for a Far Memory World. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (La Jolla, CA, USA) *(ASPLOS '24)*. 401–419. https://doi.org/10.1145/3617232.3624856

[42] Gil Tene, Balaji Iyengar, and Michael Wolf. 2011. C4: The Continuously Concurrent Compacting Collector. In *Proceedings of the International Symposium on Memory Management* (San Jose, California, USA) *(ISMM '11)*. ACM, New York, NY, USA, 79–88. https://doi.org/10.1145/1993478.1993491

[43] The KONECT Project. 2013. KONECT networks datasets. http://konect.cc/networks/.

[44] Shin-Yeh Tsai and Yiying Zhang. 2017. LITE Kernel RDMA Support for Datacenter Applications. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. 306–324.

[45] Twitter. 2011. Storm: distributed and fault-tolerant realtime computation. https://github.com/apache/storm.

[46] Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. 2019. Panthera: Holistic Memory Management for Big Data Processing over Hybrid Memories. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. 347–362.

[47] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2020. Semeru: A Memory-Disaggregated Managed Runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Banff, Alberta, 261––280.

[48] Chenxi Wang, Haoran Ma, Shi Liu, Yifan Qiao, Jonathan Eyolfson, Christian Navasca, Shan Lu, and Guoqing Harry Xu. 2022. MemLiner: Lining up Tracing and Application for a Far-Memory-Friendly Runtime. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)*. 35–53.

[49] Chenxi Wang, Yifan Qiao, Haoran Ma, Shi Liu, Wenguang Chen, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2023. Canvas: Isolated and Adaptive Swapping for Multi-Applications on Remote Memory. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 161–179. https://www.usenix.org/conference/nsdi23/presentation/wang-chenxi

[50] Mingyu Wu, Liang Mao, Yude Lin, Yifeng Jin, Zhe Li, Hongtao Lyu, Jiawei Tang, Xiaowei Lu, Hao Tang, Denghui Dong, Haibo Chen, and Binyu Zang. 2024. Jade: A High-throughput Concurrent Copying Garbage Collector. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. 1160–1174. https://doi.org/10.1145/3627703.3650087

[51] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. 2018. Espresso: Brewing Java For More Non-Volatility. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (VA, USA). ACM.

[52] Mingyu Wu, Ziming Zhao, Yanfei Yang, Haoyu Li, Haibo Chen, Binyu Zang, Haibing Guan, Sanhong Li, Chuansheng Lu, and Tongbao Zhang. 2020. Platinum: A CPU-Efficient Concurrent Garbage Collector for Tail-Reduction of Interactive Services. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 159–172. https://www.usenix.org/conference/atc20/presentation/wu-mingyu

[53] Albert Mingkun Yang, Erik Österlund, and Tobias Wrigstad. 2020. Improving Program Locality in the GC Using Hotness. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 301–313. https://doi.org/10.1145/3385412.3385977

[54] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*.

[55] Wenyu Zhao, Stephen M. Blackburn, and Kathryn S. McKinley. 2022. Low-latency, high-throughput garbage collection. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. 76–91. https://doi.org/10.1145/3519939.3523440

[56] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. 2022. Carbink: Fault-Tolerant Far Memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 55–71. https://www.usenix.org/conference/osdi22/presentation/zhou-yang