Pufferfish: Container-driven Elastic Memory Management for Data-intensive Applications

Wei Chen, Aidi Pi, Shaoqi Wang, and Xiaobo Zhou
Department of Computer Science
University of Colorado, Colorado Springs, CO, USA
{cwei,epi,swang,xzhou}@uccs.edu

ABSTRACT

Data-intensive applications often suffer from significant memory pressure, resulting in excessive garbage collection (GC) and out-of-memory (OOM) errors, harming system performance and reliability. In this paper, we demonstrate how lightweight virtualization via OS containers opens up opportunities to address memory pressure and realize memory elasticity: 1) tasks running in a container can be set to a large heap size to avoid `OutOfMemory` (OOM) errors, and 2) tasks that are under memory pressure and incur significant swapping activities can be temporarily "suspended" by depriving resources from the hosting containers, and be "resumed" when resources are available. We propose and develop *Pufferfish*, an elastic memory manager, that leverages containers to flexibly allocate memory for tasks. Memory elasticity achieved by Pufferfish can be exploited by a cluster scheduler to improve cluster utilization and task parallelism. We implement Pufferfish on the cluster scheduler Apache Yarn. Experiments with Spark and MapReduce on real-world traces show Pufferfish is able to avoid OOM errors, improve cluster memory utilization by 2.7x and the median job runtime by 5.5x compared to a memory over-provisioning solution.

CCS CONCEPTS

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

KEYWORDS

cloud computing, containerization, memory management, cluster scheduling

ACM Reference Format:
Wei Chen, Aidi Pi, Shaoqi Wang, and Xiaobo Zhou. 2019. Pufferfish: Container-driven Elastic Memory Management for Data-intensive Applications. In *ACM Symposium on Cloud Computing (SoCC '19), November 20–23, 2019, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3357223.3362730

SoCC '19, November 20–23, 2019, Santa Cruz, CA, USA

ACM ISBN 978-1-4503-6973-2/19/11…$15.00
https://doi.org/10.1145/3357223.3362730

1 INTRODUCTION

Data-intensive applications have become increasingly popular in many fields, such as data mining, machine learning, and database query. To enable data-parallel processing on large datasets and to improve hardware utilization, those applications are often executed in a shared cluster and scheduled by a modern resource manager [21, 42]. A typical resource manager relies on the estimation of workload resource (e.g., CPU cores and memory) by users to perform resource allocation and enforce resource limits. Currently, most resource managers employ static resource allocation mechanisms. Therefore, the allocated resource remains unchanged and cannot be revoked during application runtime.

Popular data-processing frameworks such as Hadoop [4], Spark [49], Dryad [25] and Hyracks [7], are implemented using managed languages (e.g., Java) in which runtimes commonly have built-in automatic memory management. For these frameworks, garbage collection (GC) often causes significant memory and CPU overhead. It even fails applications with OOM errors when the JVM heap is not configured properly. However, the JVM heap usage depends on many application-related factors, such as the size of the input data, the size of the generated intermediate results, and the execution stage, which vary largely across different applications and during application runtime.

As a result, it is almost impossible to accurately estimate the required memory for data-intensive applications before execution. As manual configuration and tuning are tedious and ineffective, users tend to provision task memory size based on the measured peak demand. However, for the prevalent static cluster resource allocation, the resource request acceptance decisions are made based on the resource availability on each individual node. The prevalent static cluster resource allocation accepts a resource request based on the resource availability on each node. A task normally has to wait until sufficient memory becomes available. Thus, memory over-provision often leads to under-utilized cluster resource and significant job queuing delay when allocated but inefficiently used cluster resources accumulate.

In this paper, we focus on data-intensive applications that have a large memory demand and a long running time (e.g., iterative machine learning applications and large batch workloads), in Java-based frameworks. We identify two major challenges: (1) improper memory setting causes individual tasks to suffer from performance degradation or even OOM; (2) dynamic and unpredictable task memory behaviors under static memory management cause significant resource wastage and queuing delay.

We present *Pufferfish*, a container-driven application-agnostic elastic memory manager. To address the first challenge, Pufferfish proposes an effective yet simple approach based on lightweight

virtualization to address the performance issue and OOM of an individual task. The key idea is to leverage OS containers to limit the memory usage of a task while setting a sufficiently large heap size to avoid OOM. In this scenario, a task under memory pressure incurs disk swapping instead of running out of memory. However, simply running the task with a large heap incurs significant overhead and degrades system performance due to excessive disk accesses. Therefore, we temporarily suspend the swapping container by capping its CPU resource to a very low level such that thrashing is throttled but the task is still alive.

To address the second challenge, Pufferfish monitors the memory usage of application containers and optimizes memory allocation by dynamically adjusting the container memory size on the fly. As long as free node memory is detected and a task demands more memory (when JVM heap usage increases), Pufferfish increases the container memory size to avoid container swapping using a memory *puff* mechanism. However, if memory contention is detected when Pufferfish serves memory requests from multiple containers on the same node, a heuristic is used to selectively increase the memory of prioritized containers and depress the rest. If a node's memory is insufficient to start a new job, Pufferfish uses *reclaim* mechanism to squeeze the memory of puffed containers. Pufferfish achieves memory elasticity so that improved cluster memory usage can be translated to higher parallelism, lower queuing delay and improved job completion time.

The ideology of Pufferfish is similar to that of VM ballooning. However, Pufferfish is designed to be deployed within an operating system, which requires different techniques and poses different challenges and opportunities. For instance, in Pufferfish, the memory consumption of containers can be obtained via Linux cgroup interfaces while VM ballooning requires a balloon driver to indirectly infer the memory demand of VMs. Pufferfish is transparent to applications and compatible with existing cluster schedulers. We implemented it on a popular cluster scheduler, Yarn [42]. Experiments with Google trace [38] on a 26-node cluster show that Pufferfish can help applications with a large memory footprint avoid OOM errors, and survive from memory pressure with less than 10% performance overhead. Through its memory elasticity, Pufferfish improves cluster utilization by 2.7x and the median job runtime by 5.5x compared to a memory over-provisioning solution.

2 IMPACT OF TASK MEMORY SIZE

For Java-based applications, the task memory size is given by setting the maximum JVM heap size (i.e., parameter -Xmx). If the JVM heap size is not set properly and there is not enough memory in the JVM heap for creating new objects, the JVM throws an OOM error and the task fails. Determining the optimal heap size for applications is not realistic. The aggregated heap size of all tasks on the same node should not be too large to cause memory swapping. Thus, setting a large per-task heap size limits the degree of parallelism for application tasks in one machine, which leaves potential concurrency unexploited. In contrast, a small heap size jeopardizes task execution with degraded performance and OOM errors.

We profile the task memory usage of representative data-intensive applications: Spark Kmeans, Spark Pagerank and MapReduce Terasort. Figure 1 illustrates how JVM heap sizes affect job performance

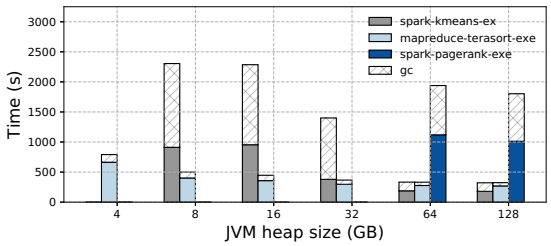

Figure 1: Job runtime of Spark Kmeans, Spark Pagerank and MapReduce Terasort with different heap sizes.

by breaking down the job runtime into execution time and GC time. For MapReduce Terasort, we only modify the heap size of reduce tasks, since Terasort is bottlenecked by the reduce phase.

For the three applications, their performance is improved by increasing the JVM heap size. Kmeans throws OOM at a 4GB heap size. Pagerank only survives when heap size is larger than 64GB. However, the 64GB heap boosts the performance of Kmeans by 5x over the 8GB heap. GC is often not triggered by a large heap size, thus GC overhead is decreased with the increase of heap size. We further find that the job execution also benefits from a large heap size. Frameworks like Spark and MapReduce reserve memory pools to store intermediate data [48] (for Spark) or to improve I/O performance (for MapReduce). In case of memory insufficiency, data cached in the memory pool is spilled to disks, resulting in excessive I/O and degraded performance. We notice that this phenomenon is significant on Spark Kmeans, since Kmeans is an iterative application and its intermediate results are immediately needed in the next iteration. A big heap size not only reduces the GC overhead but also avoids spilling.

Memory heterogeneity opportunity. Figure 2 shows the 50^{th} and the 90^{th} percentiles of memory usage of each container[1] throughout the application execution time. The results show that there is a heterogeneous memory distribution across the executors. Executor-9 uses much more memory than others do. After examining the Spark logs and repeating the experiment with various inputs, we find Executor-9 is cached with more intermediate data (as Spark RDD) and scheduled with more Spark tasks than other executors. We generally denote a task as the scheduling entity managed by a cluster scheduler, such as a Spark executor and a MapReduce task. Note that a Spark task means the execution thread in each executor. We conclude three causes of heterogeneous memory distribution: (1) Skewed input data. The intermediate results might be gathered on the hotspot executor after shuffling [15]. (2) Localization. The hotspot executor residing on the node with more localized data might be scheduled with more Spark tasks [1, 34, 48]. (3) Stragglers. The executor with less or without straggler tasks might be scheduled with more Spark tasks [27, 28]. In these situations, a unified task memory configuration makes memory over-provisioning a necessity to avert OOM.

Memory dynamics opportunity. The profiled memory usage also shows that a task's memory usage can be significantly below its assigned heap size. For one third of the execution time, Spark

[1]We ran Spark executor in a Docker container managed by Yarn.

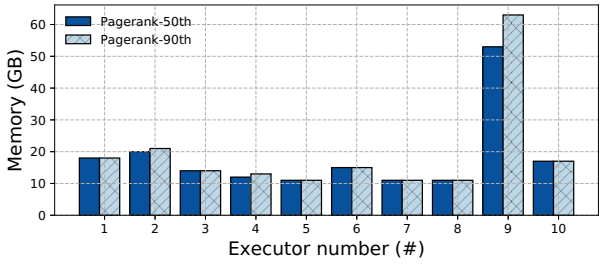

Figure 2: Memory usage of Spark Pagerank with different number of executors.

Pagerank, Spark Kmeans, and MapReduce Terasort jobs only utilize 49%, 64%, and 86% its configured heap, respectively, although the heap is not over-provisioned. It demonstrates that the memory usage for a single task is dynamic, and much memory is unused during its execution. Causes include: (1) JVM uses lazy memory allocation mechanism, which means memory is allocated on demand. As a result, application memory usage is slowly increased from a small value. (2) Application behavior is dynamic. Intermediate data might be dropped by the application and collected by the GC at runtime.

In this paper, we consider that memory heterogeneity and memory dynamics are prevalent in data-intensive applications, which provide Pufferfish the opportunities to achieve memory elasticity in datacenter scheduling.

3 CONTAINERIZATION

3.1 Avoiding OOM Errors

In Pufferfish, the key to avoiding Java OOM errors is to set a large enough heap size for JVMs running inside OS containers. The heap size (e.g., 64GB) should be larger than the physical memory limit of the container (e.g., 4GB). Thus, the JVM heap is built on the virtual memory of the container rather than on its allocated physical memory. In this case, when a task is under memory pressure, the container will start to swap recently unused data to the disk. The memory overcommitment supported by the OS allows JVMs to run with a much larger heap size than the physical memory limit. With this mechanism, a task will not run into OOM errors but start memory swapping once the task memory usage exceeds the container memory size. However, application performance could be degraded.

Note that one alternative is to simply kill the swapping JVM. However, Pufferfish targets long-running applications, such as offline batch jobs and iterative machine learning jobs that can run for hours or even days. Killing causes substantial slowdown due to the loss of execution progress. It also leads to low cluster utilization.

3.2 Suppressing Memory Thrashing

While containerization and a large JVM heap help avoid OOM errors, the problem shifts from JVM heap pressure to container memory pressure. When task memory demand grows beyond the memory limit of a container, memory swapping or thrashing significantly slows down the task execution and even affect other tasks on the same node. Further, GC is not effective since heap scanning under memory pressure does not reclaim much memory [14]. To address these issues, we temporarily *suspend* a task in trouble by reducing the CPU allocation of the container to a low level. Since memory access in a swapping container generates paging activities, the reduced CPU usage reduces the rate at which the JVM accesses memory so as to reduce its contribution to paging activities. The low CPU allocation should be sufficient so that the cluster scheduler does not consider the task as a failed one.

Identifying OCM heuristic. We use a heuristic to identify containers in which the running processes are out of container memory (OCM): *if the sum of the memory usage and swap usage are larger than the memory limit and there are swapping activities detected, the container should be suspended.* For an OCM container, Pufferfish suspends its execution to confine its swapping and GC activities. After an OCM container is suspended, Pufferfish checks the node memory availability and determines if the suspended container is able to *puff* (by puffing, the memory capacity of a container is increased). After multiple rounds of puffing, the container size will be adjusted to its actual demand during execution. By this design, containerization enables tasks to utilize temporarily available node memory (sourced from memory over-provision, memory dynamics and heterogeneity opportunities), achieving memory elasticity. Pufferfish may also trigger memory *reclaim* by forcing memory pages to be swapped to disk under node memory contention. Suspended and reclaimed containers can be resumed to normal execution with sufficient CPU and memory if memory availability is detected when completed containers release their allocated resources.

To suspend an OCM container, we minimize its CPU usage and pin task threads (including working threads and GC threads) to a single core while maintaining sufficient footprint for this task to be alive. We experimentally set the low container CPU usage to 1% and restrict the container only to run on CPU 0. Because 1% of a CPU usage still enables the task in the suspended container to send heartbeats to the cluster scheduler, the cluster scheduler will not consider this task dead and restart a new one. Note that the task performance is significantly degraded under swapping. For data-intensive applications (e.g., Hadoop, Spark), heartbeat mechanism is often implemented in a separate thread and the thread is woken up to send heartbeats every 3-5 seconds. Our experiment shows that a heartbeat thread woken up at this frequency can still be kept alive even the entire task is only assigned with 1% CPU quota. We do not expect the task to make progress in this situation, but the task and its intermediate results should be kept alive. All these resource manipulations are conducted through the `cgroup` subsystem that corresponds to the container. Although heap over-provisioning and containerization avoid OOM, they do not achieve memory elasticity. To that end, we discuss elastic memory management in section 4.3.

Address swapping. In the previous section, we advocate to suspend a swapping task by reducing its CPU usage. In this section, we use an example to demonstrate the necessity of suspension to address significant I/O traffic and JVM garbage collection. In order to emulate a scenario where a task is consecutively swapping, we run a Spark Pagerank job, then manually reclaim its memory by shrinking the container size, and finally resume its execution by

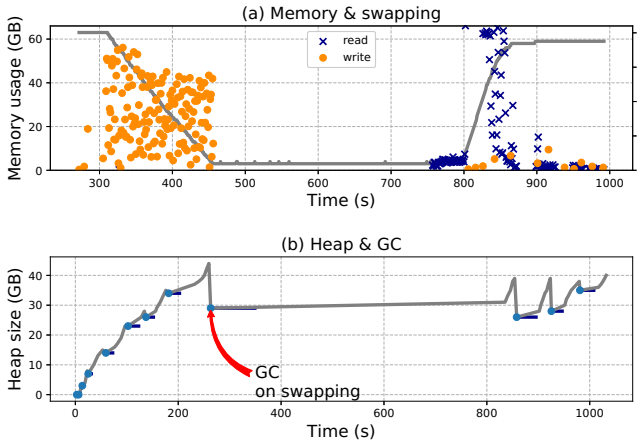

Figure 3: (a) Pufferfish effectively suspends Spark Pagerank without incurring much memory thrashing. The gray line represents the container memory size. Each dot represents an IO operation (read/write). (b) A full GC on swapping can be observed at 270^{th} second when the container starts reclaiming. The gray line represents the JVM heap size. Each dark blue line represents the time of a full GC.

recovering the container size. We configure a one-executor Pagerank with the optimal 64GB heap (a heap larger than 64GB does not yield better performance).

Figure 3-(a) shows the memory usage and the container swapping activities. We start reclaiming the container at the 265^{th} second (an arbitrary chosen moment). Significant disk write activities (as high as 2GB/s) are observed while the container is reclaimed from 64GB to 4GB because of swapping. At the 450^{th} second, we reduce the container's CPU usage to 1% and the swapping activities are quickly throttled with the disk/read rate dropping to below 1MB/s because paging is constrained under low CPU usage. During suspension, we still observe heartbeats between the Spark executor and the master. This illustrates that container suspension successfully confines the swapping activities. Note that in such a case, the working set size of the task is far beyond the container size. Thus, it is not helpful for fast memory retrieval if the task is being swapped without reducing CPU usage. Additionally, it causes significant I/O traffic. Finally, the container's memory and CPU are restored to 64GB and 100% respectively at the 750^{th} second. With sufficient memory, the container experiences a burst of disk activities in order to load the working set into memory.

Figure 3-(b) shows the full GC activities and the JVM heap usage. Even though the task is configured with a large heap, an aggressive full GC upon swapping can be observed at the 270^{th} second, which lasts for 89 seconds. As a result, the used JVM heap is dropped from 45GB to 30GB. After analyzing the GC logs, we find the full GC is caused by an OS memory allocation failure. The reason is that JVM memory allocation requests to the OS (by `mmap()` or `malloc()`) fail on container memory shortage, activating the GC to try to reclaim memory from the JVM heap. Two insights can be drawn from this experiment. (1) GC on swapping is useful in reducing the committed memory, especially when the surviving objects after GC are sparse. Since each task is given an illusion of a large available memory due to the configured large heap size, it allows tasks to exploit the memory elasticity when the host has sufficient memory. On the other hand, tasks can still leverage GC to reduce the JVM heap to confine swapping. (2) The excessive GC, which spawns many GC threads, causes strong CPU interference to other tasks running on the same node if not well handled. Pufferfish addresses this issue by reducing the container CPU usage to a low level and confining the container's CPUSET to one core.

The experiment provides following insights on our proposed container-based memory management:

- By reducing the CPU usage, the disk swapping activities are successfully throttled.
- Task is still alive by keeping a low CPU footprint of 1%.
- Swapping during suspension still triggers full GC, which effectively reduces the committed memory.

4 PUFFERFISH DESIGN AND IMPLEMENTATION

4.1 FLEX container

Pufferfish is built on top of Apache Yarn. To leverage our proposed functionalities, We define a new type of container, FLEX, whose size can be dynamically adjusted during execution by Pufferfish. In contrast, a REGULAR container is Yarn's default container, whose size is static and determined at request time. FLEX containers are designed for the long-running data-intensive applications with large memory demands. We derive the property of FLEX containers as follows:

- To avoid OOM, task running in a FLEX container is set with the same large heap size `MAX_HEAP`, representing the maximum memory demand among all application tasks. Since each container at most holds memory of an entire node, `MAX_HEAP` value is capped by memory capacity of a node.
- To avoid memory waste, all FLEX containers should be started with the same small memory size (`MIN_CONT`) , representing the minimum memory demand among all application tasks.

Currently, users need to specify the values of `MAX_HEAP` and `MIN_CONT` for FLEX containers. Cluster operators may apply their previously used over-provisioned task memory size for `MAX_HEAP`. The difference lies in that both the container size and the JVM heap size are over-provisioned in Yarn, while only the heap size is over-provisioned in Pufferfish.

4.2 System Components

We implement Pufferfish by extending Apache Yarn. Yarn is a general resource management framework that allows applications to negotiate resources on a shared cluster. Yarn uses container, a logical bundle of resources (e.g., ⟨1 CPU, 2GB RAM⟩), as the resource allocation unit. In Yarn, the ResourceManager, one per cluster, is responsible for allocating containers to each node. The ApplicationMaster, one per application, submits requests for containers to ResourceManager. The NodeManager, one per node, monitors node status and updates ResourceManager with resource availability. The key components of Pufferfish include one *node memory*

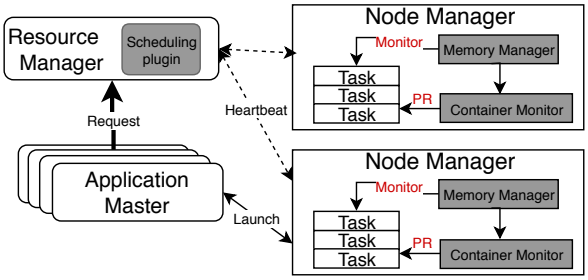

Figure 4: Architecture of Pufferfish.

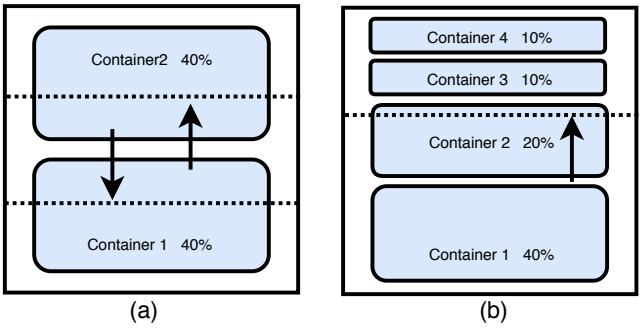

Figure 5: Demonstration of memory contentions.

manager per NodeManager, one *container monitor* per container, and a *scheduling plugin* in ResourceManager. The architecture of Pufferfish is shown in Figure 4.

Node Memory Manager, implemented in NodeManager, is responsible for monitoring node memory usage and instructing Container Monitor to resize a container. In the Node Memory Manager, we define two functions, `puff()` and `reclaim()`, shown as PR in Figure 4. Function `puff()` periodically selects FLEX containers under OCM and computes the amount of memory that can be added to them. Function `puff()` is invoked every two seconds which equals to the heartbeat interval between ResourceManager and NodeManager since containers might be released or launched during this interval. The `puff()` stops puffing when either all FLEX task memory demands are satisfied, i.e., no more OCM containers are detected, or the node memory is almost run out. Function `reclaim()` is called whenever a new container is to be launched on the node. At that moment, the Node Memory Manager needs to check if the node has enough memory. If not, it chooses one of the FLEX containers and computes how much memory belonging to this container to reclaim. We describe how the Node Memory Manager chooses containers in Section 4.3.1.

Container Monitor is a per-container daemon implemented in NodeManager that is responsible for memory monitoring and adjusting. It is instructed by the Node Memory Manager to increase or decrease container memory through the interface provided by `cgroup`. It also maintains container state and its corresponding action. For example, when a container is detected under OCM, its Container Monitor immediately suspends the container by setting its CPU allocation to 1% and limiting its CPUSET to a single core. In contrast, if a suspended container is under OCM, which means the memory demand of this container is satisfied after puffing, Container Monitor resumes its CPU and CPUSET. Pufferfish is built at the resource management layer of Yarn and is transparent to applications. We have open-sourced Pufferfish at: https://github.com/yncxcw/pufferfish.

4.3 Elastic Memory Management

For elastic memory management in clusters, Pufferfish develops two node-level mechanisms, one cluster scheduler plugin, and adopts two prioritization polices.

4.3.1 *Two Node-level Mechanisms.*

Puff. Given a set of OCM containers, function `puff()` determines the amount of memory that can be added to these containers. We present a simple method to quickly adapt the container sizes to their actual demands. Consider a container with size M that is currently suspended due to OCM and has been selected to puff. In the next round with puff ratio ϕ, the container size will be $M \times (\phi + 1)$. The container size keeps increasing in each round until its demand is satisfied.

Determining an optimal ratio ϕ for all applications is non-trivial. A large ϕ causes waste, while a small ϕ causes significant overhead because of frequent suspension and puffing. A practical method is to profile representative data-intensive applications and express the memory usage as a function of time $M(t)$. Its memory growth rate can be represented as $M(t+\delta)/M(t)$. If we set δ as the Container Monitor interval, $M(t+\delta)/M(t)$ represents the growth rate between two consecutive puffing rounds. Generally, ϕ can be set by choosing a relatively large growth rate from all profiled applications. By this setting, there is little overhead on the containers caused by memory insufficiency. The additional unused memory can later be reclaimed.

We find that applying the same puff ratio for all containers on the same node can cause memory contention. As shown in Figure 5-(a), after a few rounds of puffing, each container might hold 40% of the memory on the node machine but is still demanding more memory. In this case, both containers are suspended with OCM state and cannot make further progress, because the node has almost run out of its memory. To resolve memory contention, we develop a backoff-based puffing algorithm. It works by sorting the containers on the same node by their priorities and giving the default ratio to the container with the highest priority and backing off for the rest. For example, if there are multiple OCM containers on a node, the container with highest priority gets the default puff ratio ϕ, the second gets ratio ϕ/N_c, the third gets ratio $\phi/(N_c)^2$, and so on. By enforcing the priorities, memory contention is avoided by giving more puffing chances to high priority containers and suspending low priority containers. The priority is defined by the cluster-level policy in section 4.3.3.

Avoiding memory contention is not effective when a node memory is about to be used up, however. As shown in Figure 5-(b), container-1 with the highest priority is under OCM and demanding more memory while the node cannot satisfy the request. To avoid OOM of the node, `puff()` kills the container with the lowest priority because killing and relaunching it incurs the least performance loss.

Algorithm 1 Memory puff function: `puff()`.

```
1:  Variables: Memory puff ratio φ, Set of sorted OCM containers
    ℂ, Node allocated memory A, Node total memory M.
2:  /* ℂ is sorted by container priority */
3:  N_c = ℂ.size();
4:  if A ≥ M then
5:      if ℂ.get_first().is_ocm() then
6:          kill(ℂ.get_last());
7:      end if
8:      return;
9:  end if
10: for each container c in ℂ do
11:     if A ≥ M then
12:         break
13:     end if
14:     if !c.is_ocm() then
15:         continue
16:     end if
17:     /* Avoid one container to dominate available M − A */
18:     bm = min(c.get_memory() × φ, (M−A)/2);
19:     c.add_memory(bm);
20:     A+ = bm;
21:     φ/= N_c;
22: end for
```

Details of the `puff()` function are shown in Algorithms 1. Function `is_ocm()` implements the heuristic of identifying OCM. In line 14, we skip containers whose memory demands are satisfied. Lines 10 to 21 implement the backoff-based puffing. Lines 4 to 9 kill the container with the lowest priority if the memory demand from the container with the highest priority cannot be satisfied. To track the set of OCM containers ($\mathbb{C}$), each container is augmented with a timestamp. If a container is detected as OCM, it is put in set $\mathbb{C}$ with a timer bound. If a container is not detected as OCM for more than a time threshold (e.g., 2 minutes), which means its container size is sufficient for its memory demand, it will be removed from set $\mathbb{C}$ by Node Memory Manager. Container Monitor also shrinks the container size if memory slack is detected. For instance, if a container is puffed from 50GB to 70GB while its actual demand is 60GB, then 10GB is not actively used. Thus, to avoid memory fragmentation, the puff mechanism shrinks the container size to 62GB (2GB overhead by default).

Reclaim. Before Pufferfish launches a container on a node, it checks if the node has enough memory. If not, function `reclaim()` is called to reclaim memory based on memory availability and memory demand. For example, if a REGULAR container requests 5GB memory, Pufferfish ensures there is 5GB free memory. But for a FLEX container, Pufferfish only needs to ensure free space of `MIN_CONT`. Reclaiming starts from the OCM container with the lowest priority to avoid overhead on REGULAR containers. Once containers exit and release their occupied memory, reclaimed containers are puffed immediately. Note that containers under reclaim are suspended. To reduce the overhead, Pufferfish uses a lazy approach that delays the memory reclaim until the node memory cannot satisfy a newly scheduled container.

4.3.2 Cluster Scheduler Plugin. When Pufferfish applies puff and reclaim mechanisms for cluster memory management, there exists one major issue. The risk of task killing on some nodes would be significantly high due to memory contention on those nodes if the memory utilization is unbalanced across the cluster. The cause is that the cluster scheduler is unaware of the actual puffed memory usage of containers on individual nodes. Yarn ResourceManager keeps track of cluster-wide resource allocation based on job resource requests. However, the requested memory for a FLEX container (`MIN_CONT`) is always smaller than the actually used memory, which makes it difficult for ResourceManager to identify a suitable node for container allocation.

We develop one cluster scheduler plugin to address the issue. First, we augment the heartbeat between ResourceManager and NodeManager to convey the actual memory usage of each node. Second, to avoid FLEX containers overwhelming one node, we develop a memory-aware heuristic to direct container allocation in the cluster. When scheduling a FLEX container, Pufferfish delays the request if it cannot be satisfied due to insufficient memory of a node, to avoid meaningless memory reclaim. The heuristic also tries to balance the memory usage of FLEX containers on each node because they likely dominate one node. To this end, when choosing a node for container placement in the cluster, Pufferfish prioritizes the node with the most amount of available memory. If several nodes have the same amount of available memory, Pufferfish chooses the node with data locality. The heuristic implementation requires less than 20 lines of code. It can be easily ported to any Yarn compatible scheduler. The plugin enables the existing cluster schedulers to be memory-elasticity aware.

4.3.3 Prioritization Policies. Pufferfish can be coupled with different cluster-level policies so as to achieve various memory management objectives. In this paper, we aim to maximize cluster memory utilization and improve job runtime. We adopt two prioritization policies that determine the order of puffing for containers on the same node upon memory pressure.

- **Earliest Job First (EJF)** policy prioritizes containers based on the arrival time of the job that the containers belong to. This is the default policy in Pufferfish. Its rationale is that the oldest job may release memory first.
- **Shortest Job First (SJF)** policy sorts containers based on the expected job completion time, giving the shortest job the highest priority. This policy requires estimation of job duration which can be inferred from historical logs. Its rationale is that long jobs should be penalized as short jobs may release memory sooner. Note that the actual job completion time may vary due to various factors.

We introduce suspension tolerance as the maximum duration a container can be suspended. A task will be killed if it is suspended longer than the suspension tolerance. We expect the killed containers to be launched on another node with sufficient resources. By default, suspension tolerance is set to half of the expected duration of a job. This enhancement is useful for those FLEX containers that are allocated with a huge amount of memory but suspended for long time, which causes a lot of memory waste. Note that under SJF policy long jobs can suffer from starvation when short jobs keep arriving. Pufferfish increases a suspended job one priority

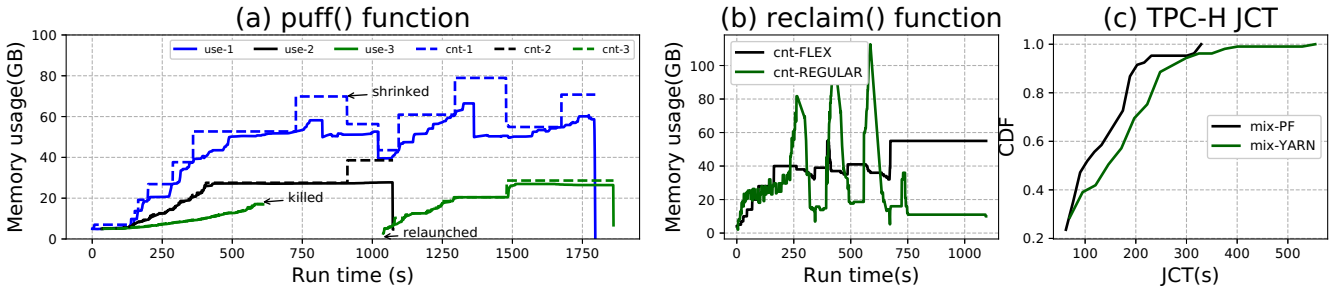

Figure 6: Efficacy of `puff()` and `reclaim()` functions in the single-node cluster. The solid line in (a) and in (b) shows the real memory usage while the dashed line draws the container memory size.

level when the job has been suspended for half of the suspension tolerance, and eventually starvation is avoided.

For producer-consumer scenario where consumer tasks wait on producer tasks (e.g., reduce tasks wait on map tasks), as Pufferfish assigns the same puff ratio to tasks belonging to the same job, their memory allocation can progress at a similar pace. However, it is possible that a producer container demands much more memory and gets suspended while the consumer container waits on it. In this case, Pufferfish simply kills any container once it has been suspended for too long in order to avoid deadlock.

5 EVALUATION

We evaluate Pufferfish by using representative MapReduce and Spark workloads. We first evaluate Pufferfish on a single-node cluster (the cluster only has one slave node) to validate efficacy of node-level puff and reclaim mechanisms. We then evaluate Pufferfish in a multi-node cluster using the Google trace [38]. We finally discuss Pufferfish overhead and parameter sensitivity.

5.1 Experimental Setup

Cluster setup The multi-node cluster in the experiment consists of 26 nodes. Each node has two 8-core Intel Xeon E5-2640 processors with hyper-threading enabled, 128GB of RAM, and 5x1-TB hard drives configured as RAID-5. The machines are interconnected by 10Gbps Ethernet. Each machine is installed with Ubuntu-16.04. Pufferfish implementation is based on Yarn-2.7.3. We use the same Yarn version for comparison. We configure each node with 120GB memory for Yarn (excluding OS and Yarn daemon usage). We use Spark-2.0.1 for Spark applications. Docker-1.12.1 is used to create OS containers. The image is downloaded from online Docker hub `sequenceiq/hadoop-docker`.

Pufferfish setting Based on our initial application profiling, we configure the puff ratio ϕ to 40%. For FLEX containers, we configure `MIN_CONT` to 4GB and `MAX_HEAP` to 64GB to ensure Spark-PR and Spark-NW do not run into OOM.

Workloads Several of our data-intensive workloads are chosen from HiBench [22]. Table 1 lists their input size, number of executors for Spark workloads (e.g., 1 and 6 for small and large Spark-Wordcount workloads, respectively), and number of reduce tasks for MapReduce workloads. Wordcount (WC), Terasort (TS), and Invertedindex (II) are both memory intensive and I/O intensive batch workloads. Kmeans (KM), Pagerank (PR), and Nweight (NW) are representative machine learning workloads with long runtime and excessive memory usage. We use the small dataset for the single-node cluster evaluation. We use both small and large datasets for the multi-node cluster evaluation. We follow the default configuration of Spark and Hadoop, except we change the application heap size in the evaluation. For evaluation of multi-tenant workloads, we use TPC-H [2] running on Spark-SQL [6] as short jobs. The total input size for TPC-H is 10GB.

Table 1: Inputs of the evaluated workloads.

Workloads (units)	Small/Executor	Large/Executor
Spark-WC(GB)	100 / 1	1000 / 6
Spark-KM(Samples)	1×10^8 / 1	5×10^8 / 6
Spark-PR(Pages)	5×10^6 / 1	5×10^7 / 8
Spark-NW(Edges)	1×10^7 / 1	1×10^8 / 8
MapReduce-TS(Records)	3.2×10^8	3.2×10^9 / 25
MapReduce-II(GB)	10×10	10×30 / 6

We map jobs in Google trace to workloads in Table 1. The application types and input scales are chosen based on the task memory usage in the trace files. Specifically, we choose a 5-hour interval from Google trace and group all jobs in that interval. We compute the memory usage of each job. The top 1% is mapped to InvertedIndex. The top 10% is mapped to TeraSort and NWeight, and the top 70% is mapped to KMeans and PageRank. The rest is mapped to WorkCount. Each job is submitted according to the timestamp in the trace.

5.2 Pufferfish in the Single-Node Cluster

We evaluate Pufferfish ('PF' for short in figures) on the single-node cluster to assess the puff and reclaim mechanisms as well as the adaptive parallelism of elastic memory management.

Puff function evaluation. In this experiment, we create a controlled environment by submitting one Pagerank job (container-1) and two Kmeans jobs (container-2 and container-3) one by one. As shown in Figure 6-(a), with default EJF container-1 is given the highest priority, while container-2 and container-3 are suppressed at the beginning. Container-3 is killed by Pufferfish at around the 600^{th} second because of the aggressive demand of container-1. However, the failover request for container-3 is blocked from the

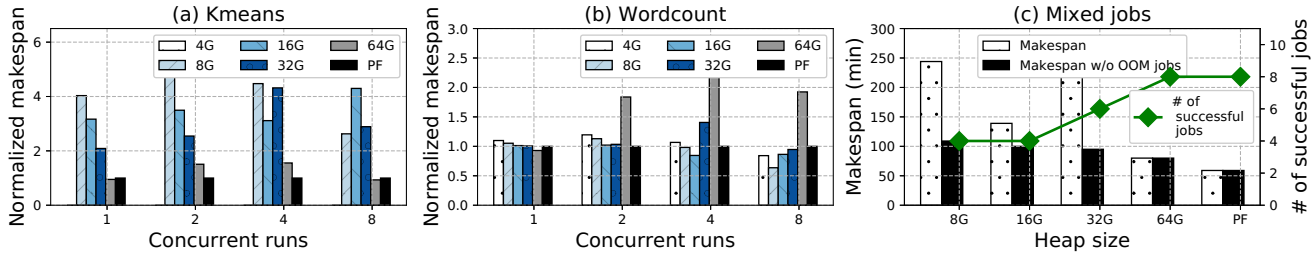

Figure 7: Performance comparison between Pufferfish and Yarn under various heap sizes. We report the median makespan over 5 runs of each experiment. A job is submitted 120s later than the previous one. In (a), Kmeans fails with the 4GB heap. With the 4GB heap, only Wordcount can finish in (c). We use normalized runtime to report the execution time for an individual job since the runtime of different jobs may vary widely. We use the absolute time in Fig 6-(c) since it is a makespan. The makespans in (a) and (b) are normalized to a single run of Pufferfish.

600^{th} second to the 1050^{th} second, as node memory is not available. After the memory demand is satisfied, the size of container-1 drops at the 880^{th} second because Pufferfish detects a memory slack in container-1 and shrinks its size. As a result, the released slack is immediately allocated to container-3.

Reclaim function evaluation We submit a Kmeans job (in a FLEX container) along with a set of TPC-H workloads (in REGULAR containers). Figure 6-(b) shows the memory usage of the FLEX container and all REGULAR containers, respectively. The FLEX-container is forced to reclaim memory by swapping memory pages to the disk when facing TPC-H jobs. Thus, the FLEX-container memory is reclaimed from 58GB to 40GB at around the 400^{th} second and further reduced to 32GB at around the 670^{th} second. As a result, we observe a 1.2x slowdown in the Kmeans job. However, the timely released memory ensures that REGULAR containers do not suffer from memory insufficiency. As shown in Figure 6-(c), under Yarn's static memory allocation, the performance of TPC-H is severely affected by queuing delay due to head-of-line blocking. With memory reclaim, Pufferfish (mix-PF) achieves an 80% improvement over Yarn (mix-YARN) for the 99^{th} percentile job completion time.

Container parallelism and performance. To show the benefits of memory elasticity, we compare Pufferfish with Yarn. In Yarn, applications are set with different heap sizes so that different degrees of container parallelism (the number of concurrently running containers) on a node are achieved. The trade-off is that less memory offers higher parallelism but suffers from suboptimal performance. We submit two sets of jobs. One only consists of Kmeans jobs and the other only consists of Wordcount jobs. Kmeans is a memory-intensive and CPU-intensive workload with a maximum profiled memory demand of 61GB per container with 64GB heap size. Figure 7-(a) shows the performance of Kmeans is significant degraded under suboptimal heap sizes because of GC activities and data spilling. However, the 64GB-heap still achieves shorter makespan than those with higher container parallelism. For 2 and 4 concurrent runs, the makespan of Pufferfish is better than that of the 64GB-heap by 33.6% and 35.6%, because Pufferfish enables a container parallelism of 2 during most of the application runtime while Yarn only admits one job (two 64GB Kmeans would exceed 120GB node capacity). However, Wordcount has a different behavior. Wordcount is an I/O intensive workload and has a maximum profiled memory usage of 24GB per container under 64GB heap size. Unlike iterative Kmeans whose cached data can be immediately used in the next stage, Wordcount only has two stages (one map and one reduce). As a result, a large memory pool, although it decreases the frequency of data spilling, does not gain obvious performance improvement. In this case, the benefit of higher container parallelism outweighs the benefit of a large heap. Pufferfish enables Wordcount to have the maximum container parallelism of 4. Thus, as shown in Figure 7-(b), it outperforms the 64GB heap by 1.75x-2.5x for concurrent runs. Due to high container parallelism, 8GB heap and 16GB heap outperform Pufferfish in the makespan. The reason is that Wordcount is an I/O intensive application without iterative execution. Oversubscribing I/O by increasing the parallelism outperforms Pufferfish. However, there does not exist one heap size for all workload scenarios. Relying on a larger heap or higher container parallelism does not necessarily improve performance. Pufferfish adaptively sets the container size at runtime to improve cluster utilization and job performance.

We conduct another experiment with mixed workloads and show it in Figure 7-(c). The mixed workloads contain two Wordcount, two Kmeans, two Pagerank and two NWeight jobs. We use the Spark default failover retry number (4). Only the 64GB-heap and Pufferfish successfully finish all of the jobs. Through adaptive parallelism, Pufferfish outperforms the 64GB-heap by 26%.

This experiment illustrates that Pufferfish exploits more memory elasticity in an environment with more memory heterogeneity (Figure 7-(b) *vs.* Figure 7-(c)). The reason is that Pufferfish can adjust container memory on the fly. In particular, it shrinks container sizes of less demanding jobs (e.g., Wordcount) for high demanding jobs (e.g., Kmeans) to improve overall concurrency. We also observe that the makespan excluding OOM jobs is 1.2x-2.2x shorter, implying that OOM causes substantial. We further find that it is difficult to detect OOM. First, failover execution launches several retries for a failed task. Second, the default garbage collector is conservative in throwing OOM until it takes 98% of CPU time but releases less than 2% of heap memory [23]. For a heap close to the size that is just enough to avoid OOM, JVM spends a lot of time on useless GC. It also explains why the 32GB heap has a longer makespan than the 16GB heap. In contrast, by allocating a large heap, Pufferfish

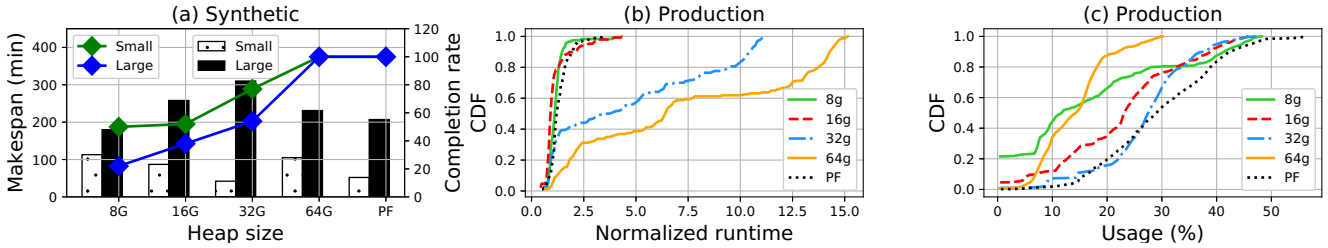

Figure 8: Experiment results of the multi-node cluster.

Table 2: Number of each application mapped in trace.

App	WC	KM	PR	NW	TS	II
Small/Large	23/26	31/20	41/24	3/5	3	2

Table 3: OOM rate under various heap size.

Heap	64GB/pf	32GB	16GB	8GB	4GB
Failure	0	5.1%	17.3%	34.1%	> 60%

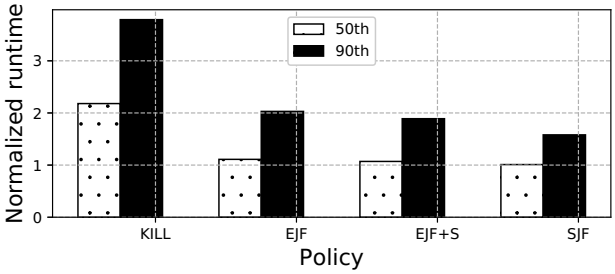

Figure 9: Normalized runtime under various policies.

ensures successful execution for all applications and shifts unused heap quota to others to improve overall performance.

5.3 Pufferfish in the Multi-Node Cluster

Synthetic workloads. We then evaluate Pufferfish using the synthetic workloads on the 26-node cluster. We build the synthetic workloads by replicating the mixed workloads in Figure 7-(c) by 20x. We run both small workloads and large workloads (corresponding to Table 1) to study the impact of job size on elastic memory management. In this experiment, each task is only allowed to fail once to exclude the overhead caused by failover. As shown in Figure 8-(a), Pufferfish achieves the best performance with large workloads as Pufferfish is configured with optimal heap and higher parallelism. For small workloads, Pufferfish is a little bit slower than 32-GB heap (42 mins *vs.* 52 mins) as the overhead of `puff()` is more pronounced when the job is short. However, most importantly, Pufferfish yields no application failure.

Production data-intensive workloads. We replay subsets of the Google trace on our 26-node cluster. The trace contains 178 data-intensive workloads, all of which are configured with FLEX containers. The number of each application with small and large input datasets is listed in Table 2 (e.g., WC stands for Wordcount). The job runtime is normalized to that when running the job in an isolated environment with a 64GB heap.

As shown in Table 3, Pufferfish does not cause OOM. Figure 8-(b) plots the runtime distribution of successfully finished applications. It shows Pufferfish outperforms Yarn with the 32GB heap and the 64GB heap by 3.2x and 5.5x respectively for the median job runtime because of memory elasticity. We further notice that the degraded performance with 32GB and 64GB heaps by Yarn is caused by significant queuing delay at ResourceManager. For Yarn with 8GB and 16GB, the job failures caused by OOM and intensive GC caused by suboptimal heap size ruin performance. Pufferfish automatically adjusts the memory allocation and container parallelism on each node, avoiding queuing delay and OOM.

Figure 8-(c) shows the cluster memory utilization reported by Docker. Pufferfish achieves the highest memory utilization. In particular, Pufferfish improves the median memory utilization over Yarn with 64G heap by 2.7x. Note that Yarn with 16GB heap and 8GB heap represent a relatively high memory utilization. However, as suboptimal heap size causes OOM, the high memory utilization is the result of useless failover and does not contribute to better job performance. The improved utilization in Pufferfish with elastic memory, on the other hand, is used to host more jobs, increasing the overall parallelism, reducing queuing delay and job runtime.

Policy evaluation. We evaluate the impact of policy discussed in section 4.3.3 using the same production trace. Figure 9 shows the median and 90^{th} percentile job runtimes. For comparison, we also implement a variant of Pufferfish in which elastic memory management is not implemented. In this variant, which we denote as KILL in the figure, each container will puff on its own without any coordination with the containers on the same node, and will be killed when its increased memory cannot be met in one puff. We also implement EJF with suspension tolerance and denote it as EJF+S. Note that SJF is enabled with suspension tolerance by default to avoid starvation.

For the 90^{th} percentile job runtime, SJF achieve the best performance (23% faster than EJF) because (1) it avoids the situation where short jobs are blocked by long jobs, and (2) the completed short jobs soon release enough memory for long jobs. EJF+S improves the performance of EJF by killing a task which is suspended too long so that the relaunched task might be placed on a less intensive node, partially resolving the issue that short jobs are blocked by long jobs. Without coordination of memory management, KILL achieves the

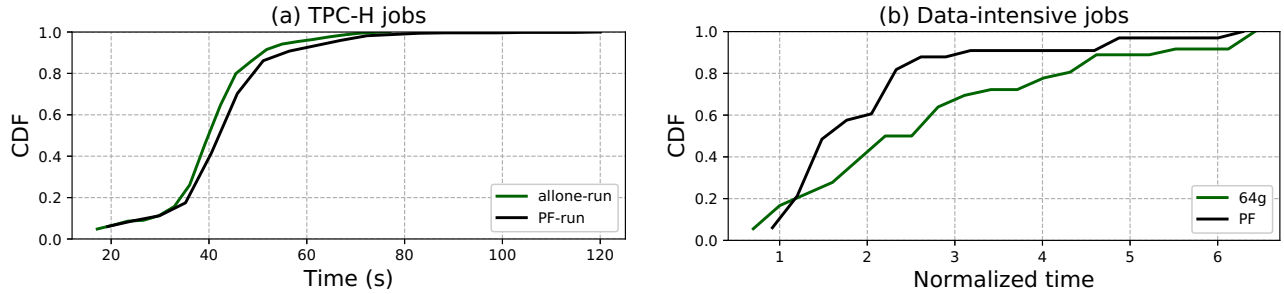

Figure 10: Results of mixed workloads. (a) TPC-H workloads. (b) Data-intensive workloads.

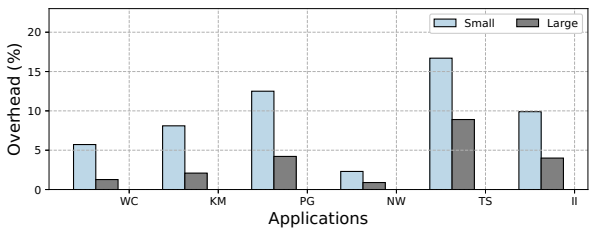

Figure 11: The overhead of different workloads.

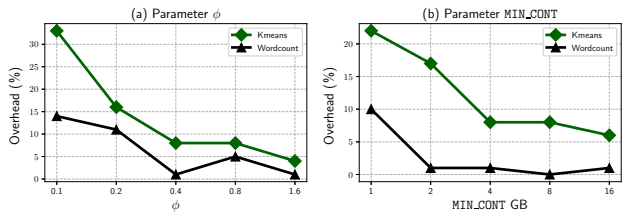

Figure 12: Parameter sensitivity.

worst performance. The drawback of SJF is that it requires prior knowledge of job duration.

Production mixed workloads. This experiment shows that Pufferfish can preserve fairness in the presence of low-latency workloads.

The trace contains 38 data-intensive workloads and 576 TPC-H workloads. We first compare Pufferfish where mixed workloads are used against Yarn where TPC-H runs alone, to examine the memory reclaim overhead. As shown in Figure 10, since Pufferfish reclaims memory from FLEX containers in a timely manner, Pufferfish achieves similar performance to that of Yarn for TPC-H jobs. However, we also observe a 1.74x slowdown in the 99^{th} percentile latency for Pufferfish due to memory reclaim overhead.

We then run the mixed workloads with both Pufferfish and Yarn. For data-intensive workloads, Pufferfish improves the 80^{th} percentile job runtime over Yarn by 2x due to elastic memory allocation. However, Pufferfish incurs as much as 8x slowdown for the 95^{th} percentile job runtime compared to the fastest (1^{st} percentile) job runtime by Pufferfish. This is due to the fact that Pufferfish reclaims memory from FLEX containers to avoid memory allocation delays for REGULAR containers. This experiment shows Pufferfish achieves global fair-share for REGULAR containers by reclaiming over-provisioned memory from FLEX containers. Due to the longer tail latency, we recommend that users only use FLEX containers for jobs without strict SLOs.

5.4 Overhead and Parameter Sensitivity

This section evaluates the overhead of Pufferfish and its parameter sensitivity. The job runtime and overhead are both normalized to those with a 64 GB heap.

Overhead. The overhead caused by Pufferfish comes from the application start phase when application containers are puffed from `MIN_CONT`. During this period, the container is consecutively suspended and puffed. Figure 11 shows that the overhead during the start phase is less than 10% across various applications.

It is also important to evaluate the latency of memory reclaim, since some latency critical applications with REGULAR containers cannot wait too long during memory reclaim. Our experiment shows it takes about 0.5 seconds to reclaim 1 GB memory on our SSD-backed swap partition. We also expect that future fast storage will further reduce the latency.

Parameter sensitivity. We evaluate Pufferfish's overhead under various parameters ϕ and `MIN_CONT`. We run CPU-intensive Kmeans and I/O-intensive Wordcount applications. As shown in Figures 12-(a) and (b), respectively, increasing ϕ and `MIN_CONT` both result in lower overhead because fewer numbers of puff operations are required. Another observation is that this overhead can be amortized with increasing of job runtime, since the overhead mainly occurs during the task start phase. To reduce puff overhead, we recommend users use a large ϕ. A large ϕ does not cause memory waste as Pufferfish will finally shrink the unused memory.

Spilling *vs.* Paging. A recently proposed spilling-based approach [24] also aims to achieve memory elasticity for data paralleled workloads. In the spilling approach, applications proactively store data from memory into disks and release memory. We run Spark Pagerank with the spilling approach and Pufferfish's paging approach. For paging, we limit the container size to force swapping but set the 64GB optimal heap size. For spilling, we limit the JVM heap size but enable spilling by using `persist()` function on RDDs [48]. Thus, we provide a fair performance comparison because the physical memory usage is the same. We scale up the memory allocation

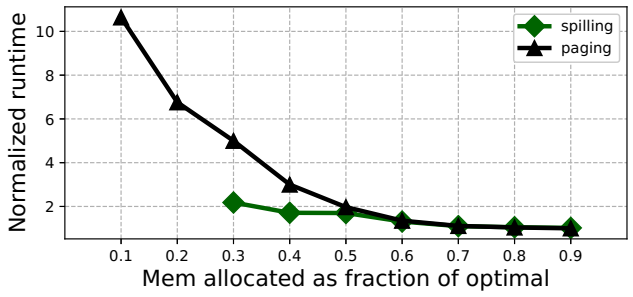

Figure 13: Paging vs. spilling.

(container size for paging and heap size for spilling) as fractions of the optimal 64GB heap. Figure 13 shows that both approaches achieve ideal performance when the allocated memory gets close to the optimal heap (0.5 or more). The spilling approach wins when less memory is allocated (0.3-0.5) because spilling uses LRU to evict data and follows the data usage pattern. However, when the allocated memory is less than 30% of the optimal, the spilling approach throws OOM while the paging approach still completes the execution, though at the cost of a 10x slowdown.

The spilling approach can achieve better performance for non-iterative applications with very few execution stages, such as Word-count or Sort. But for in-memory iterative applications, e.g. Spark Kmeans and Pagerank, the spilling approach still suffers from intensive I/O traffic or even OOM, because the data that is just spilled may be needed immediately in the next stage. Thus, the spilling approach does not provide an advantage over paging for in-memory iterative applications. In addition, we note there are two challenges to apply spilling practically, as spilling is an application-aware approach: (1) Users have to identify which data blocks (RDDs in Spark) to spill; (2) Users have to figure out an proper heap size that is small enough to achieve high parallelism but large enough to avoid OOM. Pufferfish is transparent to applications and thus does not suffer from these problems.

5.5 Discussion

REGULAR container vs. FLEX container. Pufferfish allows users to enforce fairness between FLEX containers and REGULAR containers in a multi-tenant cluster. In particular, users can launch REGULAR containers for applications that require a fixed memory budget, while launching FLEX containers for jobs that can run under a flexible memory budget.

Application-assisted container resizing. A recent improvement [3] to Apache Yarn enables resizing the container size. However, container resizing requires the applications (e.g, Spark Driver) to specify the amount of memory to be added or reclaimed, which requires intrusive modification to applications. Pufferfish, on the other hand, relies on puff and reclaim which are transparent to applications.

Container approach *vs.* OS approach. Another alternative solution is to make modifications to the OS and implement elastic memory management in the OS level (e.g., in OS memory management). However, this approach is not practical as current data-intensive application software stacks contain multiple hierarchies for memory management. As a result, the OS approach lacks contextual information about the JVM layer or the application layer. Our proposed container approach utilizes both the OS interfaces through cgroup and the application context information in user space through Yarn, while requiring no modification to the OS source code.

Container approach *vs.* application approach. It is intuitive to optimize memory management in the application layer because an application understands its memory demands [29, 37]. Although users can specify the memory size of containers upon job submission, dynamically adjusting container size without application's instruction is not supported during runtime. It is also challenging to achieve memory elasticity across different application frameworks at the application level. For example, it is hard to coordinate memory allocation between Spark tasks and Hadoop tasks when using an application approach. Checkpoint-based techniques (e.g., CRIU) requires an intrusive modification to applications. Pufferfish aims to dynamically resize the containers during job runtime.

Container approach *vs.* JVM approach. Another intuition is to build memory elasticity into the JVM. [45] proposes dynamically adjusting the JVM heap size to achieve memory elasticity for data queries. However, this approach relies on the estimation of future memory usage to determine the target heap size after resizing. It has two major drawbacks. First, the estimation requires application runtime information (e.g, column size of a database table), which is hard to (1) generalize to all applications and (2) couple with cluster schedulers. Second, if the estimation is not right, the JVMs runing on the same node collide with each other due to memory contention. Our approach, on the other hand, adjusts the container size based on the JVM memory demand, which is transparent to applications. Further, our approach avoids memory contention through (1) guaranteed memory isolation between containers; (2) dedicated memory control through puff and reclaim function. Further, tasks running in FLEX containers inherently have a large heap size. Overall, this benefits the applications as less GC and less spilling are triggered.

Container approach *vs.* virtual machine monitor. Memory management in virtual machines often employs memory ballooning techniques. Since it is difficult for a virtual machine monitor (VMM) to obtain the actual memory demand directly from the guest OS, ballooning is a mechanism that allows the VMM to reclaim memory from a guest OS based on information provided by a ballooning driver installed on the guest OS [40, 44]. Pufferfish works in an different way. It obtains the memory demand through OCM heuristics, which is not available in memory ballooning. It improves job performance by flexibly puffing OS containers so as to accelerate individual tasks when node memory is available.

6 RELATED WORK

Cluster scheduling. Yarn [42] and Mesos [21] are two widely used open-source centralized schedulers. Sparrow [35] is a fully distributed scheduler based on random sampling. Hawk [12] and Mercury [26] both implement a hybrid scheduler to avoid inferior scheduling decisions and balance the trade-off of scheduling quality and scalability. CARBYNE [19] allows applications to altruistically yield their allocated resources to achieve secondary goals. GRAPHENE [20] discusses task dependency in cluster scheduling.

3sigma [36] proposes scheduling jobs based on their runtime distribution. Medea [16] develops a framework to manage long running applications. Gandiva [46] proposes a scheduler tailored for deep learning applications. BIG-C [10] is a preemptive-based framework that improves the performance for latency-critical applications in a shared cluster. Elassecutor [29] schedules Spark executors in an elastic manner according to predicted time-varying resource demands. Pufferfish can be implemented on these representative schedulers.

Memory management. Pufferfish is closely related to studies that address memory pressure. ROLP [8], FACADE [33], Yak [32] and Broom [17] optimize GC overhead by allocating data items in regions. iTask [14] forcibly suspends tasks and performs an external GC to reduce memory demand. These approaches either require users to implement memory management-related code, which requires expertise or mainly focus on GC overhead, which still leaves memory pressure as a threat. Recent studies [24, 45] aim for memory elasticity to improve cluster memory utilization. Both approaches need application assistance to achieve memory elasticity.

Cluster utilization. To improve resource utilization, several studies [30, 31, 47] propose consolidating applications on shared resources and managing interference at the node level so that application quality-of-service can be met. Other studies [11, 50] improve resource management at the datacenter level and achieve higher utilization. SDChecker [9] profiles and analyzes the causes of latency in a multi-stack scheduling environment (e.g. Spark on Yarn). Pufferfish exploits memory elasticity for higher memory utilization.

There are efforts that estimate resource usage for long-running workloads, which have limitations because they either require domain-specific knowledge [43] or a static profiling approach [5]. Relying on estimation by profiling may still results in with OOM errors [1]. Pufferfish is a general-purpose framework that tackles these issues without any assumption about the underlying frameworks or how memory is managed in these frameworks.

Data-parallel Frameworks. Many data-parallel processing frameworks have been developed to help people facilitate data analysis, model data, and understand data. Hadoop [4] is for massive data processing on commodity PCs. Spark [49] leverages in-memory computing to speed up data processing. Shark [13] and Hive [41] provide an SQL interface to access data based on Hadoop and Spark. Tez [39] and Dryad [25] enhance parallel data processing with strong expression through a DAG of tasks. GraphX [18] enables users to build and deploy graphs and graph-parallel computation. Pufferfish is application-agnostic and can help these application frameworks survive from memory pressure and achieve better performance.

7 CONCLUSION

In this paper, we advocate Pufferfish, an elastic memory manager for data-intensive applications. It tackles both memory pressure and low cluster utilization. The core idea is to run a big JVM in a small OS container to avoid OOM and realize cluster memory elasticity with the puff and reclaim mechanisms. We have implemented Pufferfish as an independent module in Yarn. Experimental results show Pufferfish is able to help applications with large memory demands survive from memory pressure, increase task parallelism, and significantly improve job performance and cluster utilization.

8 ACKNOWLEDGMENT

We thank our shepherd Siddhartha Sen and the anonymous reviewers for the valuable feedback. This research was supported in part by U.S. NSF grant SHF-1816850.

REFERENCES

[1] Spark-19371. https://issues.apache.org/jira/browse/SPARK-19371.
[2] Tpch standard specification. http://www.tpch.org/tpcc/spec/tpcc.
[3] Yarn-1645. https://issues.apache.org/jira/browse/SPARK-19371/.
[4] Hadoop. http://hadoop.apache.org, 2009.
[5] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *Proc. of USENIX NSDI*, 2017.
[6] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In *Pro. of ACM SIGMOD*, 2015.
[7] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proc. of IEEE ICDE*, 2011.
[8] R. Bruno, D. Patrício, J. Simão, L. Veiga, and P. Ferreira. Runtime object lifetime profiler for latency sensitive big data applications. In *Proc. of ACM Eurosys*, 2019.
[9] W. Chen, A. Pi, S. Wang, and X. Zhou. Characterizing scheduling delay for low-latency data analytics workloads. In *Proc. of IEEE IPDPS*, 2018.
[10] W. Chen, J. Rao, and X. Zhou. Preemptive, low latency datacenter scheduling via lightweight virtualization. In *Proc. of USENIX ATC*, 2017.
[11] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proc. of the ACM SOSP*, 2017.
[12] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *Proc. of USENIX ATC*, 2015.
[13] C. Engle, A. Lupher, R. Xin, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: fast data analysis using coarse-grained distributed memory. In *Proc. of ACM SIGMOD*, 2012.
[14] L. Fang, K. Nguyen, G. Xu, B. Demsky, and S. Lu. Interruptible tasks: Treating memory pressure as interrupts for highly scalable data-parallel programs. In *Proc. of ACM SOSP*, 2015.
[15] R. Gandhi, D. Xie, and Y. C. Hu. Pikachu: How to rebalance load in optimizing MapReduce on heterogeneous clusters. In *Proc. of USENIX ATC*, 2013.
[16] P. Garefalakis, K. Karanasos, P. R. Pietzuch, A. Suresh, and S. Rao. Medea: scheduling of long running applications in shared production clusters. In *Proc. of the ACM EuroSys*, 2018.
[17] I. Gog, J. Giceva, M. Schwarzkopf, K. Vaswani, D. Vytiniotis, G. Ramalingam, M. Costa, D. G. Murray, S. Hand, and M. Isard. Broom: Sweeping out garbage collection from big data systems. In *Proc. of USENIX HotOS*, 2015.
[18] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *Proc. of USENIX OSDI*, 2014.
[19] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *Proc. of USENIX OSDI*, 2016.
[20] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *Proc. of the USENIX OSDI*, 2016.
[21] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proc. of USENIX NSDI*, 2011.
[22] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *Proc. of IEEE Data Engineering Workshops (ICDEW)*, 2010.
[23] C. Hunt and B. John. *Java performance*. Prentice Hall Press, 2011.
[24] C. Iorgulescu, F. Dinu, A. Raza, W. U. Hassan, and W. Zwaenepoel. Don't cry over spilled records: Memory elasticity of data-parallel applications and its application to cluster scheduling. In *Proc. of USENIX ATC*, 2017.
[25] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proc. of ACM SOSP*, 2007.
[26] K. Karanasos, S. Rao, C. Curino, C. Douglas, K. Chaliparambil, G. M. Fumarola, S. Heddaya, R. Ramakrishnan, and S. Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *Proc. of USENIX ATC*, 2015.
[27] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skewtune: mitigating skew in MapReduce applications. In *Proc. of ACM SIGMOD*, 2012.
[28] Y. Kwon, K. Ren, M. Balazinska, B. Howe, and J. Rolia. Managing skew in hadoop. *Proc. of IEEE Data Eng. Bull.*, 2013.

[29] L. Liu and H. Xu. Elasecutor: Elastic executor scheduling in data analytics systems. In *Proc. of the ACM SoCC*, 2018.
[30] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: improving resource efficiency at scale. In *Proc. of ACM ISCA*, 2015.
[31] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proc. of IEEE/ACM MICRO*, 2011.
[32] K. Nguyen, L. Fang, G. Xu, B. Demsky, S. Lu, S. Alamian, and O. Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *Proc. of USENIX OSDI*, 2016.
[33] K. Nguyen, K. Wang, Y. Bu, L. Fang, J. Hu, and G. Xu. Facade: A compiler and runtime for (almost) object-bounded big data applications. In *Proc. of ACM SOSP*, 2015.
[34] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, B.-G. Chun, and V. ICSI. Making sense of performance in data analytics frameworks. In *Proc. of USENIX NSDI*, 2015.
[35] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *Proc. of ACM SOSP*, 2013.
[36] J. W. Park, A. Tumanov, A. Jiang, M. A. Kozuch, and G. R. Ganger. 3sigma: distribution-based cluster scheduling for runtime uncertainty. In *Proc. of the ACM EuroSys*, 2018.
[37] A. Qiao, A. Aghayev, W. Yu, H. Chen, Q. Ho, G. A. Gibson, and E. P. Xing. Litz: Elastic framework for high-performance distributed machine learning. In *Proc. of the USENIX ATC)*, 2018.
[38] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proc. of ACM SoCC*, 2012.
[39] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino. Apache tez: A unifying framework for modeling and building data processing applications. In *Proc. of ACM SIGMOD*, 2015.
[40] T.-I. Salomie, G. Alonso, T. Roscoe, and K. Elphinstone. Application level ballooning for efficient server consolidation. In *Proc. of ACM Eurosys*, 2013.
[41] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. of VLDB Endowment*, 2009.
[42] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache Hadoop YARN: Yet another resource negotiator. In *Proc. of ACM SoCC*, 2013.
[43] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: efficient performance prediction for large-scale advanced analytics. In *Proc. of USENIX NSDI*, 2016.
[44] C. A. Waldspurger. Memory resource management in vmware esx server. *ACM SIGOPS Operating Systems Review*, 2002.
[45] J. Wang and M. Balazinska. Elastic memory management for cloud data analytics. In *Proc. of USENIX ATC*, 2017.
[46] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, et al. Gandiva: introspective cluster scheduling for deep learning. In *Proc. of the USENIX OSDI*, 2018.
[47] H. Yang, A. Breslow, J. Mars, and L. Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *Proc. of ACM ISCA*, 2013.
[48] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. of USENIX NSDI*, 2012.
[49] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.
[50] Z. Zhang, L. Cherkasova, and B. T. Loo. Exploiting cloud heterogeneity to optimize performance and cost of MapReduce processing. In *Proc. of ACM SIGMETRICS*, 2015.