OS-Augmented Oversubscription of Opportunistic Memory with a User-Assisted OOM Killer

Wei Chen, Aidi Pi, Shaoqi Wang, and Xiaobo Zhou
University of Colorado, Colorado Springs, CO, USA

Abstract

Exploiting opportunistic memory by oversubscription is an appealing approach to improving cluster utilization and throughput. In this paper, we find the efficacy of memory oversubscription depends on whether or not the oversubscribed tasks can be killed by an `OutOfMemory` (OOM) killer in a timely manner to avoid significant memory thrashing upon memory pressure. However, current approaches in modern cluster schedulers are actually unable to unleash the power of opportunistic memory because their user space OOM killers are unable to timely deliver a task killing signal to terminate the oversubscribed tasks. Our experiments observe that a user space OOM killer fails to do that because of lacking the memory pressure knowledge from OS while the kernel space Linux OOM killer is too conservative to relieve memory pressure.

In this paper, we design a user-assisted OOM killer (namely UA killer) in kernel space, an OS augmentation for accurate thrashing detection and agile task killing. To identify a thrashing task, UA killer features a novel mechanism, constraint thrashing. Upon UA killer, we develop Charon, a cluster scheduler for oversubscription of opportunistic memory in an on-demand manner. We implement Charon upon Mercury, a state-of-the-art opportunistic cluster scheduler. Extensive experiments with a Google trace in a 26-node cluster show that Charon can: (1) achieve agile task killing, (2) improve the best-effort job throughput by 3.5X over Mercury while prioritizing the production jobs, and (3) improve the 90th job completion time of production jobs over Kubernetes opportunistic scheduler by 62%.

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

Keywords memory management, resource sharing, Linux system, cloud computing, distributed system

ACM Reference Format:
Wei Chen, Aidi Pi, Shaoqi Wang, and Xiaobo Zhou . 2019. OS-Augmented Oversubscription of Opportunistic Memory with a User-Assisted OOM Killer. In *Middleware '19: Middleware '19: 20th International Middleware Conference, December 8–13, 2019, Davis, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3361525.3361534

1 Introduction

Modern cluster schedulers [27, 36, 37] provision a mix of diverse workloads, such as business critical jobs, customer facing services, exploratory analytics and testing jobs. To efficiently allocate memory resource and improve cluster utilization, memory oversubscription is an appealing approach that commonly reserves exclusive memory for production jobs (e.g., long running jobs or services with strict QoS restriction) and provisions the leftover transient memory to best-effort jobs (e.g., ad-hoc exploratory queries, jobs for debugging purpose). Accordingly, cluster scheduler Yarn [36] recently introduces opportunistic container to oversubscribe opportunistic memory. Tasks of best-effort jobs are assigned with opportunistic containers and are scheduled to run as soon as opportunistic memory is available. Yarn relies on static memory partition when allocating memory for a container. As a result, the memory size of an allocated container cannot be changed during task execution regardless of its dynamic memory demands. Kubernetes [3] introduces burstable class which is similar to opportunistic containers of Yarn, and allows the oversubscription of opportunistic memory in a dynamic manner. Both Yarn and Kubernetes share a common design where the core is a per-node opportunistic scheduler.

By their design, the production jobs are scheduled by a centralized scheduler with guaranteed resources to enforce the QoS while the best-effort jobs are scheduled by opportunistic schedulers to oversubscribe transient opportunistic memory. Therefore, tasks from best-effort jobs are aggressively scheduled to oversubscribe opportunistic memory in an on-demand manner when the opportunistic scheduler detects free memory. When the production jobs require more resources due to load fluctuations [14, 25, 30, 32] or newly submitted jobs, containers of opportunistic resources are preempted by killing their corresponding tasks. To maximize the cluster memory utilization, we consider a realistic situation [6, 26, 37] where there is a continuous and dynamic stream of opportunistic jobs to keep the cluster memory usage at a high load.

The core of the opportunistic scheduler is an `OutOfMemory` (OOM) monitor and a user space OOM killer. The OOM monitor inspects the memory availability via OS interface `/proc/meminfo` and informs the user space OOM killer to terminate overcommitted tasks accordingly upon memory pressure. The effectiveness of memory oversubscription highly depends on whether the tasks holding the oversubscribed memory can be timely killed so as to release memory before memory thrashing and attain high memory utilization.

However, we find that existing opportunistic schedulers with a user space OOM killer are unable to unleash the power of opportunistic memory. They cause significant memory thrashing, leading to system unresponsiveness and suboptimal application performance. There are three major reasons. First, the Linux kernel is unable to report a metric that precisely indicates the amount of available memory. To oversubscribe the opportunistic memory for best-effort jobs without harming performance of production jobs, an opportunistic scheduler has to perform container allocation and killing based on instantaneous memory availability and memory demands. Because of the inaccurate available memory reported by OS, the task killing by the OOM killer often cannot alleviate memory thrashing. Second, due to the inaccurate report of system

Middleware '19, December 8–13, 2019, Davis, CA, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-7009-7/19/12...$15.00
https://doi.org/10.1145/3361525.3361534

available memory, a user space OOM killer relies on the cluster configuration of per-node memory limit to throttle the upper limit of per-node memory usage. The OOM killer is activated once the host memory usage is beyond the limit. However, configuring a proper per-node memory limit is tedious, and it likely causes problems due to its dependencies on runtime environments, especially in a heterogeneous cluster. Misconfiguration either leads to memory thrashing or low memory utilization. Third, it is not ensured that `SIGKILL` from user space can be timely delivered to kernel space under memory pressure. The delayed `SIGKILL` results in significant memory thrashing and undesirable unresponsiveness of the system.

The intuition of addressing the issues of the user space OOM killer is to use kernel space OOM killer since (1) the killing signal can be quickly delivered without any interference, and (2) the killing decision can be made precisely with fine-grained metrics regarding to memory thrashing from kernel space. However, the existing Linux OOM killer still runs into severe memory thrashing because it is too conservative in order to minimize the lost of processes.

In this paper, we propose a user-assisted OOM killer in kernel space, namely UA killer. UA killer leverages the existing OOM killing primitives but with a careful consideration of the timing of task killing in order to avoid significant memory thrashing. UA killer features two improvements: (1) it is integrated within the kernel memory reclaimer, so that it ensures timely delivery of the killing signal, and (2) it detects both anonymous page thrashing and file page thrashing, so that it accurately identifies a thrashing condition. The core idea of UA killer is *constraint thrashing*, where *tasks are allowed to thrash for a tolerant amount of time before a victim task is chosen to be killed.* This novel mechanism allows UA killer to accurately detect both anonymous page thrashing and file page thrashing within the constraint thrashing period so that the killing decision can be made informatively. By this design, opportunistic memory can be efficiently used while memory trashing on production jobs is mitigated.

Built upon UA killer, we develop Charon, a cluster scheduler that aims to efficiently oversubscribe opportunistic memory in an on-demand manner. Charon leverages UA killer for agile task killing and accurate thrashing detection.

We implement Charon upon the state-of-the-art opportunistic scheduler Mercury [27]. Experiments with Google trace [34] on a 26-node cluster shows that, (1) by UA killer, OOM killing latency is short and guaranteed (<20ms), (2) when using Spark batch jobs as production jobs, Charon improves the throughput of best-effort jobs over Mercury by 3.5x, and it improves the 90^{th} job completion time (JCT) of production jobs over Kubernetes opportunistic scheduler by 62%, (3) when using Cassandra as the production service, Charon achieves the best throughput among all approaches because UA killer can detect file page thrashing accurately and kill the containers timely, and (4) Charon protects the performance of production jobs from memory thrashing. Finally, we conduct extensive simulations of a production cluster using Google trace. It shows that unexploited opportunistic memory is prevalent in a production cluster. Charon improves the throughput of best-effort jobs over Mercury and Kubernetes by 59% and 26%, respectively.

Overall, we make the following contributions in this paper.

- We find a user space implementation of OOM killer by Kubernetes causes significant thrashing in oversubscribing opportunistic memory.
- We design a user-assisted OOM killer in kernel space, called UA killer, that enables accurate thrashing detection and timely task killing.
- We develop Charon, an opportunistic cluster scheduler built upon UA killer. We implement Charon into popular cluster scheduler Mercury and experiments demonstrate its superior performance over Mercury and Kubernetes.

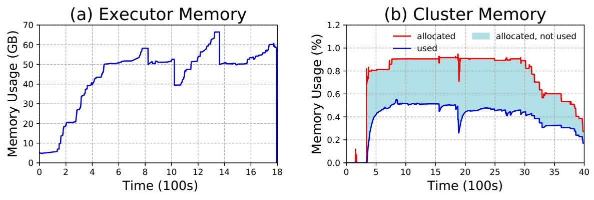

Figure 1. Memory usage of production jobs of Spark workloads. (a) physical memory usage of a single container. (b) allocated and physical memory usages of all containers.

2 Motivation

2.1 Opportunistic memory

Recent progress [10, 27, 33, 37] in cluster scheduling allows best-effort jobs to utilize the opportunistic memory. However, we identify two issues that prevent opportunistic memory from being efficiently exploited. First, static resource partition, which was built on the assumption that the actual used memory of a container should be close to its allocated memory, does not consider the nature of fluctuating task memory usage. We show the memory utilization of production jobs by replaying a Google trace [34] in a 26-node cluster (§-5.1 has the cluster setup). Fig 1-(a)shows that the physical memory usage for a single container from Spark Pagerank fluctuates during the task execution. It only reaches the maximum memory limit 64GB (limited by the maximum JVM heap size) at a few moments [14, 25, 30, 32, 38]. Fig 1-(b) shows the aggregated memory usage of all containers at runtime. It demonstrates that although the allocated memory usage (reported by Yarn) approaches the cluster limit, the amount of physical memory usage remains low. Indeed, nearly 40% of the cluster memory is idle and ineffectively utilized by application jobs. To utilize the opportunistic memory, Kubernetes introduces `burstable class` by which newly requested memory is dynamically allocated from the idle memory of production jobs.

Second, the cluster per-node memory limit, which is used to prevent the application memory usage from overwhelming a node, is difficult to be properly configured. For example, Yarn uses threshold $yarn.nodemanager.resource.memory$ to statically cap the maximum physical memory usage on each node. In Kubernetes, a node inspects its available memory ($node.capacity - node.workingset$) and kills tasks from the burstable class if the available memory is less than a threshold $mem.threshold$. The threshold is utilized to prevent excessive memory usage from causing thrashing. However, in practice, these parameters highly depend on hardware architecture, workload types and OS kernel. It causes memory waste if configured conservatively, or memory thrashing if otherwise. Given the fact that modern clusters consist of heterogeneous nodes with different memory capacities and host diverse applications, identifying a one-size-fits-all memory limit is almost impossible.

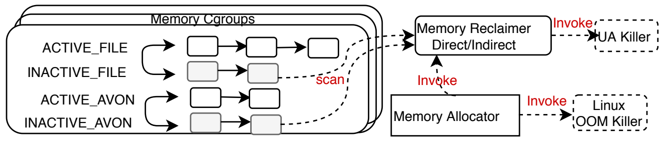

Figure 2. The Architecture of the Linux memory reclaimer and OOM killer. We also show the architectural difference between the Linux OOM killer and UA killer. The Linux OOM killer is implemented in memory allocator while UA killer is implemented in memory reclaimer.

2.2 Linux Memory Reclaimer

In Linux, by default, each process belongs to a memory cgroup and each memory cgroup manages memory pages for all of its processes. In this paper, an individual task or a service runs in an OS container that is managed by a memory cgroup. As shown in Figure 2, the memory pages of a memory cgroup belong to one of four LRU lists [7]: active_file, inactive_file, active_anon and inactive_anon. File pages (active_file and inactive_file) track the I/O related pages in the page cache. Anonymous pages (active_anon and inactive_anon) track the pages allocated by `malloc()` function. For both file and anonymous pages, the most recent accessed pages are first put in the active lists and might be then moved to the inactive lists. Conversely, a page in the inactive lists will be moved back to the active lists if the page is accessed again.

Upon memory pressure, the memory reclaimer chooses pages from the inactive list and reclaims them. For file pages, the unmodified pages are directly reclaimed, while the modified pages are firstly written back to the disk and then get reclaimed. Anonymous pages are required to be written back to the swap partition before released. There are two types of page reclaim. The direct reclaim is executed when a memory allocation request fails because of memory insufficiency. The indirect reclaim is executed by the kernel thread `kswapd` on each NUMA node. The memory reclaimer is invoked when the available memory on NUMA nodes is below the low watermark defined by the system. During both direct and indirect page reclaims, the reclaimer first calculates the number of pages to be scanned for each process (or each cgroup). It then performs scanning to examine if each page is qualified for page-out (e.g., an inactively accessed and clean page). In this step, the reclaimer equally treats all running processes. Thus, pages from each process have the same opportunities to be reclaimed. Finally, the selected pages placed at the tail of the inactive list, following an I/O write request to evict the pages to disks. To prevent the pages in active_file list from increasing too fast under intensive I/O workloads, the reclaimer moves the pages from active_file list to inactive_file list so as to balance the length of both lists. In consequence, the actively accessed file pages may be falsely reclaimed. After reclaim, memory thrashing happens when a process is accessing its working set, part of which has been swapped out to disks due to memory reclaim, resulting in constant paging and page faults.

In the worst case when the memory pressure cannot be alleviated by reclaiming memory pages, for instance when the swap partition is used up, to satisfy a memory allocation request the Linux OOM killer is invoked to terminate a process and release its memory.

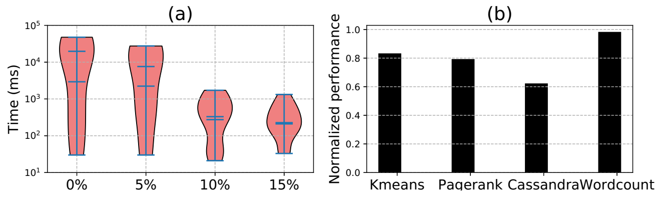

Figure 3. (a) Killing delay under different *mem.threshold* to show the delayed killing signal by user space OOM killer. The four horizontal lines from top to bottom shows the max, mean, median and min values of killing delay, respectively. The width of shade represents the percentage of specific delays. It causes significant memory thrashing and system unresponsiveness when *mem.threshold* is less than 10%. (b) Normalized performance of production jobs to performance running in isolation (the larger the better). We set *mem.threshold* as 10% to avoid swapping state (anonymous page thrashing), but still observe file page thrashing. We measure job completion time (JCT) for Spark applications, and throughput for Cassandra.

2.3 Drawbacks of User Space OOM Killer

We implement a prototype of Kubernetes burstable class on Yarn in order to assess its user space OOM killer. We augment an OOM monitor on Yarn's per-node NodeManager. It frequently (per 1ms) reads parameter `MemAvailable` from `/proc/meminfo` as the node's available memory, which is compared with the requested memory from a new task. The difference is compared with the threshold *mem.threshold*. If the difference is positive, the NodeManager launches a task. Otherwise, to free up memory, the OOM killer terminates running tasks by sending `SIGKILL` signals to OS.

We choose Spark Kmeans, Spark Wordcount and Cassandra table scanning [1] as production jobs. We submit Spark-SQL as the best-effort jobs one per second to quickly saturate the cluster nodes. Production jobs and best-effort jobs run in different CPUSETs to isolate CPU interference. To avoid slow response of NodeManager, we run NodeManager in an isolated container with parameter `swappiness` setting to zero to avoid being swapped and with a dedicated CPU core to void CPU interference from other processes. Based on the experimental results, we identify two fundamental vulnerabilities of the user space OOM killer as follows.

(1) Delayed SIGKILL Signal We assess if the user space OOM killer is able to terminate best-effort tasks fast enough to avoid significant memory thrashing. To examine how excessive the opportunistic memory can be oversubscribed without causing thrashing, we increase threshold *mem.threshold* starting from 0. The user space OOM killer causes significant memory thrashing and system unresponsiveness if *mem.threshold* is too small. It is non-trivial to find a *mem.threshold* value by which the system does not run into unresponsive state until we increase it to 10% of node memory.

To diagnose the root cause, we measure the killing delay as the period from the moment when a `SIGKILL` signal is issued to the moment when the task is killed and its resources released. As shown in Figure 3-(a), even if we increase the *mem.threshold* to 10% of machine capacity, the long-tailed killing delay with a lot variance is still not mitigated (as high as 2.7s). Because of the lengthy killing delay, the node is likely to gradually fall into memory thrashing when the memory of killed task is not yet released while the new task has been launched. Setting a large *mem.threshold* helps to reduce memory thrashing, but it also causes memory waste.

The first reason of the lengthy killing delay is that it is difficult to deliver `SIGKILL` signal to kernel space from user space under memory pressure. Under thrashing, most of CPU time is spent on the kernel mode to reclaim memory. For example, if the kernel fails to find enough memory pages, a process that is requesting memory might be blocked and ended up in sleep state waiting for memory reclaim. If the process is NodeManager itself, it is de-scheduled of CPU access, which results in delayed OOM monitoring even if we have pin the NodeManager to a dedicated CPU core.

The second reason is due to the Garbage Collection (GC) of languages with automatic memory management (Java for Yarn and Go for Kubernetes) [16, 17]. In the extreme case, the long-tailed stop-the-world GC pauses NodeManager for as long as a few hundreds of milliseconds since the multi-threaded GC is not working efficiently when most of CPU cores are dominated by the kernel mode. Thus, the GC pause delays the delivery of `SIGKILL` signals. Unfortunately, both the aggressive memory reclaim of the kernel and the GC are not predictable. `SIGKILL` signals delivery from user space, though could be sped up by techniques like jitter-reduction and CPU cores affinity (like we already used CPU cores affinity), are not always guaranteed to be timely delivered. For example, as far as we observe, a full GC can pause the NodeManager as far as 1280ms when the system is suffering from memory pressure. If there is a `SIGKILL` pending at this moment, it is anyway delayed at least 1280ms. This also explains the high variance of the killing delay we measure.

(2) Inaccurate Estimation of Available Memory in Linux Linux reports `MemAvailable` as a coarse-grained estimation of the available memory that is enough for starting a new application without swapping. It calculates the available memory as the sum of the free memory (free memory pages residing on the buddy system excluding the pages under the low watermark) and a proportion of file pages (memory pages in page cache).

However, the estimation of the available memory is often inaccurate because it is almost impossible to decide what proportion of the file pages should be accounted as the reclaimable memory (the default is 50%). On one hand, reclaiming a frequently accessed file page causes disk thrashing. We run an experiment by setting *mem.threshold* as 10% to avoid anonymous page thrashing. As shown in Figure 3-(b), Spark-Kmeans, Spark-Pagerank and Cassandra suffer 20%, 22% and 40% performance penalty, respectively, when they are consolidated with best-effort jobs. The reason is that the file pages of Kmeans, Pagerank and Cassandra in the page cache are frequently reclaimed when sharing with the best-effort Spark-SQL jobs. For Spark Kmeans, the output of one stage is shuffled to the disk and it is immediately used in the subsequent stages due to its iterative nature [42]. For Cassandra, the data that is recently accessed is automatically cached by the OS page cache, thus file page reclaim hurts the performance of the read-intensive table scanning workload. On the other hand, reclaim of a merely accessed page causes almost no side effect. Thus, Spark-Wordcount still achieves an identical performance with or without isolation since almost all of its data reads in the map stage are never used in the reduce stage and can be safely reclaimed by OS.

Motivated by our user space prototype and experiments, we propose to move the OOM killer from user space to kernel space for reasons: (1) The kernel can detect the memory pressure and kill tasks immediately, and (2) The implementation in the kernel OOM killer can leverage more fine-grained metrics for accurate detection of both anonymous pages and file pages.

2.4 Conservative Linux OOM Killer

Intuitively, as user space OOM killing is slow and inaccurate, kernel space OOM killer should be used. However, the Linux OOM killer is too conservative to relieve memory pressure in a timely manner. First, the Linux OOM killer [2] is implemented inside the memory allocator as the last resort to free up memory when the memory allocator fails to find free memory pages after trying all the other possible means (such as memory swapping). This implies that the Linux OOM killer will not kill any process until both physical memory and swap partitions are critically low. Though this design delays the task killing to the last moment as the Linux OOM killer prioritizes the consequence of losing execution progress over the consequence of losing performance, it incurs unaffordable swapping overhead. Thus, applying the Linux OOM killer to oversubscribe opportunistic memory is not practical.

Second, the Linux OOM killer is unable to alleviate memory thrashing. By default, the Linux OOM killer is invoked only if the swap partition is almost used up, under which the system has already been suffering from significant memory thrashing. An intuitive but not recommended way to avoid anonymous page thrashing is to unmount the swap partition to prohibit any anonymous page from being swapped out. Though anonymous page thrashing is avoided, the file page thrashing is still not mitigated. The reason is that the task killing of the Linux OOM killer has to defer to the moment when the memory allocator fails to find free pages by releasing file pages from page cache, during which it causes significant file page thrashing. Thus, it inflicts serious overhead to I/O intensive applications. We further find that the memory thrashing activities are the interplay of the memory allocator, the memory reclaimer and the OOM killer. In particular, the Linux OOM killer is conservatively coupled with the memory allocator to ensure a success of memory allocation in the worst situation. Since the indirect memory reclaimer is asynchronously decoupled from the memory allocator, the Linux OOM killer inside the memory allocator is thus unable to detect memory thrashing.

There are also approaches relying on working set size estimation [13, 44, 45] to inform the OOM killer in advance whether the memory will thrash. However, this approach also has problems. First, the application memory usage is highly dynamic and unpredictable. For example, garbage collection that is used by JVM to reclaim unused memory causes a sudden decrease of anonymous page usage. This event is difficult to be predicted by the estimator. Second, the online prediction method may not be a good fit for kernel space implementation because of its overhead.

The analysis of inefficiency of the Linux OOM killer further reveals that a direct employment of the Linux OOM Killer still causes significant memory thrashing. The reason is that the goal of the Linux OOM killer is to minimize task killing, which is not applicable to the use case of opportunistic memory where memory is allocated and deallocated memory in a timely manner. We list the comparison of the discussed OOM killer approaches in table 1.

3 OS-Augmented UA Killer

We then present the design and implementation of UA killer, our kernel implementation of OOM killer uses existing kernel space

Table 1. Comparison of OOM Killer Approaches.

Approaches	Implementation	Fast killing	Thrashing detection	Degree of thrashing	Programmability
User space OOM killer	User space	×	×	Strong	✓
The Linux OOM killer	Kernel space	✓	×	Strong	✓
UA Killer	Kernel space	✓	✓	Constrained	✓

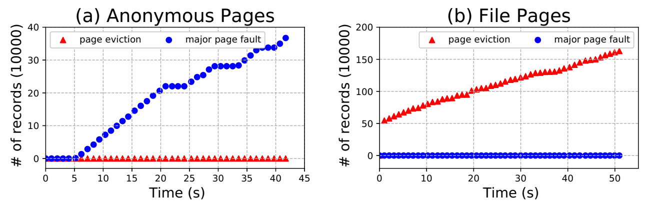

Figure 4. (a) Major page fault is constantly increasing when anonymous pages are thrashing. (b) Page eviction is constantly increasing when file pages are thrashing.

OOM killer primitives but with a careful consideration of the timing of task killing. UA killer features constraint thrashing, a mechanism that allows thrashing to take place within a limited amount of time so as to obtain key metrics for accurate thrashing detection and fast killing. UA killer is composed of a thrashing detector and a UA OOM killer. Note, UA OOM killer is a component of UA killer. The thrashing detector identifies whether a container is suffering from memory thrashing. If such a container is found, the OOM killer chooses a container to kill according to either the OOM score or our newly proposed OOM indicator.

3.1 UA Killer Design

Constraint Thrashing As it is difficult to avoid memory thrashing or predict it prior to its occurrence, UA killer features constraint thrashing to achieve fast and accurate thrashing detection. Specifically, constraint thrashing is to answer when a overcommitted best-effort task should be chosen to kill. We introduce a kernel parameter `cgth_time` as thrashing tolerance that is the period when a container is experiencing thrashing. For example, if we set `cgth_time` to three seconds, UA killer will not kill any containers until there is at least a container detected as thrashing for three seconds. Within this period of trashing tolerance, UA killer inspects both anonymous page thrashing and file page thrashing for accurate thrashing detection. UA killer has a command for users to configure the parameter of thrashing tolerance. Note that we evaluate its impact on job performance in §-5.2.

Thrashing Detector To identify a thrashing container, the thrashing detector inspects both anonymous page thrashing and file page thrashing. It samples two metrics for each container:

- **major page fault**: The major page fault occurs when the page fault handler tries to access a page that was previously swapped out (in swap partitions or in regular files mapped by `mmap`)., incurring disk I/Os.
- **page eviction**: Page eviction records the activation and eviction in the inactive file list. Specifically, it captures the file page fault of an evicted page.

Thus, an increasing number of major page faults (page evictions) indicates the thrashing of anonymous pages (file pages). To assess the effectiveness of the metrics, we run a set of background workloads to generate memory pressure, and then launch two microbenchmarks with Docker containers to force memory thrashing. The first benchmark spawns ten threads. Each thread randomly touches an one GB memory array. The second benchmark also spawns ten threads. Each thread randomly accesses an one GB file. As shown in Figure 4-(a), it is obvious that the number of the major page faults (page evictions) of a container continuously increases when its anonymous pages (or file pages) are thrashing.

Figure 4-(b) shows that the number of page evictions starts increasing from 500,000 (instead of zero) because the first round of I/O accesses that creates the file LRU list leads to the page eviction. However, the increase of page eviction caused by regular I/O is transient, and does not interfere with thrashing detection. If the benchmarks run without any memory pressure, the memory access will be served by anonymous pages and will not cause any major page faults. Similarly, the I/O accesses can be served by the page cache and will not cause actual I/Os. Thus, major page fault and page eviction can be used as metrics to identify whether a container is under anonymous page thrashing or file page thrashing.

We build two circular buffers to keep the sampled metric values for each cgroup. Each sampled value is bound with a timestamp. For a series of sampled values, the thrashing detector firstly decides whether it increases over time, and secondly decides if the series is longer than `cgth_time`, which is done by a cheap one-pass scan of the circular buffer since performance is crucial in kernel space.

Unlike the Linux OOM killer which is coupled with memory allocator, the thrashing detector of UA killer is implemented inside of the main routine of the memory reclaimer (shown in Figure 2). As the memory reclaimer is only invoked when the available memory on each NUMA node is below the low watermark, the thrashing detector is only invoked when the memory reclaimer invoked with memory shortage. This design has two benefits, (1) UA killer is able to detect memory thrashing immediately as soon as the system is experiencing memory pressure, which ensures a fast detection, and (2) UA killer remains inactive when the system has sufficient memory. In this case, it causes no over-killing and overhead for all kinds of jobs. Note that there might be multiple memory reclaimer instances running in parallel because (1) The kernel swap demon *kswapds* on each NUMA node run in parallel and (2) processes trapped into memory reclaim run in parallel. To avoid a race condition, a spinlock for each cgroup is required to protect the shared circular buffer from being concurrent accessed.

UA OOM killer After the thrashing detector identifies a thrashing container, UA OOM killer is called to choose a container to kill. UA OOM killer needs to distinguish containers of production jobs from containers of best-effort jobs in kernel space. By default, the Linux OOM killer allows users to set an OOM score (`oom_adj_score`) for individual processes to give hints to the kernel when it chooses a process to kill. The larger the value of a process, the more likely the process is to be killed. However, production containers could still be mistakenly killed by the Linux OOM killer even if we set a production container with the minimum OOM score (-2,000) because the Linux OOM killer ranks a process by both its memory footprint and its OOM score. To address this issue, UA OOM killer only kills containers whose OOM scores are larger than zero so that all containers of production tasks and critical processes (e.g., NodeManager) are excluded from being killed by setting an OOM score less than zero. Note that best-effort containers are required to configure an OOM score larger than zero.

Intuitively, there are two techniques that can be used to guide UA OOM killer. First, UA OOM killer should choose a container with the largest memory footprint to release the most memory by a single kill. Second, UA OOM killer should choose a container with the least execution time to keep the most execution progress. UA OOM killer leverages both techniques and uses a new OOM indicator $memory/execution\_time$ to choose the container with the latest launch time and the largest memory footprint. Besides the OOM indicator, we also notice that Linux OOM score (`oom_adj_score`) offers the flexibility that a user space cluster scheduler can leverage to couple with cluster-wide policies. Thus, UA OOM killer also offers an optional configuration allowing that killing decision is only based on the OOM score.

Unlike the Linux OOM killer that chooses a single process to kill, UA OOM killer targets a container. This design choice is desirable for the existing OS container based cluster schedulers [23, 27, 37], where a single task or service runs in a container. Thus, if a container is chosen to be killed, UA OOM killer terminates all processes belong to the container. After that, UA OOM killer clears the sampled major page faults and page evictions for all containers to ensure only one container is killed in each interval (`cgth_time`). This is necessary to avoid over-killing because (1) There are multiple thrashing containers. Killing them all is a waste. (2) Even though there is enough memory after a container is killed, major page fault and page eviction of a container will still increase when the container reads back its swapped-out working set. The increasing period, though is much shorter than the thrashing tolerance, interferes with the thrashing detector and thus cause over-killing.

3.2 Bypass Swapping

By default, each container has an equal opportunity to be swapped by Linux. For each container, Linux uses parameter `swappiness` to calculate the proportion of pages to scan for the file list and the anonymous list, respectively. For instance, if `swappiness` is set to 0, all scanning and reclaiming will focus on file pages, resulting in no anonymous page thrashing but significant file page thrashing. However, most critical processes (or services) cannot afford the cost of any kinds of thrashing by which the performance and the reliability are severely harmed. For example, if the Spark driver is stuck into thrashing, it may lose response and fail to communicate with the cluster scheduler, causing significant performance degradation or even application failure.

To address this issue, we make a simple yet effective technique. The idea is that we bypass scanning and reclaiming of memory pages for containers whose `swappiness` equals to zero. Therefore, CPU-intensive page scanning and IO-intensive page reclaiming will not interfere with these containers. Note, bypass swapping is only allowed for certain components, such as the job manager (Spark driver for Spark and ApplicationMaster for MapReduce) as these components involves latency-critical services (such as task scheduling, RPC and resource monitoring), causing the job failure if these services are delayed [11].

Algorithm 1 UA killer decision making heuristic.

```
1: for each $mem\_cgroup$ in $mem\_cgroup\_list$ do
2:     if $cgroup\_thrash\_threshold(mem\_cgroup)$ then
3:         $select\_cgroup\_kill()$;
4:         $clear\_memcg\_thrash()$;
5:         $break$;
6:     end if
7: end for
8: function $cgroup\_thrash\_threshold(mem\_cgroup)$;
9:     $memcg\_thrash = to\_cgthrash(mem\_cgroup)$;
10:    /* if the # of page evictions has been increasing for cgth_time */
11:    $pgev\_inc = check\_increase(memcg\_thrash.pgev)$;
12:    /* if the # of major page faults has been increasing for cgth_time */
13:    $pgmj\_inc = check\_increase(memcg\_thrash.pgmj)$;
14:    return $pgev\_inc || pgmj\_inc$
15: end function
16: function $select\_cgroup\_kill$;
17:     $cgroups = choose\_best\_effort\_cgroups()$
18:     if $score\_only()$ then
19:         $kill(\arg\max_{cgroup}(cgroup.mem\_adj\_score))$;
20:     else
21:         $kill(\arg\max_{cgroup}(\frac{cgroup.mem}{cgroup.exe\_time}))$;
22:     end if
23: end function
```

3.3 Implementation

Algorithm 1 illustrates UA killer's decision making heuristic. Each container is associated with a memory cgroup. In line 1, the thrash detector iterates the cgroup list and checks if a container is thrashing long enough to be killed. Lines 16 to 23 implement UA killer. Line 18 allows users to configure which heuristic to use. This algorithm is implemented in function `void shrink_node_memcg()` that is called by both indirect and direct memory reclaim. In the function, UA killer is called before the reclaimer iteratively searches for pages to be reclaimed. Therefore, once a container is killed by UA killer, expensive memory reclaim can be immediately avoided.

We augment a data structure `struct mem_cgroup_thrash` to hold two circular buffers and embed it into `struct mem_group`. We implement bypass swapping in function `void get_scan_count()`. This function is used to calculate the number of pages to be scanned in each memory cgroup. In this function, we set both numbers of the file pages and anonymous pages to be scanned to zero once we found a container whose `swappiness` is zero.

[Summary] UA killer allows memory thrashing to some extent, but thrashing is strictly limited by user tolerance. UA killer detects both anonymous page thrashing and file page thrashing. It exploits parameters `swappiness` and `oom_adj_score` to make itself informative of the user space cluster scheduler, and also make itself programmable. The implementation of UA killer consists of about 200 LOC based on Linux 4.12.8. Thus, UA killer supports all kinds of workloads regardless of their language runtime.

4 Charon

4.1 Overview

Upon UA killer, we develop Charon, a cluster scheduler designed for best-effort jobs to oversubscribe opportunistic memory. Charon

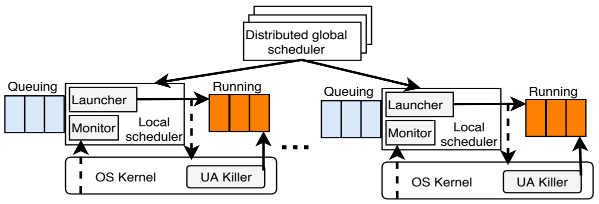

Figure 5. The Architecture of Charon.

aims to leverage UA killer to enable accurate thrashing detection and timely eviction of overcommitted opportunistic tasks. As shown in Figure 5, Charon is composed of a set of distributed global schedulers and per-node local schedulers. A global scheduler assigns best-effort tasks to nodes while a per-node local scheduler performs scheduling of containers corresponding to tasks. Charon uses UA killer to terminate overcommitted opportunistic tasks. As UA killer terminates tasks based on memory thrashing, one benefit is that it removes the need to identifying the per-node memory limit of each node (*mem.threshold*) that used to be tedious and ineffective by a user-space OOM killer.

Charon is implemented upon Mercury [27] with queue management [33]. Mercury is a state-of-art opportunistic scheduler that has been merged to the latest Apache Yarn [36]. In Yarn, ResourceManager is a centralized component, which manages the cluster resources and schedules jobs. NodeManager manages each node and its tasks. AppMaster is a per-job orchestrator that manages all tasks of the job. For production jobs, we use the centralized Capacity Scheduler which is the default scheduler in Yarn.

System Components Charon uses the same distributed global scheduler as Mercury [27, 33] but implements a novel per-node local scheduler to interact with UA killer. For placing a task, this policy prioritizes the node with the least load. The design of the per-node local scheduler is our major contribution. A local scheduler has two major components: a node monitor and a task launcher.

The node monitor inspects memory availability `MemAvailable` via *proc* file exposed by OS, which is similar to a user space implementation. However, we remove the task killing functionality from user space, and use UA killer in kernel space instead. In addition, the node monitor also inspects the memory usage for each container. This information is used to construct the task memory profile that will be used to tackle the task over-killing issue.

The task launcher launches both best-effort tasks and production tasks. For a best-effort task, it launches the task if the node has enough memory. Otherwise, it queues the task and waits until the memory is available. For a guaranteed task, since it has a higher priority, the task launcher always immediately launches it. When the number of queued containers reaches the queue size limit, a newly allocated container will be rejected by the task launcher and returned the task as a failure to AppMaster. AppMaster uses its container retry policy to decide if it resubmits the container request or fails the application. Note that we keep the queuing mechanism on each NodeManager the same as that in work [33].

In order to facilitate UA killer, the task launcher configures each Docker container to inform UA killer with task information. To avoid CPU interference, the task launcher splits the CPU cores on each node into two CPUSETs and assign them to best-effort jobs and production jobs, respectively.

Table 2. Parameter setting for UA killer.

Task Type	`oom_adj_score`	`swappiness`
Task (best-efforts jobs)	1000	60
AppMaster (best-efforts jobs)	-1000	0
Task (production jobs)	-1000	60
AppMaster (production jobs)	-1000	0

Kernel parameters Charon configures parameters `oom_ajd_score` and `swappiness` for each task depending on the application type. Table 2 shows the configuration of each parameter.

- For a best-effort task, we set `swappiness` to the default value and `oom_adj_score` to 1000 (>0) such that UA killer is able to kill the task when facing memory shortage.
- For a production task, to avoid being killed by UA killer, we set `oom_adj_score` to -1,000. We use the default `swappiness` (60) for production tasks to allow constraint thrashing since UA killer is able to release memory fast enough before memory thrashing degrades performance.
- AppMaster is vulnerable to thrashing and killing. We configure AppMaster with an `oom_adj_score` of -1,000 and a `swappiness` of zero to avoid being killed or thrashing.

Recall that UA killer can be configured to terminate tasks according to the OOM score only. For each best-effort task, we explore to configure the OOM score to its application_id since Yarn assigns the id for each application in an increasing manner. Hence, upon memory pressure, UA killer kills the latest task by choosing the task with the largest OOM score, expecting that the most execution progress can be preserved. More sophisticated killing policies can be implemented depending on the scheduling objectives. For example, Charon can support two groups of best-effort jobs. One group is superior than the other by setting two different OOM scores.

4.2 Task Over-killing

We find that launching opportunistic containers according to parameter `MemAvailable` can cause significant task over-kills due to the delayed resource charging. For example, assuming `MemAvailable` reports 10GB free memory in a node and there are ten tasks in the queue, the local scheduler decides the number of opportunistic containers could be launched. By default, it launches all tasks at once since the starting memory usage of each task is small and they could all fit into the node. However, their memory usage gradually increases and exceeds the node memory limit. Thus, OS cannot immediately tell the peak memory usage of a task. If the peak memory usage of each task is 4GB, only two tasks can complete but eight will be killed, causing significant task over-killing.

To tackle task over-killing, we develop two techniques: memory snapshot and delayed launch. Memory snapshot keeps a snapshot of of `MemAvailable` as SnapMem. SnapMem is immediately charged by the estimation of future memory usage of running containers. For each best-effort task, its estimation of future memory usage is calculated per the profiled peak memory of its job. The task launcher launches best-effort containers based on SnapMem. SnapMem is synchronized with `MemAvailable` after a predefined period (e.g., five seconds). When memory snapshot is applied to the example above, after two best-effort tasks are launched, SnapMem drops to 2GB and there is not enough memory for launching more tasks.

For delayed launch, once a task is killed by UA killer and this event is detected by Charon, it delays the launch for the next best-effort task to be scheduled for an extra predefined period of time until the memory pressure is more alleviated. As a result, the newly launched task has a better chance to survive. Note that since jobs are recurring, using the historical information as an estimation is practical and it has been widely adopted [6, 12, 33, 43].

Charon relies on OS to report available memory to launch new opportunistic task. However, unlike user space OOM killer which kills tasks by relying on OS-reported available memory, Charon relies on UA killer for accurate thrashing detection to avoid memory overwhelming. As it is challenging to tell the exact amount of free memory from OS, Charon addresses this issues in an indirect but novel manner, that is to inspect memory thrashing as an indicator for memory availability. In such a way, no matter how many opportunistic tasks are launched in user space, UA killer guarantees that the oversubscribed memory is still able to be fast released when necessary. As a result, the memory utilization can be maximized and can be allocated on-demand without worrying about the performance negative impacts due to memory overcommitment.

5 Evaluation

5.1 Experimental Setup

The implementation of Charon consists of 1,500 LOC. The project is open-source at Github `https://github.com/yncxcw/charon`.

We evaluate Charon to answer the following questions:

- Is UA killer in Charon fast enough to kill best-effort tasks to avoid significant memory thrashing?
- Is Charon able to protect the performance of production jobs/service from memory thrashing?
- Is Charon able to improve the performance of best-effort jobs in terms of throughput and job completion time?

Physical Cluster Setup We evaluate Charon in a 26-node cluster. Each node has two 8-core Intel Xeon E5-2640 processors with hyper- threading enabled, 132GB of RAM, and 5x1-TB hard drivers configured as RAID-5. The nodes are inter-connected by 10 Gbps Ethernet. UA killer is implemented on Linux-4.12.8 and Charon is implemented on Hadoop-3.0.0. We use the same version of Linux kernel and Hadoop for comparison. We use Spark-2.1.0 for Spark applications. Docker-1.12.1 is used to create OS containers. The image is downloaded from the online Docker hub sequenceiq/hadoop-docker. We limit the queue length on each node to five.

Workloads For production batch jobs, we use Spark benchmarks Kmeans, Pagerank, Bayes, Wordcount and Sort from HiBench [24]. We empirically tune the optimal heap size for each application. This tuning process ensures that the performance is not affected by a suboptimal heap size while avoiding memory over-provision. For production interactive service, We use Cassandra that is a NoSQL database with data persistence.

For best-effort jobs, we use TPC-H benchmark on Spark-SQL, and MapReduce benchmarks Wordcount, Sort, Grep, and Read. Spark-SQL is a popular query engine built on Spark for fast and in-memory processing. We use it to simulate the exploratory data queries. The input data size for Spark-SQL is 10GB. We use MapReduce benchmarks to evaluate large scale batch jobs. The input data size of each MapReduce application is 5TB. For Spark-SQL workloads, we use the default retry policy in which each container is allowed to fail twice. For MapReduce workloads, we allow each container to fail up to 40 times to resist massive task killing because each job spawns a very large number of containers.

Approaches We adopt Capacity Scheduler [41] to schedule production jobs and service. In the following experiments, we compare the following approaches in scheduling best-effort jobs.

- **Alone** Best-effort jobs run alone in the cluster without memory pressure from production jobs.
- **Mercury** The state-of-art opportunistic scheduler built on Apache Yarn. For oversubscription, we empirically set the per-node memory limit to 90% of node memory capacity.
- **Kuber** The implementation of Kubernetes burstable class upon Yarn. Kubernetes considers one container request a time during scheduling. For oversubscription, we empirically set $memory.theshold$ to 10% of the node memory capacity.
- **Charon** Charon opportunistic scheduler built upon UA killer.

There are fundamental differences between Charon and the other approaches. Mercury uses a static resource partition mechanism. As a result, the unused memory from a production container cannot be exploited by best-effort jobs. Both Kuber and Charon allow the best-effort jobs to utilize the opportunistic memory to the maximum extent before the OOM killer kicks in. However, Kuber and Mercury use a user space OOM killer while Charon uses a kernel space OOM killer (UA killer).

Metrics We use the job completion time (JCT) to evaluate the performance of production batch jobs, and use the tail latency and throughput to evaluate performance of production service. For best-effort jobs, we use the JCT to evaluate the job performance and use the job completion rate (JCR) to evaluate the throughput. JCR is the ratio of the number of completed jobs to that of the submitted jobs. Note that the objective of memory oversubscription aims to protect the performance of production jobs/service when they are consolidated with best-effort jobs, and improve the performance of best-effort jobs as much as possible.

Simulation To simulate a production cluster, we build a simulator Charon-SLS based on the Yarn scheduler load simulator (SLS). Its implementation consists of 1,000 LOC. The input specification includes the job description, task description and memory usage description. We build a tool to generate the input specification by parsing a Google production trace [34]. By doing so, we want to understand the amount of opportunistic memory that Charon can oversubscribe in a production cluster. We use MapReduce benchmarks to generate all production and best-effort jobs. We simulate a 100-node production cluster with 32 cores and 128GB memory per node. We limit the queue length on each node to ten.

5.2 Killing Delay and Thrashing Tolerance

First, we study the killing delay of UA killer and evaluate Charon with different configurations of thrashing tolerance. We run Spark Pagerank jobs as the production jobs. We then submit Spark-SQL TPC-H queries as the best-effort jobs.

Figure 6-(a) shows that the killing delay of UA killer is much lower (more than 100x) than its user space counterpart. It is less than 20ms with low variance. As its execution in kernel space cannot be preempted, it suffers from no interference.

The trashing tolerance (`cgth_time`) in UA killer represents how sensitive OS is to memory thrashing. We study its impact on the performance of the best-effort jobs. Figure 6-(b) depicts the JCT of Spark-SQL best-effort jobs when the thrashing tolerance varies

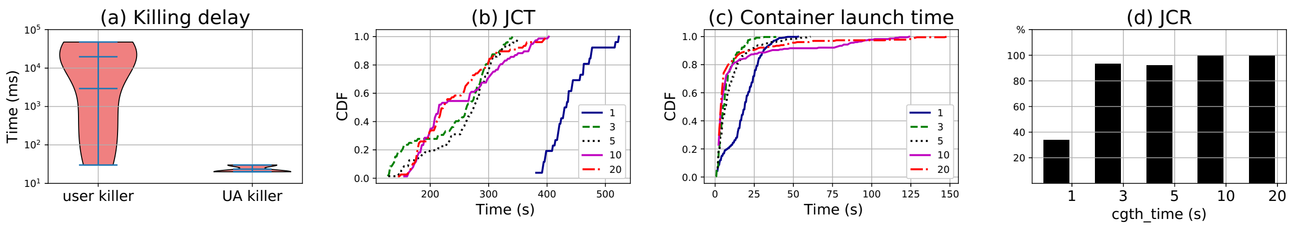

Figure 6. (a) shows comparison of the killing delay between the user space killer and UA killer (kernel space); (b)-(d) show the JCT, container launch time, and JCR for Spark-SQL best-effort jobs under various thrashing tolerance `cgth_time`.

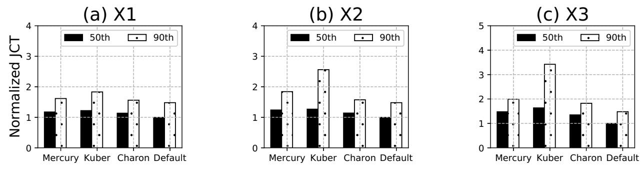

Figure 7. Normalized 50th and 90th JCT for production jobs.

from one to 20 seconds. It shows that the jobs achieve the worst JCT when the thrashing tolerance is set to one second. This is the result of excessive task killings when the thrashing tolerance is too small, such that UA killer is extremely sensitive to memory thrashing. We also find the JCT presents a long-tailed latency with a large thrashing tolerance. In particular, the tailed JCT is as high as 441 seconds when the thrashing tolerance is set to 20 seconds. After carefully examining each piece of the latency, we fine the degraded JCT is caused by the increased container launch time.

We measure the container launch delay as the period from the moment when NodeManager issues the container launch command to the moment when the newly launched container is detected by NodeManager. During container launch and JVM launch in the container, the JVM requests memory from OS to set up its heap. When tasks are conservatively killed due to a large thrashing tolerance, memory shortage cannot be immediately resolved by task killing. OS needs a long time to lock enough memory associated with each memory request. Thus, the JVM suffers a long time to obtain memory from OS. Figure 6-(c) depicts container launch delay when the thrashing tolerance varies from 1 to 20 seconds. A thrashing tolerance of 1 also causes a long container launch time because the launched containers are killed before the task inside is detected. Figure 6-(d) shows the JCR of the best-effort jobs under different thrashing tolerances. The JCR stabilizes when the thrashing tolerance is equal to or greater than 3 seconds.

[Summary] Thus, we empirically set `cgth_time` to 3 seconds as it achieves the best JCT and good JCR (72 out of 77 jobs completed).

5.3 Oversubscription - Production Jobs

We now evaluate the effectiveness of Charon in memory oversubscription with production jobs and best-effort jobs. We replay a subset of Google trace [34] by submitting jobs according to their timestamps in the trace. Note that the replayed jobs may not strictly reflect the memory usage recorded in the trace since it is difficult to generate exactly the same memory footprints. The trace contains 39 Spark HiBench production jobs and 703 Spark-SQL best-effort jobs. The memory usage of the production jobs is shown in Figure 1-(b). To increase the load of best-effort jobs, we replicate each best-effort job in the trace file by 2 times and 3 times, which are denoted as scenarios X2 and X3, respectively. Note that in all three scenarios the load of production jobs stays the same.

Performance of production jobs Figure 7 (a)-(c) show the normalized 50th and 90th JCT of the production jobs. Note that Default means the experiment runs without best-effort jobs. The JCT of production jobs is normalized to that when each job runs in isolation. Charon achieves almost identical performance to Default, which demonstrates that Charon is able to protect the performance of production jobs from memory thrashing.

We also find that Kuber leads to the worst JCT. In specific, Charon outperforms Kuber by 19%, 38% and 62% for the 90th JCT under various loads X1, X2 and X3, respectively. It is due to that the falsely reclaimed file pages result in the performance penalty to production jobs because of the inaccurate estimation of available memory in Kuber. Charon protects the active file pages of the production jobs from being reclaimed by aggressively killing best-effort tasks when file thrashing on the production jobs is detected. Furthermore, the heuristic of UA killer will not result in over-killing for workloads without large refaulted file pages (e.g., Wordcount).

Performance of Spark-SQL best-effort jobs As shown in Figure 8 (a)-(c), Charon achieves the best JCT under various loads X1, X2 and X3. In particular, Charon improves the 95th JCT over Mercury by 8.8X, 4.8X and 1.8X under loads X1, X2, X3, respectively. Mercury suffers from long-tailed performance because of its static memory partition. From the log files of NodeManager, we find that the best-effort tasks are significantly queued at each node, not able to be executed until at least one long running production task on the same node completes and releases its resource. The inflexibility impedes the opportunistic memory from being fully exploited. As a result, the queuing delay by Mercury is pronounced even under a light load (X1). In contrast, both Charon and Kuber schedule tasks according to the instantaneously available memory of each node. The aggressive task launching not only avoids the lengthy task queuing delay but also increases the chance of a task being killed. However, the fail-and-reschedule strategy increases the opportunity of a killed task to be placed on a node with a relatively lighter load, compensating the overhead caused by task killing.

Under a high load (X3), performance of Charon and Kuber is degraded because of massive task killing. The best-effort tasks overwhelm the node memory so that OOM killer has to kill en masse to avoid swapping. In addition, we observe that Kuber causes a higher variance of JCT than Charon. Though the median JCT and the 90th JCT are similar for Kuber and Charon, the 99th JCT of Kuber is up to 1.8x higher than that of Charon. The reason is that

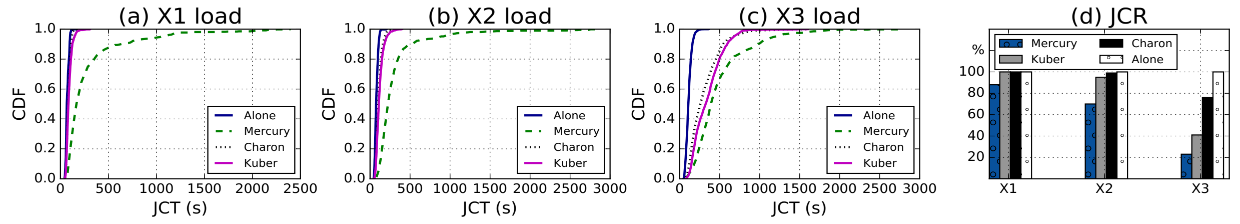

Figure 8. Performance of Spark-SQL best-effort jobs. (a)-(c) show the JCT under various loads. (d) shows the JCR.

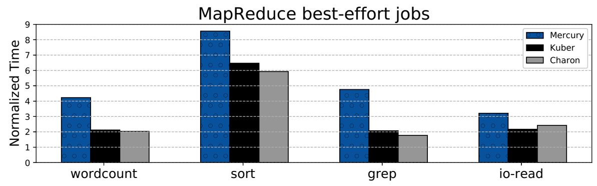

Figure 9. The normalized JCT of individual MapReduce jobs.

Kuber relies on a user space OOM killer, which introduces a high variance to the container launch time because of its varying killing delay, causing a significant increase of the tailed JCT.

Figure 8-(d) shows the JCR by different approaches. For low and moderate loads (X1 and X2), Charon yields no job failure because the opportunistic memory is sufficient for best-effort tasks. However, 12% of the best-efforts jobs fail with Mercury even under the low load (X1) because of its static memory partition. As a result, the queue on each node soon reaches its limit and tasks are immediately rejected and failed. Under a high load (X3), Charon is able to complete 76% of the submitted jobs while Mercury only completes 23%, which implies that Charon improves the throughput of the best-effort jobs by 3.1x over Mercury. Charon achieves 23% higher throughput than Kuber under a high load (X3), suggesting that Charon is able to exploit more opportunistic memory than Kuber does, because (1) Charon does not rely on the static per-node memory limit configuration, and (2) UA killer is faster than the user space killer, avoiding the situation of running into continuous memory pressure. Charon and Kuber achieve a similar JCR under loads X1 and X2, suggesting the user space killer performs under low and moderate loads.

Performance of MapReduce best-effort jobs We run MapReduce applications as the best-effort jobs and show the normalized JCT in Figure 9. Charon improves over Mercury by about 25% to 48% for different types of jobs. Kuber and Charon achieve similar performance. Unlike Spark-SQL that is usually dominated by CPU [32], MapReduce applications are dominated by file read in map phase and data shuffling in reduce phase. The intensive I/Os of MapReduce create substantial file pages that generate memory thrashing to the production jobs. Consequently, UA killer detects file page thrashing and kills best-effort tasks aggressively. As a result, less opportunistic memory can be exploited in this situation compare to that when Spark-SQL is used as the best-effort jobs.

[Summary] Compared to Mercury, Charon significantly improves the performance of best-effort jobs by aggressively exploiting opportunistic memory. Through its accurate trashing detection and timely task killing, Charon avoids significant memory thrashing that is common in Kuber.

5.4 Oversubscription - Production Service

Low-latency interactive services, such as NoSQL database, have become popular in BigData processing. As a trend that these services are sharing the infrastructure of a multi-tenant cluster, it is important to protect their performance in a consolidated environment. In this experiment, we run Cassandra as an online interactive service for the production service and use Spark-SQL for the best-effort jobs. By default, to serve a read request, Cassandra tries to locate the requested data in RowCache, an in-memory data cache. If it is a hit, the request is immediately satisfied from the memory. Otherwise, the request is served from the disk by reading the data from on-disk SSTable. For read intensive workloads, due to the limited size of RowCache, most of the requests may fall on the disk.

We deploy Cassandra in the cluster and configure each Cassandra slave daemon with an 8G heap size. We use workloads-B from YCSB, a read-mostly workloads, as the benchmark to evaluate whether UA killer is able to throttle the file page thrashing in a timely manner. We configure the YCSB client with 32 threads. We use the same 703 Spark-SQL best-effort jobs as in §-5.3. Note that CPUSET is used to isolate CPU interference in this experiment.

As shown in 10(a), the throughput of Cassandra drastically decreases with both Mercury and Kuber due to significant file page thrashing. The reason is that both Mercury and Kuber simply rely on `MemAvailable` to decide whether to launch or to kill tasks, ignoring file page reclaims. Thus, they are not aware that the file pages of Cassandra are aggressively reclaimed. As OS tends to prioritize the reclaim of read-only pages to reduce the I/O traffic, many of the file pages of Cassandra are reclaimed. Charon kills best-effort tasks when thrashing on Cassandra production service is detected, thus it improves its throughput over Kuber by 11.1%, 40.4%, and 41.4% under loads X1, X2 and X3, respectively. As shown in 10(d), Charon achieves the worst JCR, because its UA killer acts more aggressively than the user space OOM killer in killing best-effort tasks so as to protect the file pages of Cassandra from being thrashed. Recall that Cassandra production service has the priority in workload consolidation. Charon attains up to 94% of the throughput of Cassandra if there is no consolidation. Note that without consolidation, there is no throughput of best-effort jobs at all.

Figure 10(b)-(c) shows the median and the 99^{th} latency of Cassandra service by different approaches under various loads. Charon achieves similar median latency as if there is no consolidation. It also shows that all approaches incur a significantly increased 99^{th} tail latency, though Charon outperforms Mercury and Kuber under X3 load. The possible reasons include: (1) UA killer allows a constraint thrashing before killing a task, but parameter `cgth_time` of 3 seconds might be too long to protect the tail latency, and (2) there are more network congestion and disk accesses when admitting

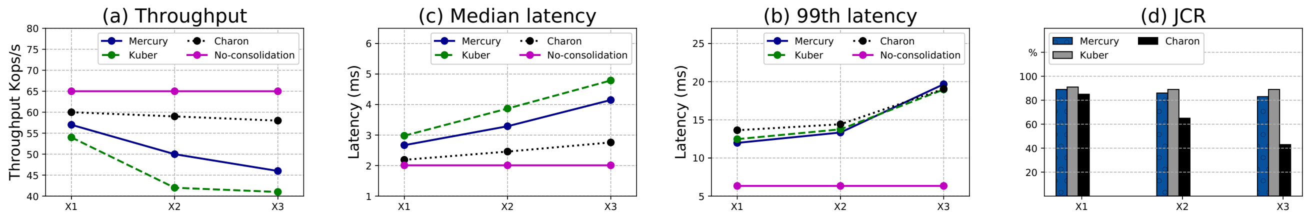

Figure 10. (a)-(c) Throughput, the median and the 99^{th} latency of Cassandra under different loads. (d) JCR of best-effort jobs.

more best-effort jobs. It is non-trivial to address the tailed latency of production service, which will be our future work.

[Summary] Charon is able to protect the performance of production interactive service at large while maximizing opportunistic memory usage, because its UA killer can detect the file page thrashing and killing tasks timely.

5.5 Production Trace Simulation

We run Charon-SLS using a subset of Google trace to conduct the production simulation on a simulated 100-node cluster. The subset contains 754 production jobs and 3,074 best-effort jobs. We derive the job profiles from Google datacenter trace. The profile includes the type of a job (i.e. best-effort or production), the task execution time, the job launch timestamp, the number of tasks in a job and the memory usage of each task. As done in §-5.3, we repeat best-effort job submissions to increase the overall load to X2, X3, and X4. Figure 11-(a) shows the allocated memory and the actually used memory for production jobs. It shows that the production jobs leaves nearly 30% of the allocated memory unexploited. It indicates that unexploited opportunistic memory is prevalent in a production cluster, advocating the need of memory oversubscription.

Figure 11 (b)-(c) shows that the tail latency of best-effort jobs suffers 20X-25X slowdown by Mercury. Due to its head-of-line blocking policy, tasks of best-effort jobs wait for memory to be released. Thus, only 51% of the best-effort jobs can be completed even under low load X1, suggesting that the static memory partition is not practical for large scale memory oversubscription deployment. Kuber and Charon address this issue by oversubscribing instantaneously available memory in an aggressive manner. Moreover, Charon improves the JCR over Kuber by 26% under heavy load X4, indicating that Charon is able to exploit more opportunistic memory in a production cluster.

Finally, we evaluate the two techniques of Charon in addressing task over-killing, i.e., memory snapshot and delayed launch. Figure 11 (e) shows the JCR of the best-effort jobs with the techniques enabled and disabled. It shows that the enabled techniques are able to improve the JCR by about 15%.

[Summary] The simulation by using the production trace reveals that unexploited opportunistic memory is prevalent in clusters. Charon with UA killer outperforms start-of-the-art approaches Mercury and Kuber in memory oversubscription.

5.6 Discussions

Overhead The overhead of Charon is due to the constraint thrashing, which only incurs under memory pressure. Production jobs suffer from thrashing overhead only during the constraint thrashing period. Our experiment shows the overhead caused by thrashing is negligible for throughput-oriented production workloads, such as machine learning workloads. We still observe increased tail latency for latency-critical production workloads. We will address this issue in our future work. When the cluster nodes run without memory pressure, UA killer causes no side effect as memory reclaimer remains sleep state. As demonstrated in §-5.2, there is a trade-off between the job performance and the thrashing tolerance: by reducing `cgth_time`, memory thrashing is reduced and performance of production jobs is improved, but the chances of best-effort tasks being killed are increased.

Swappiness Charon sets parameter `swappiness` to the default value (60) for both best-effort tasks and production tasks. A critical question remains unanswered: why do not set `swappiness` to zero for production tasks to bypass memory reclaim? In such a way, production tasks will not suffer from any page thrashing. The answer is to avoid task over-killing when I/O intensive production tasks create massive file pages which are not actively used. To validate this, we run a Spark Wordcount as the production job in a single-node cluster and submit Spark-SQL as the best-effort job. We measure the file page size of the Wordcount containers and the number of concurrent Spark-SQL tasks.

As shown in Figure 12-(a), when the parameter `swappiness` is set to 0, the file pages which belong to the production job are not allowed to be reclaimed by OS. As a result, when the production job is populating more file pages during the execution, Figure 12-(b) shows that many concurrent best-effort tasks running with opportunistic memory are killed by UA killer. In contrast, if memory reclaim on the production job is allowed when `swappiness` is set to 60, memory reclaim on the file pages will not cause over-killing to the best-effort tasks because the reclaimed file pages of the production job are not actively used and will not refault into memory. Thus, these pages are safe to be released without causing thrashing to the production job. The experiment demonstrates that UA killer is able to distinguish the memory reclaim activities that are detrimental to production jobs and allow others to reclaim so as to minimize task killing to best-effort job.

Per-node memory limit During the experimentation, we find it is painful to set an appropriate per-node memory limit for a user space OOM killer. First, the setting should reserve some memory for OS and control blobs (e.g. NodeManager for Yarn and DataNode for HDFS), whose usage is hard to profile. Second, it is highly related to the OS kernel and hardware architecture, e.g., by increasing the RAM capacity of cluster nodes, the setting needs to be updated. Last but not least, it should take application characteristics into account. However, applications have diverse memory demands. It takes us many rounds of tuning to find the best per-node memory limit for Mercury and Kuber. UA killer overcomes this issue by implementing OOM killer in kernel space where the resolution of memory pressure is self-contained.

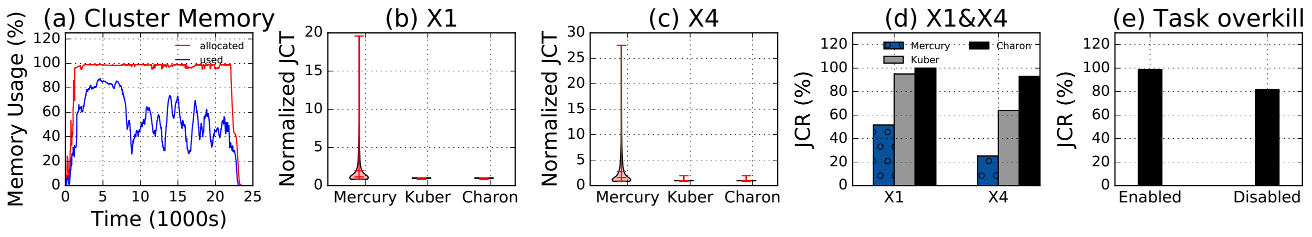

Figure 11. (a)-(d) show the production trace evaluation, and (e) show the JCR of best-effort jobs under task over-killing.

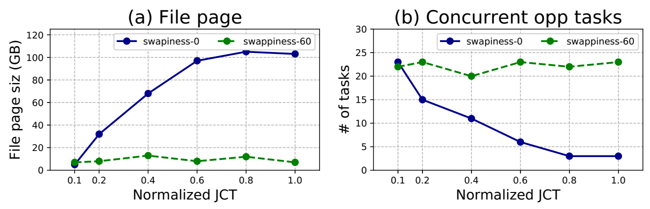

Figure 12. Illustration of the reason to set `swappiness` to 60 for production job: avoid over-killing to the best-effort job.

Application characteristics Our experimentation recommends to allocate opportunistic memory to tiny and short tasks, such as data queries on a small or a medium data set, so as to reduce the chance of a task being evicted. This intuition is two folds. First, the availability of opportunistic memory is highly dynamic and is hard to be accurately characterized. Second, the OOM indicator from UA killer favors tasks with large memory usage.

We notice an extreme case: when all opportunistic memory is fully allocated and the queue on each node is full, incoming tasks are immediately rejected. Thus, Charon can only schedule and execute a single large-scale MapReduce job at a time in our cluster. To address the issue, the local scheduler should be implemented with fair-share policies (FIFO by default).

Best-effort jobs rely on the failover mechanism to relaunch the failed tasks. For each task, the mechanism defines a maximum retry number. We find a large retry number does not improve the JCR of best-effort jobs when the opportunistic memory on each node is fully loaded because an additional retry is rejected immediately.

6 Related Work

Cluster Scheduling Yarn [36] and Mesos [23] are two widely used open-source centralized schedulers. Sparrow [31] is a fully distributed scheduler based on random sampling. Hawk [12] and Mercury [27] both implement a hybrid scheduler to avoid inferior scheduling decisions as a trade-off of scheduling quality and scalability. CARBYNE [18] allows applications to altruistically yield their allocated resources to achieve secondary goals. Study [29] proposes a framework to adaptively split large job into small jobs to reduce queuing delay. Madea [15] implements placement constraints for scheduling long running applications. SDChecker [9] profiles and analyzes the causes of latency in a multi-stack scheduling environment (e.g. Spark on Yarn). Most of the studies focus on the design of new policies and mechanisms for the global scheduler. Charon, on the other hand, focuses on the local scheduler and it can be implemented upon those representative schedulers.

Opportunistic resources Cluster schedulers Mercury [27], Apollo [6] and Borg [37] propose to utilize the leftover resources for scheduling best-effort jobs. However, they rely on the static resource partition or set a safe margin for production jobs, thus are unable to unleash the power of opportunistic resources. Pado [40] is an application-level approach to utilize opportunistic resources for machine learning jobs. CEDULe [4] proposes to leverage burstable performance instances to accommodate bursty workloads. Mos [5] is a workload-aware object store that enhances different tenants' QoS by dynamic configuration tuning and fine-grained resource allocation. Study [28] proposes a model to compute the quantity of resources that achieves the best execution time without resource over-commitment. BIG-C [8] is a preemptive-based framework that improve the performance for latency-critical applications in a shared cluster. Another group of efforts [19, 21, 22, 35] try to leverage opportunistic resources provided by cloud providers (e.g., spot instance from AWS) to reduce the financial cost. They heavily rely on the cloud provider to inform the possible eviction and the availability of opportunistic resources while our approach schedules tasks based on instantaneous resource availability.

OS transparency Charon is similar to other work that advocates more exposure of application information to kernel or vice versa. Redline [39] proposes the first class support for interactive services. It dynamically adapts to recent load, maximizing responsiveness and system utilization. Mittos [20] designs a new OS interface for applications to express their QoS to achieve predictable tail latency. It adopts a fast failover mechanism that allows requests to be served on a less busy node.

7 Conclusion

This paper presents Charon, a cluster scheduler for oversubscribing opportunistic memory. Charon is empowered by UA killer, a kernel space OutOfMemory (OOM) killer that enables timely and informative task eviction while avoiding thrashing on critical processes. System implementation of Charon with extensive evaluations show that (1) Charon achieves task killing in a timely manner and removes the dependency on static resource configuration via kernel space UA killer, and (2) Charon significantly improves the throughput of best-effort jobs and cluster utilization, while prioritizing production jobs. Charon equipped with UA killer is a general solution, which is applicable to many cluster schedulers.

8 Acknowledgments

We thank the anonymous reviewers and our shepherd Dilma Da Silva for their valuable comments on our paper. This research was supported in part by U.S. NSF grant SHF-1816859.

References

[1] Cassandra. http://cassandra.apache.org, 2008.

[2] Taming the oom killer. https://lwn.net/Articles/317814/, 2009.

[3] Kubernetes. https://kubernetes.io/, 2015.

[4] A. Ali, R. Pinciroli, F. Yan, and E. Smirni. Cedule: A scheduling framework for burstable performance in cloud computing. In Proc. of IEEE ICAC, 2018.

[5] A. Anwar, Y. Cheng, A. Gupta, and A. R. Butt. Mos: Workload-aware elasticity for cloud object stores. In Proc. ACM HPDC, 2016.

[6] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: scalable and coordinated scheduling for cloud-scale computing. In Proc. of USENIX OSDI, 2014.

[7] D. P. Bovet and M. Cesati. Understanding the Linux Kernel. O'Reilly Media, Inc., 2005.

[8] W. Chen, J. Rao, and X. Zhou. Preemptive, low latency datacenter scheduling via lightweight virtualization. In Proc. of USENIX ATC, 2017.

[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, A. Pi, S. Wang, and X. Zhou. Pufferfish: Container-driven elastic memory management for data-intensive applications. In Proc. of ACM SoCC, 2019.

[11] T. Dai, J. He, X. Gu, and S. Lu. Understanding real-world timeout problems in cloud server systems. In Proc. of IEEE IC2E, 2018.

[12] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel. Hawk: Hybrid datacenter scheduling. In Proc. of USENIX ATC, 2015.

[13] P. J. Denning. The working set model for program behavior. Communications of the ACM, 1968.

[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] 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 ACM EuroSys, 2018.

[16] L. Gidra, G. Thomas, J. Sopena, M. Shapiro, and N. Nguyen. Numagic: A garbage collector for big data on big numa machines. In Proc. of ACM ASPLOS, 2015.

[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] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In Proc. of USENIX OSDI, 2016.

[19] W. Guo, K. Chen, Y. Wu, and W. Zheng. Bidding for highly available services with low price in spot instance market. In Pro. of ACM HPDC, 2015.

[20] M. Hao, H. Li, M. H. Tong, C. Pakha, R. O. Suminto, C. A. Stuardo, A. A. Chien, and H. S. Gunawi. Mittos: Supporting millisecond tail tolerance with fast rejecting slo-aware os interface. In Proc. of ACM SOSP, 2017.

[21] A. Harlap, A. Tumanov, A. Chung, G. R. Ganger, and P. B. Gibbons. Proteus: agile ml elasticity through tiered reliability in dynamic resource markets. In Proc. ACM Eurosys, 2017.

[22] A. Harlap, A. Chung, A. Tumanov, G. R. Ganger, and P. B. Gibbons. Tributary: spot-dancing for elastic services with latency slos. In Proc. USENIX ATC, 2018.

[23] 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.

[24] 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.

[25] 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.

[26] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, Í. Goiri, S. Krishnan, J. Kulkarni, et al. Morpheus: Towards automated slos for enterprise clusters. In Proc. of USENIX OSDI, 2016.

[27] 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.

[28] N. Kremer-Herman, B. Tovar, and D. Thain. A lightweight model for right-sizing master-worker applications. In Proc. of IEEE SC, 2018.

[29] F. Liu and J. B. Weissman. Elastic job bundling: An adaptive resource request strategy for large-scale parallel applications. In Proc. of IEEE SC, 2015.

[30] 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.

[31] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In Proc. of ACM SOSP, 2013.

[32] 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.

[33] J. Rasley, K. Karanasos, S. Kandula, R. Fonseca, M. Vojnovic, and S. Rao. Efficient queue management for cluster scheduling. In Proc. of ACM Eurosys, 2016.

[34] 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.

[35] P. Sharma, T. Guo, X. He, D. Irwin, and P. Shenoy. Flint: batch-interactive data-intensive processing on transient servers. In Proc. of ACM Eurosys. ACM, 2016.

[36] 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.

[37] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with borg. In Proc. of ACM Eurosys, 2015.

[38] L. Xu, M. Li, L. Zhang, A. R. Butt, Y. Wang, and Z. Z. Hu. Memtune: Dynamic memory management for in-memory data analytic platforms. In Proc. of IEEE IPDPS, 2016.

[39] T. Yang, T. Liu, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Redline: First class support for interactivity in commodity operating systems. In Proc. of USENIX OSDI, 2008.

[40] Y. Yang, G.-W. Kim, W. W. Song, Y. Lee, A. Chung, Z. Qian, B. Cho, and B.-G. Chun. Pado: A data processing engine for harnessing transient resources in datacenters. In Proc. of ACM Eurosys, 2017.

[41] M. Zaharia. Job scheduling with the fair and capacity schedulers. Proc. of Hadoop Summit, 2009.

[42] H. Zhang, B. Cho, E. Seyfe, A. Ching, and M. J. Freedman. Riffle: optimized shuffle service for large-scale data analytics. In Proc. of ACM EuroSys, 2018.

[43] Y. Zhang, G. Prekas, G. M. Fumarola, M. Fontoura, Í. Goiri, and R. Bianchini. History-based harvesting of spare cycles and storage in large-scale datacenters. In Proc. of USENIX OSDI, 2016.

[44] Y. Zhong, X. Shen, and C. Ding. Program locality analysis using reuse distance. ACM Transactions on Programming Languages and Systems (TOPLAS), 31(6): 20, 2009.

[45] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In Proc. ACM ASPLOS, 2004.