OS-Assisted Task Preemption for Hadoop

Mario Pastorelli, Matteo Dell'Amico, Pietro Michiardi
EURECOM, France

***Abstract*—This work introduces a new task preemption primitive for Hadoop, that allows tasks to be suspended and resumed exploiting existing memory management mechanisms readily available in modern operating systems. Our technique fills the gap that exists between the two extremes cases of killing tasks (which waste work) or waiting for their completion (which introduces latency): experimental results indicate superior performance and very small overheads when compared to existing alternatives.**

I. Introduction

Data-intensive scalable computing (DISC) frameworks, such as Hadoop [1] and Spark [2], have received great attention by both industry and academia, as they allow to design and execute scalable algorithms to process large amounts of data, with the ultimate goal of better understanding users, business processes, and services, in a variety of application domains.

In many situations, organizations resort to *separate clusters* to make sure that data exploration and algorithm tuning jobs do not interfere with well-tested, production ones. Indeed, it is common to differentiate jobs that are essential for the business of an organization that runs them, from other jobs that can be seen as "best-effort": the latter category should not use resources that high-priority jobs would need.

Clearly, physical partitioning of jobs and clusters entails high overheads in terms of system administration, and an inefficient use of cluster resources. A more flexible and efficient solution is to consolidate clusters, and (manually) define job priorities to inform resource allocation. Unfortunately, while systems such as Hadoop provide ways of setting priorities for jobs, such techniques are imperfect since the only mechanisms available to implement them impose a choice between waiting for low-priority tasks[1] to finish before resources can be granted to high-priority ones, or killing such tasks, thereby wasting the work they have done so far.

The endeavor of this work is to provide a new, *transparent* solution, that allows preempting running tasks, in order to *run high-priority tasks with low latencies* while *avoiding wasting work*. As we discuss in Section II, priorities that are manually set by developers are not the only use case that benefits from an efficient preemption primitive: preemption is important for size-based and deadline-based scheduling, and for enforcing fairness in resource allocation. Still in Section II, we discuss preemption primitives that are available in Hadoop, with their merits and their shortcomings.

Our solution, elucidated in Section III, uses operating system (OS) mechanisms to suspend and resume tasks, which – in current DISC frameworks – are standard UNIX processes. Coherently with the implementation of Hadoop, which uses standard POSIX signals to communicate with processes, we perform suspension and resuming with, respectively, the `SIGTSTP` and `SIGCONT` signals.

With our approach, the state of tasks is implicitly saved by the operating system, and kept in memory. If not enough physical memory is available for running tasks at any moment, the OS paging mechanisms saves the memory allocated to the suspended tasks in the swap area. This step avoids overheads due to systematic serialization and deserialization, and is generally rare in systems with abundant memory.[2]

As the experimental results of Section IV show, our preemption primitive outperforms current approaches in both our performance goals: providing low latencies to high-priority tasks and avoiding redundant work. Even when the available memory is limited, the overhead due to paging is very small.

We further note that our preemption primitive has implications on both implementing Hadoop schedulers and writing MapReduce programs. Hadoop schedulers have a better way to perform task preemption, but they should decide *which* tasks to evict; those who are writing MapReduce programs should consider optimizing them in order to minimize the amount of allocated memory. These issues are discussed in Section V.

In Section VI, we conclude and discuss further research.

II. Related Work

Preemption is an important concept in scheduling in general, and in addition to the manual priority settings we described in the Introduction, there are several use cases in a system such as Hadoop that can benefit from such a primitive. Job schedulers, like the Hadoop FAIR and Capacity schedulers, can use preemption to warrant fairness [4]: if a job starves due to long-running tasks of another job, these latter may be preempted. In deadline scheduling [5], preemption can be used to make sure that jobs that are close to the deadline are run as soon as possible. Size-based schedulers [6], [7] in general attribute priorities to jobs according to a virtual or real size, and preemption can guarantee that higher-priority jobs are allowed to run earlier.

Currently, two preemption strategies are available for Hadoop. One technique is to wait for tasks that should be preempted to complete: this is done using the `wait` strategy. Another approach is to kill tasks, using the `kill` primitive. Clearly, the first policy has the shortcoming of introducing

[1] In Hadoop, and in DISC systems in general, a *task* is a unit of processing work which is performed on a single machine. A typical Hadoop task can last tens of seconds or minutes.

[2] Ananthanarayanan *et al.* report that "the median and 95^{th} percentile memory utilizations [in Facebook clusters] are 19% and 42%, respectively." [3].

large latencies for high-priority tasks, while the second one wastes work done by killed tasks. We refer to the work by Cheng *et al.* [8] for an approach that strives to mitigate the impact of the `kill` strategy by adopting an appropriate eviction policy (*i.e.*, choosing which tasks to kill). In Section IV, we compare our new preemption primitive to `wait` and `kill`.

A recent preemption mechanism for Hadoop is Natjam [9]: unlike in our work, where we use the OS to perform process suspension and resuming, Natjam operates at the "application layer", and saves counters about task progress, which allow to resume tasks by fast-forwarding to their previous states. Since the state of the Java Virtual Machine (JVM) is lost, however, Natjam cannot be applied seamlessly to arbitrary tasks: indeed, many MapReduce programming patterns involve keeping track of a state within the task JVM [10]; this problem is exacerbated by the fact that many MapReduce jobs are created by high-level languages such as Apache Pig [11] or Apache Hive [12]: jobs compiled by these frameworks are highly likely to make use of these "tricks", which hinders the application of Natjam.

Natjam proposes to handle such stateful tasks with hooks that systematically serialize and deserialize task state. Besides requiring manual intervention to support suspension, this approach has the drawback of always requiring the overhead for serialization, writing to disk, and deserialization of a state that could be large. In contrast, our approach does not incur in a systematic serialization overhead, since it relies on OS paging to swap to disk the state of the tasks, *if* and *when* needed.

III. OS-assisted Task Preemption

We now describe our preemption primitive, that implements task suspension and resume operations. First, we outline how process suspension and memory paging work in modern operating systems. Then, we present the implementation of our preemption mechanism. Note that this work focuses solely on preemption primitives, and glosses over *task eviction policies* that are within the scope of a job and task scheduler.

A. Suspension and Paging in the OS

Here we provide a synthetic description of the way OSes perform memory management, which motivate our design and implementation. A more in-depth description of such mechanisms can be found, for example, in the work of Arpaci-Dusseau [13, Chapters 20 and 21].

In general, system RAM is occupied by file-system (disk) cache and runtime memory allocated by processes (including map/reduce tasks); when RAM is full – for whatever reason – the OS needs to *evict* pages from memory, either by reclaiming space (and evict pages) from the file-system cache or by *paging out* runtime memory to the swap area. Since Hadoop workloads involve large sequential reads from disks, it is a best practice to configure the Linux kernel to give precedence to runtime memory, always evicting file-system cache first [14]. The system therefore only pages out runtime memory to avoid "out of memory" conditions, *i.e.* when the memory allocated by *running* processes exceeds the physical RAM.

To decide which pages to swap to disk, OSes generally employ a policy which is a variant of least-recently-used (LRU) [15]; *clean* pages – *i.e.*, those that have not been modified since the last time they have been read from disk – do not need to be written and get prioritized when performing eviction. Page-out operations are generally clustered to improve disk throughput (and amortize on seek costs) by writing multiple pages to disk in a batch. These implementation policies ensure that paging is efficient and with small overheads, especially if a suspended processes leads to swapping. Most importantly for our case, pages from suspended processes are evicted before those from running ones.

We recall that it is necessary to make sure that the aggregate memory size for all processes – both running and suspended – does not exceed the size of the swap space on disk, because in such a case the operating system would be forced to kill processes. Since Hadoop tasks can only allocate a limited amount of memory, this can be ensured by configuring the scheduler so that also the number of suspended tasks per task-tracker is limited.

Thrashing. Paging, in general, is not problematic unless *thrashing* happens, a phenomenon where data is continuously read from and written to swap space [16] on disk. Thrashing is caused by a *working set* – *i.e.*, the set of pages accessed by running programs – which is larger than main memory.

In Hadoop, thrashing is avoided because two mechanisms are in place: *i)* the number of running tasks per machine is limited (and controlled via a configuration parameter); and *ii)* the heap space size that a given task can allocate is limited (and also controlled via configuration). Proper Hadoop configuration can thus limit working set size and avoid thrashing.

The aforementioned mechanisms prevent thrashing in the same way even when suspension is used. Memory allocated by suspended processes is *outside the working set* and hence *cannot cause thrashing*; pages allocated for the suspended processes are paged out and in *at most once*, respectively after suspension and resuming. Thrashing could only happen if a given job is continuously suspended and resumed by the scheduling mechanism: the moderate cost of a suspend-resume cycle can be thus multiplied by the number of cycles. A reasonable scheduler implementation should take into account that suspending and resuming a job has a cost, and should take measures to avoid paying it too often.

B. Implementation Details

The concepts that we illustrate here are valid for both Hadoop 1 [1], which is the most widely used Hadoop implementation in production, Hadoop 2, which uses a new infrastructure for resource negotiation called YARN [17], and even other frameworks such as Spark [2]. Currently, our implementation targets Hadoop version 1.

Our preemption primitive exposes an API that can be used both by users on the command line and by schedulers. Mirroring the implementation of the `kill` primitive in Hadoop, we introduce *i)* new messages between the JobTracker (a centralized machine responsible for keeping track of system state and scheduling) and TaskTrackers (machines responsible for running Map/Reduce tasks), and *ii)* new identifiers for task states in the JobTracker.

JobTracker. Hadoop has a "heartbeat" mechanism where, at fixed intervals and every time a task finishes, TaskTrackers inform the JobTracker about their state.

As soon as the JobTracker receives the command to suspend a task from the user or the scheduler, that task is marked as being in a `MUST_SUSPEND` state. At the following heartbeat from the involved TaskTracker, the JobTracker piggybacks the command to suspend the task. The following heartbeat notifies the JobTracker whether the task has been suspended – which triggers entering the `SUSPENDED` state in the JobTracker – or whether it completed in the meanwhile.

Analogous steps are taken to resume tasks, exchanging appropriate messages and handling the `MUST_RESUME` state, returning the state to `RUNNING` when the process is over.

TaskTracker. In Hadoop, Map and Reduce tasks are regular Unix processes running in child JVMs spawned by the TaskTracker. This means that they can safely be handled with the POSIX signaling infrastructure. In particular, to suspend and resume tasks, our preemption primitive uses the standard POSIX `SIGTSTP` and `SIGCONT` signals.

These signals are used because (unlike `SIGSTOP`) they allow handlers to be written to manage external state, *e.g.*, when closing and reopening network connections.

Job and Task Scheduler. We factor out the role of task eviction policies implemented by the scheduler, which are not the focus of this work, by building a new scheduling component for Hadoop – a dummy scheduler – which dictates task eviction according to static configuration files. This allows to specify, using a series of simple triggers, which jobs/tasks are run in the cluster and which are preempted. In addition to executing jobs and preempting tasks with our `suspend/resume` primitives, the dummy scheduler also allows using the `kill` primitive and to `wait`, for the purpose of a comparative analysis.

IV. Experimental Evaluation

In our experiments, we evaluate preemption primitives in terms of the latency they introduce and the amount of redundant work they require. We show that our approach outperforms other preemption primitives and has a small overhead both when jobs are lightweight in terms of memory, and when they are memory-hungry.

A. Experimental Setup

Our `suspend/resume` primitives operate at the task level, and behave in the same way for both Map and Reduce tasks. We evaluate the behavior of the system in a simple setup: our dummy scheduler runs two single-task, map-only jobs, called t_h and t_l (h and l stand for high and low priority respectively). t_l processes a single-block file stored on HDFS, with size 512 MB; t_h processes single HDFS input block of size 512 MB. Both jobs run synthetic mappers, which read and parse the randomly generated input. We remark that this setup is analogous to the one used by Cho *et al.*, who evaluated their preemption primitive using similar synthetic jobs created by the SWIM workload generator [18].

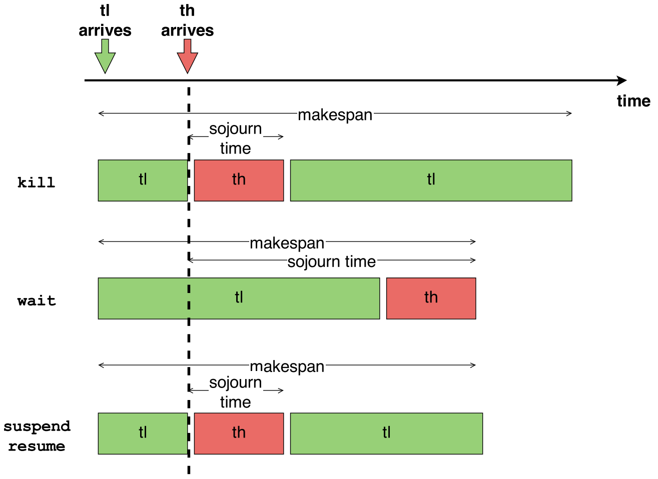


Figure 1: Task execution schedules.

In our experiments, our dummy scheduler preempts the low-priority task t_l after it has reached a completion rate $r\%$ (*i.e.*, $r\%$ of the input tuples have been processed) and grants the task slot to the high priority task t_h. Once t_h is completed, the scheduler resumes t_l, which can complete as well.

Next, we evaluate the behavior of our `suspend/resume` preemption mechanism against the two baseline primitives available in Hadoop: `wait` and `kill`. When waiting, task t_h is simply executed after t_l completes; when killing, task t_l is killed as soon as t_h is scheduled, and t_l is rescheduled from scratch after the completion of t_h. This simple experimental setup is illustrated in Figure 1.

According to Hadoop configuration best practices, in our experimental setup we prioritize runtime memory over disk cache and therefore limit swapping, as discussed in Section III-A, by setting the Linux `swappiness` parameter to 0.

B. Performance Metrics

Our goals are ensuring low latency for high-priority tasks, and avoid wasting work: we quantify them, respectively, with the *sojourn time of* t_h and the *makespan* of the workload. *Sojourn Time of* t_h is the time that elapses between the moment t_h is submitted and when it completes; *makespan* is the time that passes between the moment in which the first task t_l is submitted and when *both* tasks are complete.

C. Results

We focus on experimental results in case of light-weight tasks. This is the standard case for "functional", stateless, mappers and reducers. In this case, the amount of memory that tasks allocate is essentially due to the Hadoop execution engine (*i.e.*, JVM, I/O buffers, overhead due to sorting, *etc.*).

Stateful mappers and reducers, instead, can allocate non-negligible amounts of memory; we thus complement our experiments by studying our performance metrics and overheads for memory-hungry jobs, which represent a worst-case scenario for our preemption primitive.

All our results are obtained by averaging 20 experiment runs; we omit error bars for readability: in all data points reported, minimum and maximum values measured are within 5% of the average values.

Baseline Experiments. Figure 2a on the next page illustrates the sojourn time of t_h: the arrival rate of $_h$ is a parameter defined as a function of t_l progress, as shown on the x-axis.

The `kill` and our `suspend`/`resume` primitives achieve small sojourn times, as opposed to `wait`, in particular when t_h arrives early. However, they both incur in some overheads: `kill` runs a cleanup task to remove temporary outputs of the killed task; `suspend`/`resume` may slow down t_h in case paging out memory occupied by t_l is needed. In our baseline setup, both jobs are light-weight, hence the suspended process resides only in memory. This explains the small advantage for our mechanism, which outperforms all other primitives even when t_h arrives at 90% completion rate of task t_l.

Figure 2b on the following page illustrates our results for the makespan metric, using the same setup described above. In this case, the makespan is heavily affected by a preemption primitive that wastes work. The `wait` policy, at the cost of delaying t_h, avoids supplementary work and achieves a small makespan; the `kill` primitive, instead, wastes all the work done by t_l before preemption. Finally, our preemption primitive behaves similarly to the `wait` policy, despite the possible overhead due to page-out/page-in cycles.

For light-weight jobs, we conclude that our primitive is superior to both alternatives, as both sojourn times and makespan are small. We note that the authors of Natjam measured an overhead of around 7% in terms of makespan, in similar experimental settings as ours. Our findings suggest that the overhead in our case is negligible.

Worst-Case Experiments. The experiments discussed above are valid for simple implementations of Map and Reduce tasks, that carry out stateless computations on their input. Stateful tasks can, however, allocate memory, which may force the OS to swap. Since clusters often have plentiful available memory [3], such a situation is unlikely to be frequent. However, we still consider a "worst case" scenario to stress our primitive: both t_l and t_h allocate a large amount of memory (2 GB in our case; we note that this requires an *ad hoc* change to the Hadoop configuration since Hadoop jobs are not generally allowed to allocate such an amount of memory). This value makes sure that, when running a single task the system does not have to recur to swap;[3] conversely, when the two tasks are present in the system at the same time, one of them is forced to page out memory. We ensure that tasks allocate memory and that the OS marks pages as "dirty", by writing random values to all memory at task startup, and reading them back when finalizing the tasks.

Figures 3a and 3b on the next page present the sojourn time and the makespan for the worst-case experimental setup. While our preemption primitive still outperforms both alternatives with respect to both metrics, it is possible to notice that the overheads related to paging are visible: with respect to the sojourn time, the `kill` primitive achieves a slightly lower value; similarly, the `wait` primitive achieves slightly smaller makespan. Overall, the overhead due to our preemption primitive is marginal: we further investigate and quantify it in the next section.

Impact of Memory Footprint. We now focus on a detailed analysis of the overheads imposed by the OS paging mechanism on the performance of our preemption primitive. To do so, we vary the amount of memory a task allocates in the setup phase.[4] In our experiments t_l allocates 2.5 GB of memory, and we parametrize over the amount of memory t_h allocates. For each experimental run, we measure the number of bytes swapped by the process executing t_l, and compute the degradation of sojourn time and makespan compared to the `kill` and `wait` primitives, respectively.

Figure 4 indicates that the overheads due to paging are roughly linearly correlated to the amount of data swapped to disk. For the sojourn time, our preemption primitive degrades when t_h allocates more than 1.5 GB of RAM: in the worst-case, sojourn time is 20% larger than with the `kill` primitive. Similarly, for the makespan, our mechanism degrades when t_h allocates more than 1.3 GB: in the worst-case, makespan is 12% larger than with the `wait` primitive. Finally, we note that swapped data grows more than linearly because of an approximate implementation of the page replacement algorithm in Linux (and other modern OSes), which can lead to more swapping than strictly necessary [19, Chapter 17].

V. Discussion

We now elaborate more on the implications of the new preemption primitive we introduce in this work.

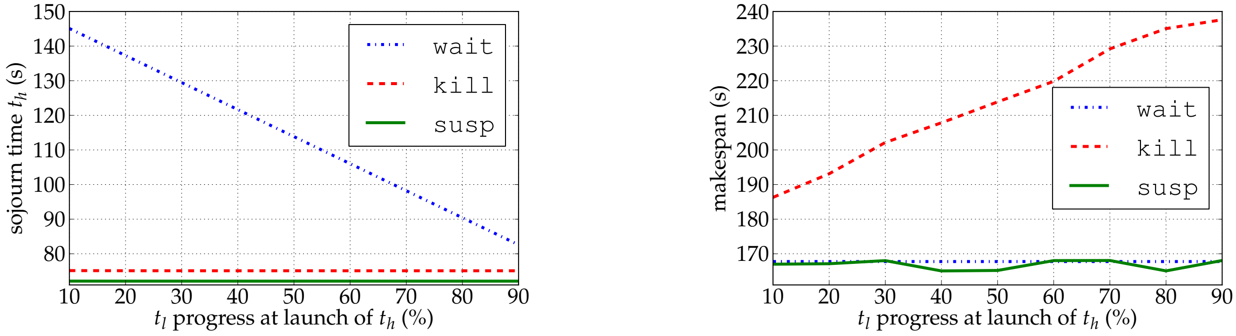
A. Scheduling and Eviction Strategies

As we discussed in the Introduction, our `suspend` primitive gives one more opportunity to the developers of schedulers, in order to perform more efficient preemption. As we have shown with the results of Section IV, our primitive generally performs close to optimally in most cases; however, for freshly started tasks, it may be preferable to use the `kill` primitive, and for tasks that are very close to completion it may be better to simply `wait` for them to finish.

Task Eviction Policies. An important topic that falls under the responsibility of the schedulers is to decide *which* task(s) to evict once a high-priority job needs time to execute. Cho *et al.* [9] propose to suspend tasks that are closest to completion, in order to have all tasks of a job as close to each other as possible, resulting in a good influx on job sojourn times. If the goal is instead to avoid redundant work and reduce makespan, another possible strategy may aim to suspend tasks

[3] The physical memory of our system is 4 GB; the rest of the memory is needed by the Hadoop framework and by the operating system services.

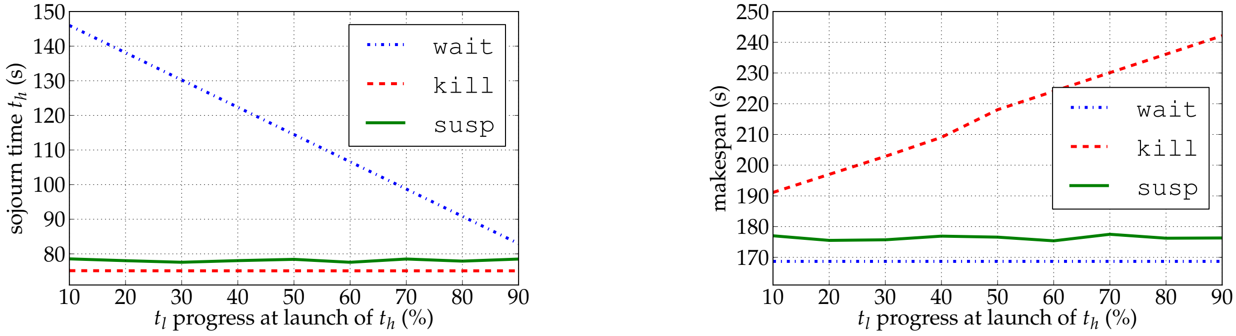
[4] This is where, generally, auxiliary data structures are created to maintain an internal state in a task.



(a) Sojourn time of t_h

(b) Makespan

Figure 2: Baseline experiments: a comparison of the three preemption primitives with light-weight tasks.



(a) Sojourn time of t_h

(b) Makespan

Figure 3: Worst-case experiments: a comparison of the three preemption primitives with memory-hungry tasks.

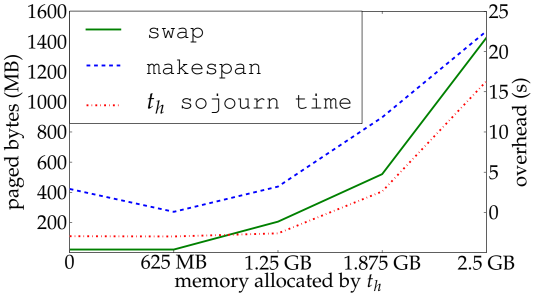


Figure 4: Overheads when varying memory usage.

with smaller memory footprints, which reduces overheads according to our experimental results.

Resume Locality. In our implementation, a suspended process can only be resumed on the same machine it was suspended on. If the same task gets scheduled on a different machine, it has to be restarted from scratch, losing work done so far: in that case, the `suspend` is effectively analogous to a delayed `kill`. We call this issue *resume locality* due to its similarity with the *data locality* issue – *i.e.*, the problem of running mappers on the machines that have local copies of data.

Hadoop schedulers generally handle data locality by using the simple technique of delay scheduling [20]: waiting a fixed amount of time before scheduling non-local copies of data. Only if that threshold is exceeded, a non-local mapper is run. The same technique can be used for our resume locality issue.

Analogously to our approach, Natjam only supports local resumes. As a future improvement, the authors suggest moving the checkpoints used to mark task state and reduce inputs over the network; a similar approach could be taken also in our case, using process migration facilities such as CRIU [21]. However, extreme care should be taken before attempting to use such a non-local resume in particular for reducers, since the cost of moving non-local inputs can be very large.

B. Implications on Task Implementation

In most cases, our `suspend`/`resume` mechanism is transparent towards the implementation, and task implementations that correctly handle error conditions and the possibility of being killed by the scheduler will also handle suspension correctly. However, we add a few notes regarding tasks with external state and ways in which task implementation can control the memory footprint.

External State. In some cases, Hadoop jobs can interact with the external world through more than inputs and outputs: they can use network connections and/or use “Hadoop Streaming”, whereby arbitrary executables can be used as mappers or reducers, interacting with the Hadoop framework through Unix pipes. In these cases, there are interactions that happen outside the control of Hadoop; in the most common case, external software would correctly pause waiting for the next input from a suspended task; however, when the interaction happens with a complex program, the fact that they correctly handle suspended programs should be tested.

Controlling Memory Footprint. We have seen that the memory footprint allocated by a process has an impact on the overheads due to suspension; when writing task implementations, it is good measure to take this into account and optimize for lower memory footprints.

Java garbage collectors differ in the way they are implemented: some of them release memory to the OS when they stop using it, others do not [22]. It is therefore a good idea to configure Java to use a garbage collector that does release memory, such as the new G1 implementation [23]. It is also possible to hint the garbage collector to run using `System.gc()`; this is advisable after disposing of large objects in memory.

VI. Conclusion

In this work we presented a new task preemption primitive that improves over existing techniques to perform both manual and automatic scheduling of Hadoop jobs.

The gist of our preemption primitive was to make use of the memory management mechanisms readily available in the OS to perform task suspension and resuming. Motivated by the limitations of current approaches to task suspension – that implement preemption at the "application level" – we argued that an OS-assisted approach could provide a general preemption mechanism that seamlessly supported a variety of workloads, including stateful tasks.

We implemented our preemption primitive for Hadoop, and discussed how to modify its core components to take into acccount the suspended state of a task, and the signalling mechanisms to trigger task suspension and resume.

Finally, we implemented a simple Hadoop scheduler that allowed us to focus on the goals of our comparative analysis of preemption mechanisms. In our experiments, we glossed over the details and variety of task eviction policies implemented by standard schedulers, and we compared the performance of the `kill`, `wait` and `suspend`/`resume` mechanisms, paying particular attention in quantifying the overheads due to the OS memory management mechanisms. We did so in a variety of experimental settings, including worst-case scenarios of memory-hungry Hadoop jobs. We showed that our technique fills the gap that exists between the two extremes cases of killing tasks (which waste work) or waiting for their completion (which introduces latency): performance is near-optimal, while overhead is small in most cases.

We have preliminary results showing that our preemption primitive performs well in the context of HFSP, our size-based scheduler for Hadoop [24]. Our next steps involve a comprehensive study of task eviction policies implemented in standard Hadoop schedulers that make use of our preemption primitive, a thorough experimental campaign with realistic workloads, and the application of our technique to additional DISC frameworks, such as Apache Spark.

References

[1] Apache, "Hadoop," http://hadoop.apache.org/.

[2] ——, "Spark," http://spark.incubator.apache.org/.

[3] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "Pacman: Coordinated memory caching for parallel jobs," in *USENIX NSDI*, 2012.

[4] M. Zaharia, "Job scheduling with the fair and capacity schedulers," 2009. [Online]. Available: https://trac.nchc.org.tw/grid/raw-attachment/wiki/jazz/09-09-22/FairScheduler_MateiZaharia_Cloudera.pdf

[5] K. Kc and K. Anyanwu, "Scheduling Hadoop jobs to meet deadlines," in *CloudCom*. IEEE, 2010.

[6] J. Wolf, D. Rajan, K. Hildrum, R. Khandekar, V. Kumar, S. Parekh, K.-L. Wu, and A. Balmin, "Flex: A slot allocation scheduling optimizer for mapreduce workloads," in *Middleware 2010*. Springer, 2010.

[7] M. Pastorelli, A. Barbuzzi, D. Carra, M. Dell'Amico, and P. Michiardi, "HFSP: size-based scheduling for Hadoop," in *Big Data*. IEEE, 2013.

[8] L. Cheng, Q. Zhang, and R. Boutaba, "Mitigating the negative impact of preemption on heterogeneous mapreduce workloads," in *Proceedings of the 7th International Conference on Network and Services Management*. International Federation for Information Processing, 2011, pp. 189–197.

[9] B. Cho, M. Rahman, T. Chajed, I. Gupta, C. Abad, N. Roberts, and P. Lin, "Natjam: Design and evaluation of eviction policies for supporting priorities and deadlines in mapreduce clusters," in *SoCC*. ACM, 2013.

[10] J. Lin and C. Dyer, "Data-intensive text processing with MapReduce," *Synthesis Lectures on Human Language Technologies*, 2010.

[11] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *SIGMOD*. ACM, 2008.

[12] 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," *PVLDB*, vol. 2, no. 2, 2009.

[13] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, 2013. [Online]. Available: http://pages.cs.wisc.edu/~remzi/OSTEP/

[14] "CDH 4 installation guide – tips and guidelines," Cloudera. [Online]. Available: http://www.cloudera.com/content/cloudera-content/cloudera-docs//CDH4/4.2.0/CDH4-Installation-Guide/cdh4ig_topic_11_6.html

[15] "Page replacement design," LinuxMM. [Online]. Available: http://linux-mm.org/PageReplacementDesign

[16] P. J. Denning, "Thrashing: Its causes and prevention," in *Fall Joint Computer Conference*. ACM, 1968.

[17] 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 *SoCC*. ACM, 2013.

[18] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, "The case for evaluating mapreduce performance using workload suites," in *MASCOTS*. IEEE, 2011.

[19] D. Bovet and M. Cesati, *Understanding The Linux Kernel*. O'Reilly & Associates Inc, 2005.

[20] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *EuroSys*. ACM, 2010.

[21] "CRIU: Checkpoint/restore in userspace." [Online]. Available: http://criu.org

[22] S. Krause, "JDK 7 GC behavior: To free or not to free," August 2011. [Online]. Available: http://www.stefankrause.net/wp/?p=14

[23] M. Beckwith, "G1: One garbage collector to rule them all," July 2013. [Online]. Available: http://www.infoq.com/articles/G1-One-Garbage-Collector-To-Rule-Them-All

[24] M. Pastorelli, A. Barbuzzi, D. Carra, M. Dell'Amico, and P. Michiardi, "Practical size-based scheduling for MapReduce workloads," *CoRR*, vol. abs/1302.2749, 2013.