True Elasticity in Multi-Tenant Data-Intensive Compute Clusters

Ganesh Ananthanarayanan*
University of California, Berkeley
ganesha@cs.berkeley.edu

Christopher Douglas*
Microsoft Corp.
cdoug@microsoft.com

Raghu Ramakrishnan*
Microsoft Corp.
raghu@microsoft.com

Sriram Rao*
Microsoft Corp.
sriramra@microsoft.com

Ion Stoica
University of California, Berkeley
istoica@cs.berkeley.edu

ABSTRACT

Data-intensive computing (DISC) frameworks scale by partitioning a *job* across a set of fault-tolerant *tasks*, then diffusing those tasks across large clusters. Multi-tenanted clusters must accommodate service-level objectives (SLO) in their resource model, often expressed as a maximum latency for allocating the desired set of resources to every job. When jobs are partitioned into tasks statically, a cluster cannot meet its SLOs while maintaining both high utilization and efficiency. Ideally, we want to give resources to jobs when they are free but would expect to reclaim them instantaneously when new jobs arrive, *without* losing work. DISC frameworks do not support such *elasticity* because interrupting running tasks incurs high overheads. Amoeba enables lightweight elasticity in DISC frameworks by identifying points at which running tasks of over-provisioned jobs can be safely exited, committing their outputs, and spawning new tasks for the remaining work. Effectively, tasks of DISC jobs are now sized dynamically in response to global resource scarcity or abundance. Simulation and deployment of our prototype shows that Amoeba speeds up jobs by 32% without compromising utilization or efficiency.

Categories and Subject Descriptors

D.4.5 [**Reliability**]: Checkpoint/restart

General Terms

Design, Experimentation, Performance

*Work done at Yahoo!

SOCC'12, October 14-17, 2012, San Jose, CA USA

Keywords

elasticity, multi-tenancy, data-intensive computing, checkpoint/restart

1. INTRODUCTION

Data-intensive computation (DISC) frameworks [3, 13, 22] partition jobs into multiple parallel *tasks*. Tasks are assigned a fixed amount of work and are the finest unit of granularity for execution. Each task is assigned to a *compute slot* in the cluster by a resource manager. These clusters are shared by multiple entities in an organization to achieve economies of scale, and the resource manager is responsible for allocation of compute slots among jobs such that SLOs are met, including a stringent and small latency for allocating the desired set of slots to every job.

The static granularity of tasks is insufficient for a resource manager to enforce SLOs while maintaining both high utilization and efficiency across the cluster. Ideally, a resource manager may grant slots to a job while the cluster is idle, but would expect the job to relinquish slots when a new job arrives *without* waiting for its running tasks to complete. We say a job is *elastic* if it can instantaneously relinquish slots without losing work. Elastic jobs help clusters more readily meet its SLOs. Elasticity can be achieved if DISC frameworks either migrate state of active tasks, or suspend tasks and resume them later when slots become available. Unfortunately, DISC frameworks support neither operation due to high and unpredictable overheads (*e.g.,* network load of migration and complex maintenance of suspended tasks' state).

In the absence of elasticity, frameworks balance utilization and efficiency using the following techniques: (1) Cap the number of slots utilized by each job based on others' SLOs, possibly leaving some idle slots fallow. Consequently, some tasks are queued even though slots are idle. This policy is employed in Yahoo!'s clusters with the achievable cluster utilization in practice being $\sim$ 70%. (2) Utilize the entire cluster, but enforce SLOs by *killing* tasks in jobs that are above their guaranteed share. Killed tasks are re-executed leading to wastage of work. Published numbers from Bing's Dryad cluster show that 21% of the tasks are killed [16]. These two approaches exhibit a hard trade-off between utilization and efficiency: with the former, the Yahoo! cluster

achieves 70% utilization with a 100% efficiency, while with the latter, the Bing cluster achieves 100% utilization with 79% efficiency. In either case, the lack of elasticity wastes operational cluster resources.

Given this dilemma, when a cloud operator allocates resources to a DISC job, the decision (and pricing) assumes that the cost of reclaiming that resource for another, more desirable allocation is high. Either the operator builds some fraction of the excess capacity into the pricing or the resources used by killed tasks cannot be billed to the users, inflating the cost of operating the cluster. In contrast, an elastic job can fill excess capacity without creating unbillable waste when that capacity is reclaimed. Also, cloud operators can offer discounts in pricing of spot instances [1] to those users whose jobs are guaranteed to be elastic. DISC frameworks could take advantage of such incentives if they were elastic, and do so without attempting to predict task runtimes.

Amoeba is a system that enables lightweight elasticity in DISC frameworks. It leverages the property that it is possible to judiciously *split* the original work assigned to tasks without altering the result of the overall computation. Amoeba identifies such *safe* points to split a task, which also obviates carrying over of state across the splitting. Using this low-overhead splitting, Amoeba *resolves contentions for slots by safely exiting a running task, committing its output and spawning a new task for its remaining work.* Amoeba achieves this by keeping a memento for task progress via periodic updates. With such elastic resource consumption by jobs, the resource manager can over-allocate slots to jobs without missing its SLOs or wasting computation. The same mechanisms used in Amoeba for scaling down the slots assigned to a job can also scale up: as slots become available (say, due to the cheap "pay-as-you-go" spot instances [1]), a running task can be safely terminated and multiple new tasks can be spawned for the remaining work.

As tasks express more constraints and preferences for their runtime environment, the utility of Amoeba's elasticity increases. These constraints are vital for both performance (*e.g.,* memory locality [14]) and also functionality (*e.g.,* machines with GPUs). Reports from Google show that more than half the tasks in their clusters are constrained [12], in turn leading to the satisfaction of constraints of tasks being included in SLOs. Elasticity will be crucial to meeting SLOs in such a setting without compromising efficiency or utilization.

Trace-driven simulations show that Yahoo!'s Hadoop cluster can use Amoeba's elasticity to improve the average completion time of its jobs by 26% without facing an utilization limit. In Facebook's Hadoop cluster, Amoeba's elasticity can improve average job completion times by 32%, measured against a baseline where tasks are killed to meet SLOs. Based on these encouraging results, we are currently implementing Amoeba inside the Hadoop framework. Measurements of our current prototype show that Amoeba imposes minimal overheads ($\leq 3\%$) and speeds up jobs by 33%.

2. WHY IS ELASTICITY HARD?

We considered many strawman solutions for achieving elasticity in DISC frameworks. However, intrinsic difficulties in applying them preclude their use.

2.1 Controlling Task Sizes

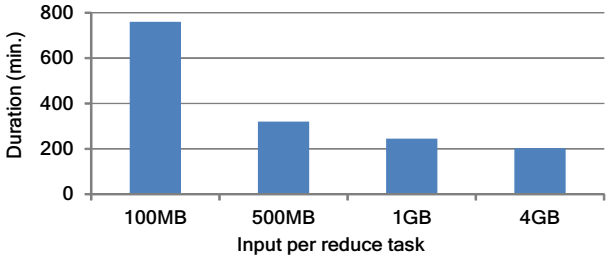


Figure 1: **Seek overheads. Duration of a job with intermediate data of 16TB, as the data per reduce task increases.**

One approach to achieve job elasticity is simply to make all task durations *small* and/or *uniform*. If task durations are small, they vacate slots quickly and SLOs can be met using frequently freed slots. If task durations are uniform, then SLOs can be revised to reflect that constraint; moreover, the uniform wait time for all tasks satisfies some resource managers' constraints on fairness. One may claim that such an approach achieves both high utilization by keeping slots occupied, and also high efficiency by never forcibly reclaiming slots from tasks.

2.1.1 Small Task Durations

Small task durations can be achieved by assigning a small amount of input data per task. Unfortunately, small I/O-bound tasks impose intrinsic overheads, often aggravated in proportion to the number of tasks.

Disk Seeks: One such overhead is the number of disk seeks in reading the input data.[1] It is common for tasks of DISC applications to read from multiple input sources (machines), *e.g.,* reduce and join operations. Many such simultaneous tasks reading small amounts of data can exacerbate the effect of disk seeks on several hosts where its inputs are present. Therefore, taking care that a task reads large and contiguous chunks from *every* input source results in its total input size becoming large.

To understand this effect, consider how a reduce task obtains its input in a MapReduce computation: *every* map task may produce input for *any* reduce task. For large jobs composed of thousands of map tasks, reading even a small amount of data from each map output rapidly inflates a reduce task's input size. The only way to keep the input size of tasks small is by paying in disk seeks. Note that large jobs account for an absolute majority of cluster utilization (and tasks) [14], so this dilemma cannot be side-stepped by resorting to a probabilistic argument that rounds them out of consideration.

We confirm the reality of this problem using a simple experiment. We run a sample Hadoop job with 16TB of intermediate data between the map and reduce phases. As the input per reduce task decreases from 4GB to 100MB, the job's duration increases by nearly 4× (Figure 1). Large tasks are necessary to offset disk seek overheads, but they are unsatisfactory for meeting SLOs.

Per-task Overheads: Notwithstanding disk seeks, small tasks inflate per-task overheads. Every task incurs overheads of scheduling (at the scheduler and on the machine

[1]SSDs, despite improvement in both price and performance, are unlikely to replace disks as the primary storage medium [5, 15].

they run on) and startup. The centralized scheduler stores state per task for management and monitoring purposes. This includes slots allocated to jobs and performance monitors for outlier mitigation [17, 24]. Recent studies have shown that these per-task overheads outweigh the benefits due to small tasks [27]. Finally, Zhou et al. show that common techniques to make reduce tasks small increases the size of intermediate data transfers [18].

2.1.2 Uniform Task Durations

As small tasks are inefficient, perhaps one may make all tasks uniform in size. Such uniform sizing ensures that even if slots are not vacated quickly, every task waits the same amount of time for a slot to be vacated, hence making it fair. As before, adjusting input sizes is an intuitive way to equalize task durations. Since we do not know about the jobs beforehand, we check if the relation between task durations and their input sizes is uniform across all Hadoop jobs in Facebook's production cluster.

We define the *progress rate* of a task to be its input size divided by its duration. Let the progress rate of a job be the median rate across all its tasks. Now we calculate the coefficient-of-variation $\left(\frac{stdev}{mean}\right)$ of progress rates across all the jobs. A low value implies that task durations are easy to predict given input sizes. Unfortunately, the coefficient-of-variation of progress rates is 2.02 and 2.35 across all map and reduce phases, respectively, indicating large difference in execution rates.

While investigating techniques to predict progress rates is beyond the scope of this paper, we do emphasize that it is challenging in the presence of single-waved jobs [14] and skewed computations [17, 30]. In light of the observations in §2.1.1 and §2.1.2, we conclude that merely depending on right sizing of tasks will not ensure high utilization, efficiency, and SLO compliance.

2.2 OS Checkpoint Mechanisms

An alternate approach to achieving elasticity is through traditional checkpointing mechanisms available in operating systems like suspending execution of running processes (tasks) and storing their state for resumption at a later time. However, to relinquish slots to other tasks, it is not sufficient to store the state of the suspended task, which often have heap sizes of multiple GBs, in memory. Doing so makes the task's memory unavailable for other tasks. The state of the suspended task needs to be paged to disk. Practitioners have well documented that the performance of machines along with their shared services (*e.g.,* DFS slave processes) are heavily degraded in the presence of such paging of multiple GBs of data, eventually triggering frequent machine reboots [21, 9].

Further, to make use of slots available on other machines, the state of the suspended task needs to be transferred across the network. Such a transfer is both complicated as well as resource-intensive. The cost of transferring the full image of a task may trigger multiple GBs of additional network traffic, significantly slowing down network flows of other jobs. For these reasons, DISC frameworks do not support suspension and transfer of task execution state of running tasks.

3. Amoeba FOR ELASTICITY

Amoeba's enables a job to dynamically adapt its resource utilization based on the available cluster resources, without wasting computation. Our approach is to use a checkpoint/restart mechanism to dynamically change the number of slots assigned to a job. That is, (1) assign available slots to jobs , and (2) when there is contention, reclaim slots from jobs by checkpointing their work.

Enabling such a checkpoint/restart mechanism requires addressing two challenges. First, tasks must be checkpointed only at those points where it is possible to complete their remaining work *without* carrying state from the current execution. Otherwise, the overhead of transferring arbitrarily complex state (§2.2) will be significant and unpredictable. Second, identifying such points even for tasks whose input is consumed from different machines, necessitates a global view. We address these challenges in §3.1 and §3.2, and then describe their incorporation within DISC frameworks (§3.3 and §3.4).

3.1 Safe Points of Interference

Amoeba terminates a task's execution at *safe* points and then creates a new task to complete its remaining work. A safe point for a task is any point in the task from where the remaining work can be executed without requiring any context from the current execution.

Tasks in DISC frameworks perform computations on non-overlapping subsets of their input. The subsets are processed independently in sequence. A subset is either a single record or a set of records, and data in the subset share a *key.* The MapReduce programming model [13] prohibits storing state across keys. Therefore, a natural safe point for many DISC operators is key boundaries, *i.e.,* when all the processing for a key is complete.

We explain this in the context of some common DISC operators. Map operators take an input list of records and output an intermediate key-value set of records. Because every sublist is also an admissible partitioning of the job's input data, a map task can be split after every record. By way of illustration, map tasks are typically composed on block boundaries without consulting the contents of the input *a priori.* Commonly, a map task will initialize its reader with a file path, offset, and length, deferring record alignment to the underlying format at runtime. Running the same job on an identical file written with a different block size will have the same output, but each map will process a different sequence of records. By maintaining a memento that records comparable information about the underlying state of the reader, one can create a checkpoint as if the original partitioning of the input data ended after the last processed record.

Similarly, reduce operators are invoked on a group of records in the intermediate set, all of which share the same key (*e.g.,* group-by).[2] Join operators, again, work on a given key across multiple sets of intermediate data. Hence, these DISC tasks can be split and suspended when they cross key boundaries. Whenever the task execution is resumed, processing begins at the next key.

Based on this observation, we construct the following lightweight, two-step checkpoint/restart mechanism. First, the execution of a task is terminated at a key boundary. Its output is saved and the next key which should be processed is recorded as the task's checkpoint. Second, a new task is created and it starts execution from the next key identified

[2]In practice, input of reduce tasks are ordered by key (either hash or sort order).

in the checkpoint. The new task is spawned whenever a slot that meets the relevant constraints (*e.g.,* locality) becomes available.

To limit overheads, we do not preserve task state across a checkpoint/restart. This is consistent with the MapReduce programming model that does not span state across key boundaries. Existing as well as new frameworks being developed [26] also conform to this model. However, we are currently in the process of extending Amoeba's design to those computational models that preserve some state across keys.

3.2 Global View of Intermediate Data

Tracking safe points, *i.e.,* key boundaries in name, is non-trivial. As mentioned in §3.1, many DISC tasks (*e.g.,* reduce and join) read their input from multiple sources simultaneously, where each may contain records in its target key-space. To checkpoint a task, one needs to restore the reader's state *globally* across all its input sources.

One could leave all records at their origin and restore the reader by remembering its position in all sources. Regrettably, the overhead involved increases in proportion to the number of input sources (map tasks) if restored at the reader; it also increases in proportion to the number of consumers (reduce tasks) if all readers' positions are saved at each input segment's origin.

Instead, we collect outputs from map tasks into a shim layer that generates an index. Reduce tasks access the intermediate data through an API that re-synchronizes the task using the last-processed key. Our shim layer also performs aggregation that improves the efficiency of the transfer into the reduce, ameliorating some of the effects of skew in map output (discussed in §3.4).

3.3 System Description

We now describe the design of our system that realizes the ideas in §3.1 and §3.2. Figure 2 shows the system architecture. DISC frameworks, conceptually, consist of two components: a cluster-wide *resource manager* and an *application manager* per job [8, 11]. The resource manager handles allocation of slots to jobs based on priorities. The application manager petitions the resource manager for slots and schedules tasks for execution.

When a task begins execution, it first obtains its work assignment from the application manager. These correspond to steps (1) to (4) in Figure 2. As task execution progresses, the task periodically reports its progress to the application manager. Map tasks send progress reports periodically after processing a specified number of records. Reduce/join tasks send progress reports at the boundary between keys (using the global view as described in §3.2), and it includes information about the next key that will be processed by the task. The application manager records this information as part of the task's checkpoint. The task's execution is complete either when its assigned work is processed completely or when the application manager requests the task to terminate execution in response to a progress report.

Termination of a task happens in the presence of contention, when resource assignments do not conform to cluster scheduling invariants. Slots are taken from the application manager that is using the most slots over the target allocation. The resource manager tolerates some deviation from its target allocations, to avoid frequent checkpointing

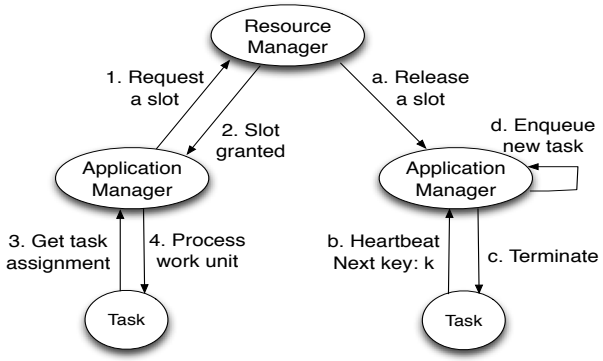


Figure 2: **System architecture for implementing Amoeba's elasticity in multi-tenanted clusters. The centralized resource manager notifies the individual application managers to relinquish compute slots based on cluster demands.**

and to allow for natural vacancies to replace resources, in proportion to target shares of jobs.

To release a slot, an application manager responds to a task's progress report with a termination request. In response, the task commits its output and terminates execution (as in §3.1). The slot is now available to the resource manager for reallocation. In addition, the application manager enqueues a new task for executing the remaining work and petitions the resource manager for another slot allocation. These correspond to steps (a)—(d) in Figure 2 and happens between steps (1) and (2). Subsequently, when the resource manager grants it a slot, the application manager schedules the new task.

A decision for the application manager is picking *which* among the running tasks to checkpoint. This has ramifications on performance because checkpointing a task too often magnifies scheduling overheads. Cluster schedulers typically wait for machines to report free compute slots before assigning them an unscheduled task. To avoid overwhelming the scheduler, machines space their reports in time [4]. While the lag in scheduling tasks is not significant for long-running tasks, they can dominatinate small tasks' runtime (§2.1.1). In the current implementation, Amoeba prefers to checkpoint the longest-running tasks. A detailed analysis of this aspect along with its performance impact is underway.

The same mechanisms described above can be used to repartition the work assigned to running tasks when there is idle or under-serviced capacity in the cluster. This is akin to work-stealing in which the number of slots assigned to a job is grown dynamically.

3.4 Implementation Status

We have developed a prototype that allows checkpoint/restart for reduce tasks.[3] Our implementation is based on the design outlined in Figure 2. For obtaining a global view of intermediate data, we use `Sailfish`, a Hadoop-based MapReduce framework that we have developed recently [28]. In `Sailfish`, output of map tasks are aggregated and augmented with an index on the keys. The application manager uses this index to determine every task's work assignment (*i.e.,* range of keys to process)

[3]The software developed for Amoeba has been open-sourced [7].

in a data-dependent manner. Reduce tasks use the index to efficiently retrieve only their portion of input from the aggregated data (*i.e.,* retrieve data *by key*).

We support checkpoint/restart of reduce task using the following two modifications. First, we modify the reduce task to send periodic progress reports to the application manager (step (b) in Figure 2). In the current implementation, the progress reports are sent at key boundaries but no frequent than 10 seconds apart. Second, we modified the per-job application manager to (1) store the memento for task progress, (2) force tasks to terminate execution by checkpointing (step (c) in Figure 2), and (3) enqueueing new tasks for the remaining work (step (d) in Figure 2). In addition, we also modified the resource manager to force per-job application managers to relinquish slots when there is contention (step (a) in Figure 2). When the new task starts, its execution is like *any* other reduce task: it starts with steps (3) and (4) in Figure 2 and then retrieves its input from the aggregated data.

4. EVALUATION

We present the benefits of Amoeba using a trace-driven simulator. In addition, we also report results from a preliminary deployment of our prototype.

4.1 Trace-driven Simulation

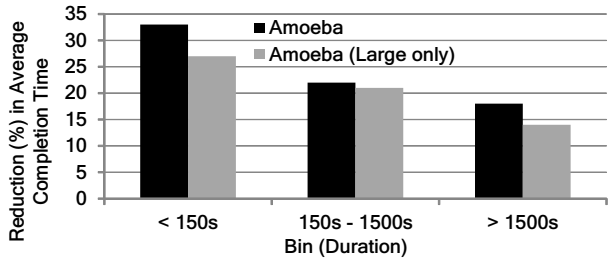
We use a trace-driven simulator to replay one week of Hadoop trace logs corresponding to Hadoop MapReduce jobs from the Yahoo! and Facebook clusters. The baseline for the Yahoo! and Facebook traces corresponds to the policies of sacrificing utilization and efficiency—killing tasks and limiting utilization to 70%— respectively. The overall improvement in average completion time in the two traces are 26% and 32%. Figure 3 shows the improvement across jobs of different sizes.

In the Yahoo! workload, small jobs benefit more from Amoeba's elasticity. This is because they are more affected by queuing delays, which can constitute a significant fraction of their execution time. Queuing delays are amortized for large jobs and that explains their lower gains due to elasticity. However, large jobs account for most tasks in the cluster [14] and hence a correspondingly high number of killed tasks as well. Therefore they benefit the most in the Facebook workload (by 58%). Tasks of small jobs are rarely killed, so their gains are modest.

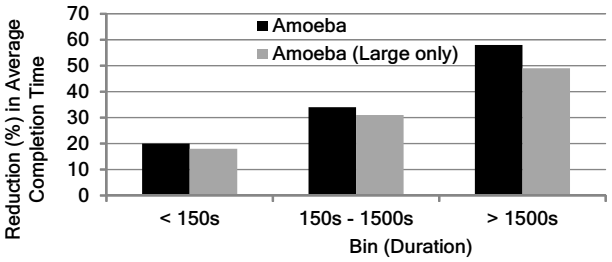
An interesting experiment is to evaluate the gains when the tasks of only the large jobs (> 1500s) are checkpointed. The rationale behind this is similar to the overheads for checkpointing small tasks (§3.3). Tasks of large jobs are likely to be long-running and hence checkpointing them minimizes overheads. Little is lost by not checkpointing tasks of small jobs because, by virtue of the heavy-tailed distribution of job sizes, they account for only a small fraction of the tasks in the cluster [14].

4.2 Preliminary Deployment

We deploy our prototype (§3.4) on a 150-node cluster at Yahoo!. Each machine has two quad-core Intel Xeon E5420 processors, 16GB RAM, 1Gbps network interface, four 750GB drives, and runs RHEL 5.6, and exposes six compute slots. For evaluation, we use a production MapReduce job which is used to build models for behavioral ad-targeting, along with actual datasets used in production at



(a) Yahoo! workload



(b) Facebook workload

Figure 3: **Improvement in completion times with** Amoeba **for the Yahoo! and Facebook Hadoop workloads. Applying** Amoeba **only to tasks of large jobs (>** 1500s**) well-approximates the strategy where we checkpoint any task.**

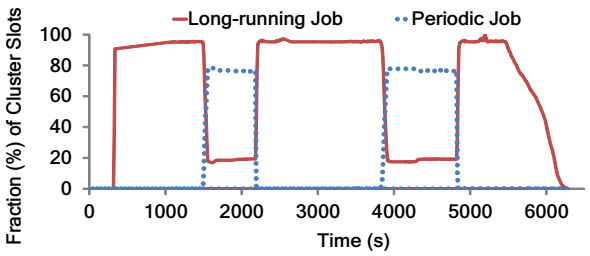
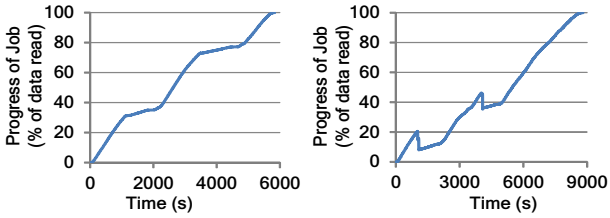


Figure 4: **Elasticity with** Amoeba**. The long-running job checkpoints its running tasks and nearly instantly gives up its slots whenever the periodic-job is submitted.**

Yahoo!. The job involves a join operation over two multi-TB datasets in which the job output is proportional to the input (*i.e.,* about 10TB of data).

We first quantify the overheads involved in checkpoint/restart. In these experiments, the application manager initially assigns 4GB of data per reduce task (based on Figure 1). After N minutes of execution, the application manager forces the task to checkpoint and terminate. A new task is spawned for the remaining work and the same checkpoint/terminate mechanism is recursively used until all the data is processed. Results with N = 3, 5, 10 and 15 minutes show that the overhead of checkpoint/restart on job completion time is 3% to 10% compared to the case where there is no interruption.

We next use Amoeba's lightweight elasticity to resolve contentions between jobs. In this experiment, we have one long-running job. Periodically, a new job is submitted to the cluster. The periodic job is entitled to 80% of the slots in the cluster. As Figure 4 shows, as soon as the periodic job enters, the long-running job checkpoints its running tasks gives up its slots, and the slots are shared in the right proportion.



(a) Amoeba (b) Killing of Tasks

Figure 5: **Job wastes no work when it yields slots. The fraction of work completed monotonically increases, albeit at a reduced slope as a new job arrives.**

When the periodic job leaves the cluster, the long-running job expands to fill the cluster.

We highlight Amoeba's efficiency by contrasting it with an implementation that kills tasks of the long-running job when the periodic job arrives. Figure 5a shows the monotonic increase in progress rate of the long-running job with Amoeba. There is no wasted computation when the job is forced to give up slots, though progress slows as the job gets fewer slots. In contrast, if tasks of the long-running job are killed to accommodate another's SLO, progress not only slows, but it is also lost, as evidenced by the dips in Figure 5b. Consequently, the job takes longer to complete when its tasks are killed, as expected. The completion time increases by 33%, and is close to the gains observed in the simulations.

5. APPLICATIONS OF ELASTICITY

While the focus thus far was on applying elasticity to bridge the trade-off between efficiency and utilization, we intend to explore its applicability in other scenarios.

1. Fragmentation: Clusters are moving towards a model that does not statically define the resources (processor, memory etc.) associated with slots allotted to tasks. This provides flexibility in accommodating the diverse requirements of tasks [8, 10]. A likely scenario in such settings is *fragmentation* of resources, *i.e.,* when the aggregate available memory in the cluster is large but no single machine (or very few machines) contains sufficient memory to schedule a task. When tasks have variable memory requirements, the resource manager would keep track of available memory in every machine and allocate every task to a machine that meets its memory requirement. Analogous to blocks in disks, this leads to memory in the cluster also getting fragmented.

We estimate the likelihood of memory fragmentation by simulating the Facebook trace, which contains memory requirements of tasks, on a cluster of 1,000 machines. For 51% of the time, more than 90% of the machines have less than 4GB of free memory; this is calculated by subtracting the memory being used by tasks from the available memory capacity of the machines. In fact, 37% of time, over 90% of the machines do not have even 2GB of free memory.

Going forward, the variance among tasks in their memory requirements is expected to increase. Coupled with location constraints, fragmentation will only worsen. Amoeba naturally lends itself to "compacting" memory by checkpointing and moving tasks around to create sufficient free memory on individual machines.

2. Data Skew: Tasks in a reduce phase are prone to seeing skews in their input sizes. This is often due to poor partitioning of keys without consider the underlying distribution, reported to be significant in Bing's cluster [17].

Amoeba's elasticity can automatically handle such skews. Tasks can be assigned arbitrary amounts of work initially. If they are deemed to take too long to execute (due to skew), they can be checkpointed/restarted with the appropriately smaller sizes. Consequently, the framework becomes completely self-tuning. A simple version of such tuning was illustrated in §4.2.

3. Speculative Executions: Amoeba can improve the efficiency of speculative executions. Speculative copies are used to mitigate the effect of outlier tasks [17, 24]. The speculative copies are identical to the original task, and hence duplicate the work done thus far by them. This duplication can be avoided using Amoeba's ability to checkpoint partial output of tasks and launch a speculative copy only for the remaining work of outlier tasks.

This leads to more effective outlier mitigation. First, it increases the probability of the speculative copy beating the original. In fact, schemes like Mantri [17] launch a speculative copy only if it is probabilistically guaranteed to beat the original. Second, the savings in resources by not duplicating work leaves more room to speculate on other outliers.

The idea of using checkpoints to increase efficiency of speculative tasks was also mooted by Condie et al. [29] but not designed or implemented.

4. Decentralized Scheduling: Increase in sizes of clusters and reduction in the duration of tasks has led to substantial interest in decentralized scheduling models. What this means is that resources in the cluster will be allocated independently by multiple individual schedulers (as many as one per job), without coordination. This naturally leads to greater likelihood of contention among tasks. For instance, a scheduler might allocate its CPU-intensive task to a machine which is already executing another CPU-intensive task alloted by another scheduler. For both efficient resolution among such contending tasks as well as to avoid overloading machines, it is desirable to be able to checkpoint/restart tasks.

6. RELATED WORK

Prior work has leveraged specific workload characteristics to meet SLOs without losing efficiency or utilization. Scarlett's [16] data replication leverages their skewed popularity to avoid locality-based contentions. Scheduling techniques [25, 20] leverage small and uniform task durations to trade short-term SLO violation for efficiency. However, these workload characteristics are not universal. For example, the Yahoo! cluster has more data popularity skew than the Bing cluster, and the Bing cluster has longer and less uniform task durations than the Facebook cluster. In contrast, Amoeba's solution is generic and independent of workload characteristics.

Cluster schedulers [23, 19] meet locality constraints of tasks using an optimization problem that includes the cost of migrating tasks, approximated as the elapsed duration of the tasks. Amoeba's lightweight elasticity would help accurately estimate this cost.

Finally, cluster providers like Amazon EC2 offer Elastic MapReduce [2] that helps jobs provision resources statically

at startup. In contrast, Amoeba dynamically adapts the resource usage of a job during its execution.

7. ONGOING WORK

We are extending our implementation of Amoeba to support checkpoint/restart for map tasks as well. To make the demonstrated advantages of this prototype more widely available, we will integrate with the new generation of Hadoop infrastructure [8]. Specifically, we will decouple the intermediate data collection from the reduce phase and introduce *collator* tasks into MapReduce workflows [6]. Collator tasks will aggregate the map outputs and gather statistics on intermediate data to effect the lightweight elasticity of Amoeba. Depending on the depth of the integration, collator tasks may themselves be elastic, though one could implement a variant of Amoeba that treats them as an optional optimization step.

Finally, we are also exploring the "scale-up" aspects of Amoeba. That is, to avoid statically sizing tasks, dynamically scale-up the number of tasks in a job based on resource ability. For instance, when slots become available, safely terminate a long-running task and then launch multiple tasks for the remaining work.

Acknowledgments

We thank Facebook and Yahoo! for access to their job traces from their production clusters. For feedback on the draft and insightful discussions on the topic, we thank our anonymous reviewers and Carlo Curino of Microsoft. This research was partially supported by the sponsors of the AMP Lab at Berkeley: SAP, Amazon Web Services, Cloudera, Huawei, IBM, Intel, Microsoft, NEC, NetApp and VMWare, and by DARPA (contract #FA8650-11-C-7136).

8. REFERENCES

[1] Amazon ec2 spot instances. `http://aws.amazon.com/ec2/spot-instances/`.

[2] Amazon elastic mapreduce. `http://aws.amazon.com/elasticmapreduce`.

[3] Hadoop. `http://hadoop.apache.org`.

[4] Hadoop heartbeat. `https://issues.apache.org/jira/browse/HADOOP-5784`.

[5] Implications of storage class memories (scm) on software architectures. `http://bit.ly/i63sgH`. *HPTPS*, 2009.

[6] Preemption and restart of mapreduce tasks. `http://issues.apache.org/jira/browse/MAPREDUCE-4585`.

[7] Sailfish. `http://code.google.com/p/sailfish`.

[8] The Next Generation of Apache Hadoop MapReduce. `http://developer.yahoo.com/blogs/hadoop/posts/2011/02/mapreduce-nextgen/`.

[9] Personal Communication with Yahoo! Datacenter Operators, Sunnyvale. 2012.

[10] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *USENIX NSDI*, 2011.

[11] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *USENIX NSDI*, 2011.

[12] B. Sharma, V. Chudnovsky, J. L. Hellerstein, R. Raifaat, C. R. Das. Modeling and Synthesizing Task Placement Constraints in Google Compute Clusters. In *ACM SOCC*, 2011.

[13] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *USENIX OSDI*, 2004.

[14] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, I. Stoica. PACMan: Coordinated Memory Caching for Parallel Jobs. In *USENIX NSDI*, 2012.

[15] G. Ananthanarayanan, A. Ghodsi, S. Shenker, I. Stoica. Disk Locality Considered Irrelevant. In *USENIX HotOS*, 2011.

[16] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, E. Harris. Scarlett: Coping with Skewed Popularity Content in MapReduce Clusters. In *EuroSys*, 2011.

[17] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, E. Harris, B. Saha. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *USENIX OSDI*, 2010.

[18] J. Zhou, P. Larson, R. Chaiken. Incorporating Partitioning and Parallel Plans into the SCOPE Optimizer. In *ICDE*, 2010.

[19] K. Amiri, D. Petrou, G. R. Ganger, G. A. Gibson. Dynamic Function Placement for Data-intensive Cluster Computing. In *USENIX ATC*, 2000.

[20] L. Cheng, Q. Zhang, R. Boutaba. Mitigating the Negative Impact of Preemption on Heterogeneous MapReduce Workloads. In *CNSM*, 2011.

[21] M. Isard. Autopilot: Automatic Data Center Management. In *Operating Systems Review*, 2007.

[22] M. Isard, M. Budiu, Y. Yu, A. Birrell and D. Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *ACM Eurosys*, 2007.

[23] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, A. Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *ACM SOSP*, 2009.

[24] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *USENIX OSDI*, 2008.

[25] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, I. Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *ACM EuroSys*, 2010.

[26] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica. Spark: Cluster Computing with Working Sets. In *USENIX HotCloud*, 2010.

[27] S. Agarwal, S. Kandula, N. Bruno, M-C. Wu, I. Stoica, J. Zhou. Re-optimizing Data Parallel Computing. In *USENIX NSDI*, 2012.

[28] S. Rao, R. Ramakrishnan, A. Silberstein, M. Ovsiannikov, D. Reeves. Sailfish: A framework for large scale data processing. In *SoCC*, 2012.

[29] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein. MapReduce Online. In *USENIX NSDI*, 2010.

[30] Y. Kwon, M. Balazinska, B. Howe, J. Rolia. A Study of Skew in MapReduce Applications. In *Open Cirrus Summit*, 2011.