Local Resource Shaper for MapReduce

Peng Lu*, Young Choon Lee*, Vincent Gramoli†, Luke M. Leslie* and Albert Y. Zomaya*

*Center for Distributed and High Performance Computing, School of Information Technologies
The University of Sydney, NSW 2006, Australia
pelu1144@it.usyd.edu.au, young.lee@sydney.edu.au, luke.leslie@gmail.com, albert.zomaya@sydney.edu.au
†School of Information Technologies
NICTA and the University of Sydney,
NSW 2006, Australia
vincent.gramoli@sydney.edu.au

Abstract—Resource capacity is often over provisioned to primarily deal with short periods of peak load. Shaping these peaks by shifting them to low utilization periods (valleys) is referred to as “resource consumption shaping”. While originally aimed at the data center level, the resource consumption shaping we consider focuses on local resources, like CPU or I/O as we have identified that individual jobs also incur load peaks and valleys on these resources. In this paper, we present Local Resource Shaper (LRS), which limits fairness in resource sharing between co-located MapReduce tasks. LRS enables Hadoop to maximize resource utilization and minimize resource contention independently of job type. Co-located MapReduce tasks are often prone to resource contention (i.e., load peak) due to similar resource usage patterns particularly with traditional fair resource sharing. In essence, LRS differentiates co-located tasks through active and passive slots that serve as containers for interchangeable map or reduce tasks. LRS lets an active slot consume as much resources as possible, and a passive slot make use of any unused resources. LRS leverages such slot differentiation with its new scheduler, Interleave. Our results show that LRS always outperforms the best static slot configuration with three Hadoop schedulers in terms of both resource utilization and performance.

I. INTRODUCTION

The underutilization of resources remains a major issue in computer systems. The term “resource consumption shaping” was originally coined by James Hamilton [9] to name the idea of smoothing the resource consumption otherwise alternating between peaks and valleys. At internet scale, this alternance is explained by the time-of-day that sweeps around the world, with the load valleys corresponding to periods of day-time in the least populated regions of the globe (such as the Pacific ocean). The key idea behind resource consumption shaping, or resource shaping for short, is to smooth spikes by “knocking off peaks” and “filling valleys” [3]. The fact that resource utilization in data centers is usually lower than 10% [8] promises great potential for resource shaping in reducing the amount of required resources.

In this paper, we tackle the problem of shaping resource consumption at each individual node. We identify peaks and valleys in the utilization of local resources, like CPU or I/O. In response to this observation, we smooth resource consumption by automatically tuning the execution of concurrent tasks to increase performance without over-provisioning. The main challenge is twofold as it consists of characterizing concurrent local tasks and scheduling them appropriately to maximize resource utilization while minimizing resource contention.

Our focus lies on MapReduce applications, where each task processes a chunk of data using the same predefined (map/reduce) function. Processes of a single node are usually fairly treated, in that each receives an identical CPU time slice (quantum), without the explicit consideration of its resource usage pattern. We argue that this fair resource sharing is detrimental to MapReduce applications. In particular, the inherent synchronous nature of map/reduce rounds forces these tasks with similar resource utilization patterns to occur almost simultaneously, thus increasing contention. Typically, I/O-bound tasks incur significant contention at concomitant periods of time when trying to access the same disk, translating into idle CPU time. By filling valleys where one resource is underutilized, one can reduce contention and job duration.

To this end, we develop Local Resource Shaper (LRS) as a novel resource management solution. LRS interlaces the resource usage of multiple workloads to maximize resource utilization with low resource contention. LRS deals with two slot/container priorities: Active and Passive, the latter being able to use resources only when the former is not using them. The rationale behind this two-tier slot differentiation is (1) MapReduce tasks typically consume more than 50% of a CPU resource [7], and (2) the fairness in resource sharing at the task level within the same job is not a major concern.

We illustrate LRS and its new, complementary MapReduce scheduler (Interleave) by implementing them on a Hadoop cluster consisting of 11 Amazon EC2 instances. We demonstrate LRS capability in comparisons with three well-known Hadoop schedulers: FIFO, Fair and Capacity. Experiments have been conducted using six MapReduce benchmarks (Table 1). These benchmarks are specialized in text retrieval, decryption, sorting, scientific computation, etc., and all are taken from the MapReduce literature [10], [11], [4], [14]. Our results indicate that LRS improves these Hadoop-based alternatives in three main ways:

1) **Increasing CPU usage.** LRS improves CPU utilization by about 10 percentage point (reaching up to 89% in comparison with a range between 77% and 81% of those three Hadoop schedulers).
2) **Lowering I/O contention.** LRS with help of Interleave halves the I/O wait time of Hadoop.
3) **Reducing job duration.** LRS always performs better compared with the original design of Hadoop. Reduction in job execution time by an average of 10%, and up to 18%.
The rest of this paper is organized as follows. Section II describes issues in fair resource sharing that motivate our work on shaping local resource consumption. Section III presents LRS and describes its implementation in Hadoop. In Section IV, we evaluate LRS, with and without Interleave, against existing alternatives. We discuss related work in Section V and present the conclusions in Section VI.

II. ON THE PROBLEM OF FAIR RESOURCE SHARING

To illustrate the problem of allowing MapReduce tasks to fairly share resources, we analyzed the resulting resource usage pattern of Hadoop when running those six benchmarks. We use a 4-core node and set each job to have 4 GB of input data (PiEst is configured with 64 map tasks). We only plot results for Grep and WordCount in this section due to their representativeness and space limitation.

The default scheduler in Hadoop uses a FIFO queue to dispatch tasks to slots (or containers in Hadoop 2.x (YARN [13]) terminology). The maximum number of tasks running concurrently is upper-bounded by the number of slots. The resources of each worker are uniformly partitioned into slots (i.e., fair resource sharing), and the number of slots is statically configured before launching the Hadoop system.

Following Yahoo!’s recommendation of choosing the number of slots between half and twice the number of cores [2], we perform experiments using the Hadoop FIFO scheduler with three distinct configurations: 4m4r (4 map slots and 4 reduce slots), 6m6r, and 8m8r. Figure 1 depicts the CPU utilization and execution time of a Grep job running on a 4-core node. As expected, we observe that the idle CPU time decreases as the number of slots increases, resulting in a decrease in execution time. However, Figure 2 illustrates degrading performance when running a WordCount job, which experiences significant I/O activity, in the same settings. This degradation is due to the dramatic increase in wasted CPU time spent waiting for I/O as the number of slots increases. An interesting observation is that the decrease in job duration between 6m6r and 8m8r is most likely due to the higher CPU utilization of 8m8r paying off. However, both 6m6r and 8m8r have higher durations than 4m4r due to their I/O contention.

### TABLE I: A summary of the 6 MapReduce benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Resource</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grep</td>
<td>CPU-bound</td>
<td>Search text matching reg. exp.</td>
</tr>
<tr>
<td>PiEst</td>
<td>CPU-bound</td>
<td>Estimate Pi</td>
</tr>
<tr>
<td>WordCount (WC)</td>
<td>Moderate CPU</td>
<td>Count words</td>
</tr>
<tr>
<td>Crypto (Crpt)</td>
<td>Moderate CPU</td>
<td>Decrypt cipher text</td>
</tr>
<tr>
<td>Sort</td>
<td>I/O-bound</td>
<td>Sort input data</td>
</tr>
<tr>
<td>TeraSort (TS)</td>
<td>I/O-bound</td>
<td>Sort input data</td>
</tr>
</tbody>
</table>

Fig. 1: CPU utilization for Grep with different slot configurations. Execution times (in seconds) shown in parentheses.

Fig. 2: CPU utilization for WordCount with different slot configurations.
Fig. 3: Resource usage of WordCount. Write rate is in bytes.

Note that CPU resource utilization towards the end of Figures 1 and 2 is deteriorating and heavily fluctuating because reduce tasks mostly complete their execution in a short time and only one reduce task is assigned in a scheduling cycle.

To confirm our contention hypothesis, in Figure 3 we report the write rate (i.e., the number of bytes written, or expected to be written, to disk per second as returned by the Linux command `pidstat`) for a short time window. In both configurations, each core executes four tasks (for a total of 16 tasks). The 4m4r slot configuration makes tasks run sequentially, while the 8m8r slot configuration always runs two tasks concurrently. Figure 3(b) indicates that task1 and task2 have a similar CPU usage pattern (they both sort and merge at the same time), resulting in I/O contention (confirmed by high and bulky I/O wait in Figure 2(c)). Although the CPU utilization is increased with 8m8r, I/O contention increases; specifically, CPU I/O wait time accounts for 9.01% compared to 0.11% with 4m4r (despite disk scheduling or network command queuing). This poses the issue of incompatibility between resource utilization and resource contention exacerbated by the fair resource sharing. Note that there are more than one hundred parameters in Hadoop and changing the values of some of them like `io.sort.mb`, `io.file.buffer.size` and `io.sort.record.percent` may affect the performance. As the tuning of Hadoop parameters is out of the scope of this paper, we simply selected the default values for the parameters.

III. THE LOCAL RESOURCE SHAPER

In this section, we present LRS (Figure 4) with its two main components: Splitter and Interleave. Splitter at the core of LRS defines Active/Passive slots to shape resource consumption. Interleave encompasses the slot manager, to adapt the number of passive slots dynamically in order to maximize CPU usage, and the task dispatcher, to dispatch tasks to the appropriate Active and Passive slots.

A. Splitter

A major issue with the current slot configuration is that the best choice is subject to job characteristics, and thus there is no rule of thumb. Moreover, resource utilization is essentially limited by the underlying fair resource sharing strategy even with the “best” slot configuration. To tackle the problem of slot configuration, LRS uses Splitter as a ’pluggable’ resource manager. Splitter pairs up slots in two priority modes: Active slot and Passive slot. A task in an Active slot takes up as much resources as possible to keep its original usage, and a task in a Passive slot makes use of any unused resources while the task in the Active slot is either waiting I/O operations to be completed, or has completed its execution. Active and Passive slots are realized using `cgroups` and their resource sharing ratio (for CPU and I/O resources) is 100:1.

Splitter works with TaskTracker to allocate resources to slots. Before a TaskTracker is launched, Splitter collects the CPU information of the current worker machine using the `lscpu` Linux command to determine the numbers of Active and Passive slots, respectively. In our implementation, we have configured two slots per core and layered them in Active and Passive priority modes. We adopt this two-slot-priority approach as most MapReduce tasks consume more than 50% of available CPU resources [7].

Splitter is triggered by a change in the status of a running task. When receiving new tasks from JobTracker, Splitter follows a FIFO policy to first fill Active slots and then Passive slots. The transition of a task from a Passive slot to an Active slot takes place when a task running in the Active slot finishes. The early-assigned task in Passive is switched to the idle Active slot and that Passive slot is allocated to a new task. This transition takes place repeatedly.

The focus of this paper is on CPU and disk I/O. Other resources, like memory or network bandwidth, are not considered but LRS can easily incorporate previous work, including Capacity scheduler [1], Mantri [4] and Sailfish [12]. The Capacity scheduler enforces a limit on the percentage of memory allocated to a user/job. Delay scheduler delays a task to favor high data-locality and reduce network usage. Mantri and Sailfish avoid network hotspots by decreasing intermediate data transmission.

1 `cgroups` is a Linux kernel feature capable of limiting, accounting and isolating resource usage.
The Interleave MapReduce Scheduler

The Interleave scheduler implements a slot manager (SM) and a task dispatcher (TD) on top of Splitter (Figure 1). Here, we adopt ‘task slot’ that serves as a container for interchangeable map or reduce task. We refer to task slot when we use the term ‘slot’ in the context of Interleave.

Before a TaskTracker starts to work, its corresponding Splitter configures the number of slots as described in Section III-A. SM keeps track of the overall resource usage. Once it detects spare resources (i.e., the CPU is underutilized) in its worker machine, it notifies TaskTracker to increase the maximum number of Passive slots to obtain more tasks from TD in JobTracker. TD dispatches tasks accounting for the existence of dual-purpose task slots.

1) Slot Manager: The slot manager seeks to further increase resource utilization by dynamically configuring (expanding and shrinking) the maximum number of Passive slots. As the resource usage for Active slots is guaranteed and the resource contention between Passive slots is a lesser concern, an increase in the maximum number of Passive slots on a particular worker node helps make use of every spare resource (particularly with I/O intensive jobs). Such an increase has no impact on the resource usage of Active slots as all Passive slots must wait so long as Active slots are using resources.

SM uses 3 seconds as a monitoring cycle, the same interval as the cycle of heartbeat. For each cycle, we calculate the (average) effective CPU utilization (i.e., CPU_{eff} = user mode + system mode) and average I/O wait (IO_{wait}). The actual usage of CPU (CPU_{used}) is then defined as the summation of CPU_{eff} and IO_{wait}. If all slots are occupied but there is some spare resource, SM calculates the number of additional Passive slots as follows:

\[ N = \begin{cases} \left\lfloor \frac{1 - CPU_{used}}{CPU_{used} \times cores / Slot^{MAX}} \right\rfloor & \text{if } CPU_{eff} < 0.9 \land IO_{wait} \leq T \\ 1 & \text{if } IO_{wait} > T \end{cases} \]

where Slot^{MAX} is the maximum number of allocated slots. T is a threshold configured by the user to determine the characteristic of running tasks. The empirical value for T that we have obtained from our experiments is 30%. Note that if this threshold is too high, there is no performance impact on a single node but resource usage spikes may make the slot manager ask for too many tasks, hence potentially raising the issue of stragglers [6], [16].

2) Task Dispatcher: The LRS-aware task dispatcher resides in JobTracker and is triggered by heartbeats sent from TaskTrackers. For each worker, TD dispatches tasks to either Active slots or Passive slots, but not both at any given scheduling event. Tasks of all submitted jobs are organized in a FIFO queue. The dispatcher processes tasks in order and is data locality aware. The dispatcher consists of two phases: reduce task scheduling and map task scheduling.

The behavior of TD is presented in Algorithm 1. The first part is the reduce task scheduling. Since slots in Interleave are able to run either map tasks or reduce tasks, reduce tasks need to be first dispatched in case map tasks of the latest jobs keep occupying all slots and earliest jobs hang due to insufficient slots to run reduce tasks. Only one reduce task is dispatched per heartbeat, as in the original design of Hadoop.

The second part is map task scheduling, which has two stages. Stage 1 assigns tasks to run on Active slots in a FIFO manner. Stage 2 assigns tasks to run on Passive slots but data-local tasks from all submitted jobs take priority in order to improve data locality. Note that we never dispatch map tasks to both Active slots and Passive slots in the same scheduling cycle, which enables tasks to be evenly distributed across all workers when the number of tasks is less than the number of slots in the cluster.

IV. Evaluation

In this section, we evaluate LRS extensively with four different schedulers (three Hadoop built-in schedulers and our own Interleave scheduler), and under six different benchmarks (Table 1). Each of these benchmarks has been previously used to evaluate MapReduce [10], [11], [4], [14].

We performed all our experiments on a Hadoop cluster consisting of 11 EC2 m1.xlarge instances. Each instance has four cores, 15 GB RAM, and is running Hadoop-1.0.0 with a block size of 64MB. The cluster was configured such that one node is dedicated to run JobTracker and NameNode, and each of the remaining 10 nodes hosts a TaskTracker and a DataNode. Based on the empirical rule provided in [2], we varied the slot configuration from 4 map slots and 4 reduce slots (4m4r) to 8 map slots and 8 reduce slots (8m8r) in our experiments. This makes the capacity of our tested cluster equal to 80-160 slots.

Algorithm 1: LRS-aware Task Dispatcher

<table>
<thead>
<tr>
<th>When a heartbeat is received from worker n: /* Reduce task scheduling */</th>
</tr>
</thead>
<tbody>
<tr>
<td>if n has free Active/Passive slots then</td>
</tr>
<tr>
<td>for j in jobs do</td>
</tr>
<tr>
<td>if j has unassigned reduce task t then</td>
</tr>
<tr>
<td>assign t on n</td>
</tr>
<tr>
<td>/* Map task scheduling */</td>
</tr>
<tr>
<td>/* Stage 1: assigning map tasks to Active slots */</td>
</tr>
<tr>
<td>for slot in Active slots do</td>
</tr>
<tr>
<td>if j has unassigned map task t then</td>
</tr>
<tr>
<td>assign t on n</td>
</tr>
<tr>
<td>/* Stage 2: assigning map tasks to Passive slots */</td>
</tr>
<tr>
<td>if no map task is assigned to Active slots in this scheduling cycle then</td>
</tr>
<tr>
<td>for slot in Passive slots do</td>
</tr>
<tr>
<td>if j has unassigned map task t with data on n then</td>
</tr>
<tr>
<td>assign t on n</td>
</tr>
<tr>
<td>for slot in Passive slots do</td>
</tr>
<tr>
<td>if j has unassigned map task t then</td>
</tr>
<tr>
<td>assign t on n</td>
</tr>
</tbody>
</table>
By contrast, LRS achieves high CPU utilization while incurring a low amount of resource contention. In particular, we can see in Figure 5 that the I/O wait duration with LRS remains lower in both experiments than in the motivating Section 1, regardless of the chosen slot configuration.

To better illustrate that CPU utilization valleys may arise from I/O resource contention, Figure 6 depicts the CPU and disk resource utilization of a single core running WordCount (cf. Figure 3 for comparison). By distinguishing between Active and Passive slots, LRS lets the task in the Active slot fully exploit the CPU resource, while the task in the Passive slot keeps waiting until the active task shows usage valleys due to, for example, I/O wait. Once the active task’s CPU consumption decreases as it terminates, LRS switches the oldest passive task to active mode to keep leveraging the CPU resource. This behavior is reproduced cyclically (a third incoming task would become passive until the active task finishes, and so on) and it contributes to fewer context switches compared to fair resource sharing. Local resource shaping is illustrated by the complementary variations in CPU utilization of the 4 tasks in Figure 6(a), as expected, this harmonious shape contrasts significantly with the disharmony present without LRS (Figure 3(b)).

LRS also shapes I/O resource consumption in the same way as CPU utilization. In fact, this I/O resource shaping allows LRS to decrease the portion of CPU time spent waiting for I/O from 9.07% with a 6m6r configuration, to 0.86%. Thus, LRS helps minimize the contention of simultaneous disk writes as depicted in Figure 6(b) which would otherwise significantly limit performance.

To conclude, the combination of low I/O resource contention with increased CPU resource utilization translates directly into performance improvement. We observe that LRS can decrease by 10 times the I/O waiting time, and can achieve 13% higher CPU utilization over a seemingly appropriate slot configuration (6m6r) on the same non-CPU-bound application (WordCount). As a result, LRS outperforms by 12% the execution time of WordCount running with 6m6r (i.e., 435 vs. 496 seconds, see Figure 7).

B. Boosting Performance of Existing Schedulers

In this section, we show that the core component of LRS (Splitter) is complementary to its scheduler. To this end, we incorporate three state-of-the-art Hadoop schedulers into LRS: the FIFO scheduler, the Fair scheduler and the Capacity scheduler. Since these schedulers still use separate map and reduce slots, their incorporation with LRS is realized by configuring 4m4r for Active and 4m4r for Passive. The schedulers were run on our 11-node cluster with the 6 benchmark jobs. We compare job execution time using Splitter to manage resources with 3 optimal configurations based on the number of cores. Results (Figure 8) are normalized based on job execution time with LRS. Even though we did not modify these schedulers, Splitter improves the overall performance by managing resources more effectively. The FIFO scheduler achieves performance improvement of 8% on average for the 6 jobs compared with 3 different configurations. The Fair scheduler and Capacity scheduler achieve, on average, performance improvements of 7% and 5%, respectively.
C. An LRS-Specific Scheduler to Limit I/O Contention

For Crypto and the I/O-bound jobs (Sort and TeraSort), part of the unused CPU resources caused by resource contention still exist when using LRS’s core resource shaping component, Splitter. The Interleave scheduler is used to alleviate this by supplementing LRS with its slot manager and task dispatcher (Figure 9). Interleave further improves resource utilization by 4% on average for effective CPU utilization for Crypto and the I/O-bound jobs (Sort and TeraSort), and further decreases I/O wait by half for Crypto, 23% for Sort and 29% for TeraSort, compared to the case when the FIFO scheduler \(LRS^{FIFO}\) is used. Due to small amounts of unused CPU resources, results for PiEst, Grep and WordCount using the Interleave scheduler are similar to that using LRS without Interleave, and thus are not presented.

In Figure \[\text{9(e)}\] we use Sort with the Interleave scheduler as an example to show the variation in resource usage and the change in the maximum number of slots. In the first 80 seconds, the maximum number of slots is 8 and the number of concurrently running tasks varies. Although unused CPU resources appear around 70 seconds, the maximum number of slots is 8 and the number of slots is still 8 because the number of currently running tasks is less than the maximum number of slots. However, the number of concurrently running tasks reaches 10 and 9 from
D. Improving the Performance of Slot Configurations

In another experiment, we validate LRS with the Interleave scheduler (simply LRS) on our 11-node cluster with the same benchmarks as that of Section IV-A except that we increased the input data to 20 GB and the number of tasks for each job to 320 map tasks and 160 reduce tasks. Additional test cases for multiple job combinations were added to make this experiment more comprehensive. Results are shown in Figures 10 and 11. We compare Interleave against the default FIFO scheduler and observe that the job execution time with the Interleave scheduler (LRS) remains lower than with the default FIFO scheduler (LRSFIFO) with the optimal slot configuration by 9% on average and by up to 17%. We also observe that the effective CPU utilization increases by 11% on average, and by up to a 22% (for the combination of Crypto and WordCount). Finally, I/O wait for moderate CPU jobs and I/O-bound jobs is reduced by a factor of 2 on average and by up to a factor of 5 (for the combination of Sort and WordCount).

As the capability of Splitter to improve resource utilization and performance has been shown in Section IV-A (Figures 5 and 6) and the Interleave scheduler achieves yet more improvement, we only present the performance of Interleave scheduler (LRS) in the following sections.

We observe for all experiments that each configuration is best suited to execute a certain job. For example, in our 11-node cluster, 8m8r is the best configuration for Grep, 6m6r is the best for the combination of Sort and WordCount, and 4m4r is the best for Sort. As workloads change over time in real systems, any one of these static configurations will cause performance degradation. Even if we try to profile a job to get a best configuration before we ran it on a production system, the best configuration still could be wrong. For example, 8m8r is the best for Sort with small input data size on a single node, but it performs the worst with large input data on our cluster. Moreover, job combinations will make the problem more complex. In our experiments, we observed an average slowdown of 9% (up to 22%) caused by different configurations. LRS allows us to overcome this problem.

70 seconds to 120 seconds because unused CPU resources still exist when the number of concurrently running tasks reaches the maximum number of slots. All map tasks finish at 120 seconds and, after that, the number of concurrently running reduce tasks gradually increases as only one reduce task is assigned in a scheduling cycle.
V. RELATED WORK

Resource consumption shaping was proposed as an extension to network-traffic shaping for data center utilization [3, 8]. The underlying idea behind resource consumption shaping is that resource consumption in data centers can be smoothened by deferring non time critical workloads in the peak usage period. Although our work is inspired by this work, our focus is at the finer node level for distributed systems like MapReduce that are concerned more about job-level performance than that of individual tasks.

Hadoop is very popular for large-scale distributed computing particularly to process ever-increasing data volumes, hence managing resources within Hadoop has been a challenge of practical importance. There is a large body of work on resource management [1, 15, 4, 12, 5, 7], especially at the scheduling level. In contrast with these solutions, our Interleave scheduler exploits the fact that LRS trades map/reduce slots off for Active/Passive slots.

The Capacity scheduler [1] supports job memory resource requirement. Jobs are able to be dispatched in a way to reduce memory interference between running tasks. The Delay scheduler [15] takes into account data locality of map tasks. It replaces relatively slow-speed network I/O with local disk I/O to achieve efficient resource utilization for performance improvement. More recently, Mantri [4] and Sailfish [12] achieve performance improvement by decreasing intermediate data transmission between map and reduce tasks to avoid network hotspots. Even automatic solutions [5] that tune Hadoop parameters to improve performance cannot disable fair resource sharing and existing resource allocation techniques, like DRF [7], share various resources but always in a fair manner. We thus believe that these solutions could also benefit from LRS to reduce their job duration by shaping their resource consumption instead of fairly consuming them.

VI. CONCLUSIONS

Local resource consumption shaping aims at leveraging the resources of each node of a distributed system, despite unpredictable workload usage. Our LRS solution maximizes resource utilization and minimizes resource contention by exploiting Active/Passive slots to reduce job duration.

We conducted an extensive analysis of LRS on a cluster of machines using 6 MapReduce benchmarks and evaluating their performance with 3 different state-of-the-art schedulers. We draw a number of interesting conclusions:

- The homogeneous nature of map tasks and reduce tasks make them prone to resource contention. LRS starts improving performance by limiting fairness, whereas fair resource sharing forces homogeneous tasks to acquire similar resources in overlapping periods of time, leading to contention peaks.
- The problem of local resource consumption shaping is orthogonal to the scheduling problem in that simply differentiating Active from Passive slots leads to performance improvements regardless of the scheduler. A scheduler, like ours, can leverage this differentiation to reduce I/O contention substantially.

- Letting tasks run on any slot gives room for optimization: in our experiments, LRS always outperformed the most efficient static slot configuration both in terms of performance and resource utilization. Interestingly, the concomitant development on Hadoop 2 (YARN) [13] seems to confirm our observation as YARN replaces the slot implementation/configuration of Hadoop by the introduction of containers with its creation based on resource capacity. However, YARN still treats resource usage pattern as fixed and it remains limited by fair resource sharing.

Acknowledgments

NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program

REFERENCES