THE UNIVERSITY OF
SYDNEY

APPLICATION PROFILING AND RESOURCE MANAGEMENT FOR MAPREDUCE

A thesis submitted in fulfilment of the requirements for the
degree of Doctor of Philosophy in the School of Information Technologies at
The University of Sydney

Peng Lu

February 2015

Dedication

To my parents, Long Lu and Xiaohua Miao;

To my wife, Xi Wang and newborn daughter, Xitong Lu

Abstract

Application profiling and resource management for MapReduce

Scale of data generated and processed is exponential growth in the Big Data ear. It poses a challenge that is far beyond the goal of a single computing system. Processing such vast amount of data on a single machine is impracticable in term of time or cost. Hence, distributed systems, which can harness very large clusters of commodity computers and processing data within restrictive time deadlines, are imperative. In this thesis, we target two aspects of distributed systems: application profiling and resource management. We study a MapReduce system in detail, which is a programming paradigm for large scale distributed computing, and presents solutions to tackle three key problems.

Firstly, this thesis analyzes the characteristics of jobs running on the MapReduce system to reveal the problem—the Application scope of MapReduce has been extended beyond the original design goal which was large-scale data processing. This problem enables us to present a Workload Characteristic Oriented Scheduler (WCO), which strives for co-locating tasks of possibly different MapReduce jobs with complementing resource usage characteristics.

Secondly, this thesis studies the current job priority mechanism focusing on resource management. In the MapReduce system, job priority only exists at scheduling level. High priority jobs are placed at the front of the scheduling queue and dispatched

first. Resource, however, is fairly shared among jobs running at the same worker node without any consideration for their priorities. In order to resolve this, this thesis presents a non-intrusive slot layering solution, which dynamically allocates resource between running jobs based on their priority and efficiently reduces the execution time of high priority jobs while improves overall throughput.

Last, based on the fact of underutilization of resource at each individual worker node, this thesis propose a new way, Local Resource Shaper (LRS), to smooth resource consumption of each individual job by automatically tuning the execution of concurrent jobs to maximize resource utilization while minimizing resource contention.

Acknowledgements

First of all, I would like to express my deep appreciation to my supervisors Prof. Albert Y. Zomaya and Dr. Young Choon Lee. I would not be here to complete the long journey of my PhD candidature without their incredible guidance, patience, intelligence, and thoughtfulness. I am grateful to them for introducing me to the world of research.

Next, I would like to thank all the academics and researchers in centre for distributed and high performance computing, and administrative staff members in the school of information technologies for their friendliness and kindness. I want to give particular thanks to Assoc Prof. Bing Bing Zhou, Dr. Chen Wang and Dr. Vincent Gramoli for their valuable discussions and instructions. I enjoy meetings with them.

There are also many friends who emotionally and technically supported me, in particular Jun Liang Chen, Zhong Li Dong, Luke M. Leslie, Fei Wang and Xin Yang Li. I thank them for being there for me.

Last but not least, I would like to thank all my family members. As a student studying overseas, it is impossible to go through this degree without their selfless care and warm encouragements. I can never repay the endless love from my parents. Their full support makes me to reach this stage. I am also want to thank my wife. She never gives up on me but make me to be confident when I felt confused.

List of Publications

1. P. Lu, Y. C. Lee and A. Y. Zomaya, Workload Characteristic Oriented Scheduler for MapReduce, in the Proceedings of the 18th International Conference on Parallel and Distributed Systems (ICPADS), December 17-19, 2012.

2. J. Chen, BB.Zhou, C.Wang, P. Lu and A. Y. Zomaya, Throughput Enhancement Through Selective Time Sharing and Dynamic Grouping, in Proceedings of the 27th IEEE International Parallel & Distributed Processing Symposium (IPDPS), May 20-24, 2013.

3. L. Leslie, Y. C. Lee, P. Lu and A. Y. Zomaya, Exploiting Performance and Cost Diversity in the Cloud, in Proceedings of the 6th International Conference on Cloud Computing (CLOUD), 2013.

4. P. Lu, Y. C. Lee and A. Y. Zomaya, Non-Intrusive Slot Layering in Hadoop, in the Proceedings of the International Symposium on Cluster Computing and the Grid (CCGRID), May 14-16, 2013.

5. P. Lu, Y. C. Lee, V. Gramoli, L. Leslie and A. Y. Zomaya, Local Resource Shaper for MapReduce, in the Proceedings of the 6th IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Dec 15-18, 2014.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This chapter presents a high-level overview of this thesis. It first introduces background knowledge. It then provides the motivation why application profiling and resource management in distributed computing systems need to be further improved. At last, the principal research contributions are identified.

1.1 Background

1.1.1 MapReduce Programming Model

As the exponential growth and availability of data, processing "big data" on a single machine is becoming impracticable in term of time or cost. This trend makes high demand for distributed computing system that can harness very large clusters of commodity computers and processing data within restrictive time deadlines. MapReduce [1] is a programming paradigm for large scale distributed computing, which is originally developed by Google for facilitating to process vast amount of data in parallel. It propose a new abstraction that easily expresses distributed computations on massive amounts of data but hides the messy details of parallelization, fault tolerance, data distribution and load balancing in a framework. The execution of a MapReduce job consists of the map function and the reduce function. Input data sets are modeled as

collections of key/value pairs. The map function first processes all key/value pairs one by one and produces a set of intermediate key/value pairs. All intermediate value are grouped by their key and pass to the reduce function. The reduce function will merge these values together based on their key and generate the final result which is still in key/value format. The whole process can be summarized in the following equations:

$$Map(< k1, v1 >) \rightarrow list(< k2, v2 >) \tag{1.1}$$

$$Reduce(< k2, list(v2) >) \rightarrow < k2, v3 > \tag{1.2}$$

MapReduce programming model is inspired by functional languages. Many large-scale data problems can be mapped onto this model to take advantage of distributed computing. As an example that could be implemented as a MapReudce job, consider the problem of counting the number of occurrences of each word in a book. First, we need to find out the input key/value pairs for the map function. It depends on your computing resource. We can use the content of the whole book as a single record to pass to a map function like *Map (<book name, content>)* if computing resource is limited or we can divide the book into smaller blocks like chapters, paragraphs and even lines as a set of records to pass them to many map functions which are executing on a distributed environment and each map function processes the input records like *Map (line number, content of the line)* in parallel. The map function processes the records and output the intermediate results like *list (<word, 1>)*. The MapReduce framework will shuffle and sort the outputs based on their key and produce the input for reduce function like *<word, list (1,...,1)>*. At last, according to the rule you configure, one or many reduce functions will be launched to process the intermediate results and generate the final results like *list (<word, the number of occurrences of the word>)*. Obviously, MapReduce makes programming easier to extract the capacity of distributed computations and accelerate for processing vast amounts of data.

1.1.2 Hadoop

Hadoop [2] is an open source implementation of MapReduce. It has been widely used in production environment of many companies like Facebook, Yahoo!, etc. Based on the theory published in [1], Hadoop implements a framework that provides a clear and simple API to program a MapReduce-sytle job to process vast amounts of data in parallel on large clusters of commodity hardware in a reliable, fault-tolerant manner. It divides a job into a large number of small tasks and then run them in parallel to make the overall job execution time smaller than it would otherwise be if the tasks ran sequentially. A job usually has a map and a reduce phase (the reduce phase could be omitted).

In the map phase, the input data set of a job is divided into a large number of small input splits (default is 64 MB) and distributed on different worker machines. Each input split is assigned to a map task, which runs the user-defined map function to handle each record of the input split. The output of user-defined map function, presented by key/value pairs, is first partitioned based on hashcode of each key (default configuration) and then is sorted according to the keys. Sorting and Partitioning are operated in memory at first until a threshold is reached, part of data in memory is flushed to a separate temporary file stored in local file system. To the end, all temporary files are merged into an output file for the reduce phase. On the other hand, the reduce phase is broken into three sub-phases: *shuffle*, *sort* and *reduce*. Each reduce task starts with *shuffle* sub-phase, fetching a sorted data partition from output of the map phase via HTTP, which distribute on the worker machines where the map tasks are executed. All these received data will be grouped by their keys in *sort* sub-phase. The *sort* and *shuffle* sub-phases could occur simultaneously. When the needed data partition is sorted, *reduce* sub-phase proceeds: the user-define reduce function is executed and final results are written to the distributed file system (HDFS).

A job in Hadoop framework is divided into smaller grain tasks in order to be dispatched across a Hadoop cluster to utilize distributed resource. A Hadoop cluster

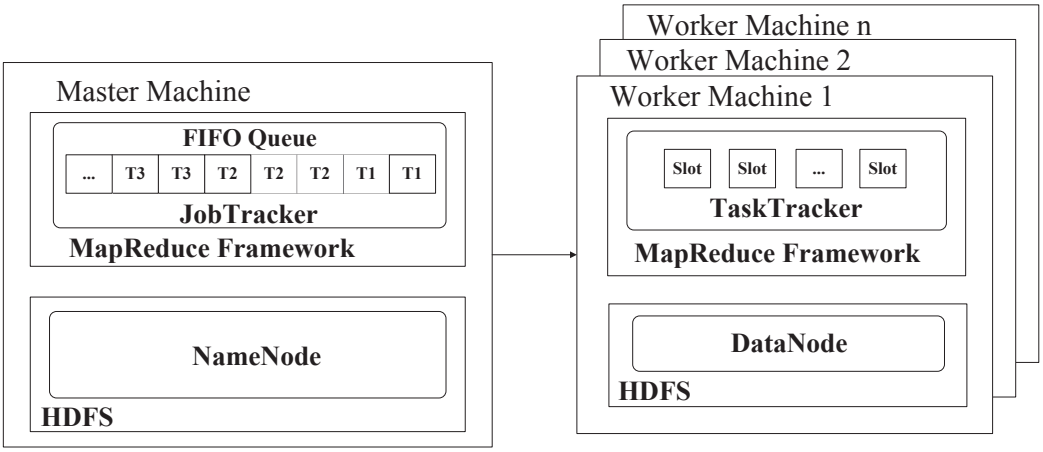


Figure 1.1: The architecture of Hadoop.

complies with the master/slave paradigm as seen in Figure 1.1. A master machine (JobTracker in Hadoop) is responsible for dispatching tasks according to scheduling strategies while a set of worker machines (TaskTracker in Hadoop) are in charge of managing resource and processing tasks assigned by the master. As the capability of the computer has grown rapidly, more tasks can be executed simultaneously in order to maximize resources utilization. Hence, Hadoop uses slots as the finest granularity to manage resources and execution a task. The number of slots across the cluster represents the cluster's capacity and the number of slots per TaskTracker determines the maximum number of concurrent tasks that are allowed to run in the worker machine. Moreover, the number of slots needs to be statically configured before launching TaskTracker and takes effect during the lifetime of the TaskTracker.

1.2 Motivations

As suggested by the thesis title, this PhD work target two aspects of distributed systems: application profiling and resource management.

1.2.1 Application profiling

The characteristics of applications need to be detected in order to allocating suitable resources to execute them. It helps to build a contributed system to achieve efficient resource utilization and improve its performance. Application profiling is an efficient approach which provides feasible and reliable methods to extract and evaluate the characteristics. Moreover, most of distributed systems are originally design for a certain purpose. But due to the complexity of changing a new distributed system that includes deployment, user experience, compatibility, reliability, etc, users tend to stick to a familiar distributed system to meet their diverse needs. This could result in a great decrease in performance when executing applications that are not suitable. Application profiling is also a way to know the need of new applications in order to make the improvement for existing contributing systems. Considering the example of Hadoop, applications in many areas are increasingly developed and ported using Hadoop to exploit parallelism. The Application scope of MapReduce has been extended beyond the original design goal which was large-scale data processing. This extension inherently makes a need for the MapReduce framework to explicitly take into account characteristics of job for two main goals of efficient resource use and performance improvement.

1.2.2 Resource management

Resource management is the core of distributed systems. In this thesis, we make a deep study on current Hadoop framework. Hadoop architecture complies with the master/slave paradigm as described in section 1.1.2. Each worker node uses slots to manage resource for running tasks. Resource is uniformly partitioned into slots in the sense that they fairly share the resource. One slot is able to run one task so the number of slots in a Hadoop cluster worker node specifies the concurrency of task execution. The number of slots needs to be statically configured before launching the Hadoop cluster. This design imposes severe limitations in term of resource management particularly at the worker node level. First the current static slot configuration inaccurately

represents resource sharing with diverse applications. The number of slots in a worker node dictates the maximum number of tasks that are allowed to concurrently execute; and this slot configuration is fixed throughout the lifetime of the TaskTracker residing in the worker node. In theory, the number of slots is configured in the way that the best performance is achieved by maximally using resource among slots. In reality, such static configuration is a bottleneck for efficient resource utilization because it is impossible to find a rule of thumb when dealing with a diverse set of jobs. Many dynamic factors need to be considered like different types of job, different orders of job and even different sizes of input data for the same job. Clearly, there is a need to find a better way to manage resource instead of static slot configuration.

Second, resource is shared fairly regardless of job priority. The current scheduler in Hadoop dispatches tasks in a FIFO manner, and resource is uniformly partitioned based on slots and allocated to both high-priority/early-submitted tasks and low-priority/late-submitted tasks at the worker node level. Such way to manage resource brings extra delay in the execution time of high-priority/early-submitted jobs, and it is also unsuited to execute ad-hoc query jobs expecting fast response time. Clearly, there is a need to find a better way to manage resource in consideration of job priority.

Last, as the capability of the computer has grown rapidly, distributed systems are able to concurrently run more tasks on the same machine. Improvement by efficiently exploiting this local resource is becoming more valuable for the improvement of overall performance. Current operating systems allow multiple jobs (even more than there are CPUs) to run at a time. This is generally done via time-sharing—each job is given an equally short CPU time in turn. Although some costs get involved when switching jobs, this is very useful for these interactive applications, making them have a quick response. It is also a good approach for concurrently running jobs to fairly share resources on a machine. However, such fair resource sharing is detrimental to distribute systems running batch jobs like Hadoop. Tasks from the same job usually have the similar resource consumption pattern. They tend to be executed at the same time across

all resource in order to make the job done as soon as possible. This would make these tasks have more change to use CPU or I/O resource at the same time as the operating system makes all resource is fairly shared among them. Therefore, resource competitions increase and performance decrease. Clearly, there is a need to find a better way to efficiently utilize resource among running tasks instead of fair resource sharing that the operating system makes.

1.3 Contributions

In this thesis, we make the following research contributions towards the understanding and advance of application profiling and resource management in distributed systems:

- We study Hadoop scheduling strategies to effectively deal with different workload characteristics-CPU intensive and I/O intensive. We present a Workload Characteristic Oriented Scheduler (WCO), which strives for co-locating tasks of possibly different MapReduce jobs with complementing resource usage characteristics. WCO is characterized by its essentially dynamic and adaptive scheduling decisions using information obtained from its characteristic estimator. Workload characteristics of tasks are primarily estimated by sampling with the help of some static task selection strategies, e.g., Java bytecode analysis. Results obtained from extensive experiments using 11 benchmarks in a 4-node local cluster and a 51-node Amazon EC2 cluster show 17% performance improvement on average in term of throughput in the situation of co-existing diverse workloads.

- We study the priority mechanism of Hadoop focusing on resource management at work node level and present a non-intrusive slot layering solution. Our solution approach in essence uses two tiers of slot (active and passive) to increase the degree of concurrency with minimal performance interference between them. Tasks in the passive slots continue their execution when tasks in the active slots are not fully using (CPU) resource, and tasks/slots in theses tiers are dynamically

and adaptively managed. To leverage the effectiveness of slot layering, we develop a layering-aware task scheduler. Our non-intrusive slot layering approach is unique in that (1) it is a generic way to manage resource instead of static slot configuration and (2) both overall throughput and high-priority job performance are improved. Our experimental results with 6 representative jobs show 3%-34% improvement in overall throughput and 13%-48% decrease in the executing time of high-priority jobs compared with static configurations.

- We present a new way, Local Resource Shaper (LRS), to allocate resource, which limits fairness in resource sharing between co-located Hadoop tasks. LRS enables Hadoop to maximize resource utilization and minimize resource contention independently of job type. Co-located Hadoop 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.

1.4 Structure of the thesis

The rest of this thesis is organized as follows. Chapter 2 focuses on application profiling. We first describe and discuss the problem due to application diversity based on the analysis on the characteristic of different types of job running on Hadoop. We then present our workload estimation module and the WCO scheduler which co-locates running tasks according to their characteristics to improve the overall performance. Chapter 3 and 4 target to resource management. Chapter 3 resolves the problem of

resource allocation based on priorities of running tasks. We present the design, implementation and validation of the non-intrusive slot layering solution. It uses two tiers of slot to allocate as much resource as needed to high-priority tasks for reducing their execution time and make low-priority tasks to take advantage of the unused resource for improving overall performance. Chapter 4 first gives a deep analysis on resource usage of distributed systems running batch jobs like Hadoop. It reveals the issues in fair resource sharing that could result in resource contention and decrease system performance. Then, LRS, a new way to manage resource for running tasks, is presented and experimental results validate its performance. Finally chapter 5 summarizes the thesis and point out future directions.

Chapter 2

WCO: a scheduler for MapReduce based on application profiling

2.1 Introduction

MapReduce [1] has become increasingly popular not only for traditional large-scale data processing, but also for scientific computing, machine learning, and graph processing [3, 4, 5]. Some popular examples are PageRank, Page Indexing, Chukwa, Hama, Mahout, Hbase, and Hive. This application diversity implies that MapReduce applications running on the same platform may exhibit different characteristics, i.e., I/O intensiveness or CPU intensiveness. Such diversity is largely ignored in the current scheduler in Hadoop—an open source implementation of MapReduce. The scheduler simply uses a single job queue to dispatch jobs in a FCFS (First Come First Serve) manner. There is more probability that tasks of the same workload characteristic (resource usage pattern or resource utilization) are dispatched to the same machine leading to resource contention and in turn reducing throughput. For example, if a node in a Hadoop cluster has two empty map slots, TaskTracker on the node will send a request to scheduler. Once the scheduler receives the request, the first two tasks in the job queue are dispatched to that node. These tasks typically come from the same job. They have most

likely the same workload characteristic since a job is divided into multiple map tasks, and they are placed in the job queue in sequence. In such a case, the throughput would be reduced due to resource sharing/contention. Rather, different types of application should be combined to run on the same node. CPU intensive tasks and I/O intensive tasks are often complementary since a task that has more I/O operations tends to have low CPU utilization.

In the recent past, many notable works on MapReduce scheduling and resource allocation strategies have been reported, e.g., [6, 7, 8]. The primary goal of these works is high performance/throughput either by minimizing data staging overheads, or exploiting resource abundance and/or heterogeneity. The fair scheduler [6] uses a delay strategy to achieve optimal data locality, and provides a policy to allocate resources fairly for multi-user workloads. The scheduler designed in [7] is aware of resource heterogeneity, and authors in [8] consider multiple users. One thing in common in most of these previous efforts, if not all is that MapReduce jobs are simply considered as a single application type (data or I/O intensive).

Accounting for workload characteristics becomes more complex when Hadoop clusters are shared by multiple users with diverse (MapReduce) applications. The scheduler must consider characteristics of running jobs. In this chapter, we examine different methods and techniques to enhance MapReduce scheduling for Hadoop cluster. To this end, we present the Workload Characteristic Oriented (WCO) scheduler for MapReduce applications. The WCO scheduler pays fine attention to application diversity. It improves resource utilization and application performance by co-locating jobs of different workload characteristics, i.e., less resource contention and performance interference. The WCO scheduler incorporates approaches to detect the characteristic of a job and to balance CPU usage and I/O usage of the whole system by combining different types of job. We also present a task selection submodule, which may contribute to additional improvement. Experimental results show that our scheduler is able to increase the system throughput by 17% on average in the situation of co-existing diverse

workloads.

The rest of the chapter is organized as follows. Section 2.2 describes the related work. Section 2.3 describes and discusses the analysis on the characteristic of different types of job. Section 2.4 presents our workload estimation module that plays a major role in our WCO scheduler. Section 2.5 details our scheduler. Section 2.6 presents experimental results that validate the efficacy of our new scheduler. In Section 2.7 we draw the conclusion.

2.2 Related Work

MapReduce applications have been increasingly popular as data volume increases dramatically and large-scale data processing is a core and crucial business activity. The original scheduler in Hadoop essentially uses a single job queue with a FCFS strategy. Specifically, tasks of the same characteristic in terms of resource usage pattern or resource utilization (belonging to a single job) tend to be dispatched to the same machine; this is clearly prone to resource contention, and in turn reducing throughput.

A number of scheduling strategies (e.g., [6, 8, 9]) have also been proposed to enhance the performance of MapReduce with various new features. Authors in [6, 8, 9] focus on 'fairness' for users or resources. Scheduling strategies introduced in [7, 10, 11] are used to exploit the heterogeneity of resource to improve performance. In [7] authors propose a heterogeneity-aware MapReduce scheduling policy that assigns tasks considering different machine types. The Progress Share is introduced as a metric to describe the characteristic of a job running on different types of machine. However, resource contention from co-located tasks of the same type still exists. Our work differs primarily that we explicitly consider different types of MapReduce application while these previous works treat MapReduce applications are of a single type (data/IO intensive) of application.

Since the identification of application characteristics greatly leverages decision making for scheduling in particular several recent studies have addressed application

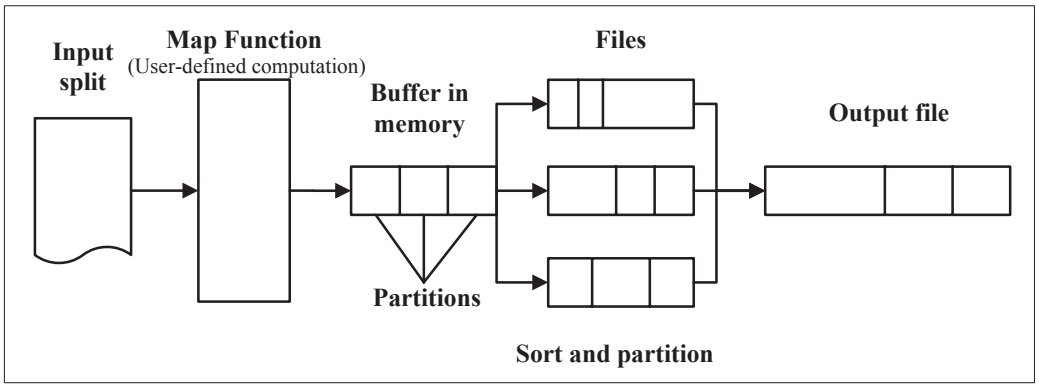
profiling in multi-user environments, e.g., [12, 13]. In [12], authors design a prediction mechanism based on I/O rate to detect the workload type, and implement a system to improve the usage of both CPU and disk I/O resource by combining different types of workload to run on the same machine. But the prediction mechanism cannot provide accurate estimates due to ignoring intermediate process. The work in [13] modeled the correlation between application performance and resource usage patterns using a statistical analysis technique (CCA or cannonical correlation analysis).

Static program/code analysis is yet another well studied technique to identify program logic (resource usage patterns). It is generally divided into high level and low level techniques. High level static analysis focuses on program logic and usually builds control flow graph (CFG) to identify resource demand [14, 15], whereas low level static analysis considers the execution time of instructions and cost of runtime, such as Instruction Level Parallelism (ILP), Memory Level Parallelism (MLP) and branch predictability [16, 17]. Our static analysis strategy incorporated into the WCO scheduler balances between efficacy and complexity (analysis overhead) using CFG and relative computational intensity, and sampling adjustment.

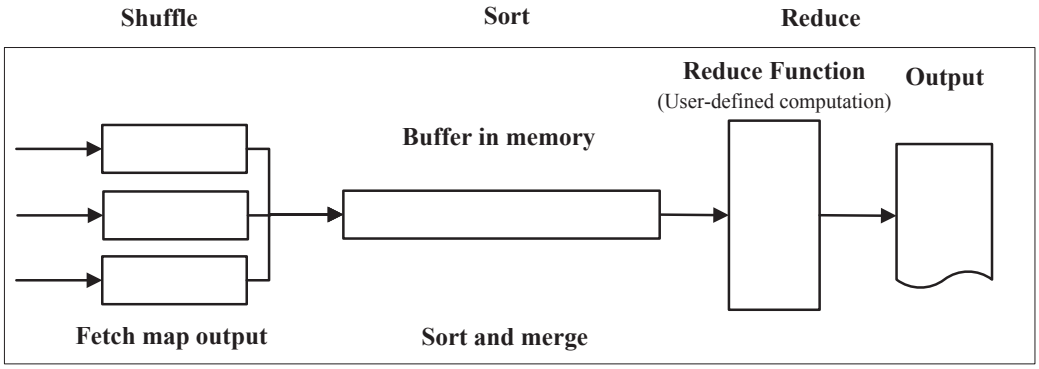
2.3 Characterization of MapReduce Workloads

MapReduce consists of two phases: Map and Reduce (Figure 2.1). In the map phase the input data set is divided into a large number of small *input splits* (default is 64MB), and these *input splits* are processed by map tasks of a user-defined function across compute nodes (Figure 2.1(a)). Intermediate results produced by map tasks are then processed by tasks in the reduce phase for final results (Figure 2.1(b)).

In this chapter, the execution time of a map task is defined as the amount of time taken from reading the input split to outputting intermediate results to local file system; and the execution time of a reduce task is defined as the amount of time taken from fetching the output of map tasks to writing results to HDFS. In the following we decouple the time consumption on I/O operations and CPU operations of a MapReduce



(a) Map phase



(b) Reduce phase

Figure 2.1: MapReduce workflow.

application. The execution time for a map/reduce task is then defined as:

$$TaskExecutionTime = OT + CT + IOT \tag{2.1}$$

where OT is the fixed overhead in running a task, and CT and IOT are times taken in CPU and IO operations, respectively. OT is independent of data size, which mainly includes JVM (Java virtual machine) initialization overhead, and scheduling time. CPU-related operations mostly occur in the user-defined map and reduce function. Broadly, IO operations can be classified into the following: 1. Input and output for a map/reduce task, 2. Reading and writing for sorting data in a map/reduce task, and 3. Shuffle for a reduce task (see Figure 2.1).

CT and IOT are two parts, distinguishing from other types of task, to represent

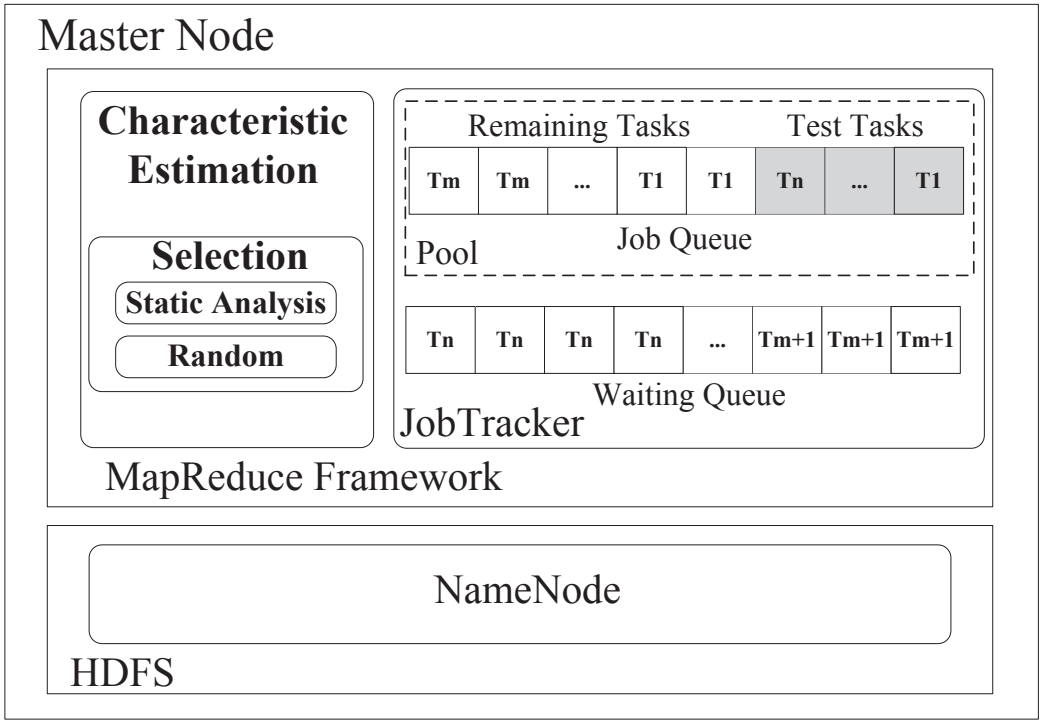


Figure 2.2: WCO scheduler overview. Those two queues (Job Q and Waiting Q) are only conceptually separated to distinguish between running jobs and waiting jobs; however, they both are part of a single priority queue.

the characteristic of a task. The ratio between them are denoted by computing rate (CR) and I/O rate, respectively. The I/O rate of a task is the total amount of input and output of a task divided by task execution time. Since the Hadoop framework uses cache mechanism and temporary files for sorting, the accurate total amount of input and output of a task is difficult to be counted. Thus, in this chapter, we adopt CR to represent the characteristic of a task, which is defined as:

$$CR = \frac{CT}{OT + CT + IOT} = \frac{CT}{TaskExecutionTime} \tag{2.2}$$

If a task's *CR* reaches to 1, the task is regarded CPU intensive, or I/O intensive if *CR* is close to 0.

2.4 Workload Characteristic Estimation

In this section, we present our workload characteristic estimation module incorporated into our WCO scheduler (Figure 2.2).

To estimate the characteristic of a job, values of some variables in Equation 2.2 must be known in advance. One way to obtain these values is to derive from prior executions of the job. However, there are some potential problems: 1. we cannot guarantee that the historical data exist from prior executions of the same job; 2. prior executions of the same job were likely performed over different input data sets and may therefore have completely different characteristics; 3. noisy historical data may exist because of prior execution environments—if prior executions were concurrently run with jobs having the similar characteristics, the result would be different due to resource contention.

Therefore, we design a module as part of our scheduler to estimate workload characteristics. The estimation process consists of two phases: sampling and adjustment.

2.4.1 Sampling

In the sampling phase, we take advantage of the fact that MapReduce jobs are divided into small tasks. When a job is submitted, one of map tasks is selected (and run) to estimate the execution time and CPU usage; and sampling result can represent the characteristic of the rest of map tasks in the job as discussed in later of this section. This sampling also applies to reduce tasks.

2.4.1.1 Task Selection

Our scheduler targets a highly dynamic environment, in which new jobs can be submitted at any time, and in which resources of a node are shared by slots to concurrently execute tasks. In such an environment, a sampling task tends to be combined with other tasks to run on a node. If the running tasks have similar characteristic, the execution

time will be longer than expected because of resource contention. It is an overhead introduced by our sampling phase. If the number of tasks in a job is very large, this overhead is insignificant. However, actual Hadoop systems could be quite contrary as observed in [6] that 70% jobs only have less than 20 tasks. Considering this kind of high proportion of execution time taken by the sampling task, we need to find an efficient strategy to select tasks.

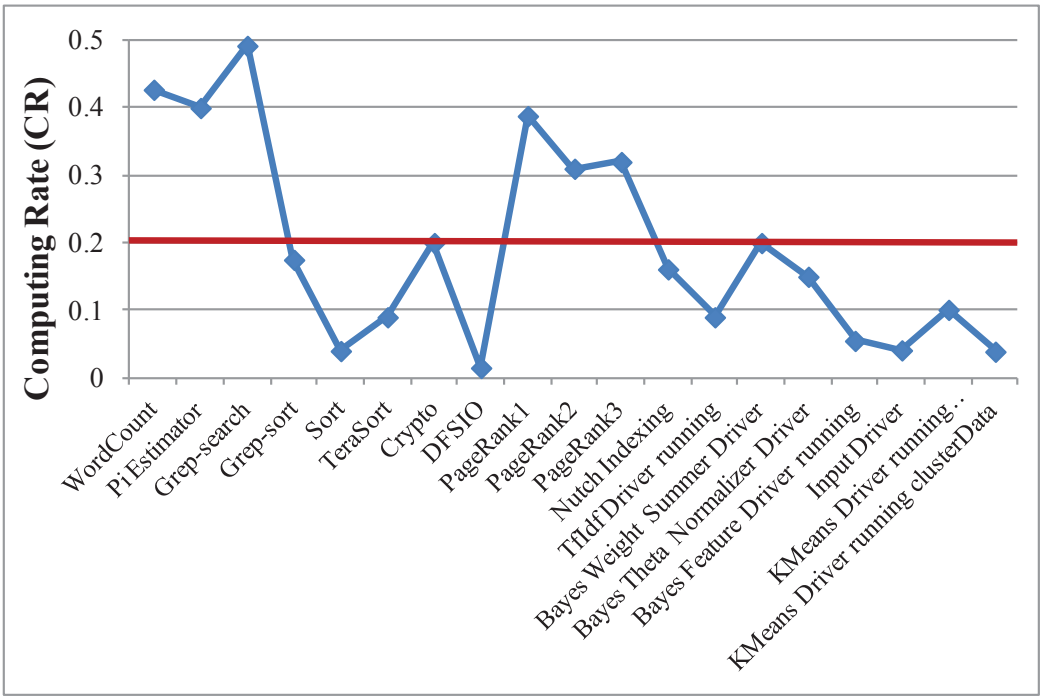
In our scheduler, we adopt a 'user-definable' selection submodule as an assistant approach to reduce the impact of such sampling noise. This submodule allows users to (design and) specify a task selection strategy based on their actual execution environment.

In this work, we implement two strategies: random and static analysis. The random strategy simply selects tasks based on the order of their arrival. The static analysis strategy exploits inherent attributes of jobs in Hadoop, i.e., Java bytecode.

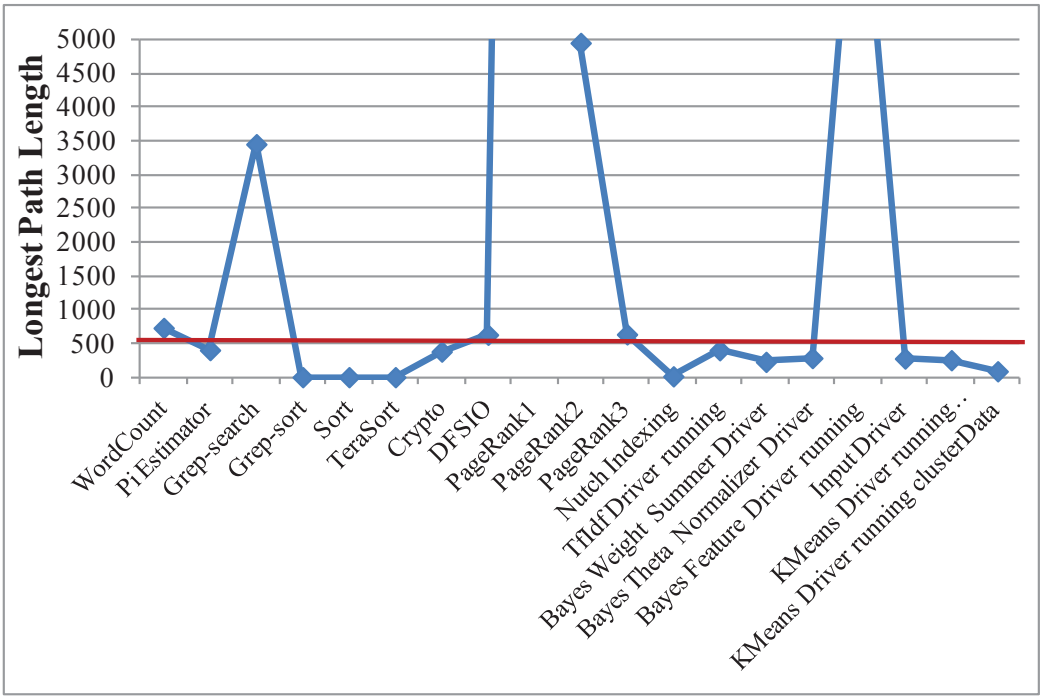
The static analysis takes more time to select a sampling task than the random. For a given task, the static analysis first build CFG and identify the longest path length (LPL) to represent application characteristic, i.e., amount of computation. LPL is most likely the worst case execution path. The longer the LPL, the higher the computing rate. Thus, LPL is a good measure to estimate computing rate prior to task execution.

To validate our LPL-based static analysis, we have run 19 benchmarks—including machine learning jobs, web search jobs and some typical MapReduce benchmark jobs—in a 4-node local Hadoop cluster and compared them in terms of CR and LPL.[1] Results for map tasks are shown in Figures 2.3. We define a job as I/O intensive if its CR is less than 0.2 and LPL is fewer than 500. Based on this, map tasks in 16 jobs of those 19 benchmarks (84%) are classified correctly. The definition of characteristic for reduce tasks is a little different because reduce tasks involve more I/O intensive operations (*Shuffle* and *Sort* steps both are I/O intensive). In Figure 2.4, we define a job as I/O intensive if its CR less than 0.2 and LPL fewer than 1,000. Now, reduce

[1] Detailed experimental setup can be found in Section 2.6.1.

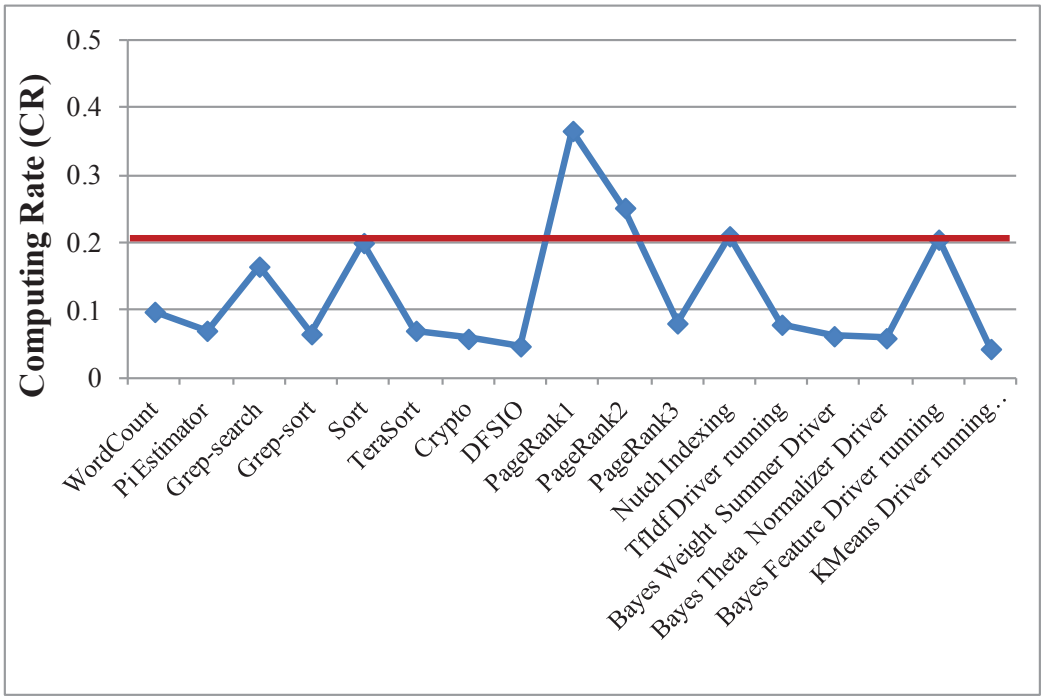


(a) Computing rate

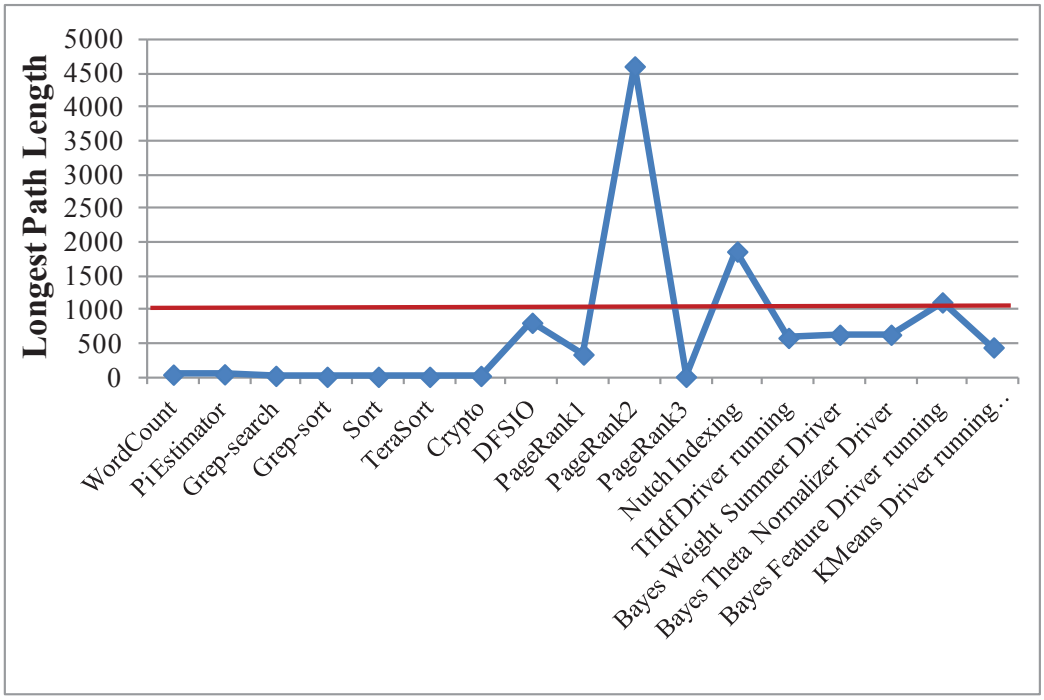


(b) Longest path lengths

Figure 2.3: Relationship between CR and LPL for map tasks of 19 jobs.



(a) Computing rate



(b) Longest path lengths

Figure 2.4: Relationship between CR and LPL for reduce tasks of 17 jobs.

tasks in 16 jobs of 17 benchmarks [2] (94%) are able to be reflected correctly.

It is obvious that the static analysis strategy can provide more accurate task selection than the random strategy, but it introduces more overhead.

2.4.1.2 Generalization of Workload Characteristic

Now, we need to answer the following question: can the characteristic of one map/reduce task represent the characteristic of the rest of map/reduce tasks belonging to the same job? The characteristic of a task (or workload characteristic) in this chapter indicates a tendency of resource utilization for the task because we acknowledge the fact that map or reduce tasks vary in their execution characteristics depending on the input data set they process. Note that we do not expect CR for each task is perfectly accurate. Rather, we expect CR to represent a tendency of resource utilization for map/reduce tasks belonging to the same job.

In the map phase, all map tasks deal with the input data of the same size, and they have the same user-defined map function to process their workload. The patterns of workload are uniformly distributed among all input splits. Each map task consequently has the approximate execution time. Although some extreme cases exist, these cases are very rare. We take Grep job as an example of such extreme cases: all words in a split are matched with the pattern for the Grep while no one word is found in another split. These two tasks could have completely different execution times because the divergence of their workloads. Because the frequency of occurrence of these cases is very low, there is no major impact on our scheduler even if they happen. Therefore, in this chapter we consider that each map task of a job has the same characteristic.

The reduce phase is similar. Each reduce task has the same user-defined reduce function. Because workload patterns of map tasks are uniformly distributed and the algorithm in the Partitioner, the output of map tasks tends to be well-proportioned and consequently the input of each reduce task is approximate. Thus, reduce tasks of a job

[2] Note that two benchmarks (`Input Driver` and `KMeans Driver running clusterData`) only have the Map phase

```
When a heartbeat is received from worker n:
if n has a free slot then
  /* Dispatch a new job in Pool (Fig. 2.2) for sampling */
  Find new jobs in Pool
  for j in jobs do
    Launch a task in j using task selection strategy
  end for
  Find jobs in Pool that have CR
  for j in jobs do
    Let newCR = (CR of j + total CR of tasks running on n) / #tasks running on n
    if newCR < average CR for all running jobs + threshold then
      Launch task of j on n
    end if
  end for
  /* # resources is larger than # running tasks*/
  Find jobs in Pool
  for j in jobs do
    Launch a task of j using task selection strategy
  end for
end if
```

Figure 2.5: Pseudo-code of the scheduler

have the same characteristic. Moreover, *Shuffle* and *Sort* (Figure 2.1(b)) both are I/O intensive; hence, most reduce tasks are I/O intensive.

2.4.2 Adjustment

After the sampling of a job the rest of tasks in that job are dispatched to run with tasks of other jobs that were thought to have complementary workload characteristics. In the adjustment phase sampling results are further calibrated with actual runtime data, e.g., task execution time and CR. This phase is necessary because sampling results may be impacted by unpredicted facts, such as resource contention, and diversity of input data. Our scheduler keeps track of every finished task in order to calculate average CR to represent the characteristic for the rest of tasks.

2.5 WCO Scheduling

In this section we detail the actual scheduling of MapReduce jobs using the WCO scheduler.

Our scheduler resides in JobTracker of master node and is triggered by heartbeats sent from TaskTrackers on worker nodes. This mechanism is the same as the FIFO scheduler in Hadoop but the heartbeats include CR information for tasks running (and completed after the last heartbeat) on the worker. Tasks of all submitted jobs are organized in a priority queue (Figure 2.2), and tasks that need to be tested (sampling) have high priority. Once a heartbeat arrives, the scheduler first selects test tasks according to the result of the task selection strategy, and then selects the rest of tasks based on CR (the first two *for* loops in Figure 2.5).

Once a task finishes, the CR of the task is retrieved from a heartbeat and collected by the Characteristic Estimation module for adjustment purposes as described in Section 2.4.2. When a heartbeat indicating free slots is received, our scheduler dispatches complementing tasks based on workload characteristics. Our scheduler uses the average CR of all running tasks to measure which tasks can be complementary for each other to run on the same node. The workings of WCO scheduler are shown in Figure 2.5.

The case for ample resource capacity (i.e., the number of available resources is larger than the number of submitted tasks) may happen, and it has been considered in our scheduling strategy (the last *for* loop in Figure 2.5). In such case, the scheduler directly dispatches tasks based on results from the selection submodule to use resource as much as possible instead of the approach of sampling one task and then dispatching the rest of tasks. In addition, as our scheduler is designed for dynamic environments in which new jobs constantly arrive, sampling tasks with high priority may occupy all resources, i.e., a classic scheduling problem of starvation. To avoid this problem, we adopt a user-defined threshold that sets the maximum number of concurrently running jobs in Pool (in Figure 2.2). The rest of jobs have to wait in the queue, and one or more

jobs are added to Pool if the number of running jobs deceases below the threshold.

2.6 Experiments

In this section, we describe experimental setup with 11 benchmarks and two testbeds, and present results.

2.6.1 Experimental Setup and Applications

We have used two testbeds for our experiments: a small private cluster with 4 nodes and a large cluster with 51 EC2 nodes (`m1.small`). In both environments Hadoop-1.0.0 with a block size of 64 MB was running; and each node has either two map slots or two reduce slots (i.e., 4 map slots and 4 reduce slots in the 4-node cluster, and 50 map slots and 50 reduce slots in the 51-node cluster). In the 4-node cluster, we used Xen to deploy four nodes on two physical machines, and each node was configured with one 3GHz core and 1.5 GB RAM. One of the nodes was configured to run the JobTracker, the TaskTracker, the NameNode and the Datanode. The rest of nodes were set to run TaskTrackers and DataNodes. The 51-node EC2 cluster was similarly configured, but one of the nodes was configured to be both the JobTracker and the NameNode, and the 50 remaining nodes were used as TaskTrackers and DataNodes.

The set of applications we used for our experiments was diverse. In addition to typical MapReduce benchmarks—Sort, WordCount, TeraSort, Grep, Crypto and Pi Estimator—we used 5 additional jobs from a benchmark suite [18], which includes two web applications: Nutch Indexing and PageRank, two machine learning applications: Bayesian Classification and K-means Clustering, and one HDFS Benchmark application: DFSIO. There are 11 benchmarks (jobs), and some contain sub jobs; hence 19 jobs in total (Table 2.1).

Table 2.1: benchmark applications

Type	Job	Sub-job
Typical	Sort	
	WordCount	
	TeraSort	
	Grep	Grep-search Grep-sort
	Crypto	
	Pi Estimator	
Web application	Nutch Indexing	
	PageRank	PageRank1 PageRank2 PageRank3
Machine learning	Bayesian Classification	TfIdf Driver running Bayes Weight Summer Driver Bayes Theta Normalizer Driver Bayes Feature Driver running
	K-means Clustering	Input Driver KMeans Driver running runIteration over clusters KMeans Driver running clusterData
HDFS	DFSIO	

2.6.2 Experiment One: Job Characteristic Analysis

The aim of the first experiment is to evaluate our analysis on generalization of workload characteristics. The experiment was carried out in the 4 node cluster. The input data is 512 MB for each job except Pi Estimator. [3] The size of input data leads eight map tasks for each job and we manually set eight reduce tasks.

We have verified the similarity in the characteristic of tasks in an individual benchmark. For simplicity's sake, We only present six representative applications among

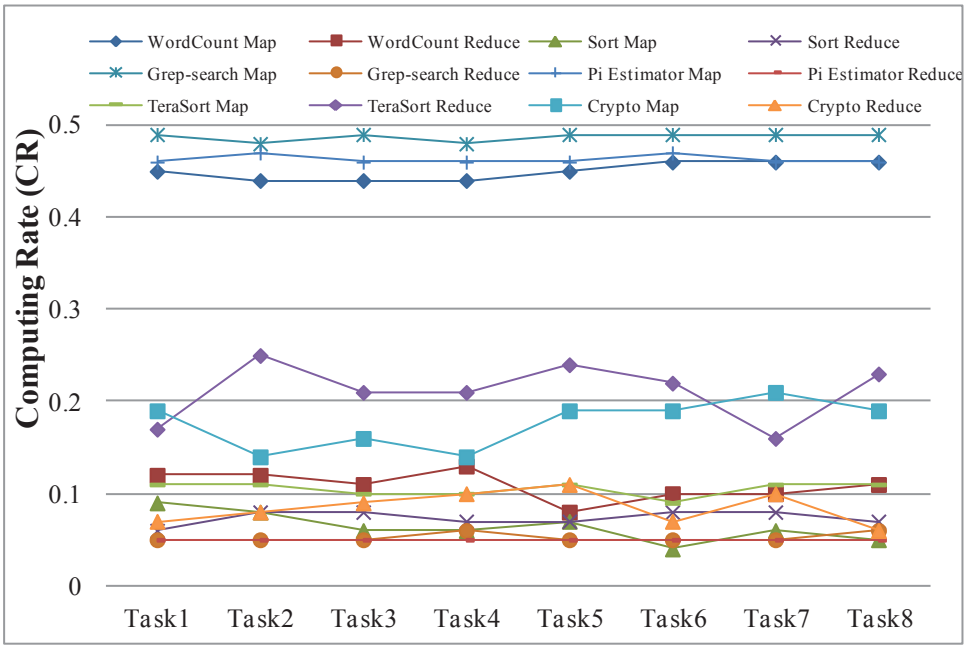
[3]There is no input data needed for Pi Estimator. The numbers of map tasks and reduce tasks are decided by configuration.

19 benchmarks. Those six benchmarks are WordCount, Pi Estimator, Grep-search, Sort, TeraSort and Crypto. The selection is based on their relativeness to CR. That is, while the latter three can represent I/O intensive, the first three are relatively more CPU intensive. We separately ran each of these six benchmarks in isolation to identify the computing rate and the execution time of each (map/reduce) task in a given benchmark (Figures 2.6(a) and 2.6(b)). Apparently, there is no significant variation for tasks belonging to the same job in terms of both CR and execution time. The variation of reduce tasks tends to be greater than of map tasks. It is because the input data of reduce tasks, outputted by map tasks, is probably not partitioned evenly. However, the variation of execution time for map tasks and reduce tasks in the same job still remains in a very similar level. Therefore, we can use the characteristic of a task to represent the characteristic of the rest of tasks.

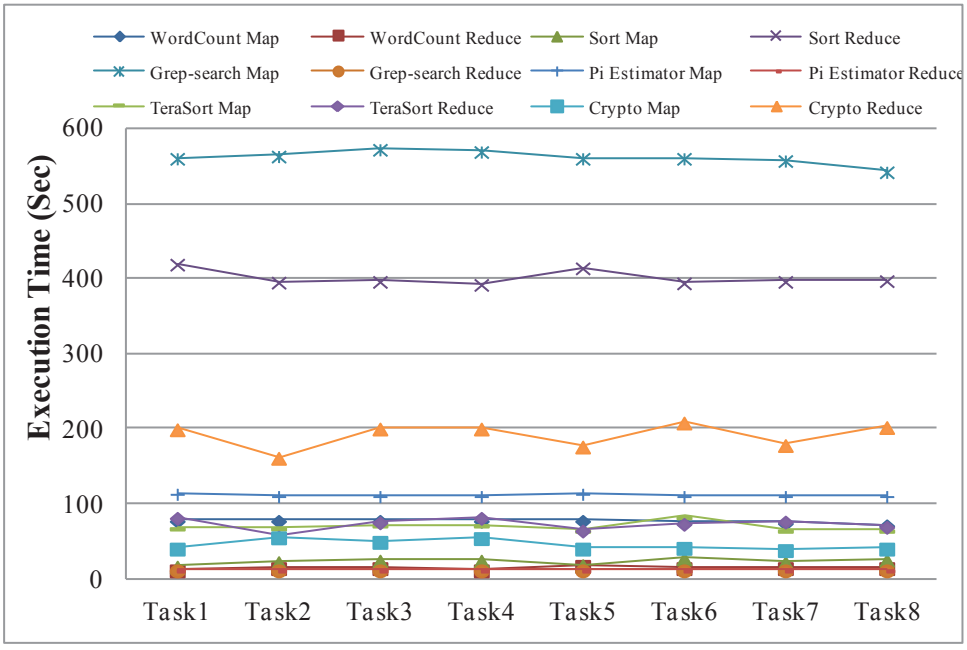
2.6.3 Experiment Two: Scheduling With the 4 Node Cluster

In this experiment, we used all 11 benchmarks on the 4 node cluster, and the static analysis technique as a task selection strategy was used in our scheduler. The rest of experimental settings were the same as those in the first experiment (Section 2.6.2). We used three different combinations of workload: I/O intensive, CPU intensive, and mixed (I/O + CPU) to simulate the environment with application diversity. It is feasible because our scheduler is mainly based on a job queue. If all resources of the cluster are used, subsequent jobs have to wait in the queue and no immediate impact on running tasks. In our experiments, we submitted jobs at the same time to simulate the workload at a certain moment of the system (snapshot). The order of job submission is not very important for our scheduler because the scheduler dispatches jobs based on characteristics first, and then the order (if characteristics are the same).

The snapshot with mixed workload simulates the moment when multiple types of job are in the job queue. We set four different test cases with all 11 jobs, the mixture of WordCount (CPU intensive) and TeraSort (I/O intensive), two I/O intensive workloads



(a) Computing rate



(b) Execution time

Figure 2.6: Characteristic analysis for 6 typical benchmarks. Computing rate and execution time both have no sharp variation (similar) between map/reduce tasks of the same job.

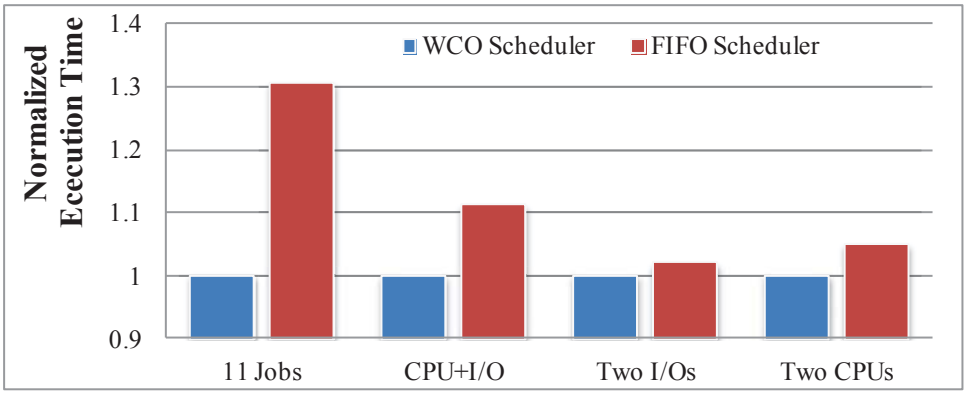
Table 2.2: execution time (sec) with respect to different schedulers. Task selection is not applicable in the case with 2 jobs.

	Scheduler	11 jobs	CPU+I/O	2 I/Os	2 CPUs
4 nodes	Hadoop FIFO	4,668	1,412	2,489	1,805
	WCO (random)	3,763	-	-	-
	WCO (static anls.)	3,575	1,269	2,440	1,719
51 nodes	Hadoop FIFO	3,008	634	868	2,065
	WCO (random)	2,711	-	-	-
	WCO (static anls.)	2,672	547	859	2,286

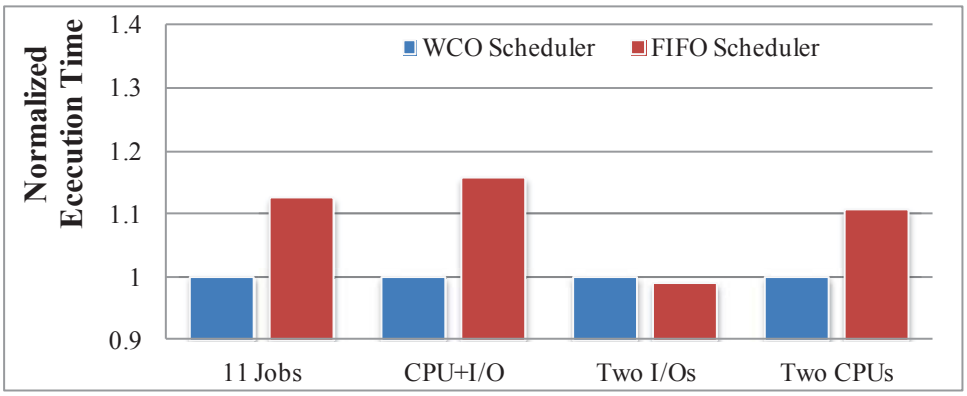
(Sort and TeraSort), and two CPU intensive workloads (WordCount and Grep). We see that WCO scheduler has a negligible effect when the workload has a similar characteristic (Figure 2.7(a) and Table 2.2). However, significant speedups are seen for the mixed workload, 23% faster with 11 jobs and 10% faster with the mixture of a CPU and an I/O intensive job, compared with the FIFO scheduler.

2.6.4 Experiment Three: Scheduling With the 51 Node Cluster

The third experiment was conducted on the 51 node cluster. Workload combinations, task selection strategy and benchmark jobs remained the same as those in Experiment two (Section 2.6.3). The difference is the input data (2GB) that resulted in 32 map tasks for each job. We manually configured 32 reduce tasks for each job. Because the disks and network are shared with others users' VMs on EC2, test results are not stable. In our experiments, we used the average value of multiple tests to reduce the interference of the shared I/O. Experimental results are similar to those in Experiment two. We got 11% improvement with the 11 job workload, 13% improvement with the mixture of a CPU and an I/O intensive job (Figure 2.7(b) and Table 2.2).



(a) 4 node cluster with 512 MB input data for each job



(b) 51 node cluster with 2 GB input data for each job

Figure 2.7: Comparison on three types of workload mixture: mixed, I/O intensive and CPU intensive. Results have been normalized against WCO.

2.6.5 Experiment Four: Comparisons and Discussion

In the fourth experiment, we conducted a more in-depth analysis on our scheduling performance. In previous experiments 11 job workload was used, but the size of input data was varied, 512 MB for the 4 node cluster and 2 GB for the 51 node cluster, respectively. Results are shown in Table 2.2. In the 4 node cluster, the WCO scheduler with the static analysis strategy is 23% faster than the FIFO scheduler. If we use the random strategy to select tasks during the sampling phase instead of the static analysis, the scheduler is 19% faster than the FIFO scheduler. In 51 node cluster, the WCO scheduler with the static analysis strategy has 11% improvement, and the scheduler with the random strategy obtained a similar experimental result, 10% improvement.

Based on these results, we raise two questions. The first one is why the 4 node cluster gains more benefit with the static analysis strategy compared with the 51 node cluster. It is because of the resource proportion for the sampling phase. In order to keep simple, we assume that map and reduce tasks of a job have the same execution time and the capacity of each node for both clusters is identical. We define the resource consumption for a node that provides resource R to process a task per unit time as the product of task execution time (Equation 2.1) and R. The ratio of the resource consumed in the sampling phase to the resource consumed by all jobs is:

$$\beta = \frac{2 \times R \times \sum_{n=1}^{k} TT_n}{R \times \sum_{n=1}^{k} \sum_{n=1}^{m} TT_n} \tag{2.3}$$

$$= \frac{2 \times R \times \sum_{n=1}^{k} TT_n}{R \times N \times a \sum_{n=1}^{k} \sum_{n=1}^{m} TT_n} \tag{2.4}$$

where TT is the task execution time, m is the number of tasks in a jobs, k is the number of jobs, a is an task execution parallelism coefficient which makes that the total time taken to finish all jobs equals $a\sum_{n=1}^{k}\sum_{n=1}^{m} TT_n$, N is the number of nodes in a cluster. For both clusters, the task input split is 64 MB and the number of submitted jobs is the same. This enables the resource consumption for the sampling phase is approximate. Based on Equation 2.4, if k is fixed, increasing m and/or N results in decrease of proportion of the resource consumed by the sampling phase. Typically, resource consumption and execution time have linear relationship. Therefore, similarly, the proportion of the time taken by the sampling phase reduces with increasing m and/or N. Apparently, the improvement of performance caused by the accuracy of task selection strategy can be ignored if m and N are relative large. In such a case, we can choose the strategy which is simple and introduces the minimum overhead. Conversely, for the 4 node cluster, we should take advantage from the accuracy of the static analysis strategy to get additional improvement.

The second question is why we obtain different improvement rates in these two clusters with the same workload. It is due to the ratio of CPU intensive tasks and I/O

intensive tasks. If the ratio is 1:1 (50% and 50%), we can always combine one CPU intensive task with one I/O intensive task to run on the same node, and consequently the result would be the best. In these two clusters, the ratio is different. The size of input data partly decides how many tasks we can have. Hence, the speedup is different.

Based on our experimental results, the performance of our scheduler heavily relies on workload. If the workload is diverse and the ratio of CPU intensive tasks and I/O intensive tasks is approximately even, the speedup of our scheduling strategy is significantly high. Moreover, if there are relatively more jobs and fewer resources, additional benefit can be obtained from a more sophisticated task selection strategy.

2.7 Conclusions

In this chapter we have presented a MapReduce scheduler that has been implemented on top of Hadoop. The scheduler with its characteristic estimation module dynamically and adaptively dispatches tasks of MapReduce application in the way that tasks with complementing resource usage are co-located to improve performance/throughput. The estimation module adopts a task selection strategy in addition to sampling and runtime adjustment. Our experiments have demonstrated that our scheduler effectively exploits workload diversity. Moreover, our ‘simply’ static analysis technique can contribute to reducing sampling overhead particularly when there are relatively more jobs than resources. The experimental results have validated our approach with static analysis and verified our claims on the efficacy of our scheduler.

Chapter 3

Non-intrusive slot layering in Hadoop

3.1 Introduction

MapReduce [1] is a compelling parallel and distributed computing solution to harness large-scale commodity machines for processing big data. It makes programming easier to extract the capacity of distributed computations and accelerates to process vast amounts of data. Hadoop [2], an open source implementation of MapReduce, has been widely used. In the Hadoop framework, a job is divided into smaller grain tasks and they are dispatched to multiple machines to be executed simultaneously in order to reduce job execution time. Resource in Hadoop clusters is also uniformly partitioned into slots in the sense that they fairly share the resource.

The Hadoop architecture complies with the *master*/*slave* paradigm. A *master* (JobTracker in Hadoop) is responsible for dispatching tasks according to scheduling strategies while a set of worker machines (TaskTrackers in Hadoop) are in charge of managing resource and processing tasks assigned by the *master*. In such a design, MapReduce presents a splendid mechanism to harness large-scale commodity machines to process big data that has been done by supercomputer traditionally. However, as the capacity of a single machine rapidly increases (e.g., #cores), the original Hadoop design imposes two severe limitations in term of resource management particularly at

the worker level. First, the current static slot configuration inaccurately represents resource sharing with diverse applications. The number of slots in a worker machine dictates the maximum number of tasks that are allowed to concurrently execute; and this slot configuration is fixed throughout the lifetime of the TaskTracker residing in the worker machine. In theory, the number of slots is configured in the way that the best performance is achieved by maximally using resource among slots. In reality, such static configuration is a bottleneck for efficient resource utilization because it is impossible to find a rule of thumb when dealing with a diverse set of jobs. Clearly, an inappropriate slot configuration easily leads to severe overall performance degradation (up to 37% performance decrease in our experiments) as resource contention tends to be aggravated if there are too many slots, or on the contrary resource sharing is not fully realized if there are too few slots.

Second, resource is shared fairly regardless of job priority. The current scheduler in JobTracker dispatches tasks in a FIFO manner, and resource is uniformly partitioned and allocated to both high-priority/early-submitted tasks and low-priority/late-submitted tasks at the worker level. Clearly, such fair resource sharing brings extra delay in the execution time of high-priority/early-submitted jobs, and it is also unsuited to execute ad-hoc query jobs expecting fast response time. In our pilot study it is observed that the execution time of a high-priority job with the current Hadoop scheduler is almost 3 times longer compared with its execution time in a dedicated environment, i.e., a single slot per core.

Apparently, these limitations reduce overall throughput and high-priority job performance, and therefore the explicit consideration of them is crucial for the further development of MapReduce programming paradigm. Some previous efforts ([19, 20, 10, 21]) have been made to mitigate the detrimental impact. The studies in [19, 20, 10] focus on a single job performance, whereas [21] consolidates tasks in term of their characteristics for improving resource utilization and overall throughput. Nevertheless, these previous studies are still based on using slots to uniformly partition resource and

therefore suffer from the aforementioned limitations.

In this work we propose a non-intrusive slot layering approach with the goal of improving both overall throughput and high-priority job performance. It unifies Map slots and Reduce slots as task slots and prioritizes them into two tiers—Active and Passive—using `cgroups` [22]. Sufficient resource is dynamically allocated to tasks running in Active slots according to the variation of their resource usage pattern, and tasks running in Passive slots are executed exploiting the unused resource of Active slots; hence the name non-intrusive slot layering. In addition, we devise a layering-aware scheduler which arranges and schedules jobs based on priority to make effective use of two priority tiers. To the best of our knowledge, this is the first work in MapReduce that uses two tiers of slots (layering) based on job priority. Our approach advances the state of the art in several ways:

- Leverages slot layering to achieve better resource utilization, 1%-36% improvement for overall throughput,
- Extends job priority from JobTracker to TaskTracker; high-priority jobs are allocated sufficient resource on each worker; and as a result their execution time significantly decreases by 13%-52%,
- Eases the configuration burden of the system administrator by setting the numbers of Active/Passive slots to be the same as the number of cores, respectively, and
- Improves data locality (placing tasks on workers that contain their input data).

The rest of the chapter is organized as follows. Section 3.2 discusses related work. Section 3.3 describes the design and implementation for our system. Section 3.4 presents experimental results that validate the efficacy of our approach followed by detailed analysis and discussion of such an efficacy in Section 3.5. In Section 3.6 we draw the conclusion.

3.2 Related Work

The original scheduler in Hadoop essentially uses a single job queue with a FIFO strategy to dispatch tasks to slots. Resource of each worker is uniformly partitioned into slots, and the number of slots is statically configured before launching the Hadoop system.

As the number of users sharing a MapReduce cluster increases, efficient resource sharing is essential. Schedulers presented in [23, 6, 9] strive for more efficient cluster sharing. The Capacity scheduler in [23] enables resource to be separately partitioned for each user or organization while schedulers in [6, 9] pay more attention to fairness.

Much of the recent work has shown interest in either increasing a single job performance or overall resource utilization. Works in [19, 20] use job profiling techniques to determine the size of resource in order to finish a job with specific performance goals. However, techniques used in [20] do not pay much attention to the performance degradation caused by over-utilized or under-utilized resource between slots, and consequently, performance improvement for a single job may be achieved at the cost of reducing overall throughput. J. Polo et al. in [19] present techniques to dynamically allocate slots instead of the original approach of static configuration. These techniques decouple the resource sharing between map and reduce slots. But the profiling techniques it used only take into account the case for over-utilized resource, detecting the upper bound of resource utilization of a job to determine the best number of collocating slots at a worker. This approach is able to efficiently avoid resource contention and yet, resource could be under-utilized if the upper bound only appears at a certain time. By contrast, the WCO scheduler in [21] collocates tasks of possibly different MapReduce jobs with complementing resource usage characteristics to improve overall throughput with sacrifice of a single job performance to a certain degree. Apparently, all these works have to achieve improvements at the cost of one or more other performance goal. The main reason is that they are still based on slots of uniformly partition resource. In this work we provide the non-intrusive slot layering approach to improve both a single

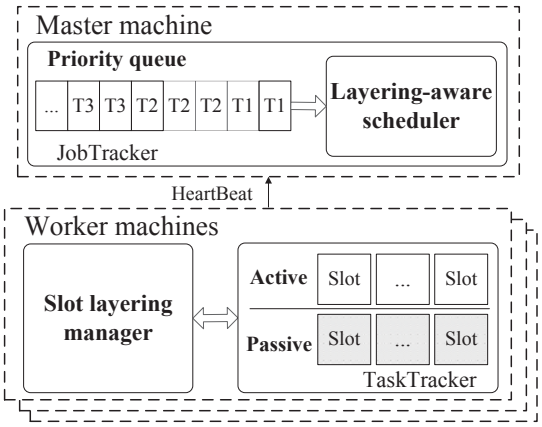


Figure 3.1: Overview of non-intrusive slot layering.

job performance and overall resource utilization.

3.3 Non-intrusive Slot Layering

In this section, we present a non-intrusive slot layering approach, which has been implemented on top of Hadoop.

3.3.1 Overview

Hadoop tasks often do not fully utilize CPU resource during their execution and exhibit certain CPU usage patterns [18]. Resource sharing between two tasks running on the same processor (core) as an attempt to maximize utilization is a primary source of performance interference/degradation due to resource contention. In essence, our approach, non-intrusive slot layering, isolates resource (a processor core or simply core)[1] into two tiers with different resource sharing priorities: *Active* and *Passive*. That is, there are two slots on a core with different resource usage priorities. Moreover, Hadoop introduced two-type of slots (Map slots and Reduce slots) to run tasks in different phases. This way is easier to lead to an inaccurate configuration to represent

[1]In this work, we only consider to isolate CPU capacity within each TaskTracker. Note that extending our current work to accommodate for other resource, e.g. disk bandwidth, network bandwidth, is straightforward. Hereafter, resource refers to CPU resource.

resource sharing among a worker and possibly aggravates the performance interference/degradation. For example, we can configure 2 Map slots and 2 Reduce slots for a single core machine in order to share CPU resource between two running tasks. Accordingly, two tasks at most (two Map tasks or two Reduce tasks) are concurrently executed on the worker when running a job. However, four tasks (two Map tasks and two Reduce tasks) could be executed at the same time once there are multiple jobs. Therefore, in order to have a finer-grained resource model, we extend such slots based on type to unified task slots which are used as container to run both Map and Reduce tasks. In the rest of the chapter we will use the terms 'task slot' and 'slot' interchangeably.

In this work, we use two components to implement non-intrusive slot layering as shown in Figure 3.1: Slot Layering Manager and Layering-aware Scheduler. The former, working with a TaskTracker, is used to dynamically prioritize slots into two tiers for isolating resource based on priorities of running tasks. The latter residing in the JobTracker of master machine is used to decide task placement on TaskTrackers explicitly taking into account two slot tiers.

3.3.2 Slot Layering Manager

The slot layering manager dynamically manages tasks with different tiers. It works with TaskTracker of each worker machine and is triggered by new tasks assigned by JobTracker of the master machine. After a TaskTracker is initialized, the slot layering manager automatically collects the CPU information of current worker machine using the `lscpu` Linux command to determine the number of slots in Active and Passive, respectively. We adopt `cgroups`—that is capable of limiting, accounting and isolating resource usage—for managing two slot tiers. This Linux kernel feature enables for two concurrent tasks on a single core to complement resource usage contributing to utilization improvement.

The maximum number of task slots allocated to the Active tier is automatically

Algorithm 1: Task scheduling run at each scheduling cycle.

```
When a heartbeat is received from worker n:
/* Reduce task scheduling */
if n has free Active/Passive slots then
    for j in jobs do
        if priority of j is greater than the lowest priority in Active slots then
            assign unassigned reduce task t of j on n
/* Map task scheduling */
/* Stage 1: assigning map tasks to Active slots */
for slot in Active slots do
    if number of running and assigned map tasks on n is less than TR then
        for j in jobs do
            if j has unassigned map task t with data on n then
                assign t on n
            else if j has unassigned map task t then
                assign t on n
/* Stage 2: assigning high-priority or data-local map tasks to Passive slots */
if n has free Passive slots & no map task is assigned to Active slots in this
scheduling cycle then
    for slot in Passive slots do
        if number of running and assigned map tasks on n is less than TR then
            for j in jobs do
                if priority of j is greater than the lowest priority in Active slots
                then
                    assign unassigned map task t of j on n
                else if j has unassigned map task t with data on n then
                    assign t on n
/* Stage 3: assigning map tasks to Passive slots */
if n has free Passive slots & no map task is assigned to Active slots in this
scheduling cycle then
    for slot in Passive slots do
        if number of running and assigned map tasks on n is less than TR then
            for j in jobs do
                assign unassigned map task t of j on n
```

determined by the number of cores. The maximum number of task slots in the Passive tier has the same number as that in the Active tier or is manually configured. The total

number of task slots is twice as the number of cores by default. The CPU resource sharing ratio between the two tiers is 100:1; this enables that tasks running in Active slots take the priority of utilizing CPU to keep their original CPU usage pattern as much as possible while tasks running in Passive slots use as much unused CPU resource as possible; hence, the resource usage between slots is non-intrusive. Clearly, our non-intrusive slot layering approach is able to improve resource utilization and reduce the overhead caused by process switching latency.

When receiving new tasks from JobTracker, high-priority tasks are assigned to Active slots and the rest in Passive slots. If all the tasks have the same priority, the slot layering manager follows a FIFO manner, placing early-assigned tasks to Active slots. Furthermore, the transition of a task across these two tiers could occur during its execution. If a high-priority task arrives later and all Active slots are occupied, the lowest-priority or latest-assigned task in an Active slot is switched to a Passive slot in order to free up an Active slot for the high-priority task. If one of tasks running in Active slots finishes, the highest-priority or earliest-assigned task in Passive slot is switched to the Active slot. This transition takes place constantly.

3.3.3 Layering-aware Scheduler

The layering-aware scheduler resides in JobTracker and it is triggered by heartbeats sent from TaskTrackers. This mechanism is the same as the FIFO scheduler in Hadoop but each heartbeat includes the lowest priority in Active slots and the number of free slots on the worker machine grouped by Active slots and Passive slots. Tasks of all submitted jobs are organized in a priority queue (Figure 3.1). Users can give a job a specific priority when submitting the job or the system will assign a timestamp as its priority. Users also can change the priority during the lifetime of the job. Map and Reduce tasks have the same priority as the job they belong to has. The layering-aware scheduler dispatches tasks according to priorities and is data locality aware. The scheduler consists of two phases: map task scheduling and reduce tasks scheduling.

The task scheduling is outlined in Algorithm 1. The first part is reduce tasks scheduling. We allocate reduce tasks first because reduce tasks usually have higher priority than map tasks. Only a highest-priority reduce task is allocated per heartbeat. This makes reduce tasks are distributed to as many workers as possible and thus reduces I/O resource contention (mainly existing in `Shuffle` and `Sort` phases) between reduce tasks.

The second part is map tasks scheduling. which has three stages. First, we assign tasks to run on Active slots. In this stage, the scheduler strictly respects priority selecting the highest-priority job considering data locality. In Stage 2, we implement an algorithm to improve the response time for late-arriving but high-priority jobs and increase the number of data-local tasks. In this way, those high-priority jobs assigned to Passive slots in this stage are switched to Active slots by the slot layering manager on that worker. If there is no high-priority job, we select data-local tasks from all submitted jobs in order to improve data locality. The remaining Passive slots if any are filled by tasks in our priority queue in their order (Stage 3). Note that we never dispatch map tasks to both Active slots and Passive slots in the same scheduling cycle, which enables that high-priority jobs are evenly distributed across TaskTrackers to maximize resource utilization within a cluster. A threshold `TR` in the three stages, which is calculated by a ratio of map tasks to all tasks, is used to limit the maximum number of running map tasks on the current worker. It can help the scheduler to reserve sufficient idle slots for unassigned reduce tasks.

Preemption can't be supported in this scheduler. High-priority jobs have to wait until any slot becomes available. As to supporting preemption, there is a tradeoff between overall performance and waiting time for a high-priority job: whether to kill running tasks in order to free up slots for the high-priority job or to wait until they finish. In this work, we have proposed the slot layering manager, which can allocate more slots per worker while reducing the execution time of tasks running in Active slots. With a sufficient cluster size, more slots means higher possibility to obtain a free slot for a

high-priority task, and non-intrusive slot layering guarantees resource provisioning for Active slots. Moveover, recall the design in Hadoop, a job is divided into small tasks to run in parallel for reducing the overall job execution time and consequently, the execution time of a task should be relatively much shorter compared to the execution time of a job. Based on these reasons, the time for waiting slots will make up a small percentage of the total execution time of a high-priority job. Therefore, we prefer the overall utilization in this work.

3.4 Experiment

In this section, we describe results from two experiments that validate our claims of improvement of overall resource utilization (overall throughput) and the reduction in the execution time of high-priority jobs.

3.4.1 Experimental Setup and Applications

We have used two testbeds composed of EC2 m1.large instances for our experiments: a small cluster with 4 nodes and a large cluster with 40 nodes. In both environments Hadoop-1.0.0 with a block size of 128 MB was running. In the 4-node cluster, one node was configured to run JobTacker, TaskTracker, NameNode and DataNode, whereas the rest were set to run only TaskTrackers and DataNodes. The 40-node cluster was similarly configured, but one of the nodes was dedicated to run JobTracker and NameNode, and the 39 remaining nodes were used as TaskTrackers and DataNodes. Based on the common practice provided in [24], we varied the slot configuration from 2 map slots and 2 reduce slots (2m2r) to 4 map slots and 4 reduce slots (4m4r) in our experiments. The set of applications we used for the two testbeds was 6 typical MapReduce benchmarks: Crypto, Grep, Sort, TeraSort, WordCount and DFSIO.

Note that disks and network could be shared with other users' VMs on EC2, which may result in unstable results. Therefore, all the experimental results presented in this

chapter are averaged over at least three tests to reduce the interference of such I/O sharing.

3.4.2 Experiment One: Scheduling with a 4-node cluster

There are three objectives for the first experiment:

- Proof of that there is no rule of thumb to statically configure slot count,
- Evaluation of the overall performance improved by non-intrusive slot layering for a small-size cluster and,
- Validation of the effectiveness of our approach for high-priority jobs in a small-size cluster.

We first separately ran the 6 benchmark jobs with 2GB or 10GB input data (writing 100MB data per map task for DFSIO), and varied the maximum number of slots per TaskTracker from 2m2r to 4m4r. The execution time of each job is shown in Table 3.1. The best performance is observed for jobs with non-intrusive slot layering. Those execution times in Table 3.1 are plotted in a normalized form for more effective comparisons (Figure 3.2). Specifically, job execution times of different configurations are normalized based on the execution time with non-intrusive slot layering. It is observed that each configuration can be the best configuration for a certain job, e.g., 2m2r is the best configuration for Crypto with 2GB input data, 3m3r is for TeraSort with 10GB input data, and 4m4r is for Grep with 2GB or 10GB input data. Note that variation of input size could also make the best configuration different. 4m4r is the best for Sort with 2GB but 2m2r for Sort with 10GB. Therefore, there is no rule of thumb to set the best static configuration for a particular job and an inappropriate static configuration results in up to 21% performance degradation in the experiment.

Further, we ran jobs with different workload combinations. The results are shown in Table 3.2 and Figure 3.3. The size of input data for each job is 1GB except the last combination with 2GB input data. The jobs were submitted by an interval of 2 seconds,

so that they were executed in a certain order. Obviously, the best configuration always changes along with job characteristics, order, frequency and the size of input data. There is also no rule of thumb to set the best static configuration for multiple jobs and an inappropriate static configuration results in up to 37% performance degradation in the experiment.

These two experiments prove that there is no absolute best static configuration for a dynamic environment and the overall throughput certainly decreases due to an inappropriate configuration.

As with overall performance, our non-intrusive slot layering approach performs the best in all situations. More importantly, our approach not only removes the configuration burden, but also delivers its performance close to or even beyond the overall performance with the best static configuration. As seen in Tables 3.1 and 3.2, our approach obtains up to 12% and 36% improvement compared with the best configuration and other configurations, respectively. Moreover, the rate of data-local tasks (Figure 3.4) increases 6%-13% using our layering-aware scheduler.[2]

To evaluate the execution time of high-priority jobs, we used two users to simulate a typical scenario: ad-hoc query jobs are submitted with a high priority when long-time routine jobs keep running. In this test, one user kept submitting jobs. After all task (Map and Reduce) slots were occupied, the other one randomly submitted a high-priority job 10 times to get an average execution time. For simplicity, we only used WordCount jobs. The results are presented in Table 3.3. The execution time of a WordCount job with the dedicated resource is 80 seconds. Running with static configurations is 103%-139% slower and our system is 76% slower. Moreover, since high-priority jobs are executed in Active slots, our average Map and Reduce tasks execution time is the closest one to the execution time running on dedicated resource, 5% increase for Map tasks and 13% increase for Reduce tasks. Apparently, Running with non-intrusive slot layering is the best because tasks running in slots can be finished

[2]Data replication was configured as 2.

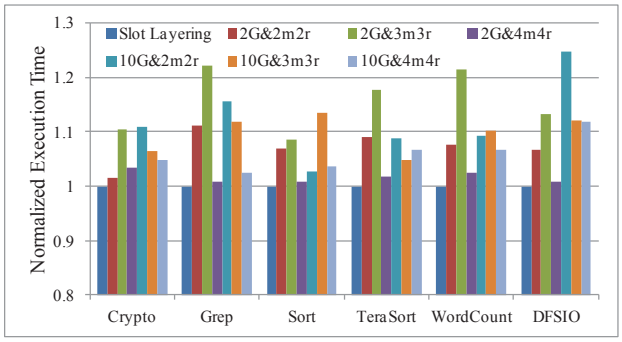


Figure 3.2: Normalized execution time comparisons for 6 jobs with 2GB or 10GB input in 4-node.

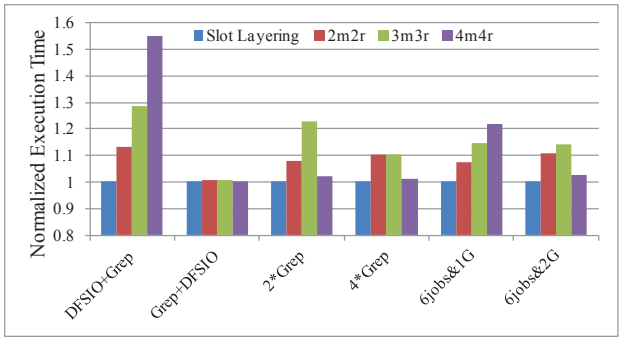


Figure 3.3: Normalized execution time comparisons for job combinations in 4-node cluster.

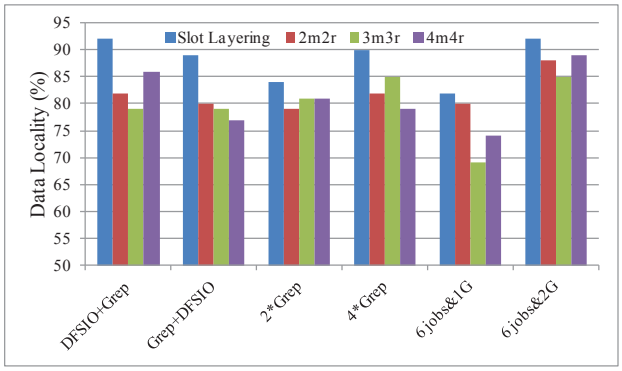


Figure 3.4: Percentages of data local tasks in 4-node cluster.

more quickly and there are more slots in the cluster (compared with 2m2r).

Table 3.1: Jobs execution time (Sec) in 4-node cluster.

	2GB / 10GB			
	Slot Layering	2m2r	3m3r	4m4r
Crypto	128 / 578	130 / 641	141 / 615	132 / 607
Grep	326 / 1267	362 / 1465	398 / 1418	329 / 1300
Sort	108 / 471	116 / 484	118 / 535	110 / 489
TeraSort	115 / 449	125 / 489	135 / 471	117 / 479
WordCount	125 / 430	135 / 470	153 / 474	129 / 459
DFSIO	127 / 133	136 / 166	144 / 149	128 / 148

Table 3.2: Jobs execution time (Sec) in 4-node cluster. The order of 6 jobs combination is Crypto, WordCount, DFSIO, Grep, TeraSort and Sort.

Combination	Slot Layering	2m2r	3m3r	4m4r
DFSIO+Grep	209	237	270	325
Grep+DFSIO	237	239	239	238
2*Grep	349	376	428	357
4*Grep	610	673	671	617
6 jobs with 1GB input data	365	393	418	445
6 jobs with 2GB input data	620	686	708	637

Table 3.3: Average jobs execution time (Sec), average Map tasks execution time (Sec) and average Reduce tasks execution time (Sec) comparisons for a high-priority job (WordCount) in 4-node cluster.

	Job		Map Task		Reduce Task	
	time	inc.	time	inc.	time	inc.
Dedicated Resource	80	-	40	-	15	-
Slot Layering	141	76%	42	5%	17	13%
2m2r	162	103%	68	70%	20	33%
3m3r	181	126%	108	170%	22	47%
4m4r	191	139%	128	220%	26	73%

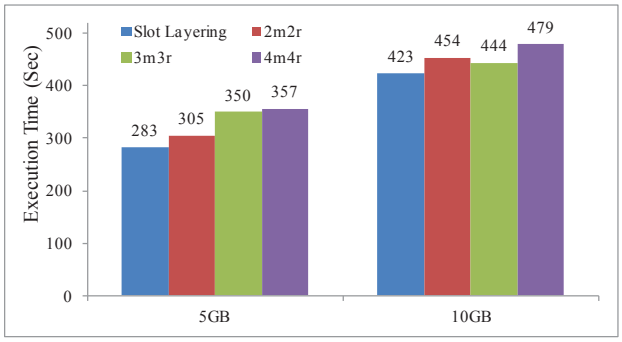


Figure 3.5: Comparisons for 6 jobs execution time in 40-node cluster.

3.4.3 Experiment Two: Scheduling with a 40-node cluster

Experiment One presents the improvements achieved by our non-intrusive slot layering approach for a small-size cluster. In this experiment, we further validate that our approach is able to be widely used for any cluster size. We used a bigger-size cluster (40-node), and the 6 benchmark jobs with 5GB and 10GB input data for each were submitted in a random order with an interval of 2 seconds. We tested each configuration for 10 times and results were averaged. Job execution time and rate of data-local tasks are shown in Figures 3.5 and 3.6. Our approach still outperforms others by 5%-21% in term of execution time, and increases the rate of data-local tasks by 3%-9%[3].

Experimental results (Figure 3.7) for high-priority jobs are similar to those in Experiment one. Running a high-priority job with our approach is 27% slower than running it in the dedicated resource, but 23%-52% faster than the original scheduler in Hadoop.

3.5 Analysis and Discussion

In this section, we show and discuss in depth how our approach overcomes shortcomings of the current way in Hadoop to represent resource capacity.

[3]Data replication was configured as 4.

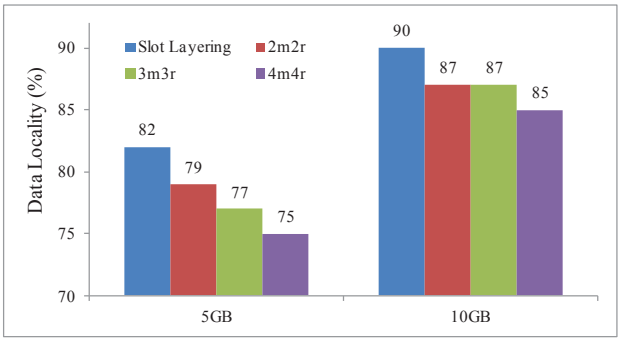


Figure 3.6: Percentages of data local tasks in 40-node cluster.

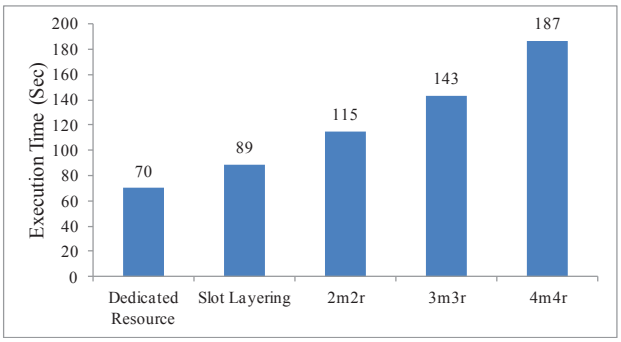


Figure 3.7: Comparisons for a high-priority job in 40-node cluster.

3.5.1 Best static slot configuration

One of goals of our approach presented in this chapter is to remove the static slot configuration burden without performance degradation. A necessary indicator we need to compare with is the performance with the "best" static slot configuration. However, there is no rule of thumb [24]. The best slot configuration varies in a range according to the capacity of machines and the characteristic of jobs. In order to narrow down the range for our experiments in this chapter, we use two-core machines (EC2 m1.large) to run the same job (WordCount) on a 4-node cluster with different slot configurations, varying the maximum number of slots per TaskTracker from 1 map slot and 1 reduce slot (1m1r) to 8 map slots and 8 reduce slots (8m8r). As can be seen in Figure 3.8, the best static configuration is using 4 concurrent map tasks and 4 concurrent reduce tasks per TaskTracker. Apparently, the configuration of 1 map slot and 1 reduce slot leads to very poor resource utilization while the configuration with more than 4 map slots and

4 reduce slots causes over-utilized resource. We also found the same behavior in other jobs we used in this chapter. The best static slot configuration only varies from 2 map slots and 2 reduce slots to 4 map slots and 4 reduce slots.

3.5.2 Efficacy of non-intrusive slot layering

In Figures 3.9(a) and 3.9(b) we separately show the details of CPU utilization and system context switches among four configurations during the execution of 8 map tasks (WordCount) on a worker. The average CPU utilization and the number of system context switches in total can be seen in Figure 3.9(c). The non-intrusive slot layering approach has the best CPU utilization (as CPU usage pattern of tasks running in Active and Passive can complement to keep the maximum CPU utilization) and fewer context switches among these four configurations (as the tasks running in Active slots take the priority of utilizing CPU and can run longer without interruption), and consequently it delivers the shortest execution time as seen in Figure 3.8. In addition to this performance improvement, the number of slots is determined by the number of processors or cores. There is no more need to figure out a rule of thumb and statically configure the slot count. Moreover, because tasks running in Active slots are provisioned sufficient resource, their execution time is very close to that in the dedicated resource (2 map slots) as seen in Table 3.4. Meanwhile, the task execution in Passive slots is carried out taking advantage of unused resource; hence, the execution time of 75 seconds compared 80 seconds with 4 map slots. This non-intrusive layering of slots enables the scheduler to give more resource to high-priority jobs without reducing overall throughput (Section 3.3.2). It also remedies the violation against the scheduling strategy—high-priority or early-submitted tasks fairly share resource with low-priority or late-submitted tasks.

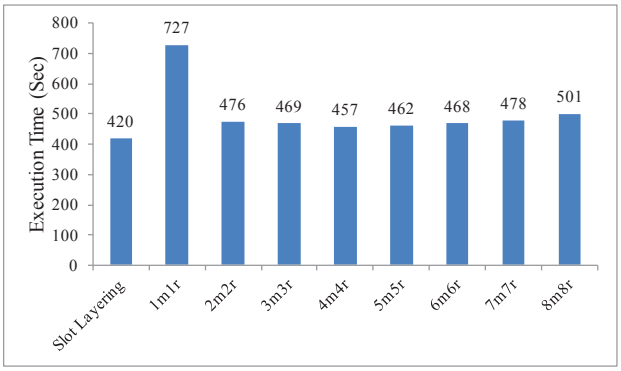
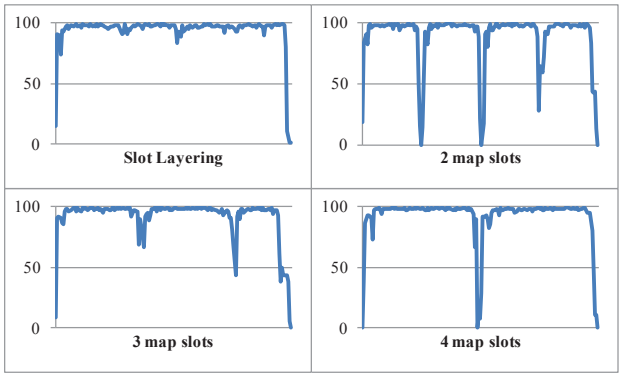


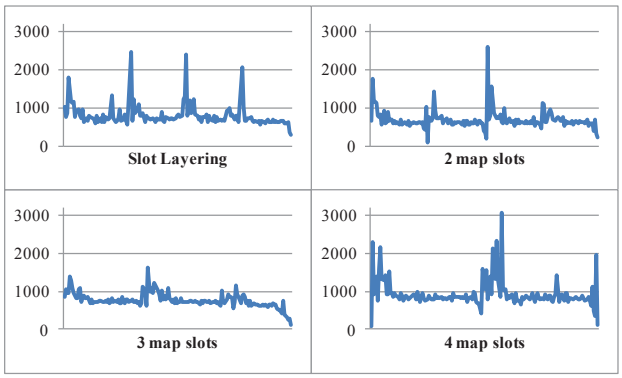
Figure 3.8: Comparison of executing a WordCount job with different static slot configurations and non-intrusive slot layering.

3.5.3 Benefit of task slots

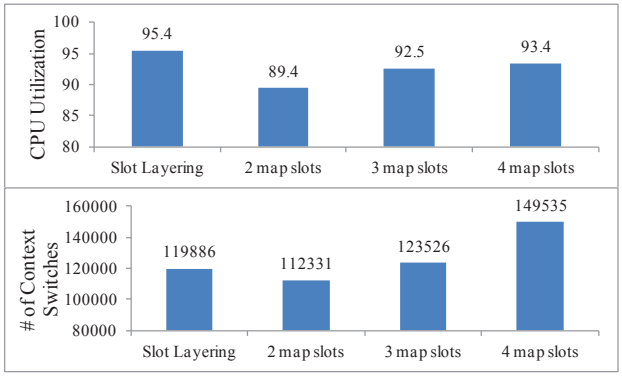
Coexistence of Map and Reduce slots easily produces wrong configuration to represent the resource capacity of a worker and accordingly decreases overall system throughput. As shown in Figure 3.10, we configure 2 Map slots and 2 Reduce slots for a worker and submit a Wordcount and TeraSort job in order, including 2 Map tasks and 2 Reduce tasks for each of them. If we assume 4 concurrently running tasks is the best situation to maximize resource utilization on the worker, area B in Figure 3.10 is an ideal case to take fully advantage of the capacity of the worker. However, in area A, although part of resource is idle, TeraSort Map tasks still have to wait. It is because there are only two Map slots occupied by two WordCount Map tasks. On the other hand, if we assume 2 concurrently running tasks is the best situation, area A becomes the ideal case while area B possibly over-utilizes the resource since there are 4 simultaneously running tasks. Apparently, the current approach based on separate Map and Reduce slots could represent inaccurate resource capacity in some cases and thus results in performance decrease. In our work, we combine Map and Reduce slots as task slots to run both Map and Reduce tasks. It can properly represent the resource capacity in all of the above situations.



(a) CPU utilization.



(b) Number of context switches.



(c) Summaries for average CPU utilization and the number of system context switches in total.

Figure 3.9: Comparisons of CPU utilization and context switches among different static slot configurations and non-intrusive slot layering.

Table 3.4: Map task execution time with different static slot configurations and non-intrusive slot layering.

Configuration		Map Task Execution Time (Sec)
2 map slots		42
3 map slots		60
4 map slots		80
4 map slots	2 Active Slots	45
	2 Passive Slots	75

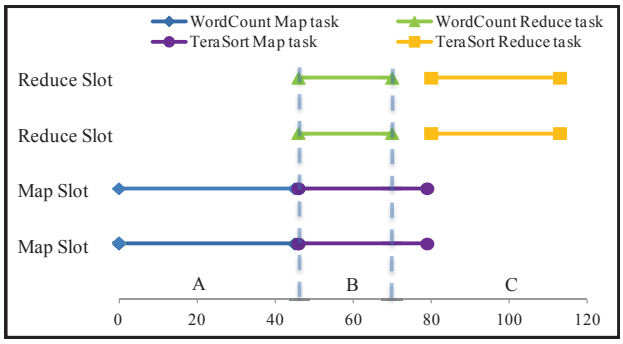


Figure 3.10: Slot usage when running WordCount and Terasort

3.6 Conclusions

In this chapter we have proposed a non-intrusive slot layering approach that aims to improve both overall throughput and high-priority job performance. Its slot layering manager efficiently isolates resource for slots in Active and Passive tiers. Resource is no longer fairly shared by running tasks. Instead, high-priority tasks take needed resource to perform as if they run in dedicated resource while the unused resource by those high-priority tasks is utilized by low-priority tasks. Such a way can improve overall resource utilization as well as resolve limitations in the original Hadoop design.The layering-aware scheduler in our approach further helps leverage the improvement delivered by our non-intrusive slot layering approach taking into account job priority and data locality. Our experiments in two different cluster sizes with representative MapReduce jobs have validated our goals of efficient overall resource use

and high-priority job performance improvement.

Chapter 4

Local Resource Shaper for MapReduce

4.1 Introduction

The underutilization of resources remains a major issue in computer systems. The term "resource consumption shaping" was originally coined by James Hamilton [25] 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" [26]. The fact that resource utilization in data centers is usually lower than 10% [27] promises great potential for resource shaping in reducing the amount of required resources.

In this chapter, we tackle the problem of shaping resource consumption at each individual node. We identify peaks and valleys 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 overall job duration.

Our solution to this problem is called *Local Resource Shaper (LRS)*. LRS interlaces the resource usage of multiple workloads to maximize resource utilization with low resource contention. LRS is a novel resource management solution in the following ways: (1) LRS tackles a different problem from global resource consumption shaping. It aims at reducing resource contention, rather than resource usage. By contrast, it makes sense to lower peaks at internet scale to reduce, for example, the power consumption of a data center. (2) The main novelty of LRS lies in its differentiation of slots, rather than in its scheduler, and is aimed at speeding up the execution of tasks within a single job, whereas others [28, 29, 30, 19, 6] guarantee fair resource sharing. (3) As opposed to reactive solutions [31, 32, 33] that react to resource contention *a posteriori* by migrating the load, LRS takes a preventive approach by minimizing resource contention.

We illustrate LRS and its new MapReduce scheduler (*Interleave*) by implementing them in the Hadoop framework. We first demonstrate LRS capability without Interleave by combining it with three well-known Hadoop schedulers: FIFO, Fair, and

Table 4.1: A summary of the 6 MapReduce benchmarks.

#	Benchmark	Resource	Description
1	Grep	CPU-bound	Search text matching regular expression
2	PiEst	CPU-bound	Estimate Pi using Monte Carlo method
3	WordCount (WC)	Moderate CPU	Count words in the input file
4	Crypto (Crpt)	Moderate CPU	Decrypt cipher text in the input file
5	Sort	I/O-bound	Sort input data
6	TeraSort (TS)	I/O-bound	Sort input data

Capacity. This implementation deals with two slot priorities: *Active* and *Passive*, the latter being able to use resources only when the former is not using them. Slots can be viewed as the container of a single task. As MapReduce tasks typically consume more than 50% of a CPU resource [28], this simple two-tier solution is enough to fully leverage the resources. Hence, Active/Passive slots can effectively deal with the tradeoff between resource utilization and resource contention.

We then incorporate Interleave as a complementary MapReduce scheduler to leverage the Active/Passive slots differentiation. This scheduler adopts a dual-purpose 'task slot', which serves as a container for interchangeable map and reduce tasks. Interleave implements two components, a slot manager and a task dispatcher. The slot manager is in charge of adaptive allocation of Passive slots when it detects spare resources at runtime, while the task dispatcher implements a scheduling algorithm that exploits the Active/Passive differentiation with the consideration of task slots.

We have conducted an extensive analysis of MapReduce to evaluate LRS. Our platform consists of a Hadoop cluster of 11 nodes in Amazon EC2. We have compared LRS against existing Hadoop alternatives on the six MapReduce benchmarks depicted in Table 4.1. These benchmarks are specialized in text retrieval, decryption, sorting, scientific computation, etc., and all are taken from the MapReduce literature [21, 19,

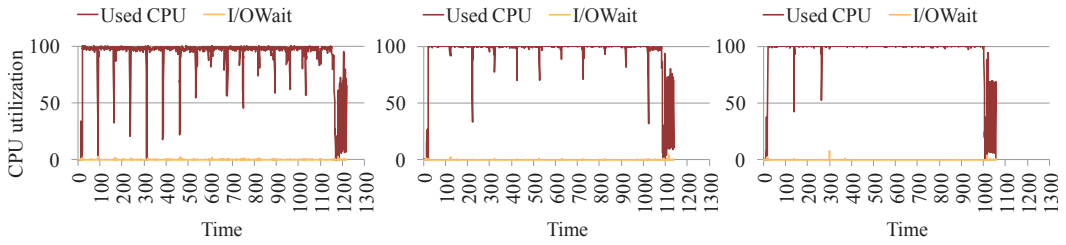
34, 20]. In addition, we have also compared LRS to a recently proposed Hadoop scheduler, called Delay [6], in treating a Facebook workload. Our results indicate that LRS improves these Hadoop-based alternatives in three main ways:

1. **Increasing CPU usage.** LRS allows us to achieve CPU utilization of up to 89% when considering both system and user CPU times. Even without Interleave, LRS (with the default Hadoop FIFO scheduler) still achieves an average CPU utilization of 85% which remains higher than the peak CPU utilization one could obtain without LRS with any of the three Hadoop schedulers.

2. **Lowering I/O contention.** Our MapReduce scheduler, Interleave, exploits the Active/Passive slots differentiation to reduce I/O contention by filling the valleys where I/O do not occur. This reduces the time each task spends waiting on I/O. Specifically, LRS benefits from Interleave by halving the I/O wait time of Hadoop.

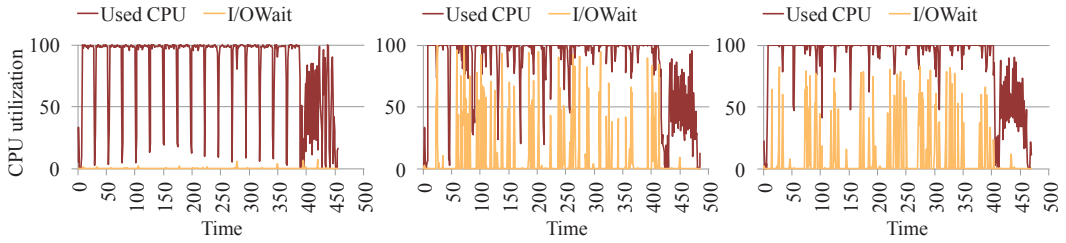
3. **Reducing job duration.** We have experimentally tested LRS against Delay [6], which was shown to perform well under a Facebook workload. We thus have evaluated the job completion time using LRS against Hadoop using Delay and observed that LRS reduces the job duration by up to 20% under the Facebook workload.

An interesting conclusion of our work is that the constraints of Hadoop slot configuration seemingly impact performance. We have tested all possible static configurations of map/reduce slots, as recommended by Yahoo! [35], and have observed a performance variation of 22% based on the configuration the user could choose, hence motivating the search for the best configuration. LRS relieves the programmer from the burden of finding such a best configuration. We have also observed that LRS, with the adoption of task slot, always outperforms the best static slot configuration both in terms of resource utilization and performance. In that respect, our work tends to support the recent attempt of developers to trade map/reduce slots for containers [36].



(a) 4m4r (exec: 1279) (b) 6m6r (exec: 1198) (c) 8m8r (exec: 1118)

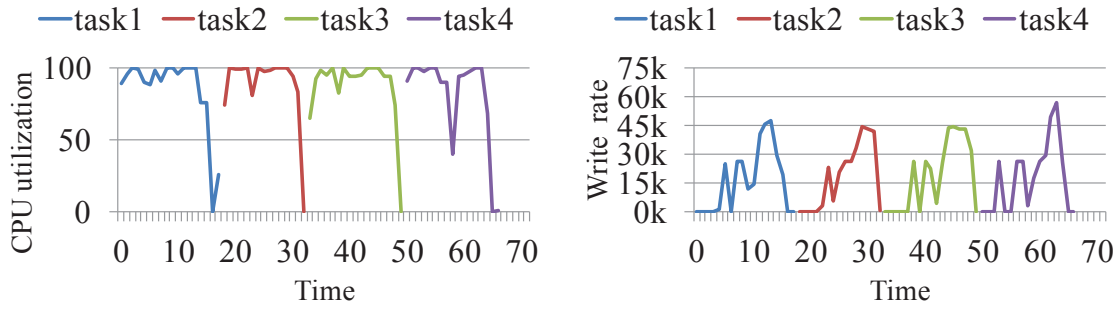
Figure 4.1: CPU utilization for Grep with different slot configurations. Execution times (in seconds) shown in parentheses. CPU resource utilization towards the end is deteriorating and heavily fluctuating because reduce tasks mostly complete their execution in a short time and only one reduce task is assigned in a scheduling cycle.



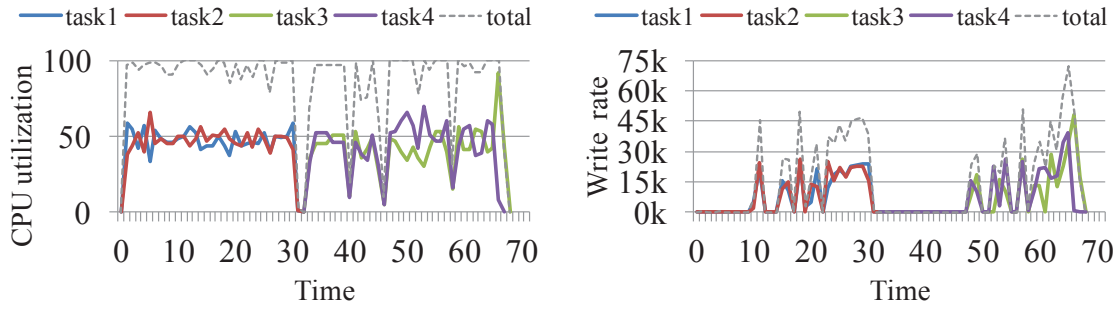
(a) 4m4r (exec: 464) (b) 6m6r (exec: 496) (c) 8m8r (exec: 479)

Figure 4.2: CPU utilization for WordCount with different slot configurations.

The rest of this chapter is organized as follows. Section 4.2 describes issues in fair resource sharing that motivate our work on shaping local resource consumption. Section 4.3 presents LRS and describes its implementation in Hadoop. In Section 4.4, we evaluate LRS, with and without Interleave, against existing alternatives. We discuss related work in Section 4.5 and present the conclusions in Section 4.6. Appendix A depicts the resource utilization and job duration of the benchmarks omitted in Section 4.2.



(a) tasks running sequentially (exec: 67)



(b) 2 concurrent tasks per core (exec: 69)

Figure 4.3: Resource usage patterns of WordCount. Write rate is in bytes.

4.2 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 benchmarks from Table 4.1. 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 and defer the remaining results to Appendix A.

Hadoop is an open-source implementation of MapReduce that follows the master/slave paradigm where the master machine (JobTracker) executes a job by scheduling its different *tasks*, or sub-processes, and a set of slave machines (TaskTrackers) manage resources and perform tasks based on fair resource sharing. The default scheduler in Hadoop uses a FIFO queue to dispatch tasks to slots. 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, and the number of slots is statically configured before launching the Hadoop system. Unfortunately, adequately making such a static choice remains an open problem [35].

Following Yahoo!'s recommendation of choosing the number of slots between half and twice the number of cores [35] , we perform experiments using the Hadoop FIFO scheduler with three distinct configurations: 4m4r (4 map slots and 4 reduce slots), 6m6r, and 8m8r. Figure 4.1 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 4.2 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.

To confirm our contention hypothesis, in Figure 4.3 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 4.3(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 4.2(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

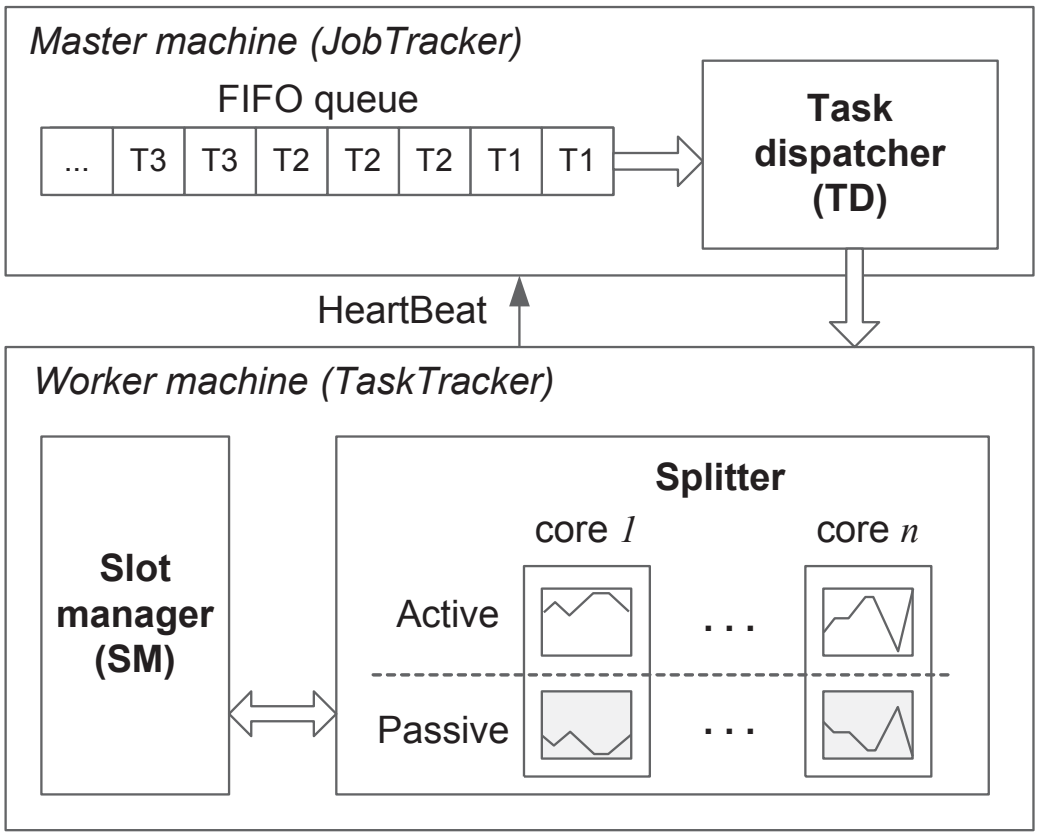


Figure 4.4: The architecture of LRS.

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 chapter, we simply selected the default values for the parameters.

4.3 The Local Resource Shaper

In this section, we present LRS (Figure 4.4) 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.

4.3.1 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 [28].

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 chapter 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 [37], Mantri [34] and Sailfish [38]. 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.

4.3.2 The Interleave MapReduce Scheduler

The Interleave scheduler implements a slot manager (SM) and a task dispatcher (TD) on top of Splitter (Figure 4.4), with the adoption of a 'task slot'. As the coexistence of map and reduce slots leads to resource contention when both map and reduce tasks are running concurrently on a core, we merge the map slot and reduce slot into an undifferentiated and dual-purpose task slot. The incorporation of task slots with LRS helps eliminate such resource contention. A task slot takes any task at a time, regardless of task type (map or reduce). 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 4.3.1. 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.

4.3.2.1 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:

$$N = \begin{cases} \lfloor \frac{1-CPU^{used}}{CPU^{used} * \#cores/Slot^{MAX}} \rfloor & \text{if } CPU^{eff} < 0.9 \ \wedge \ IO^{wait} \leq T \\ 1 & \text{if } IO^{wait} > T \end{cases}$$

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 [1, 11]

4.3.2.2 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 2. 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

Algorithm 2: LRS-aware Task Dispatcher

```
When a heartbeat is received from worker n:
/* Reduce task scheduling */
if n has free Active/Passive slots then
    for j in jobs do
        if j has unassigned reduce task t then
            assign t on n
/* Map task scheduling */
/* Stage 1: assigning map tasks to Active slots */
for slot in Active slots do
    for j in jobs do
        if j has unassigned map task t then
            assign t on n
/* Stage 2: assigning map tasks to Passive slots */
if no map task is assigned to Active slots in this scheduling cycle then
    for slot in Passive slots do
        for j in jobs do
            if j has unassigned map task t with data on n then
                assign t on n
    for slot in Passive slots do
        for j in jobs do
            if j has unassigned map task t then
                assign t on n
```

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.

4.4 Evaluation

In this section, we evaluate LRS extensively with five different schedulers (three Hadoop built-in schedulers, Delay [6] and our own Interleave scheduler), and under seven different benchmarks. Each of the first six benchmarks has been previously used

to evaluate MapReduce [21, 19, 34, 20]. The last benchmark is based on a workload from Facebook [6].

In Section 4.4.1, we show that LRS, even without our Interleave scheduler,[1] addresses our motivating problem by shaping resource consumption. In Section 4.4.2, we observe that this resource shaping translates into performance improvements regardless of the underlying scheduler used. In Section 4.4.3, we measure how our LRS (with Interleave scheduler from Section 4.4.3 onward) further reduces the I/O utilization. In Section 4.4.4, we show that LRS effectively alleviates the need for manual slot configuration. Finally, in Section 4.4.5, we compare LRS to a solution that was proved efficient in handling Facebook workloads [6].

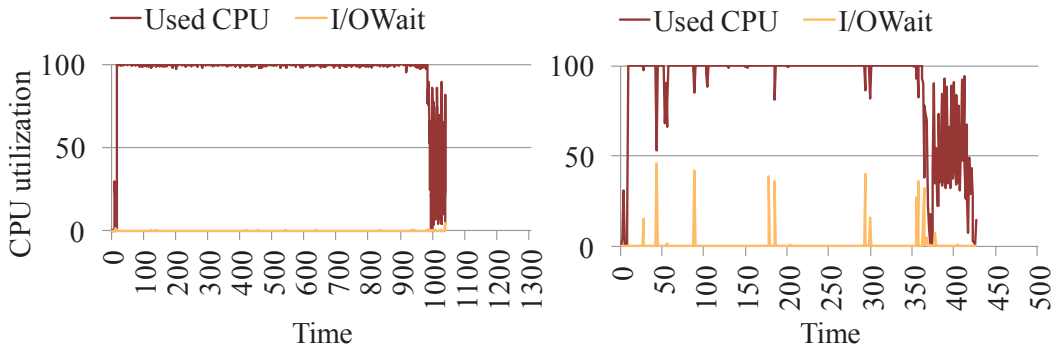
We performed all our experiments on a Hadoop cluster consisting of 11 EC2 m1.xlarge instances. Each instance has four cores, 15 GB RAM, and is running Hadoop-1.0.0 with a block size of 64MB. The cluster was configured such that one node is dedicated to run JobTracker and NameNode, and each of the remaining 10 nodes hosts a TaskTracker and a DataNode. Based on the empirical rule provided in [35], 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.

4.4.1 Shaping Resources with Active/Passive Slots

To observe the effect of local resource consumption shaping, we reproduce the same motivating experiments of Figures 4.1 and 4.2 but with our LRS solution. As Splitter essentially enables such shaping, we simply integrate it with the FIFO scheduler in Hadoop, i.e., LRS^{FIFO}. In the rest of this section, we refer to LRS as LRS^{FIFO}.

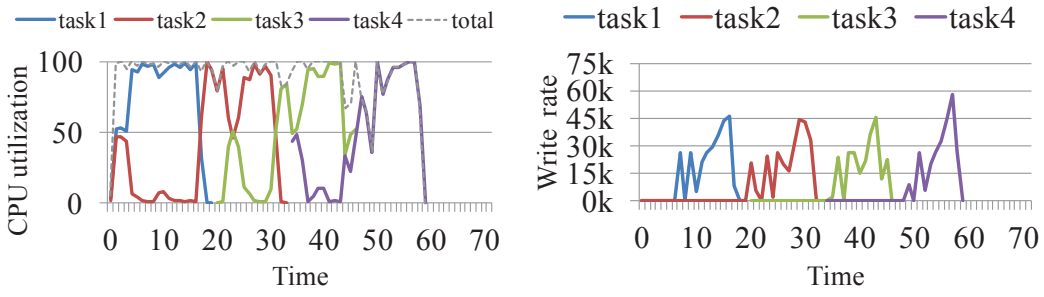
The results are depicted in Figures 4.5 and 4.6. All results for the single node experiment with concrete values are presented in Figure 4.7. These results show that Splitter alone substantially improves resource utilization.

[1] We use the notation LRS^{FIFO} to denote LRS when it uses the Hadoop FIFO scheduler (instead of



(a) Grep (exec: 1090) (b) WordCount (exec: 435)

Figure 4.5: CPU utilization using LRS^{FIFO}.

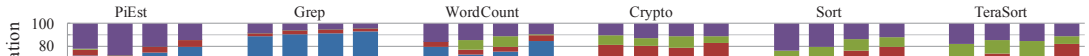


(a) CPU utilization (b) Write rate

Figure 4.6: Resource usage pattern of WordCount when running two tasks concurrently on a single core using LRS^{FIFO}. In comparison with resource usage patterns based on fair resource sharing (Figure 4.3(b)), resource consumption using LRS^{FIFO} is well shaped resulting in performance improvement (job execution times: 69 vs. 60).

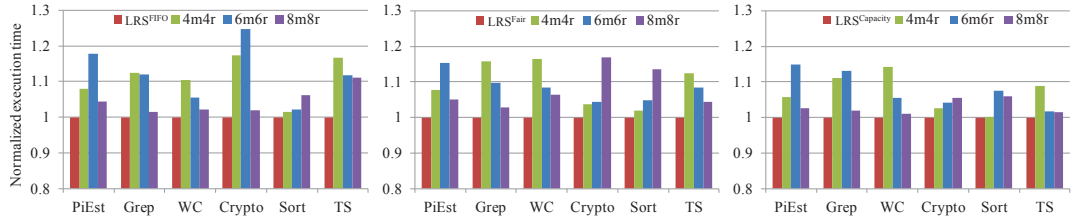
An immediate observation is that LRS maximizes CPU resource utilization (Figure 4.5); i.e., effective CPU utilization is 95.71% for Grep and 89.15% for WordCount (Figure 4.7). In particular, LRS utilizes CPU resources similarly to the 8m8r configuration for a CPU-bound application (8m8r has 94.42% effective CPU utilization for Grep). By contrast, LRS always exploits two different slots per core (Active and Passive), and thus ensures maximum CPU resource utilization in the general case. Note that more than two slots per core would not bring much CPU utilization improvement

Interleave).



	PiEst				Grep				WordCount				Crypto				Sort				TeraSort			
	4m4r	6m6r	8m8r	LRS^{FIFO}	4m4r	6m6r	8m8r	LRS^{FIFO}	4m4r	6m6r	8m8r	LRS^{FIFO}	4m4r	6m6r	8m8r	LRS^{FIFO}	4m4r	6m6r	8m8r	LRS^{FIFO}	4m4r	6m6r	8m8r	LRS^{FIFO}
exec	177	192	177	164	1279	1198	1118	1090	464	496	479	435	412	425	439	393	349	330	308	292	461	426	472	384
idle	22.8	28.4	20.5	14.9	9.23	6.66	5.53	4.26	16.4	14.6	11.6	9.99	10.6	12.7	11.5	11.2	24.2	20.4	13.9	12.6	17.8	15.0	15.6	11.0
iowait	0.05	0.03	0.03	0.01	0.04	0.02	0.02	0.01	0.11	9.07	9.01	0.86	8.45	6.95	10.4	5.88	12.8	9.22	10.1	8.30	12.3	11.6	15.7	7.27
system	5.10	4.81	5.47	5.60	2.44	2.78	3.12	3.01	4.64	4.16	4.51	4.95	18.3	21.1	19.5	18.6	15.6	18.0	20.2	19.8	15.1	15.6	15.4	16.9
user	72.0	66.7	73.9	79.4	88.2	90.5	91.3	92.7	78.8	72.1	74.8	84.2	62.5	59.1	58.4	64.2	47.2	52.2	55.6	59.2	54.6	57.6	53.2	64.7

Figure 4.7: CPU utilization for 6 jobs running on a single node. Comparisons are made between plain FIFO (with three static configurations) and LRS^{FIFO}. While 'exec' in the figure indicates execution times in seconds, the rest are CPU usage for idle, iowait, system and user times, respectively.



(a) FIFO scheduler (b) Fair scheduler (c) Capacity scheduler

Figure 4.8: Normalized job execution time comparisons for different schedulers. Job execution times are normalized due to their large differences between different benchmarks. The actual execution times can be found in the data table in Figure 4.7.

as it is known that a MapReduce task generally exploits more than half of the CPU resource [28].

On non-CPU-bound applications, LRS tends to obtain higher CPU resource utilization than the 8m8r slot configuration. More precisely, on WordCount, the 8m8r configuration only achieves 79.31% effective CPU utilization while LRS achieves 89.15%. The reason for this disparity is that, when LRS is not used, the adequate number of slots to use while ensuring fair resource sharing changes depending on various parameters, such as the type of running tasks and the size of input data, and thus creates valleys in CPU utilization (Figure 4.2).

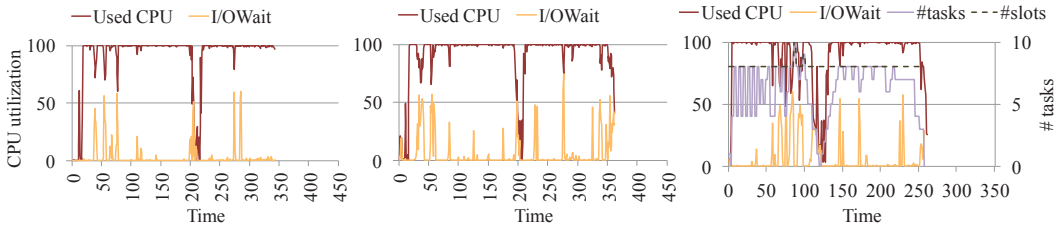
By contrast, LRS achieves high CPU utilization while incurring a low amount of

resource contention. In particular, we can see in Figure 4.5 that the I/O wait duration with LRS remains lower in both experiments than in the motivating Section 4.2, regardless of the chosen slot configuration.

To better illustrate that CPU utilization valleys may arise from I/O resource contention, Figure 4.6 depicts the CPU and disk resource utilization of a single core running WordCount (cf. Figure 4.3 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 4.6(a); as expected, this harmonious shape contrasts significantly with the disharmony present without LRS (Figure 4.3(b)).

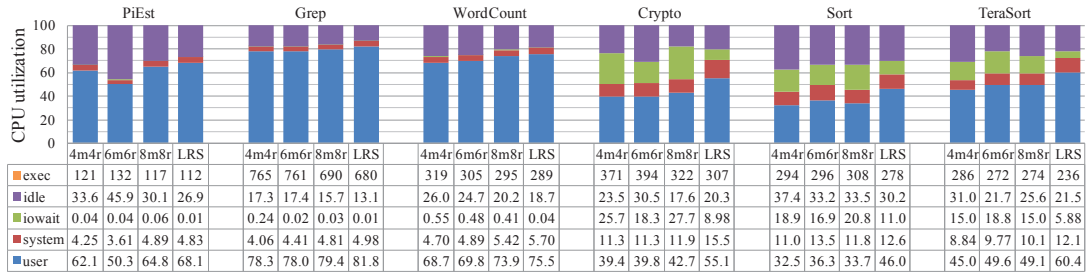
LRS also shapes I/O resource consumption in the same way as CPU consumption. In fact, this I/O resource shaping allows LRS to decrease the portion of CPU time spent waiting for I/O from 9.07% with a 6m6r configuration, to 0.86%. Thus, LRS helps minimize the contention of simultaneous disk writes as depicted in Figure 4.6(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 4.7).



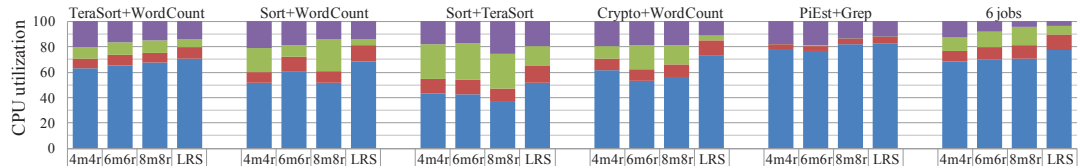
(a) Crypto (b) TeraSort (c) Sort

Figure 4.9: CPU utilization using LRS (with Interleave). The adaptive passive slot allocation of SM is shown in Figure 4.9(c). The maximum values on x-axes are intentionally set to 450, 450 and 350 for effective comparisons with other figures in Appendix A.



	PiEst				Grep				WordCount				Crypto				Sort				TeraSort			
	4m4r	6m6r	8m8r	LRS	4m4r	6m6r	8m8r	LRS	4m4r	6m6r	8m8r	LRS	4m4r	6m6r	8m8r	LRS	4m4r	6m6r	8m8r	LRS	4m4r	6m6r	8m8r	LRS
exec	121	132	117	112	765	761	690	680	319	305	295	289	371	394	322	307	294	296	308	278	286	272	274	236
idle	33.6	45.9	30.1	26.9	17.3	17.4	15.7	13.1	26.0	24.7	20.2	18.7	23.5	30.5	17.6	20.3	37.4	33.2	33.5	30.2	31.0	21.7	25.6	21.5
iowait	0.04	0.04	0.06	0.01	0.24	0.02	0.03	0.01	0.55	0.48	0.41	0.04	25.7	18.3	27.7	8.98	18.9	16.9	20.8	11.0	15.0	18.8	15.0	5.88
system	4.25	3.61	4.89	4.83	4.06	4.41	4.81	4.98	4.70	4.89	5.42	5.70	11.3	11.3	11.9	15.5	11.0	13.5	11.8	12.6	8.84	9.77	10.1	12.1
user	62.1	50.3	64.8	68.1	78.3	78.0	79.4	81.8	68.7	69.8	73.9	75.5	39.4	39.8	42.7	55.1	32.5	36.3	33.7	46.0	45.0	49.6	49.1	60.4

Figure 4.10: CPU utilization for 6 jobs running on a cluster.



	TeraSort+WordCount				Sort+WordCount				Sort+TeraSort				Crypto+WordCount				PiEst+Grep				6 jobs			
	4m4r	6m6r	8m8r	LRS	4m4r	6m6r	8m8r	LRS	4m4r	6m6r	8m8r	LRS	4m4r	6m6r	8m8r	LRS	4m4r	6m6r	8m8r	LRS	4m4r	6m6r	8m8r	LRS
exec	541	529	523	483	535	521	574	479	522	557	619	502	555	551	558	505	859	857	769	758	1760	1844	1864	1648
idle	20.1	16.6	15.1	14.3	20.9	18.8	13.9	14.3	17.7	17.1	25.3	19.2	19.6	18.3	18.6	11.3	17.6	18.4	13.0	12.1	12.9	8.09	4.17	3.81
iowait	9.18	9.50	9.30	5.83	18.7	9.10	25.3	4.67	27.3	28.8	27.4	14.9	9.69	19.6	14.9	3.67	0.07	0.65	0.02	0.01	10.5	12.3	14.2	6.39
system	7.60	8.07	8.24	8.88	8.68	10.8	8.84	12.3	11.8	11.4	10.0	14.2	9.38	8.79	10.0	12.3	4.12	4.28	5.05	5.14	8.26	9.89	10.9	12.0
user	63.0	65.7	67.3	70.9	51.6	61.1	51.8	68.6	43.0	42.5	37.1	51.4	61.3	53.2	56.4	72.6	78.1	76.6	81.8	82.7	68.2	69.6	70.6	77.8

Figure 4.11: CPU utilization for job combinations running on a cluster.

4.4.2 Boosting Performance of Existing Schedulers

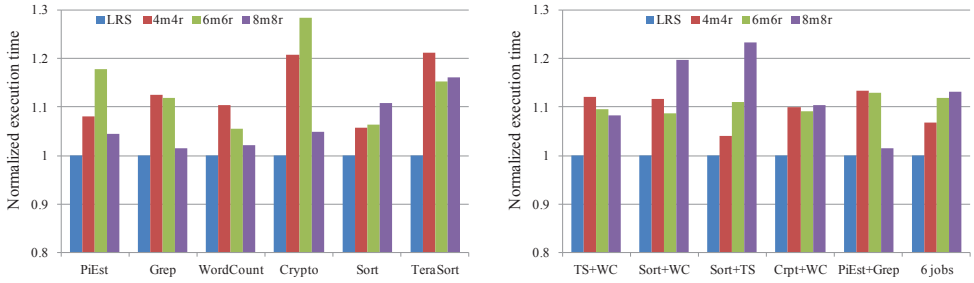
In this section, we show that the core component of LRS (Splitter) is complementary to its scheduler. To this end, we incorporate three state-of-the-art Hadoop schedulers into LRS: the FIFO scheduler, the Fair scheduler and the Capacity scheduler. Since

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 jobs in Table 4.1. We compare job execution time using Splitter to manage resources with 3 optimal configurations based on the number of cores. Results (Figure 4.8) 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.

4.4.3 An LRS-Specific Scheduler to Limit I/O Contention

For Crypto and the I/O-bound jobs in Table 4.1, part of the unused CPU resources caused by resource contention still exist when using LRS's core resource shaping component, Splitter (please refer to Appendix A for details). The Interleave scheduler is used to alleviate this by supplementing LRS with its slot manager and task dispatcher (Figure 4.9). 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 (see LRS^{FIFO} results in Appendix A), and thus are not presented.

In Figure 4.9(c), 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 still 8 because the number of currently running tasks is



(a) single job (b) combination

Figure 4.12: Normalized execution time comparisons for jobs running on a cluster. The actual execution times can be found in the data tables in Figures 4.10 and 4.11, respectively.

less than the maximum number of slots. However, the number of concurrently running tasks reaches 10 and 9 from 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.

4.4.4 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 4.4.1, 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 4.10, 4.11, and 4.12. 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 (LRS^{FIFO}) 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

Table 4.2: Distribution of benchmark jobs.

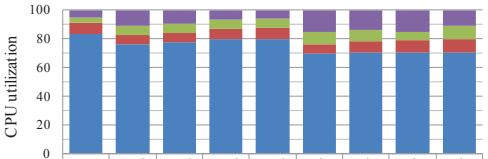
# maps	%	# benchmarks (# maps)
1-2	54%	8 WordCount (1) 6 TeraSort (2)
3-20	14%	2 Sort (8) 2 WordCount (16)
21-150	15%	1 Crypto (80) 2 TeraSort (120)
151-300	6%	1 WordCount (240)
301-500	4%	1 PiEst (400)
>500	7%	1 TeraSort (520) 1 Grep (640)

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 4.4.1 (Figures 4.5 and 4.6) 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.

(a) Normalized execution time



	LRS	FIFO 4m2r	FIFO 4m4r	FIFO 6m6r	FIFO 8m8r	Delay 4m2r	Delay 4m4r	Delay 6m6r	Delay 8m8r
exec	1998	2230	2218	2168	2130	2357	2514	2491	2343
idle	5.17	10.94	9.73	7.02	6.27	15.59	13.73	14.97	10.84
iowait	3.48	6.51	6.22	6.18	6.11	7.94	7.71	6.12	9.26
system	7.96	6.59	6.82	7.38	7.91	7.02	7.95	8.32	9.33
user	83.39	75.96	77.23	79.42	79.71	69.45	70.61	70.59	70.57

(b) CPU utilization

Figure 4.13: Facebook workload results.

4.4.5 When Running the Facebook Workload Model

In this experiment, we evaluate LRS through a set of benchmarks based on the workload trace from Facebook, which was reported in [6]. We scaled down the total number of jobs based on our cluster's scale and generated a job submission scheduler of 25 jobs. According to the Facebook trace, the distribution of job inter-arrival times was roughly exponential with a mean of 14 seconds. This makes our submission schedule 373 seconds long. The 6 benchmark jobs are mixed with different job input sizes (64 MB input block for a map task) and the job input sizes was generated based on the Facebook workload model. Table 4.2 lists the number of map tasks per job in the Facebook workload trace, the percentage of the total jobs, benchmark name and the actual number of running benchmarks.

We compare LRS (with Interleave) against the FIFO scheduler and the Delay scheduler [6]. Besides the optimal slot configuration range we used before, we added a 4m2r configuration according to the original configuration [6]. The results are shown in Figure 4.13. LRS outperforms these two schedulers with all configurations. More precisely, it decreases jobs execution time by 12% on average and by up to 20% compared with the Delay scheduler with 4m4r. Additionally, effective CPU utilization increases by 9% on average and I/O wait is about two times lower with LRS than with others.

4.5 Related Work

Efficient resource management has been studied for different purposes, such as maximizing resource utilization and minimizing resource contention [26, 27, 31, 32, 33, 39, 40, 41, 42, 19, 43, 21]. These results span across various granularities including data center, server, virtual machine (VM) and job level. Maximizing resource utilization is often sought by intensifying workload consolidation (concurrency), and thus tends to cause high resource contention and, in turn, performance degradation. The incompatibility of resource utilization and resource contention hinders the identification of the optimal concurrency level.

Resource consumption shaping was proposed as an extension to network-traffic shaping for data center utilization [26, 27]. 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.

VM placement and scheduling strategies (e.g., [31, 32, 33]) are probably the most common way to improve resource utilization. They essentially consolidate workloads/VMs in the way that the number of active servers is minimized. This consolidation is facilitated by the use of VM migration [39]. Previous works is still coarser grain (virtual machine monitor level) than ours. In the meantime, resource management approaches [40, 41, 42] are designed with the awareness of performance interference among co-located workloads. Unlike the preventive approach in our work, these works are concrete and reactive focusing on the exclusiveness and isolation of resource use between co-located applications by explicitly controlling resource usage. Unless the resource usage of co-located applications perfectly complement each other, when using previous solutions, resource contention and performance degradation is inevitable.

There were attempts to maximize resource utiliztion by profiling jobs in advance to find the resource bottleneck [44, 19, 45] and Cake [43] uses a two-level scheduling

scheme to dynamically adjusts the level of concurrency based on measured resource contention (device latency). In order to fully utilize one type of resource, they tend to face the underutilization of other resources. The WCO scheduler [21] combines workloads with different characteristics to reduce resource contention whereas Choosy [46] aims at satisfying job placement constraints. These previous results are all limited in their resource management capacity by fair resource sharing. By contrast, LRS enables multiple workloads to harmoniously share resources by non-uniformly interlacing their resource usage. This is markedly distinct from fair resource sharing.

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 [37, 6, 34, 38, 47, 28], 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 developers of Hadoop have recently decided to get rid of map/reduce slots [36]. The beta version of Hadoop 2.x does not aim at shaping local resources, but instead relies on the user to leverage "containers" appropriately. Map/reduce slots will most likely not be part of the next stable release of Hadoop in part because of the constraints they impose on schedulers.

The Capacity scheduler [37] supports job memory resource requirement. Jobs are able to be dispatched in a way to reduce memory interference between running tasks. The Delay scheduler [6] 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 [34] and Sailfish [38] achieve performance improvement by decreasing intermediate data transmission between map and reduce tasks to avoid network hotspots. Even automatic solutions [47] that tune Hadoop parameters to improve performance cannot disable fair resource sharing and existing resource allocation techniques, like DRF [28], 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.

4.6 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 7 MapReduce benchmarks and evaluating their performance with 4 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 [36] seems to confirm our observation as the stable release of Hadoop 2.x will seemingly get completely rid of map/reduce slots.

Chapter 5

Conclusions and future work

5.1 Summary and conclusions

This thesis focuses on application profiling and resource management in distributed systems. We identify and study a series of existing problems by investigating one of the most popular distributed systems - Hadoop. Meanwhile, solutions that overcome these problems are presented.

In Chapter 2, we find that the application scope of MapReduce has been extend beyond the original design goal which was large-scale date processing. Besides I/O intensive applications, more CPU intensive applications start to take advantage of MapReduce to utilize distributed computing resource for reducing execution time. Such application diversity reveals the possibility to improve the performance based on characteristics of applications. Therefore, we present the workload characteristic oriented scheduler. The scheduler with its characteristic estimation module dynamically and adaptively dispatches MapReduce applications in the way that tasks with complementing resource usage are co-located to improve overall performance. Our experimental results show it outperforms existing approaches.

In Chapter 3, we present a new way, non-intrusive slot layering, to manage resource at worker node level. It resolves two limitations derived from the original design in Hadoop: 1. the static slot configuration with dynamic resource usage of workloads results in inefficient resource utilization. 2. Job priority only exists in scheduling algorithms, not extending down to worker node level-resource of workers is fairly shared between high-priority/early-submitted tasks and low-priority/late-submitted tasks. These limitations reduce overall throughput as well as high-priority/early-submitted jobs performance. In our solution, we dynamically allocate resource at worker node level to improve resource utilization and give high-priority/early-submitted tasks as much resource as they need to reduce their execution time. Our experimental results obtained with six benchmark applications validate these objectives.

During our investigation in Chapter 3, we notice currently running tasks, especially from the same job, have the similar resource usage pattern. Fairly sharing resource between them could make them to reach their peak or valley of resource consumption at the same time that increase resource contention and decrease resource utilization. In Chapter 4, we propose Local Resource Shaper (LRS), a resource management technique (as an alternative to traditional fair resource sharing) that aims at maximizing resource utilization with minimal resource contention. LRS enables multiple workloads to non-uniformly and harmoniously interlace their resource usage on a single worker node. LRS is best suited for distributed systems that are concerned more about job-level performance than that of individual tasks. We implement LRS on top of Hadoop implementation of LRS and demonstrate its capability by integrating it with three well-known Hadoop schedulers: FIFO, Fair, and Capacity. LRS avoids the need for static slot configuration in Hadoop and always outperforms the best static slot configuration. We also develop the interleave scheduler to take full advantage of LRS. Experiments run in Amazon EC2 using six MapReduce benchmarks, with jobs based on the Facebook workload model, confirm that our solution improves both resource

utilization and performance.

5.2 Future directions

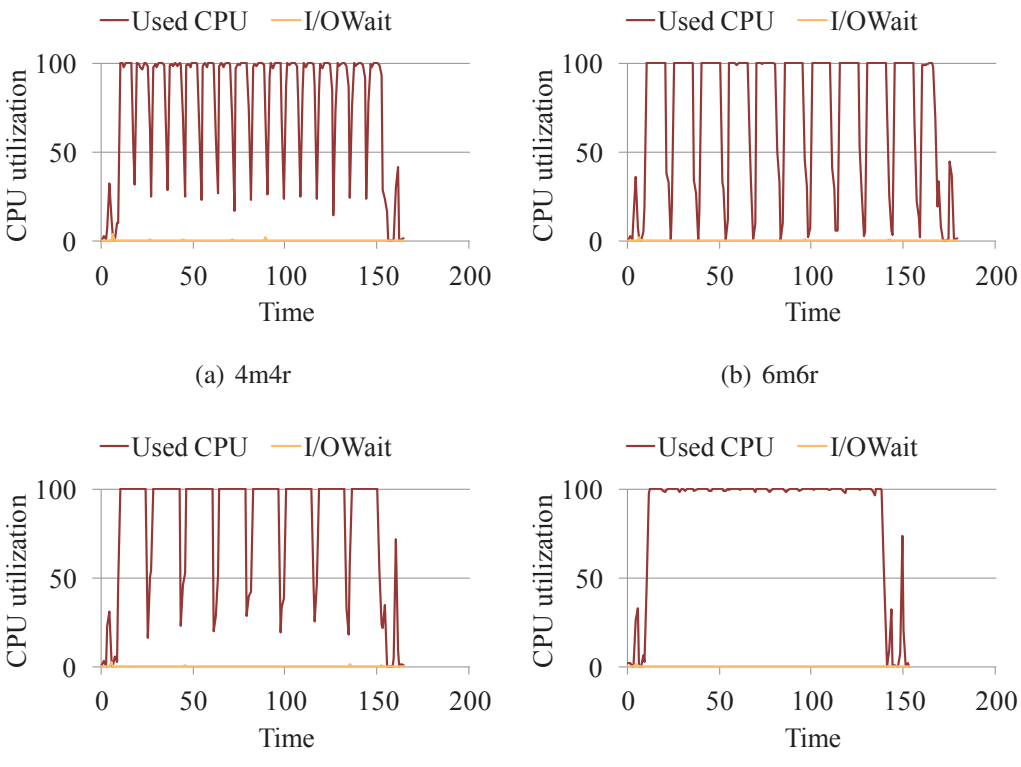
Throughout the project, we investigate a series of problems existing in application profiling and resource management. While this project scrutinized problems and developed a set of solutions to them, other issues remain, and there are opportunities for extending our advances.

The target clusters in Chapter 2 make assumption that all worker nodes are homogeneous. Although such an environment is easy to be established by virtual technologies or a cloud service provider like Amazon, this constraint still narrows down its applicability. Further research could take into account the cluster with hardware heterogeneity like GPU, Solid State Drives (SSD) to remove this constraint.

We develop and present two approaches for improvement of resource management in Chapter 3 and Chapter 4. There is still potential to extend them in following two ways: (1) Resource management in this thesis focuses on CPU and I/O resource. Extending it to other resource like network bandwidth and memory is able to further improve the system performance. (2) We implement and validate our solutions based on Hadoop clusters. Extending them to operating system kernel level is able to make other distributed systems to directly take advantage of the improvement we make.

Appendix A

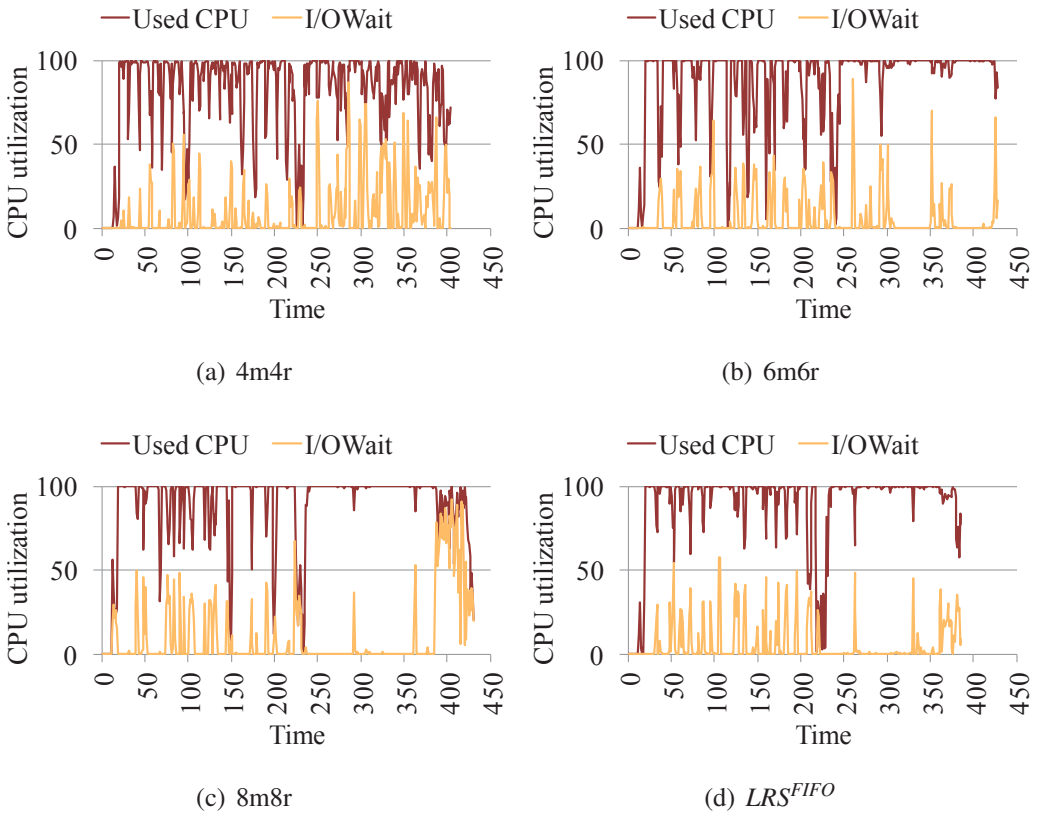
CPU Utilization with Different Slot Configurations and LRS



(a) 4m4r (b) 6m6r

(c) 8m8r (d) LRS^{FIFO}

Figure A.1: CPU utilization for PiEst.



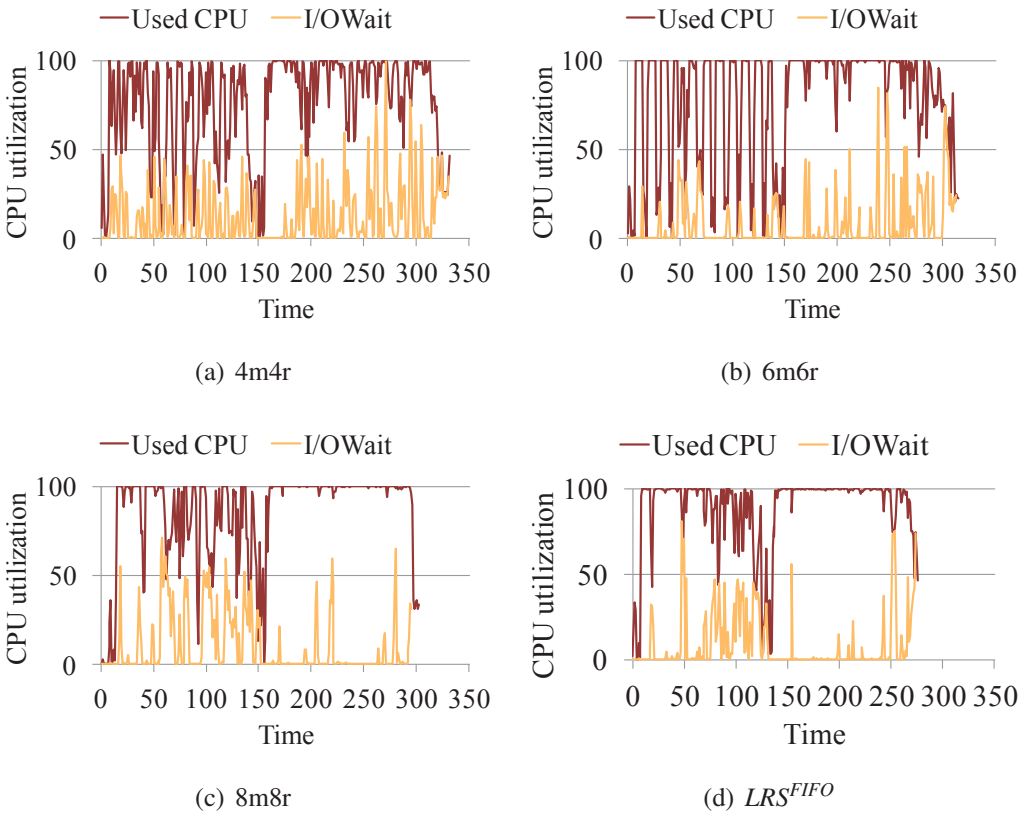
(a) 4m4r

(b) 6m6r

(c) 8m8r

(d) LRS^{FIFO}

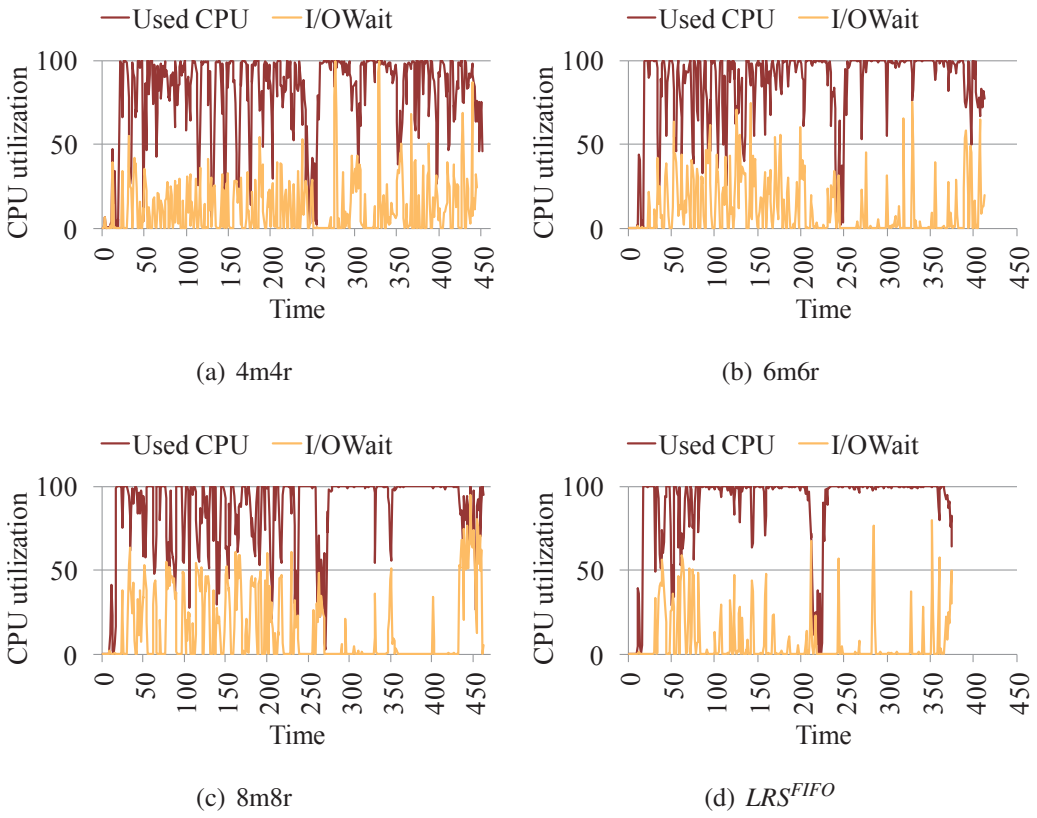
Figure A.2: CPU utilization for Crypto.



(a) 4m4r (b) 6m6r

(c) 8m8r (d) LRS^{FIFO}

Figure A.3: CPU utilization for Sort.



(a) 4m4r (b) 6m6r

(c) 8m8r (d) LRS^{FIFO}

Figure A.4: CPU utilization for TeraSort.

Bibliography

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *USENIX OSDI*, pp. 10–10, 2004. 1.1.1, 1.1.2, 2.1, 3.1, 4.3.2.1

[2] "Apache Hadoop." http://hadoop.apache.org/. [Online; accessed 2012]. 1.1.2, 3.1

[3] J. Ekanayake, S. Pallickara, and G. Fox, "Mapreduce for data intensive scientific analyses," in *IEEE Fourth International Conference on eScience*, pp. 277–284, 2008. 2.1

[4] C. Chu, S. Kim, Y. Lin, Y. Yu, G. Bradski, A. Ng, and K. Olukotun, "Map-reduce for machine learning on multicore," *Advances in neural information processing systems*, vol. 19, p. 281, 2007. 2.1

[5] J. Cohen, "Graph twiddling in a mapreduce world," *Computing in Science & Engineering*, vol. 11, no. 4, pp. 29–41, 2009. 2.1

[6] 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 *Proceedings of the 5th European conference on Computer systems*, pp. 265–278, ACM, 2010. 2.1, 2.2, 2.4.1.1, 3.2, 4.1, 4.1, 3, 4.4, 4.4.5, 4.5

[7] G. Lee, B. Chun, and R. Katz, "Heterogeneity-aware resource allocation and scheduling in the cloud," in *Proceedings of the 3rd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud¡¯11)*, 2011. 2.1, 2.2

[8] T. Sandholm and K. Lai, "Dynamic proportional share scheduling in hadoop," in *Job scheduling strategies for parallel processing*, pp. 110–131, Springer, 2010. 2.1, 2.2

[9] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, no. November, Citeseer, 2009. 2.2, 3.2

[10] J. Polo, D. Carrera, Y. Becerra, V. Beltran, J. Torres, and E. Ayguadé, "Performance management of accelerated mapreduce workloads in heterogeneous clusters," in *39th International Conference on Parallel Processing (ICPP2010)*, 2010. 2.2, 3.1

[11] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pp. 29–42, USENIX Association, 2008. 2.2, 4.3.2.1

[12] C. Tian, H. Zhou, Y. He, and L. Zha, "A dynamic mapreduce scheduler for heterogeneous workloads," in *Grid and Cooperative Computing, 2009. GCC'09. Eighth International Conference on*, pp. 218–224, IEEE, 2009. 2.2

[13] A. V. Do, J. Chen, C. Wang, Y. C. Lee, A. Y. Zomaya, and B. B. Zhou, "Profiling applications for virtual machine placement in clouds," in *Proceedings of IEEE CLOUD*, pp. 660–667, 2011. 2.2

[14] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini, "Cost analysis of java bytecode," in *Proceedings of the 16th European conference on Programming (ESOP)*, pp. 157–172, 2007. 2.2

[15] E. Albert, P. Arenas, S. Genaim, and G. Puebla, "Closed-form upper bounds in static cost analysis," *Journal of Automated Reasoning*, vol. 46, no. 2, pp. 161–203, 2011. 2.2

[16] J. Chen, L. John, and D. Kaseridis, "Modeling program resource demand using inherent program characteristics," in *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pp. 1–12, ACM, 2011. 2.2

[17] M. Schoeberl and R. Pedersen, "Wcet analysis for a java processor," in *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pp. 202–211, ACM, 2006. 2.2

[18] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pp. 41–51, IEEE, 2010. 2.6.1, 3.3.1

[19] J. Polo, C. Castillo, D. Carrera, Y. Becerra, I. Whalley, M. Steinder, J. Torres, and E. Ayguadé, "Resource-aware adaptive scheduling for mapreduce clusters," in *Proceedings of the 12th ACM/IFIP/USENIX international conference on Middleware*, Springer, 2011. 3.1, 3.2, 4.1, 4.1, 4.4, 4.5

[20] A. Verma, L. Cherkasova, and R. H. Campbell, "Resource provisioning framework for mapreduce jobs with performance goals," in *Proceedings of the 12th ACM/IFIP/USENIX international conference on Middleware*, Springer, 2011. 3.1, 3.2, 4.1, 4.4

[21] P. Lu, Y. C. Lee, C. Wang, B. B. Zhou, J. Chen, and A. Y. Zomaya, "Workload characteristic oriented scheduler for mapreduce," in *Proceedings of the 18th International Conference on Parallel and Distributed Systems*, IEEE, 2012. 3.1, 3.2, 4.1, 4.4, 4.5

[22] P. Menage, "Adding generic process containers to the linux kernel," in *Proceedings of the Linux Symposium*, 2007. 3.1

[23] "Capacity Scheduler Guide." http://hadoop.apache.org/docs/r0.20.2/capacity_scheduler.html. [Online; accessed 2012]. 3.2

[24] "Managing a Hadoop Cluster." http://developer.yahoo.com/hadoop/tutorial/module7.html. [Online; accessed 2012]. 3.4.1, 3.5.1

[25] J. Hamilton, "Internet-scale service efficiency," in *LADIS*, 2008. Keynote talk. 4.1

[26] "Resource Consumption Shaping." http://perspectives.mvdirona.com/2008/12/17/Resource ConsumptionShaping.aspx. 4.1, 4.5

[27] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The cost of a cloud: research problems in data center networks," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 1, pp. 68–73, 2008. 4.1, 4.5

[28] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: fair allocation of multiple resource types," in *USENIX NSDI*, 2011. 4.1, 4.1, 4.3.1, 4.4.1, 4.5

[29] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *USENIX NSDI*, 2011. 4.1

[30] P. Lu, Y. C. Lee, and A. Y. Zomaya, "Non-intrusive slot layering in hadoop," in *Proceedings of the 13th IEEE/ACM International Symposium on Cluster, Cloud and the Grid Computing*, pp. 253–260, 2013. 4.1

[31] F. Hermenier, X. Lorca, J.-M. Menaud, G. Muller, and J. Lawall, "Entropy: a consolidation manager for clusters," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pp. 41–50, 2009. 4.1, 4.5

[32] N. Bobroff, A. Kochut, and K. Beaty, "Dynamic placement of virtual machines for managing sla violations," in *Proceedings of the 10th IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pp. 119–128, 2007. 4.1, 4.5

[33] A. Verma, P. Ahuja, and A. Neogi, "pmapper: power and migration cost aware application placement in virtualized systems," in *ACM/IFIP/USENIX Middleware*, pp. 243–264, 2008. 4.1, 4.5

[34] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using mantri," in *USENIX OSDI*, pp. 1–16, 2010. 4.1, 4.3.1, 4.4, 4.5

[35] "Managing a Hadoop Cluster." `http://developer.yahoo.com/hadoop/tutorial/module7.html`. 4.1, 4.2, 4.4

[36] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, T. G. Robert Evans, J. Lowe, H. S. andSiddharth Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: Yet Another Resource Negotiator," in *ACM SoCC*, 2013. 4.1, 4.5, 4.6

[37] "Capacity Scheduler Guide." `http://hadoop.apache.org/docs/stable/capacity_scheduler.html`. 4.3.1, 4.5

[38] S. Rao, R. Ramakrishnan, A. Silberstein, M. Ovsiannikov, and D. Reeves, "Sailfish: A framework for large scale data processing," in *ACM SoCC*, p. 4, 2012. 4.3.1, 4.5

[39] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *USENIX NSDI*, pp. 273–286, 2005. 4.5

[40] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: managing performance interference effects for qos-aware clouds," in *EuroSys*, pp. 237–250, 2010. 4.5

[41] A. Verma, P. Ahuja, and A. Neogi, "Power-aware dynamic placement of hpc applications," in *ACM/IEEE Supercomputing*, pp. 175–184, 2008. 4.5

[42] S.-H. Lim, J.-S. Huh, Y. Kim, and C. R. Das, "Migration, assignment, and scheduling of jobs in virtualized environment," in *Proceedings of the 3rd USENIX conference on Hot topics in cloud computing*, pp. 2–2, 2011. 4.5

[43] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, and I. Stoica, "Cake: enabling high-level slos on shared storage systems," in *ACM SoCC*, 2012. 4.5

[44] R. C. Chiang and H. H. Huang, "Tracon: Interference-aware scheduling for data-intensive applications in virtualized environments," in *proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 47, 2011. 4.5

[45] B. Sharma, T. Wood, and C. R. Das, "Hybridmr: A hierarchical mapreduce scheduler for hybrid data centers," in *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*, pp. 102–111, 2013. 4.5

[46] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Choosy: max-min fair sharing for datacenter jobs with constraints," in *EuroSys*, pp. 365–378, 2013. 4.5

[47] S. Babu, "Towards automatic optimization of mapreduce programs," in *ACM SoCC*, pp. 137–142, 2010. 4.5