J Supercomput (2017) 73:3526–3546
DOI 10.1007/s11227-016-1653-7

SLA-aware energy-efficient scheduling scheme for Hadoop YARN

Xiaojun Cai[1] · Feng Li[1] · Ping Li[1] · Lei Ju[1] · Zhiping Jia[1]

Published online: 8 February 2016

Abstract Apache Hadoop becomes ubiquitous for cloud computing which provides resources as services for multi-tenant applications. YARN (a.k.a. MapReduce 2.0) is one of the key features in the second-generation Hadoop, which provides resource management and scheduling for large-scale MapReduce environments. Two enormous challenges in the YARN scheduler are the abilities to automatically tailor and control resource allocations to different jobs for achieving their Service Level Agreements (SLAs), and minimize energy consumption of the overall cloud computing system. In this work, we propose an SLA-aware energy-efficient scheduling scheme which allocates appropriate amount of resources to MapReduce applications with YARN architecture. In our task scheduling policy, We consider the data locality information to save the MapReduce network traffic. Furthermore, the slack time between the actual execution time of completed tasks and expected completion time of the application is utilized to improve the energy-efficiency of the system. An online userspace governor-based dynamic voltage and frequency scaling (DVFS) scheme is designed in the YARN per-application ApplicationMaster to dynamically change the CPU frequency for upcoming tasks given the slack time from previous completed tasks. Experimental evaluation shows that our proposed scheme outperforms the existing MapReduce scheduling policies in terms of both resource ultization and energy-efficiency.

Keywords Hadoop YARN · SLA · Resource allocation · DVFS · Task scheduling

✉ Lei Ju
julei@sdu.edu.cn

[1] School of Computer Science and Technology, Shandong University, Jinan, China

1 Introduction

MapReduce [1] is a data-driven programming model originally proposed by Google to handle large-scale web search applications. It allows automatic parallelization of tasks on a large cluster and has been proved to be especially well suited for distributed data-analytic applications. Therefore, an increasing number of companies are following the new trend of using MapReduce and its most popular open-source implementation Apache Hadoop [2], which is designed to be deployed on low-cost hardware, to offer cloud computing services. Although Hadoop has become one of the widely-adopted cluster computing frameworks for managing Big Data, it has some limitations in terms of scalability, reliability, and resource utilization. Hadoop proposed Yet Another Resource Negotiator (YARN, a.k.a. MapReduce 2.0) in 2013 to overcome such limitations [3]. YARN utilizes a global ResourceManager (RM) and per-application ApplicationMaster (AM) to replace the centralized JobTracker for resource management and job scheduling/monitoring. Furthermore, instead of separated map or reduce slots, it adopts a resource abstract called container for resource provisioning, which encapsulates multidimensional resources of a node including CPU and memory.

In a cloud computing environment, it is particularly important for service providers to meet the deadlines specified in the Service Level Agreements (SLAs). Furthermore, the ever-increasing and large-scale deployments of High-Performance Computing (HPC) systems bring in huge energy consumption and contribute to a significant electric bill, making the reduction of energy cost a high priority. According to the results in [4], the electricity consumption of worldwide data centers in 2012 was about 270 TWh, which corresponds to almost 2 % of the global electricity consumption and has an approximated annual growth rate of 4.3 %. So generally, the providers always expect to ensure a high level of adherence to the SLAs while taking up as little cost as possible.

Resource schedulers available in Hadoop YARN include the simple First-In-First-Out (FIFO) Scheduler, the Capacity Scheduler [5], and the Hadoop Fair Scheduler (HFS) [6]. The Capacity Scheduler allows multiple tenants to make the best possible use of a large cluster, governed by the constraints of allocated capacities. HFS is a method of assigning resources such that all applications can get an equal share of resources over time. However, the current YARN schedulers provide no completion time guarantee for individual MapReduce applications, and ignore the impact of resource provisioning on the system energy consumption. Over the last decades, many works on real-time and/or energy-efficient scheduling schemes for general distributed systems or the first generation of MapReduce have been proposed. However, many existing scheduling strategies may not be applicable to YARN environment due to the differences of system architecture and resource management mechanism; while some others may lead to sub-optimal scheduling and resource allocation decisions. For example, [7] shows that current CPU dynamic voltage and frequency scaling (DVFS) techniques fail to reflect respective design goal and may even become ineffective to manage the power consumption in Hadoop clusters.

In this paper, we propose a framework of resource management and scheduling in the YARN environment. For a given MapReduce application (job) and its profiling

information, the scheduler tries its best to allocate enough resources (containers) to meet the (soft) deadline specified in the application's SLA. Meanwhile, since many map/reduce tasks may complete earlier than the worst case completion time obtained in the profiling, we integrate a userspace governor-based DVFS controller into the YARN resource provisioning in order to utilize the slack time for system energy optimization. To the best of our knowledge, this is the first work on SLA-aware and energy-efficient scheduling for Hadoop YARN. The main contributions of this paper are as follows.

- We adopt the job profiling technique in [8] to characterize the performance of an application during its map, shuffle, and reduce phases. Instead of the average completion time for each task phase, we adopt the worst-case completion time during resource provisioning to achieve higher SLA conformance.
- We make the most of data locality in Hadoop for saving MapReduce network traffic, which can improve the performance of applications and produce more slack time. And our proposed resource scheduler can avoid accepting jobs that will lead to deadline misses and improve the cluster utilization.
- We design an online userspace governor-based DVFS scheme, which utilizes the slack time between the actual completion time during task execution and the estimated completion time for energy optimization.
- We have integrated the proposed framework into Hadoop 2.2.0 (with YARN) to obtain task profiling information with various CPU frequencies. Based on the profiling information, we use CloudSim [9] to simulate and evaluate the overall performance and energy consumption of the proposed scheduling scheme. Experimental results show that our framework leads to better SLA conformance and energy consumption compared with [8].

The remainder of the paper is organized as follows. We discuss the related work in Sect. 2 and give the research motivation in Sect. 3. In Sect. 4, we introduce the SLA-aware framework for Hadoop and its five interacting components. In Sect. 5, We present two algorithms for SLA-aware resource allocation and DVFS-based task scheduling, continuing with an evaluation of the proposed framework in Sect. 6. Finally, we discuss our future work and conclude the paper in Sect. 7.

2 Related work

Resource provision in MapReduce environments is a relatively new research topic, but it has already received much attention in the last few years. Polo et al. [10] introduced a new task scheduler that dynamically collects the performance data of MapReduce jobs and adjusts the resource allocation accordingly. But they only focus on the map phase and have no control over the reduce phase. Jockey [11] is designed for single job to maximize its economic utility while minimizing the impact on the data parallel clusters. While Jockey is effective at guaranteeing job latency, it lacks scheduling mechanism among multiple jobs. In [12], task dependencies between MapReduce jobs have been considered in the resource allocation to minimize the overall completion time of the application. Verma [8] proposed a framework, called ARIA, to estimate and allocate appropriate number of map and reduce slots for MapReduce applications so that they can meet their required deadlines. However, the job's actual execution

time in ARIA may exceed 7 % of the deadline because of inaccurate predictions and various uncertainties. Besides, all of these studies only support resource inference and allocation, and give no consideration to overtime budget and energy cost.

Hard real-time scheduling policies for multiprocessor systems have been well-studied in the past decade [13]. For heterogenous multi-core systems with Quality of Service (QoS) requirements, the execution time variation of tasks on different processing elements has been considered to achieve a optimal system resource allocation for general-purpose applications [14]. Furthermore [15], presents an evaluation framework which measures the robustness of heterogeneous multi-core scheduling policies if the input statistical execution time information is inaccurate. Driven by ever-increasing operating costs and awareness of energy conservation, many energy-aware approaches have been developed to improve the energy efficiency of a system. Although DVFS was originally designed for energy-efficient task scheduling on single processor platform [16,17], it was shown that clever scheduling of CPU power modes can save significant amounts of energy in parallel and distributed computing systems as well [18,19]. In [20], DVS (Dynamic Voltage Scaling) and DPM (Dynamic Power Management) schemes have been proposed for energy optimization of real-time steaming tasks on multiprocessor System-on-Chip. Wirtz and Ge [21] compared the energy efficiency among three DVFS scheduling policies for MapReduce framework. Their experimental results indicate that intelligent DVFS scheduling can achieve significant energy savings for computation intensive applications, and show that CPU Miser [22] works best for systems with large idle power as it seeks performance oriented energy savings by collecting fine grain CPU activity information. However, CPU Miser only supports setting the same frequency for all cores of a node and has to be run on every node in the cluster, which is inefficiency and a waste of resources.

Modern processors are usually equipped with the per-core DVFS technique, which enables each core of the processors to be operated at multiple frequencies under different supply voltages. The research in [23] shows that depending on the heterogeneity of workload characteristics, per-core DVFS offers substantial additional savings compared to global DVFS schemes by better adapting to the different requirements of each core. But to precisely control all of the CPU cores in a cluster is quite a challenging job. Another research in [7] investigates the impact of existing DVFS governors on the performance and energy consumption of a Hadoop cluster. It reveals that some CPU governors do not exactly reflect their design goal and may even become ineffective to manage the power consumption.

Our work differs from the previous studies in several respects. In this work, we take into account both resource provision and energy conservation, and give a clear framework for the trade-off between hardware resource and energy consumption in MapReduce environments. We apply our own strategy in adjusting the per-core frequency through the CPUFreq subsystem, and implement macro-control of all running applications. Besides, we make the most of data locality in Hadoop for saving MapReduce network traffic, which can improve the performance of applications and produce more slack time. And our proposed resource scheduler can avoid accepting jobs that will lead to deadline misses and improve the cluster utilization.

Many of the recent works also focus on data placement [24], virtual machine placement [25], single VM migration and dynamic VM consolidation [26] in cloud data

centers, as well as on parameters optimization [27] and testbed configurations [28]. While these works are orthogonal to our research, they play an important role in energy efficiency optimization of MapReduce framework.

3 Background

A MapReduce job is composed of two kinds of subtasks: map tasks and reduce tasks. Map tasks read input data blocks and generate intermediate results, which become input data for reduce tasks. Reduce tasks consist of three phases: shuffle, sort, and reduce. They merge associated intermediate results and generate the final results of the application. Since the shuffle and sort phases are interleaved, we do not consider the sort phase separately.

As shown in Fig. 1, the map phase and shuffle phase are running in parallel, while the shuffle phase and reduce phase are running in serial. And in a realistic MapReduce environment, reduce tasks usually cannot be launched until the number of finished map tasks exceeds a certain value, which can make the most of free containers and improve the cluster utilization. It is obvious that the parallelism degree of tasks can drastically impact the job progress in parallel computing model. As for MapReduce, there are two parallelism metrics (P_M/P_R) for map phase and reduce phase respectively. When the number of map and reduce tasks is in given conditions, low level of parallelism may lead to missed deadlines while high level of parallelism is certainly a waste of resources.

It is not easy to infer and allocate appropriate resources to different applications for meeting their completion deadlines. Because the execution time of each map/reduce task can be very different even though all the tasks performed the same function. This can be caused by the uncertainty of data locality, network traffic, cache hit ratio, memory access latency and so on. To deal with this problem, we need to extract some performance metrics from the past application executions, or we can execute the application on a smaller input dataset to get some necessary information. Then, we can figure out the minimum value of $P_M^j + P_R^j$ based on the predicted completion time for jobs.

Verma proposed three bounds for the prediction of completion time, and employed the average bound which is the mean value of lower bound and upper bound in ARIA

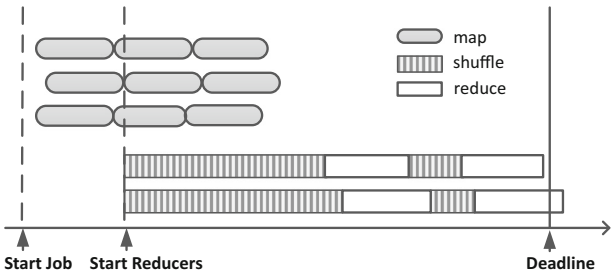

Fig. 1 Three phases of a MapReduce job

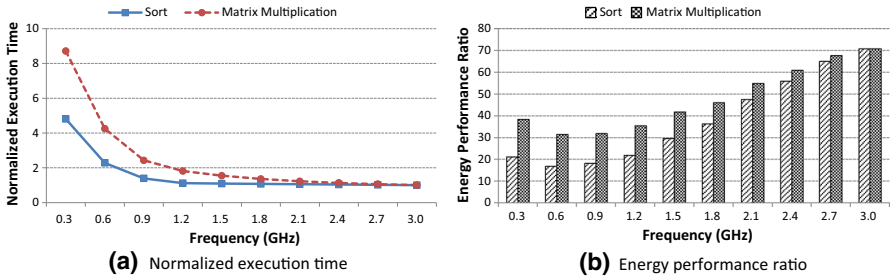

Fig. 2 Normalized execution time of reduce tasks and energy performance ratio of different frequencies

[8]. In this paper, we optimize the way of prediction and take the advantage of upper bound. However, the upper bound usually leads to larger parallelism and results in a job completion time that is much smaller than upper bound. As DVFS has already been incorporated into recent commodity processors and become one of the most commonly used power reduction techniques, we can integrate DVFS to achieve energy savings by running tasks during slack times without affecting applications' SLAs.

Previous approaches usually use the past CPU utilizations to predict future CPU requirements, which can lead to missed deadlines because of inaccurate predictions. In this work, we propose a new slack reclamation algorithm to deal with this problem from a different angle. First, we need to maintain an appropriate execution frequency for map tasks according to the expected completion time of the map phase. Then, we make the most of data locality of reduce tasks for saving network traffic and improving performance in shuffle phase. Finally, we can dynamically adjust the execution time of remaining reduce tasks by using the slack time of already completed reduce tasks and the shuffle stage. This idea was inspired by the master/slave architecture of MapReduce, and it works well based on the observation that a program's execution time is not inversely proportional to each available frequency [29]. We verify this conclusion by an experimental approach in the following work.

We execute Sort benchmark and Matrix Multiplication (MM) with multiple settings, where each setting is identified by a different frequency. The MapReduce implementation of Matrix Multiplication splits input matrices into sub-matrices and uses blocking technique to take advantage of memory cache locality [30]. Figure 2a gives the normalized execution time of reduce tasks at different frequencies. The experimental results show that the execution time is not inversely proportional to each available frequency. As the CPU frequency drops down, the execution time increases slowly at first, and then faster.

The power consumption of modern multi-core based systems has rarely been modeled due to the difficulty of building precise analytical models for multi-core CPUs. Therefore, instead of using an analytical model of power consumption by a server, we utilize real data of IBM server X3250 (Intel Xeon 3480, 4 cores $\times$ 3067 MHz, 8 GB). The configuration and power consumption characteristics of the server are shown in Table 1. Based on the normalized task execution time (as shown in Fig. 2a) and power consumption, we give the (normalized) energy performance ratio of different frequencies in Fig. 2b, where the energy performance ratio of a job j running on a

Table 1 Server's power consumption at different load levels in Watts

Server	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
IbmX3250XeonX3480	42.3	46.7	49.7	55.4	61.8	69.3	76.1	87	96.1	106	113

CPU with frequency f_i is computed as

$$(P_{f_i} - P_{f_0}) \times T_j \tag{1}$$

In general, a higher energy performance ratio indicates that completion of a task at the corresponding CPU frequency requires a potentially higher energy consumption. In practice, the energy performance ratio is co-related to both the characteristics of the application (e.g., data or control intensive application, locality of memory accesses, etc.) and the technology of the hardware server (e.g., CPU technologies, memory system design, etc.). Existing DVFS work show that running the CPU at a lower frequency is typically more energy-efficient compared with completing the task execution at full capacity and remain idle for the rest of time [16,17].

4 Framework

4.1 Framework structure

Considering the importance of SLA in MapReduce environments, we design and implement an SLA-aware framework for Hadoop 2.0. The idea is to make rational use of hardware resources and reduce energy consumption while meeting the requirements of SLA.

As shown in Fig. 3, ResourceManager (RM) is responsible for unified management and allocation of all resources in the cluster, it receives information from each NodeManagers (NM) and assigns resources to per-application ApplicationMaster (AM) in accordance with a certain strategy. AM is responsible for negotiating appropriate resource containers from the RM and working with NMs. In this paper, our framework adds five interacting components: a job profiler, a parallelism estimator, an SLA-aware scheduler, a performance monitor, and a frequency estimator. We will introduce them one by one in the following subsections.

4.2 Job profiler

In this work, we define $J = (I, M, R, D, P_M, P_R)$ is a MapReduce job, where I donates the input dataset, M/R donates map/reduce task set, D donates deadline of the job and P_M/P_R donates the minimum degree of parallelism of map/reduce tasks. In cloud computing clusters, each job j is associated with a completion time goal (D_j). The number of map tasks M_j is defined by the size of input dataset I_j and the number of reduce tasks R_j is specified by users. If j is expected to be completed

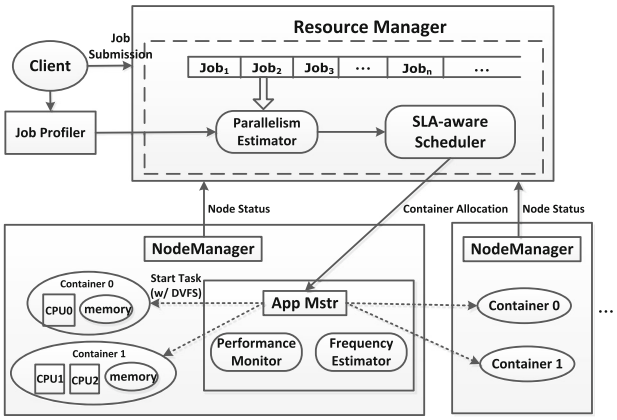

Fig. 3 An SLA-aware framework for Hadoop

Table 2 Performance metrics of MapReduce jobs

Phase	Metric	Description
Initialize	$I_{\max}$	The maximum duration of initialization
Map	M_{avg}	The average duration of map tasks
	$M_{\max}$	The maximum duration of map tasks
	Selectivity_M	The ratio of map output size to the input size
Shuffle	$\text{Sh}^1_{\max}$	The maximum duration of first shuffle
	$\text{Sh}^{\text{typ}}_{\text{avg}}$	The average duration of typical shuffle
	$\text{Sh}^{\text{typ}}_{\max}$	The maximum duration of typical shuffle
Reduce	R_{avg}	The average duration of reduce tasks
	$R_{\max}$	The maximum duration of reduce tasks
	Selectivity_R	The ratio of the reduce output size to its input

before D_j, its minimum parallelism of map/reduce tasks (P_M^j/P_R^j) should be given by the parallelism estimator.

The job profiler is in charge of parsing performance metrics of jobs from the past logs. Alternatively, profiling can be done by executing a given application with a smaller input dataset than the original one. All the performance metrics (see Table 2) are independent of the amount of resources and can reflect all phases of the given job: initialization, map, shuffle and reduce. Besides, we also need to extract the associations between performance and frequency of map/reduce tasks.

4.3 Parallelism estimator

We adopt the module of parallelism estimator as proposed in [8]. The parallelism estimator is in charge of calculating the minimum parallelism of map/reduce tasks to refine the job model. For simplicity of explanation, we omit the step of measured durations with respect to Selectivity_M and Selectivity_R. When a job is submitted and

added to the queue, the parallelism estimator will estimate the upper bound for the duration of the job j (T_j^{up}) as Eq. (1) according to Makespan Theorem as proposed in [8]. As shown in [8], the profiling based estimator is stable (with less the 5 % execution time variation) for various benchmarks executed with different dataset in various environments. Furthermore, compared with the average execution time bound as used in [8], we choose the upper bound in the estimation so that the SLA is guaranteed to be satisfied. Based on the upper bound, we can figure out the minimum value of $P_M^j + P_R^j$ over Eq. (2). Finally, a job may complete before its estimated upper execution time bound, which leads to a slack in the scheduling to be potentially utilized by DVFS for energy-efficient system design.

$$T_j^{\text{up}} = \frac{A}{P_M^j} + \frac{B}{P_R^j} + Q \tag{2}$$

$$\frac{A}{P_M^j} + \frac{B}{P_R^j} = C \tag{3}$$

where $A = (M_j - 1) \times M_{\text{avg}}$, $B = (R_j - 1) \times (\text{Sh}_{\text{avg}}^{\text{typ}} + R_{\text{avg}})$, $Q = I_{\max} + M_{\max} + R_{\max} + \text{Sh}_{\max}^1 + \text{Sh}_{\max}^{\text{typ}} - \text{Sh}_{\text{avg}}^{\text{typ}}$, and $C = D_j - Q$. Then, we can work out P_M^j and P_R^j by the Lagrange's method. Remember to round up the values because P_M^j / P_R^j has to be integral in practice.

$$P_M^j = \frac{\sqrt{A} \times (\sqrt{A} + \sqrt{B})}{C} \tag{4}$$

$$P_R^j = \frac{\sqrt{B} \times (\sqrt{A} + \sqrt{B})}{C} \tag{5}$$

4.4 SLA-aware scheduler

Resource scheduler is one of the most crucial components of YARN. It is a plug-in and defines a set of interface specifications as necessary so that users can achieve their own scheduler. We follow the interface specification and write a new resource schedułthe SLA-aware scheduler which assigns tasks according to the results of parallelism estimator. Workers periodically send a heartbeat to the RM reporting their available resources. In response, the SLA-aware scheduler returns a list of tasks to be assigned to the workers. The detailed resource allocation and task scheduling algorithm will be discussed in Sect. 5.

4.5 Performance monitor

In per-application AM, there is a performance monitor that can collect information for the currently running job and divide the job's map/reduce tasks into already completed tasks (C_j^m / C_j^r), not yet started tasks (U_j^m / U_j^r) and currently running tasks (R_j^m / R_j^r), where $M_j = C_j^m + R_j^m + U_j^m$ and $R_j = C_j^r + R_j^r + U_j^r$. When a map/reduce task

of job j is going to be launched, the performance monitor will figure out the "ideal" execution time of remaining tasks (u_j^m/u_j^r) pursuant to P_M^j/P_R^j and D_j. Suppose the upper bound of completion times for the map phase is denoted as T_M^{up} and the current time is T, we have

$$T_M^{\text{up}} = \frac{(M_j - 1) \times M_{\text{avg}}}{P_M^j} + M_{\max} \tag{6}$$

$$u_j^m = \frac{T_M^{\text{up}} - T}{\left\lceil \frac{R_j^m + U_j^m}{P_M^j} \right\rceil} \tag{7}$$

$$u_j^r = \frac{D_j - T}{\left\lceil \frac{R_j^r + U_j^r}{P_R^j} \right\rceil} - \text{Sh}_{\text{avg}}^{\text{typ}} \tag{8}$$

Specifically, when $\lceil (R_j^r + U_j^r)/P_R^j \rceil = 2$, let the execution time of N_j reduce tasks twice longer than the others, where $N_j = 2P_R^j - (R_j^r + U_j^r)$.

4.6 Frequency estimator

Starting with the 2.6.0 Linux kernel, users can dynamically scale processor frequencies through the CPUFreq subsystem. The DVFS technique enables processors to be operated at multiple frequencies under different supply voltages, thus gives opportunities to reduce the energy consumption of high performance computing by scaling processor supply voltages. The theoretical basis of the technique is the following formula:

$$E = P \times t = \alpha C V^2 F \times t \tag{9}$$

From the above formula, we can see that the energy consumption will not be reduced unless the frequency and voltage are reduced at the same time. Because for a given task, $F \times t$ is a constant. Fortunately, the frequency scaling of CPUFreq module is based on the ACPI driver of each CPU manufacturers (such as Intel's SpeedStep and AMD PowerNow), these advanced power management drivers can automatically adjust the voltage of motherboard depending on the CPU operating frequency.

The frequency estimator needs to get the associations between performance and frequency when a job was submitted and identify the target processor speed (f_j^m/f_j^r) based on the performance monitor. For a cloud computing cluster equipped with DVFS cores, we assume each of its compute nodes has N processor frequencies available: $\{f_0, f_1, \ldots, f_n\}$, satisfying $f_0 < f_1 < \cdots < f_n = f_{\max}$. Since the processor only supports a finite set of frequencies, we set $f_j^m = f_2$ to guarantee the completion time when $\forall f_j^m \subseteq [f1, f2]$, where f_1 and f_2 is a pair of adjacent available CPU frequencies. The settings of f_j^r is the same with f_j^m.

5 SLA-aware scheduler

5.1 Data locality

MapReduce applications can get better performance if map tasks run on the nodes that store the input data, which is referred as the data locality optimization. However, Hadoop does not consider data locality when scheduling reduce tasks, because the input to a reduce task is typically the output of many map tasks generated at multiple nodes while the input to a map task exists at a solo node. The flow of data between map tasks and reduce tasks is called shuffle, and this might lead to increased network traffic which is typically a bottleneck in MapReduce-based systems.

Hadoop assumes a master-slave architecture and a tree-style network topology. The single master and multiple worker nodes are spread over different racks contained in one or more data centers. Figure 4 demonstrates a data center with two racks each including five nodes. Each rack switch has uplinks connected to the core switch connecting the other rack with uniform bandwidth. The bandwidth between two nodes is dependent on their relative locations in the network topology.

The total network distance of a reduce task (Ri) includes the network distance required to shuffle all partitions to Ri. In this paper, we define the total network distance of Ri as TND_{Ri} like [31]. Apparently, the bigger TND_{Ri} is, the more time will be taken to shuffle Ri's partitions, and the network bandwidth will be dissipated additionally. Suppose there are 6 map tasks Mi ($1 \leq \text{i} \leq 6$) and 4 reduce tasks Ri ($1 \leq \text{i} \leq 4$) scheduled on distinct nodes as shown in Fig. 5, and every map task is feeding every reduce task. Assuming the distance from a node to its parent is 1 and the distance between any two nodes can be calculated by adding up their distances to their

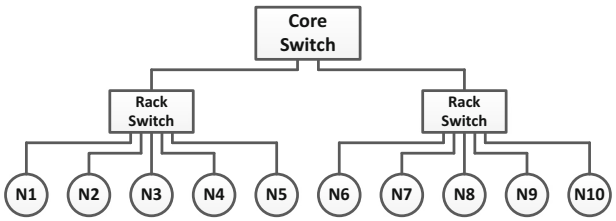

Fig. 4 A tree-style network topology of Hadoop

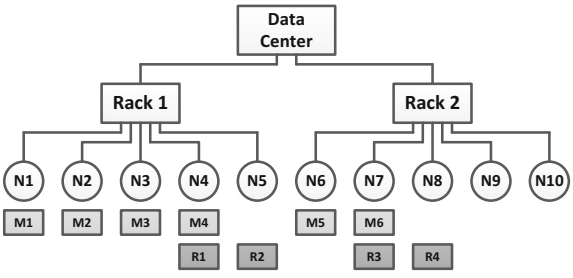

Fig. 5 The nodes at which Hadoop schedules tasks

closest common ancestor, we can ultimately get the total network distance of reduce tasks TND_{Ri} $(1 \leq \mathrm{i} \leq 4)$: $\mathrm{TND}_{R0} = 14$, $\mathrm{TND}_{R1} = 16$, $\mathrm{TND}_{R2} = 18$, and $\mathrm{TND}_{R3} = 20$.

From the above, we can see that the total network distance of reduce tasks can be very different caused by different locations, which shows that reduce tasks also have the advantage of data locality. But unlike map tasks, reduce tasks care more about the data on a whole rack than that on a single node. And we found that a reduce task can get the best performance on the rack that has the most feeding map tasks of it. Actually, Hadoop divides the data locality into three grades: node locality, rack locality, and data center locality (temporarily not implemented). When per-application ApplicationMaster applies to ResourceManager for resources, it needs to send a resource request list which describes the needs of each resource unit in detail. Each resource request can be seen as a serialized Java object, and then the resource scheduler will return a description of the allocated containers to ApplicationMaster.

In this paper, we make the most of data locality in Hadoop for saving MapReduce network traffic. Suppose every map task has a resource request priority defined as PRI_{Mi}, and every reduce task has a resource request priority defined as PRI_{Ri}. Our ApplicationMaster will give the desired node for each map task and the desired rack for each reduce task in the resource request list when applying for resources. Then the SLA-aware scheduler can allocate resources according to the request list.

5.2 Deadline constraint

Generally, the number of datanodes determines the computing power of a Hadoop cluster, and workloads that exceed the computing capacity may lead to missed deadlines for any jobs. To solve this problem, the Deadline Constraint scheduler [32] was proposed to ensure deadlines for real-time MapReduce jobs, which, however, may lead to resource under-utilization as well as deadline violations. He et al. developed a RTMR scheduler [33] to avoid accepting jobs that may lead to deadline misses, but the RTMR scheduler is not applicable to YARN environment. In this paper, our proposed SLA-aware scheduler determines whether the job can be completed within the specified deadline before a job is added to the job queue.

For convenience, the total required resources of job j are denoted as $f(P_M^j, P_R^j)$, and the number of residual containers in the cluster are denoted as C_r, where $f(P_M^j, P_R^j) = P_M^j + P_R^j$. The SLA-aware scheduler is able to achieve control of frequency scaling leveraging a boolean variable (FX). FX is set to 1 by default, and in this case, per-application ApplicationMasters can dynamically adjust CPU frequencies for energy saving. When FX is set to 0, it indicates insufficient resources in the current status of the cluster, and the SLA-aware scheduler will notify all ApplicationMasters to run tasks at the highest frequency ($f_{\max}$).

When job j is submitted, if its total required resources ($f(P_M^j, P_R^j)$) are no more than the number of residual containers (C_r) in the cluster, the SLA-aware scheduler will add j to the job queue and allocate resources to it according to its degree of parallelism. Conversely, if $f(P_M^j, P_R^j)$ is greater than C_r, there will be two cases. In the first case, job j can be completed in time if all jobs stop controlling the frequencies

of CPU cores and set their processors to the highest frequency available. Meanwhile, FX will be set to 0 until the total required resources of a new job are less than C_r. As for the second case, it is impossible to finish job j no matter what we do, so the SLA-aware scheduler will not change the value of FX and reject job j.

Through the above analysis, here comes a question: how does the SLA-aware scheduler manage to distinguish between the two cases? In this paper, the parallelism of map/reduce phase is expected to remain unchanged during the job's execution, so we first check whether there are enough resources for the map phase of job j. If $C_r < P_M^j$, the SLA-aware scheduler will simply reject job j for its unreasonable deadline. Instead, if $P_M^j \leq C_r < f(P_M^j, P_R^j)$, we will need to estimate whether there are enough resources to start the reduce phase while other jobs accelerate their progress.

Although all running applications accelerate their progress for job j, we only care about the jobs that can be finished before starting the reduce phase. Given that reduce tasks cannot be launched until the number of finished map tasks exceeds a certain value (denoted as CMFR), now the question becomes how many free containers are available at the point when the number of finished map tasks reaches CMFR. Let $T_{\text{CMFR}}^{\text{up}}$ be the upper bound of completion times for CMFR numbers of map tasks, we have

$$T_{\text{CMFR}}^{\text{up}} = \frac{(\text{CMFR} - 1) \times \text{M}_{\text{avg}}}{P_M^j} + M_{\max} \tag{10}$$

Suppose there are N numbers of jobs that can be finished before $T_{\text{CMFR}}^{\text{up}}$, then job j can make full use of the containers released by these jobs, which is denoted as RC. Let EC_r be the estimated residual containers in the cluster when starting the reduce phase, and apparently RC is part of EC_r, then we can calculate the value of RC and EC_r by the following formulas.

$$\text{RC} = \sum_{j=0}^{N} P_R^j (0 \leq j \leq N) \tag{11}$$

$$\text{EC}_r = C_r - P_M^j + \text{RC}. \tag{12}$$

5.3 Resource allocation

The SLA-aware scheduler needs to refine the job model through the parallelism estimator after job j is submitted and determine whether job j can be completed within the specified deadline or not before it is added to the job queue. The SLA-aware Resource Allocation Algorithm orders jobs by the Earliest Deadline First (EDF) algorithm, which is an optimal dynamic scheduling algorithm for real-time processing. The detailed resource allocation schema is shown in Algorithm 1. (For job j, M_j^r represents the number of running map tasks, R_j^r represents the number of running reduce tasks, M_j^f represents the number of finished map tasks, C_r represents the number of

Algorithm 1 SLA-aware Resource Allocation

Input:
$jobQueue$: job queue;
$freeContainers$: number of free containers;
Output:
$assignedContainers$: assigned container list;
1: When job j is submitted:
2: Refine the job model $(I_j, M_j, R_j, D_j, P_M^j, P_R^j)$;
3: **if** $f(P_M^j, P_R^j) \leq C_r$ **then**
4: Set $FX = 1$ and add j to the $jobQueue$;
5: **else if** $P_M^j \leq C_r$ & $P_R^j \leq EC_r$ **then**
6: Set $FX = 0$ and add j to the $jobQueue$;
7: **else**
8: Reject job j;
9: **end if**
10: Sort $jobQueue$ in order of earliest deadline;
11: **while** $freeContainers > 0$ **do**
12: **for** each free container c in $freeContainers$ **do**
13: **for** each job j in $jobQueue$ **do**
14: **if** $M_j^r < P_M^j$ **then**
15: Allocate c to AM for running the map task that has the highest PRI_{Mi};
16: $assignedContainers.add(mapTask)$;
17: **else if** $M_j^f > CMFR$ and $R_j^r < P_R^j$ **then**
18: Allocate c to AM for running the reduce task that has the highest PRI_{Ri};
19: $assignedContainers.add(reduceTask)$;
20: **end if**
21: **end for**
22: **end for**
23: **end while**
24: **return** $assignedContainers$

residual containers in the cluster, CMFR represents the threshold at which to start the reduce phase, and $f(P_M^j, P_R^j) = P_M^j + P_R^j$).

As shown in Line 11–23, for each free container and each job, if the number of running map tasks of job j is lesser than P_M^j in the job model , a new task is launched. As long as the number of finished map tasks reaches the pre-set threshold, reduce tasks can be launched as required. Preference is given to map tasks which have node locality to the worker node, while the assignment of reduce tasks is more concerned about rack locality.

5.4 Task scheduling

YARN uses a double-layer resource scheduling model: in the first layer, the resource scheduler in ResourceManager allocates resources to per-application ApplicationMasters; then in the second layer, ApplicationMasters will allocate containers to each task of their jobs. The SLA-aware scheduler is focused on resource allocation in the first layer. As for the task scheduling of the second layer, it is completely determined by per-application ApplicationMasters. In this work, we implement an

Algorithm 2 DVFS-based Task Scheduling

Input:
$assignedContainers$: assigned container list;
$unassignedTasks$: unassigned task list of job j;
Output:
$assignedTasks$: assigned task list of job j;
When job j is started:
Add M_j and R_j to the monitor set;
Set $f_j^m = f_j^r = f_{\max}$;
for each free container c in $assignedContainers$ **do**
  if c is for map tasks **then**
    Performance monitor gives u_j^m;
    Frequency estimator resets f_j^m;
    Launch map task on c with f_j^m;
    $assignedTasks.add(mapTask)$;
  else if c is for reduce tasks **then**
    Performance monitor gives u_j^r;
    Frequency estimator resets f_j^r;
    Launch reduce task on c with f_j^r;
    $assignedTasks.add(reduceTask)$;
  end if
end for
return $assignedTasks$

DVFS-based ApplicationMaster with the performance monitor and frequency estimator. The detailed task scheduling schema is shown in Algorithm 2.

As stated earlier, we want to implement an application-centric DVFS scheme which applies macro-control to the whole map/reduce phase. To guarantee the performance goal for MapReduce jobs, we will keep the parallelism of map/reduce tasks constant during an application's execution. When assigning new map/reduce tasks of a job: the performance monitor will figure out u_j^m/u_j^r at first; then, frequency estimator can reset the target frequency f_j^m/f_j^r; finally, ApplicationMaster will treat f_j^m/f_j^r as one of the environment variables that are encapsulated into the container launch context. By this way, multiple cores on the same node can be tuned to different frequencies, and the task scheduling control loop ensures efficiency of the framework.

According to Algorithm 1 and 2, Fig. 6 shows an example task scheduling for the matrix multiplication example discussed in Sect. 3. Assume that the parallelism estimator determines at least two nodes are required to meet the deadline ($D = 120$) of a particular MapReduce application with 6 map tasks and 4 reduce tasks, based on the upper bound execution time of the jobs given by the job profiler. Figure 6a show the task execution without DFVS, where both nodes run at the highest CPU frequency. Since some of the map tasks may complete their execution before the estimated upper bound, there is an aggregated slack time between the time when all map tasks finish (i.e., 60) and the required completion time of map task in order for the entire application to meet its SLA (i.e., $T_M^{\text{up}} = 75$). The execution of reduce tasks is similar, and the entire application is finished at time 95, which is energy-inefficient. On the other hand, with the DVFS-based scheduling, the CPU frequency is dynamically altered to achieve energy saving. In particular, given the current slack time resulted

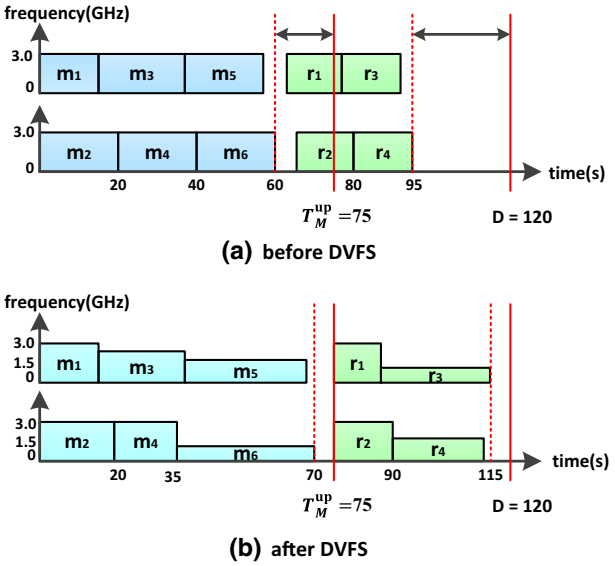

Fig. 6 Task scheduling example

from m_1's early completion, m_3 can be executed at a lower frequency of 2.4GHz, which still guarantees that all map tasks will be completed before T_M^{up}. Similarly, the minimal possible execution frequency is determined for each map task according to T_M^{up}, as well as the reduce tasks according the the overall deadline of the application.

6 Evaluation

6.1 Experimental setup

We evaluate the performance and energy efficiency of the proposed scheduling framework via CloudSim simulator [9], which supports modeling and simulation of different resource provisioning schemes and power management techniques like DVFS. Based on the ideas presented in [34], we extend CloudSim with YARN resource management and scheduler. We create a YARN environment in CloudSim with 30 homogenous physical servers and the network speeds among each machine are randomly generated between 100 MBps and 200 MBps. Each server is equipped with a Intel Xeon X3480 (4-core) CPU with capacity computation power of 3000 MIPS. Since we focus on CPU utilization in this work, we assume each YARN resource container has 1 core and unlimited memory space. Based on the energy performance ratio of servers stated in the built-in power model of CloudSim, we select the following five different CPU frequency scaling levels: {0.9, 1.2, 1.8, 2.4, 3.0 GHz}.

We use the MapReduce version of *Matrix Multiplication* [30] and the *Sort* Benchmark from Hadoop distribution for our experimental evaluation. Each benchmark is associated with four different deadlines of 4, 5, 6, and 7 min. For each benchmark, we

profile and obtained the execution time of its *map* and *reduce* tasks under different CPU frequency levels, which are used to determine the minimum parallelism of jobs based on various SLA deadlines. For a comparison, we have also implemented the ARIA approach which uses the average completion time (average bound) for MapReduce resource provisioning [8] and running on fixed 3.0 GHz CPU frequency.

There are two basic metrics used in our analysis: execution time (T) as performance metric, and the work-induced energy (E_W) for energy metric similarly as in [21]. The power consumption of servers in a cloud computing cluster is mostly determined by the CPU, memory, disk storage, power supplies and cooling systems [35]. As the idle power P_{idle} can account for up to 40 % of the system power under load in our cluster, we use the work-induced energy to provide a more direct indication of energy consumption by running applications. In particular, the work-induced energy E_W can be defined as the following equation, where E is the overall energy consumption of the cluster.

$$E_W = E - T \times P_{\text{idle}} \tag{13}$$

6.2 Experimental results

In order to prove the validity of data locality, we first evaluate the network bandwidth requirements of *Sort* benchmark in two cases. In the first case, we have a group of applications that run on the original Hadoop cluster which does not consider data locality when scheduling reduce tasks; in the second case, we improve the resource allocation algorithm and adopt rack locality for reduce tasks. Figure 7a reveals that the network accesses are very active during most of the execution period, especially in the beginning. But when we look at it in detail, we find that the network accesses of reduce phase are very different in the two cases while the network accesses of map phase are about the same. In fact, our improved algorithm can decrease network traffics of the shuffle phase by about 12 %. Then we compare the performance of *Sort* benchmarks that are specified with different deadlines and give the contrast data in Fig. 7b. As the experimental results suggest, our scheme can improve the performance of applications by 9 %, which produces more slack time for the reduce phase.

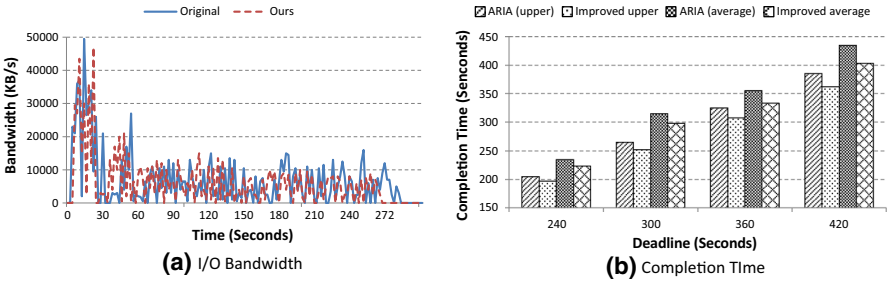

Fig. 7 The network bandwidth requirement and performance of *Sort* benchmark

Table 3 The parallelism of sort and matrix multiplication

Deadline(s)	Sort		Matrix multiplication	
	Ours	ARIA	Ours	ARIA
240	(56, 68)	(48, 62)	(63, 50)	(43, 34)
300	(43, 52)	(38, 47)	(48, 38)	(32, 25)
360	(34, 44)	(30, 41)	(34, 27)	(25, 20)
420	(28, 38)	(25, 33)	(27, 21)	(21, 17)

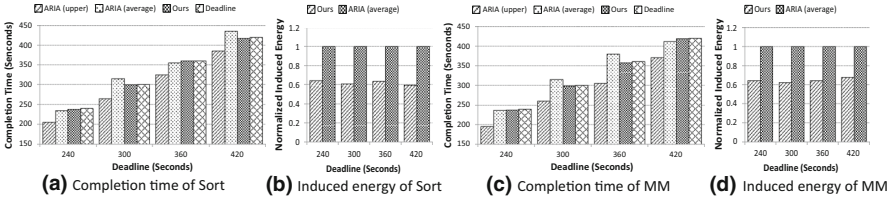

Fig. 8 The execution time and energy consumption of Sort and Matrix Multiplication based on SLA and ARIA

In Table 3, we compute the minimum parallelism of jobs based on SLA and ARIA (average bound) accordingly. For example, the *Sort* benchmark with 240 s deadline requires 56 and 68 containers in the map and reduce phases, respectively in our proposed framework; while it requires 48 map slots and 62 reduce slots in ARIA. It is because our framework considers the worst-case bound for resource provisioning, compared with the average bound used in ARIA. We can also observe that resource requirements for both approaches reduce with the relaxation of the deadline. Note that high resource demands in our approach will not cause any performance issues when the entire cluster is not heavily loaded. In a scenario with resource shortage, our scheme runs high-priority applications with more CPU resources at the highest possible frequency so that they will complete faster; then the vacant resources will be assigned to the low-priority applications. However, ARIA may lead to better fairness between applications in such scenarios.

The performance and energy consumption comparisons are shown in Fig. 8. For *Sort*, our proposed scheme achieves better SLA conformance than ARIA(average) for various deadline settings. For example, ARIA with resource provisioning based on the average task completion time (average bound) violate the SLAs with 300 and 420 s deadlines. We have also shown the ARIA (upper) approach with resource provisioning based on the worst-case task completion time. Although ARIA (upper) leads to a fast completion time, it may not utilize the resource efficiently unless the entire cluster is heavily loaded by various applications. In particular, keeping a CPU run at the highest frequency to complete a task and idle it for the rest of time is not as energy-efficient as running it at a lower frequency for a longer period (which is the fundamental reasoning behind the DVFS technology).

Furthermore, we achieve significant improvement on the energy reduction with our online DVFS scheme. The work-induced energy reduces by 36 % (which is equivalent

to a 8 % total system energy saving) when the deadline is 240 s. With the increase of available time, the resources needed to meet the deadline decreased, and the reduction ratio on work-induced energy over ARIA changes between 36 and 41 % (Fig. 8b). Similarly for *Matrix Multiplication*, our policy reduces the work-induced energy between 32 and 38 % (Fig. 8d).

7 Conclusion

In this work, we proposed an SLA-aware energy-efficient scheduling scheme for shared MapReduce applications in Hadoop YARN. Based on the job profiling information, our scheduling scheme enables automatic resource inference and allocation and makes the most of data locality in Hadoop for saving network traffic. Besides, we integrate DVFS to reduce energy by running tasks at low CPU frequency during slack times without violation of the application's SLA. And our proposed SLA-aware scheduler can avoid accepting jobs that will lead to deadline misses and improve the cluster utilization. The experimental results show that our scheme achieves better SLA conformance with low resource cost and energy consumption.

In the future work, we plan to extend the proposed scheme on heterogeneous clusters where the available resources and energy performance ratio may vary on different nodes. As a result, the resource allocation scheme should be aware of the variation of execution time when a job is assigned to different nodes. Moreover, the energy performance ratio for a particular application running on various nodes need to be incorporated into the DVFS-based task scheduling. While this paper have focused on MapReduce tasks, we will study the scheduling problem of other types of applications (e.g., Storm and Spark) which can be also deployed on YARN.

Acknowledgements This research is sponsored by the Natural Science Foundation of China (NSFC) under Grant no. 61202015 and 61533011, Shandong Provincial Natural Science Foundation under Grant no. ZR2013FM028 and ZR2015FM001, the Fundamental Research Funds of Shandong University under no. 2015JC030.

References

1. Dean J, Ghemawat S (2008) MapReduce: simplified data processing on large clusters. Commun ACM 51(1):107–113
2. Apache Hadoop. http://hadoop.apache.org/. Accessed 5 Feb 2016
3. Vavilapalli VK et al (2013) Apache hadoop yarn: yet another resource negotiator. In: Proceedings of the 4th annual Symposium on Cloud Computing. ACM, p 5
4. Van Heddeghem W et al (2014) Trends in worldwide ICT electricity consumption from 2007 to 2012. Comput Commun 50:64–76
5. Capacity Scheduler. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoopyarn-site/CapacityScheduler.html. Accessed 5 Feb 2016
6. Fair Scheduler. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoopyarn-site/FairScheduler.html. Accessed 5 Feb 2016
7. Ibrahim S et al (2014) Towards efficient power management in MapReduce: investigation of CPU-frequencies scaling on power efficiency in Hadoop. In: Adaptive resource management and scheduling for cloud computing. Springer, pp 147–164
8. Verma A, Cherkasova L, Campbell RH (2011) ARIA: automatic resource inference and allocation for mapreduce environments. In: Proceedings of the 8th ACM international conference on Autonomic computing. ACM, pp 235–244

9. Calheiros RN, Ranjan R, Beloglazov A, De Rose CA, Buyya R (2011) Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Softw Pract Exp 41(1):23–50
10. Polo J, et al (2010) Performance-driven task co-scheduling for mapreduce environments. In: Network operations and management symposium (NOMS). IEEE, pp 373–380
11. Ferguson AD, Bodik P, Kandula S, Boutin E, Fonseca R (2012) Jockey: guaranteed job latency in data parallel clusters. In: Proceedings of the 7th ACM european conference on Computer Systems. ACM, pp 99–112
12. Yao Y, Wang J, Sheng B, Lin J, Mi N (2014) Haste: Hadoop yarn scheduling based on task-dependency and resource-demand. In: IEEE 7th International Conference on Cloud Computing (CLOUD). IEEE, pp 184–191
13. Davis RI, Burns A (2011) A survey of hard real-time scheduling for multiprocessor systems. ACM Comput Surv (CSUR) 43(4):35
14. Qiu M, Sha EH-M (2009) Cost minimization while satisfying hard/soft timing constraints for heterogeneous embedded systems. ACM Trans Des Autom Electron Syst (TODAES) 14(2):25
15. Li J, Ming Z, Qiu M, Quan G, Qin X, Chen T (2011) Resource allocation robustness in multi-core embedded systems with inaccurate information. J Syst Archit 57(9):840–849
16. Krishna CM, Lee Y-H (2000) Voltage-clock-scaling adaptive scheduling techniques for low power in hard real-time systems. In: 19th Real-Time and Embedded Technology and Applications Symposium (RTAS). IEEE, pp 156–156
17. Kim W, Shin D, Yun H-S, Kim J, Min SL (2002) Performance comparison of dynamic voltage scaling algorithms for hard real-time systems. In: Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium. IEEE, pp 219–228
18. Ge R et al (2010) Powerpack: energy profiling and analysis of high-performance systems and applications. IEEE Trans Parallel Distrib Syst 21(5):658–671
19. Wang L, Von Laszewski G, Dayal J, Wang F (2010) Towards energy aware scheduling for precedence constrained parallel tasks in a cluster with DVFS. In: IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid). IEEE, pp 368–377
20. Wang Y, Liu H, Liu D, Qin Z, Shao Z, Sha EH-M (2011) Overhead-aware energy optimization for real-time streaming applications on multiprocessor system-on-chip. ACM Trans Des Autom Electron Syst (TODAES) 16(2):14
21. Wirtz T, Ge R (2011) Improving mapreduce energy efficiency for computation intensive workloads. In: 2011 International Green Computing Conference and Workshops (IGCC). IEEE, pp 1–8
22. Ge R, Feng X, Feng W-C, Cameron KW (2007) Cpu miser: A performance-directed, run-time system for power-aware clusters. In: International Conference on Parallel Processing (ICPP). IEEE, pp 18–18
23. Kim W, Gupta MS, Wei G-Y, Brooks D (2008) System level analysis of fast, per-core DVFS using on-chip switching regulators. In: IEEE 14th International Symposium on High Performance Computer Architecture. IEEE, pp 123–134
24. Maheshwari N, Nanduri R, Varma V (2012) Dynamic energy efficient data placement and cluster reconfiguration algorithm for MapReduce framework. Future Gener Comput Syst 28(1):119–127
25. Cardosa M, Singh A, Pucha H, Chandra A (2012) Exploiting spatio-temporal tradeoffs for energy-aware mapreduce in the cloud. IEEE Trans Comput 61(12):1737–1751
26. Beloglazov A, Buyya R (2012) Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers. Concurr Comput Pract Exp 24(13):1397–1420
27. Babu S (2010) Towards automatic optimization of MapReduce programs. In: Proceedings of the 1st ACM symposium on Cloud computing. ACM, pp 137–142
28. Belalem G, Tayeb FZ, Zaoui W (2010) Approaches to improve the resources management in the simulator CloudSim. In: Information computing and applications. Springer, pp 189–196
29. Singleton LC, Poellabauer C, Schwan K (2005) Monitoring of cache miss rates for accurate dynamic voltage and frequency scaling. In: Electronic imaging 2005. International Society for Optics and Photonics, pp 121–125
30. Norstad J (2009) A MapReduce algorithm for matrix multiplication. http://www.norstad.org/matrix-multiply/. Accessed 5 Feb 2016
31. Hammoud M, Rehman MS, Sakr MF (2012) Center-of-gravity reduce task scheduling to lower mapreduce network traffic. In: International Conference on Cloud Computing (CLOUD). IEEE, pp 49–58

32. Kc K, Anyanwu K (2010) Scheduling hadoop jobs to meet deadlines. In: 2010 IEEE Second International Conference on Cloud Computing Technology and Science. IEEE, pp 388–392
33. He C, Lu Y, Swanson D (2013) Real-time scheduling in mapreduce clusters. In: High performance computing and communications and embedded and ubiquitous computing ($HPCC\_EUC$). IEEE, pp 1536–1544
34. Jung J, Kim H (2012) MR-CloudSim: Designing and implementing MapReduce computing model on CloudSim. In: 2012 International Conference on ICT Convergence (ICTC). IEEE, pp 504–509
35. Minas L, Ellison B (2009) Energy efficiency for information technology: how to reduce power consumption in servers and data centers. Intel Press