Sustainable Computing: Informatics and Systems xxx (2012) xxx–xxx

Contents lists available at SciVerse ScienceDirect

Sustainable Computing: Informatics and Systems

journal homepage: www.elsevier.com/locate/suscom

Improving resource efficiency in data centers using reputation-based resource selection☆

Tung Nguyen*, Weisong Shi

Department of Computer Science, Wayne State University, United States

ARTICLE INFO

Article history:
Received 6 February 2011
Received in revised form 13 August 2011
Accepted 12 March 2012

Keywords:
Reputation
Energy efficiency
Waste

ABSTRACT

Today data centers are consuming a lot of energy but not very efficiently. Much of energy is wasted. There are several types of energy waste at different levels including infrastructure-, machine- and system-level waste. The former two levels have been improved significantly in the last few years, however, few efforts have been put on the last level, especially the resource waste caused by failures in a data center. In this paper, we attack the problem proactively by leveraging a reputation-based resource selection scheme to reduce the number of resubmissions of tasks, resulting from the failure during the course of their execution. To capture the characteristics of resources, we introduce Opera, an OPEn ReputAtion model. Opera characterizes itself with two important novelties: a *vector* representation of the reputation and the *just-in-time* feature that represents the real-time system status, which, to our knowledge, has never been considered in conventional reputation systems. To demonstrate the effectiveness of Opera, we have integrated the Opera trust model into the scheduler of Hadoop. The experimental results showed that Opera enables the scheduler to select appropriate nodes which helped to reduce not only the number of re-executed tasks but also the execution time of Hadoop's jobs under the presence of failures and heavy workload up to 59% and 32%, respectively. This improvements, in turn, can improve the energy efficiency of the whole system and the network by 16.17% and 53.32% for the sort application respectively.

© 2012 Published by Elsevier Inc.

1. Introduction

With the increasing demand in computing, the number of data centers is also increasing [16]. However, according to the EPA report in 2007 [20], the data centers are going to consume more than 100 billion kWh by 2011 and account for 23% of global information and communications technology (ICT) CO_2 emissions to the environment [4]. As a result, making the operation of data centers "greener" has become the main focus of many research activities recently. The main idea is to consume energy in an efficient way since current systems are efficiently [11,12,10]. First, let us take a look the resource inefficiency in current systems.

1.1. Resource inefficiency

The inefficiency is caused by the waste. There are several types of waste at different levels such as infrastructure-, machine- and system-level. At the infrastructure level, one half of energy is spending on cooling [24]. At the machine level, 50% energy is used during the idle [11]. At the system level, the system could be useless because of checkpointing [38] or context switching [23]. To improve resource efficiency, we have to minimize these waste.

To measure the data center energy efficiency, some metrics have been proposed. Essentially, energy efficiency of a data center is defined as the amount of computational work performed divided by the total energy used in the process:

$$\text{Energy efficiency} = \frac{\text{Computation}}{\text{Total } \textit{energy}}$$

Recent work [15,18] has defined two widely used metrics (power usage effectiveness (PUE) and data center infrastructure efficiency (DCiE)):

$$\text{PUE} = \frac{\text{Total facility power}}{\text{IT equipment power}} \quad (1)$$

However, these metrics only capture the efficiency in data center at infrastructure level. Therefore, Barroso and Hölzle [10] have proposed another way to compute efficiency which takes into account different levels of efficiency as shown in Eq. (2). In the formula, energy efficiency is factorized into three parts. The first two factors account for the efficiency of power dissipation to electronic components in a system. SPUE is server PUE which is calculated in the same way as PUE (see Eq. (1)), but its denominator only accounts for power consumed by the electronic components directly involved

☆ This work is in part supported by the US National Science Foundation CAREER Grant No. CCF-0643521.
* Corresponding author.
E-mail addresses: nttung@wayne.edu (T. Nguyen), weisong@wayne.edu (W. Shi).

2210-5379/$ – see front matter © 2012 Published by Elsevier Inc.
doi:10.1016/j.suscom.2012.03.001

Please cite this article in press as: T. Nguyen, W. Shi, Improving resource efficiency in data centers using reputation-based resource selection, Sustain. Comput.: Inform. Syst. (2012), doi:10.1016/j.suscom.2012.03.001

in the computation such as motherboard, CPU, memory and so on. The last factor represents the efficiency when running workload in the data centers:

$$\text{Energy efficiency} = \frac{\text{Computation}}{\text{Total } energy} = \left(\frac{1}{\text{PUE}}\right) \times \left(\frac{1}{\text{SPUE}}\right) \times \left(\frac{\text{Computation}}{\text{Total } energy \text{ to electronic comp.}}\right) \quad (2)$$

$$\text{Energy efficiency}' = \left(\frac{1}{\text{PUE}}\right) \times \left(\frac{1}{\text{SPUE}}\right) \times \left(\frac{\text{Computation}}{\text{Total } energy \text{ to electronic comp.}}\right) \times \left(\frac{\text{Required resource}}{\text{Actual resource used}}\right) \quad (3)$$

While this definition is good enough to characterize the first two levels of waste, i.e., infrastructure- and machine-level waste, we argue that it is not the end of the story since the execution of a job (i.e., run-time efficiency) is not considered. For example, a job will have to be rescheduled or resubmitted if it fails in the middle of the execution, which will cause system-level waste. We augment Eq. (2) by adding another definition of usage efficiency, which is defined as required resource/actual resource used as shown in Eq. (3).

Our work focuses on reducing the waste caused by failures because future systems will be expected to have higher failure rate [38]. We will show that the failure does cause increasing in energy consumption and that reducing the failure indeed helps to reduce it. Hence, our objective is to minimize the number of job resubmission, i.e. re-execution, while still maintaining the performance. Basically, this type of waste comes from inefficient resource allocation. For example, if the resource allocator assigns slow, unreliable nodes to handle critical tasks, the whole application can easily be delayed or even fail. As a result, we need to choose proper resources to execute our tasks to achieve the goal.

1.2. Our approach

Selecting appropriate resources is not new. In commercial systems, the consumers choose the service providers (SPs) by their reputation. In computational systems, resource selection selects the slow-progress node for speculative execution based on their progress score [43], a group of nodes for computational replication based on their reputation [39], or the locations to store replicas [27].

However, in general, the existing methods rely on a single-value reputation to capture the differences (the heterogeneity) between the service providers in terms of a property such as performance or availability. However, this single value representation is inadequate for the multi-property based selection criteria. For example, there are conflictions, such as the one between data locality and fairness as shown in [42], that make it unsuitable to apply conventional single value based scheduling. In addition, the reputation computed from the existing models is generally based on the past information, which may be inconsistent with the current status of the system. For example, if a node has had a good record for a long time and now it becomes unreliable because of a virus or recent compromise, the history-based computation model would still return a high reputation.

As a result, we use a vector instead of a scalar to represent the reputation and capture different concerned properties. For example, with a reliability concern, an SP has a higher reputation if it has more *successful* transactions than others. With the performance concern, the higher the reputation, the higher performance a resource has. We developed a new trust model called *Opera* (OPen ReputAtion Model) that allows users to query the reputation vector of any registered component. Opera also allows the addition of new elements to the reputation vector, which expresses its *openness*. To reduce management intervention, Opera employs a popular, open-source monitoring tool to capture the changes of the system and dynamically update the reputation vectors accordingly. Based on the reputation vectors, the system can easily select resources that conform to the users' requests.

To illustrate our idea, we modified an existing scheduler, which can be considered as a resource allocator, to leverage Opera. The chosen scheduler is the default scheduler of Hadoop, a popular framework for running MapReduce [19] applications on large scale clusters of commodity machines [2]. We conducted the experiments by executing different MapReduce applications with different configuration requirements on our local cluster, under the presence of failure and heavy-load nodes. The results show that the original Hadoop, although built with failure toleration, can still suffer from failures and increases the execution time to 50%. Moreover, the energy consumption of the whole system can increase 17%, too. The results also confirm that our modified scheduler actually selects appropriate nodes and, therefore, improves the performance up to 32%, reduced up to 59% of the number of failed/killed tasks as well as improve the (energy) usage efficiency by up to 53.32%.

The contributions of this paper are three-fold: (1) a new reputation model including *vector representation* and *just-in-time feature* (JR); (2) the design and implementation of Opera; and (3) the incorporation of it with Hadoop and comprehensive experiments.

The remainder of the paper is organized as follows. Section 2 introduces the reputation background with related work, followed by the analysis and design of Opera in Section 3. Section 4 describes the details of the implementation of Opera and the modification of Hadoop. Evaluation results are presented in Sections 5 and 6. Finally, the conclusion will be given in Section 7.

2. Background and related work

Reputation systems give information about the past behavior of an entity, helping one to decide which entities to trust. Often reputation is used in commercial systems in which the entities are SPs and customers, and in P2P systems where peers are the entities. As a customer wants to do her business in new environments where she has no prior experience with the SPs, a conventional way is to rely on the reputation of the SPs or their history behaviors. Next, we will give a brief overview of how reputation has been calculated and used in scheduling in previous work.

2.1. Reputation computation

In this subsection, based on the previous work of Sonnek [40] and Liang [31], we briefly review some existing reputation models before introducing their applications. For convenience, let R be the reputation of a node. It is worth noting, to our knowledge, R that has been always defined as a scalar in previous work. Different approaches focus on how to calculate this value from multiple inputs. In this paper, as we will see in Section 4, we propose to use a vector to represent reputation.

Ebay's rating mechanism: Ebay, a popular online auction and shopping company [3], uses a simple yet efficient feedback mechanism to compute reputation. Their model is considered the most popular and successful to date. In their model, after using a service or finishing a transaction, users are allowed to rate the SP. In particular, for the ith transaction, a user's rate can be calculated as follows:

$$r_i = \begin{cases} +1 & \text{if she is satisfied} \\ -1 & \text{otherwise} \end{cases}$$

Please cite this article in press as: T. Nguyen, W. Shi, Improving resource efficiency in data centers using reputation-based resource selection, Sustain. Comput.: Inform. Syst. (2012), doi:10.1016/j.suscom.2012.03.001

and the reputation of the corresponding SP is

$$R = \sum_{i=1}^{\text{total executed tasks}} r_i$$

Beta rating: This model was proposed by Jøsang and Ismail [17]. It is based on the beta density function governed by two parameters, α and β. The feedback of client X giving to the SP T is expressed by the function:

$$\varphi(p \mid r_T^X, s_T^X)$$

in which r_T^X and s_T^X are the degree of satisfaction and dissatisfaction of X about T, respectively. The reputation of T is also the probability expectation value:

$$R = E(\varphi) = \frac{r+1}{r+s+2} \quad (4)$$

In the bootstrap state, both r and s are equal to 0; therefore, we have $R = 0.5$. Note that $r+s$ is the total amount of feedback so far. From Eq. (4), we can see that as the amount of satisfied feedback increases, the reputation increases, which is intuitively reasonable.

Weighted rating: This method was proposed by Azzedin and Maheswaran [9]. In this method, they differentiate two important concepts: direct experience and reputation. Each of them has a weight in the reputation formula. The reputation value R is computed as follows:

$$R = \alpha S + \beta F(\alpha, \beta \text{ are the weight values})$$

where $S = r/(r+s)$: the self experience of a node when it communicates with the SP (r and s are the number of satisfied and dissatisfied transactions, respectively). $F = \sum r/(\sum r + \sum s)$: the experiences of other clients when using that SP's service.

In addition, there are other models that take into account risks [30], employ recommendations [25], and filter bad raters or abnormal ratings [6], and so on.

2.2. Reputation-based scheduling

There are many applications for reputation, but we only focus on applications regarding scheduling. Our study was inspired by the recent work of Matei et al. [43], Ananthanarayanan et al. [8] and Sonnek et al. [39] in the sense of using reputation-based scheduling. Matei et al. showed that the MapReduce implementation of Hadoop [2] suffered a significant decrease in performance in heterogeneous environments; therefore, they proposed a LATE scheduler that is based on a new progress score computation method. These progress scores, which are only valid within a specific job execution, can also be considered as the reputation of a node. Like Matei's work, Ananthanarayanan [8] (Mantri) also focused on improving the accuracy of the progress. It used real-time progress reports to detect outliers earlier and act accordingly. We also chose Hadoop as our evaluation framework because we want to tame the heterogeneity that it is suffering from. Apart from their work, we modify its scheduler to leverage Opera instead of proposing a new computation of the progress scores. Another important difference over related work in MapReduce scheduling such as [43,8] is that Opera uses information captured about the cluster "before" executing the job to make scheduling decisions, rather than reacting to poor performance "during" the job execution. Doing this enables us to decide whether to assign small tasks to slow or failure-prone nodes in the cases of executing time-consuming jobs.

Sonek et al.[39] also adopted reputation-based scheduling, but they exploited reputation to decide the size of a group in that if we schedule the same task to each member of that group, we are likely to obtain the correct result thanks to the majority vote. Intuitively, the group size should be small if its members have high reputation and verse versa.

Another closely related work is that of Alunkal et al. [7]. They also proposed a reputation-based resource selection service for Grid, which was based on [26,9], but they did not have any real implementation for the proposed service. Generally, it is difficult to evaluate a work based on trust or reputation because of the lack of real feedback data from users.

Besides, as a resource selector, our work is also similar to Nimrod/G [13], Condor-G [21] and Matchmaker [36]. However, Nimrod/G employed computational economy which used one-value (cost) to represent the resource. Users specify price and deadline, Nimrod finds resource based on the matching between cost and price. It does not care about the quality of the resources and the reputation of resource providers/sellers. All three of them focus on how to match the resource requirement to resource status before the execution of jobs. They did not mention about during the execution time of the jobs. In addition, Opera considers from both the systems and application perspective (e.g. data locality), while Matchmaker focuses on the system perspective.

To detect the failing nodes, our system uses heartbeat as GFS [22] and Hadoop [2]. However, as shown later in Section 4, we are also employing a monitoring tool, Ganglia [1], to provide more information about the behavior of the cluster to the user. Beside Ganglia, there are many network monitoring software such as Zenoss (zenoss.com), Nagios (www.nagios.org), etc. In addition, we share the idea of using load to drive the scheduling decision with other works such as [14,41]. However, the goal of [14] is to improve the grid load balancing, and that of [41] is to improve the utilization of the network.

3. Opera design

In this paper, we argue that it is insufficient to assess an entity via only a scalar value as in previous methods. Existing reputation systems such as eBay[3] use this successfully because they are only concerned about one behavior (successful buy/sell transactions) of an entity. As long as users have many successful transactions, they will have a high reputation in the "eBay world". It does not matter how good or bad they are in the other *worlds* (or contexts). The more information we have about the entities, the better decision we can make when selecting them. In addition, as we focus on scheduling systems, in which the entities are nodes and scheduler, by capturing many different aspects of a node, the system can offer different types of services to users, such as highly available service, powerful computation service, secure service, and trust service, to name a few. Moreover, mapping many components into one value as used in beta or weighted rating does not provide enough flexibility in the selection criteria. Therefore, in our new *Opera* model, each entity or node has a *vector* of reputation instead of a single scalar value. Actually, this representation is somewhat similar to the ClassAd [36] in the Grid Computing world.

In this section, we give a general design of such a model by answering three questions: (1) Who is going to use our model? (2) What services will our model provide or what are the functionalities of our model? and (3) How can we leverage it?

3.1. Opera users

Three types of users are involved in the operation of Opera, including *ratees*, *usual clients* and *reputation system designers*. *Ratees* are those who need to be rated or for whom the reputation is computed. They can be a machine, a service or a service provider. The *usual clients* are the customers of the *ratees*. They may be called *raters* if they provide rating information or *feedback* to the system.

Please cite this article in press as: T. Nguyen, W. Shi, Improving resource efficiency in data centers using reputation-based resource selection, Sustain. Comput.: Inform. Syst. (2012), doi:10.1016/j.suscom.2012.03.001

The *usual clients* can also be the end-user or other components in the system such as the scheduler or resource manager. For example, the scheduler may use Opera to choose suitable nodes in order to improve the performance or the throughput. The resource manager may use Opera to improve the utilization of the system. *Reputation system designers* are researchers who define how to compute the reputation.

3.2. Opera objectives and approach

Ultimately, Opera provides a service that helps clients select a set of suitable candidates, among available resources or services that meet their requirements. In order to do that, Opera first gathers as much information about the ratees as possible, then calculates the *reputation vectors*, and finally selects candidates that have reputation elements agreeable to the user's *criteria*.

The information about the ratees is obtained either from inside the system or the feedback provided by the raters. In other words, the judgment is enabled from both the system and the user's point of view. The information inside the system is collected by *agents* installed at each component. With coarse granularity, these agents can be the user-developed software or built-in services such as the "SNMP" (simple network management protocol) on a machine. With finer granularity, these agents can be the sensors attached on specific components of the machine. Consequently, Opera needs to employ an extensible monitoring tool as an *internal rater* and have appropriate APIs to receive rating information from *external raters*. By doing this, Opera is equipped with self-adaptability, meaning that it does not need any management intervention, which is crucial in unsupervised open environments. For example, a resource manager is capable of feeding the monitored information to statistical machine learning algorithms to achieve self-optimizing [35] in resource utilization. Further, since the extensible monitoring tool can dynamically add new monitoring metrics, it also contributes to the *openness* of Opera by facilitating the Opera users, e.g., researchers in the reputation community, to collect any information they want.

Users can either provide feedback (rating information) to Opera to use reputation built from these feedback or simply use the rating information provided by agents. If users are ratees, i.e. they provide feedback, there are many known issues, such as selfishness, maliciousness, collusion, badmouthing or ballot-stuffing clients, that need to be solved but are out of the scope of this paper. However, as users also need to register with Opera to receive service, they are able to use this information to support models that deal with these listed issues. For example, one can weigh the clients to decide their effect on the reputation of SPs [32].

Using the collected information, Opera calculates the reputation values based on both *predefined* and *user-defined* models of computation. The predefined models are built in as a part of Opera and are mainly used by usual clients. The user-defined models are developed by reputation system designers, and this feature is another aspect of the *openness* of Opera. To enable this extendibility as well as to capture the heterogeneity of ratees, Opera uses a vector structure to store the reputation elements calculated by different models. Each reputation value is an element of the reputation vector and represents a specific point of view about the ratees. For instance, as seen in Section 4, we define the reputation R as $\langle J_{app}, J_{sys}, H_{app}, H_{sys}\rangle$, in which J and H represent the current (just-in-time) and past behaviors of the system; *app* and *sys* denote the application and system point of view, respectively. When reputation system designers want to develop a new model, they add a new element to this reputation vector and provide a method to compute it. The usual clients most likely choose a predefined model that best fits their needs. One may ask these questions: Why do we need to calculate reputation? Why do not we use the collected information directly to select resource? This is because some models may require more than single monitoring information in their computation.

Table 1
The Opera's APIs.

Application programming interfaces	Description
`set_time_window_size(duration)`	Set the size of the time window in which we calculate the reputation
`register(host,port)`	Register newly joined service providers (SPs)
`subscribe(host,port)`	Subscribe by usual clients or raters
`rate(rater_id,ratee_id,trans_id, time, value)`	Rate a transaction of a SP
`getReputationLength()`	Get the current length (number of dimensions) of the reputation vector
`getRateesList()`	Get list of ratees or SPs
`addNewDimension(startupValue)`	Add a new element to the reputation vector
`getReputation(rateeID,dim)`	Get the specific reputation of a specific node
`setReputation(rateeID,dim)`	Define new model of computation

After getting the reputation of each ratee, the users (either usual clients or reputation system designers) specify the criteria to select the resources or use the predefined ones. It is this ability that enables the diversity of a service. The criteria may be based on only one reputation model (an element of the reputation vector) or a combination of many elements. In addition, they also have different types of constraint. For example, a criterion may be choosing "*top three nodes that have the highest availability*," "*nodes that have availability greater than or equal to 3 nines*," "*nodes that have availability higher than the average availability*," "*nodes that have above average availability and below average load*" or "*two nodes that have the smallest number of failed tasks*," etc. Note that in these examples of criteria, availability, load, or the number of failed tasks is considered as a model of reputation.

To summarize, Opera collects the information both from inside and outside of the system, calculates the reputation of the system from it based on predefined and user-defined computation models, and selects candidates based on predefined or user-defined criteria. The main (not all) APIs of Opera are listed in Table 1. It is the user's responsibility to choose which model and criterion are suitable for them as usual clients, or they can define their new model to compute reputation as reputation system designers. This also means that we do not provide solutions to the traditional problems such as how to know if a rating is correct, how to force a client to provide rating information, how to avoid the fact that an SP performs very well at first to obtain high reputation and then turns bad or malicious, how to maintain fairness between newly joined SPs and the old ones, and so on. We envision that the answers to most of these questions are domain-specific and need to have domain knowledge to address them. Resilient reputation management is still an open problem [34].

3.3. System architecture

The overall architecture of the system is shown in Fig. 1. From the figure, we can see how Opera communicates with the existing system. Opera applies the client/server model in which the server (the circle named Opera) communicates with 2 clients represented by 2 circles named Service Providers and Clients/Rater. However, Opera only manages the reputation of the registered SPs only. This model was chosen because it is simple and provides a global and consistent view of the system. Applying this centralized approach, one may challenge Opera's scalability and availability. Actually, they

Please cite this article in press as: T. Nguyen, W. Shi, Improving resource efficiency in data centers using reputation-based resource selection, Sustain. Comput.: Inform. Syst. (2012), doi:10.1016/j.suscom.2012.03.001

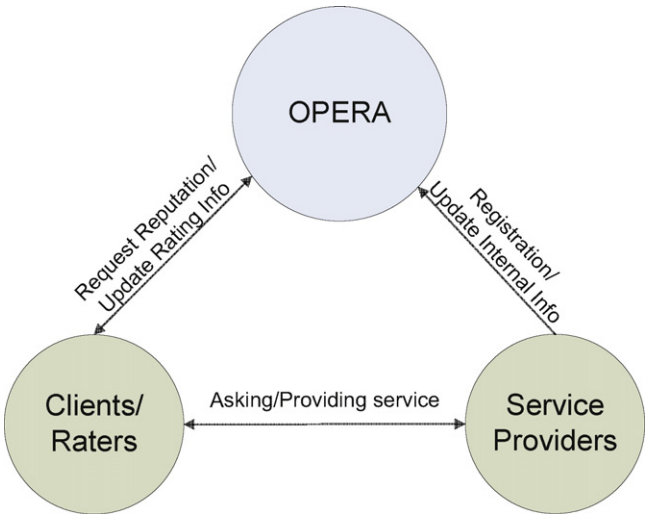


Fig. 1. Opera system design.

depend on the implementation; therefore, it will be discussed later in Section 4.

3.4. Assumptions

Before going to the implementation, it is useful to clarify all the assumptions we made.

The first assumption is that the future behaviors of a node can be inferred from the past behaviors, which is the basic assumption for many other research fields such as machine learning and data mining.

The second assumption is that the Opera server, which is responsible to calculate and store the reputation values of all participating nodes, is secure and trustworthy to simplify the design. This assumption is reasonable given the fact that data-intensive computing systems like Hadoop (which will be used as a specific application scenario in our implementation) often reside in data centers which are highly secure.

4. Materialization and implementation

To evaluate Opera, we implemented Opera using Java and modified the existing scheduler in Hadoop (version 0.20.0) to leverage it. However, we focus only on selecting nodes, *not the time*, to assign tasks because the scheduler of Hadoop is using the *pulling model*, which means as soon as a worker is free, it contacts the master to ask for tasks periodically. The time to assign the tasks cannot be decided by the scheduler. In this case, the usual client of Opera is the Hadoop scheduler, and the ratees (SPs) are nodes in the cluster.

Normally, reputation is often used in the world where participants can cooperate without knowing each other in advance. Nevertheless, in this paper, we chose Hadoop, which is an open-source platform being widely used in data intensive computing, to interact with OPERA because we would like to demonstrate OPERA applicability in different scenarios. As another reason, we want to show that although Hadoop is a highly fault-tolerant system, we can still improve its robustness with OPERA.

As in the design part, Opera also needs to employ a monitoring tool to observe the internal system behaviors (beside the feedback of raters), and we chose *Ganglia* [1], an open-source scalable distributed monitoring system, to fulfil this task.

Generally, in manipulating the reputation data, Opera applies the Share Memory model with the Producer-Consumer mechanism. The producers are the threads that implement the predefined or user-defined reputation computation models. Each thread corresponds to one model. It computes the reputation and periodically updates the corresponding element in the reputation vector. The consumers are usual clients who use reputation elements in their criteria to select candidate SPs.

In this section, we are going to show the detailed computation of each element in the reputation vector of Opera, the selection criteria, what modifications we did on Hadoop to connect with Opera, and the scalability as well as the availability of the model.

4.1. Reputation calculation

After setting up the system, we are now able to define the reputation vector quantitatively. As mentioned in the design section, we recommended a four-element vector for the reputation. Let vector $R = \langle J_{app}, J_{sys}, H_{app}, H_{sys} \rangle$ be the reputation of a node. The first part, J_{app}, is computed based on the requirements of the tasks/jobs that are going to be processed such as the resource type (32 or 64 bits), the OS (Windows, Linux, AIX, Mac), and the data required. For consistency, from now on, we define that a job is composed of many tasks. In Hadoop, there are two types of tasks: map and reduce. Within a job, all map tasks are the same, and so are all reduce tasks. The second part, J_{sys}, is computed based on the current system status such as CPU usage, network activities, free memory percentage. The third part, H_{app}, is the reputation of the node from the application aspect or the usual client (scheduler) point of view. Apparently, H_{app} is calculated based on the feedback of the usual client (scheduler) over a long time. The last part, H_{sys}, denotes the long-term system characteristics such as the reliability or availability of the node.

J_{app} and J_{sys} can be categorized as just-in-time reputation (JR) of a node, which describes its current status or recent behaviors within a small time window. H_{app} and H_{sys} are history-based reputation used to capture the past behaviors of a node. This classification arises from the observation that sometimes a reputation of a node can be affected temporarily. For example, if node A has a good reputation so far, the scheduler very likely assigns the task to it. Unfortunately, as we prefer to schedule tasks to A, it starts getting a heavier workload and, hence, processes the tasks very slowly. In fact, the reputation of that node at that particular time should be low so that our reputation-based scheduler does not assign tasks to it. Another observation is from the data locality point of view. Despite the fact that node A has had a high reputation so far, if it does not host the data required by the task, the scheduler should not choose it. In general, there are two dimensions of this JR. One relates to the task requirements, and the other relates to the current status of the nodes. To show the *openness* of Opera, we implemented JR (J_{app} and J_{sys}) as the predefined components and history-based reputation (H_{app} and H_{sys}) as user-defined components. The next step is how to quantitatively calculate them.

Let vector $R = \langle J_{app}, J_{sys}, H_{app}, H_{sys} \rangle$ be the reputation of a node within $[t_0, t]$ and n be the number of node.

As mentioned above, J_{app} is computed from the job description. There are many requirements in a job description. Some of them are single-value, and the others are multi-value. For example, while the required OS belongs to the single-value type, the required data blocks belong to the multi-value type. Let x_i be the indicator variable of a single-value requirement and the target node be the node we want to calculate the reputation for:

$$x_i = \begin{cases} 1 & \text{if the target node fits the requirement } i\text{th} \\ 0 & \text{otherwise.} \end{cases}$$

In certain circumstances, it is critical to meet multiple requirements simultaneously. For example, if a job requires running on a

Please cite this article in press as: T. Nguyen, W. Shi, Improving resource efficiency in data centers using reputation-based resource selection, Sustain. Comput.: Inform. Syst. (2012), doi:10.1016/j.suscom.2012.03.001

64-bit system, we cannot assign it to the 32-bit. If that is the case we compute:

$$p_{\text{critical}} = \prod_{i=1}^{\text{The total critical requirements}} x_i$$

Note that p_{critical} is equal to 1 only if all critical requirements are met; otherwise, its value is 0.

Another important multi-value requirement is the data required by the task. If a node has the data the task requires, it should have a high reputation toward that specific task. We capture this as:

$$p_{\text{data}} = \frac{\text{The \# of blocks stored on the target node}}{\text{The total \# of required blocks of the task}}$$

At the end, the first part, J_{app}, is computed as the following:

$$J_{\text{app}} = p_{\text{critical}} \times p_{\text{data}}$$

The second part, J_{sys}, demonstrates the current system status of a node. By current status, we mean in the short time $[t_0', t_1']$ which is much smaller than the window size $[t_0, t]$ in the computation of H_{sys} and H_{app}. In Ganglia, for example, these short periods are 5, 10 and 15 min. Basically, J_{sys} is computed from Ganglia's metrics. Since there are many metrics available in Ganglia, we can derive many ways to compute J_{sys}. One simple example, which takes into account both CPU and memory, is

$$J_{\text{sys}} = \text{the idle CPU percentage} \times \text{free memory}$$

If the CPU of a node is too busy and has less free memory, it should have a low reputation. It is worth noting that Opera can easily integrate other methods of computing J_{sys}, but that is beyond the scope of this paper.

The third part, H_{app}, is the history-based reputation from the user point of view. We utilized the information in the log files of Hadoop to calculate it as followings:

$$H_{\text{app}} = 1 - \frac{\text{total number of failed/killed tasks}}{\text{total assigned tasks}}$$

From the scheduler point of view, a node that has more failed/killed tasks over its total given tasks should have a lower reputation. As initializing, Opera reads this information from the history directory of Hadoop and calculates the start-up reputation of all nodes. Then, after finishing each job, it reads the Hadoop log file and updates this type of reputation accordingly. This type of reputation is *augmented* after each job execution.

The last part, H_{sys}, is a purely system-related reputation. We can use either the availability or reliability of a node for this value:

$$H_{\text{sys}} = \text{availability}(t) = \frac{\text{actual up time}}{t - t_0} = \frac{\text{total replies}}{\text{total heartbeats sent}}$$

For the reliability, let $f(t)$ be the unconditional failure rate of a node

$f(t)$ = the probability that a node will fail between t and $t + d_t$ ($f(t)$ is a probability density function.)

Let $F(t)$ be the cumulative probability that a node has failed by the time t:

$$F(t) = \int_{t_0}^{t} f(t)dt$$

The cumulative probability that the node has not failed from t_0 to t, given the fact that the node was up at t_0, is the reliability of that node during $[t_0, t]$:

$$H_{\text{sys}} = \text{reliability}(t) = 1 - F(t)$$

From this formula, if we know the distribution of the failure rate function, we can easily compute the reliability of a node. With the assumption of *memoryless* failures, we may apply another simpler formula:

$$H_{\text{sys}} = \text{reliability}(t) = e^{-(t/\text{MTTF})}$$

in which MTTF is the mean time for the node to fail.

Table 2 summarized the computation of the reputation vector.

Table 2
The summary of reputation computation.

J_{app}	$p_{\text{critical}} \times p_{\text{data}}$
J_{sys}	the idle CPU percentage × free memory
H_{app}	$1 - \frac{\text{total number of failed/killed tasks}}{\text{total assigned tasks}}$
H_{sys}	$e^{-(t/\text{MTTF})}$

4.2. The selection criteria

We implemented three selection criteria "above average", "top one-third" and "mixed". The first two were implemented as part of Opera representing predefined criteria. The last one was implemented in Hadoop as user-defined criteria.

The "above average" criterion return nodes that has the reputation greater than or equal to the average reputation of available nodes. The reputation here refers to a specific dimension of the reputation vector.

The "top one-third" criterion first ranks all candidates according to a specific dimension of the reputation vector and chooses nodes among the top one-third of the candidate set.

While the first and second criteria consider only one element of the reputation vector, the third one takes two of them into account. Since this is the user-defined criterion, it depends on the specific situation.

4.3. Hadoop modification

After implementing Opera, we need to modify Hadoop to use it. In this subsection, we first discuss the Hadoop scheduler briefly then explain how to change it. The modification in the speculative execution is explained last.

Hadoop has two main parts: the file system (HDFS) and the MapReduce framework [19]. In Hadoop, a MapReduce job contains many tasks of two types: map or reduce. Hadoop uses a master/slave model and a polling mechanism (pull model) to schedule tasks. Every time a worker (Tasktracker) has available execution slots, it contacts the master (Jobtracker) to ask for a task periodically. After performing certain checks such as `is-blacklisted`, the master finds a task to assign to it.

To use Opera in such a model, the idea is that when a node comes to ask for tasks, the master asks Opera for its reputation, evaluates it against a selected criteria, and assigns tasks to it accordingly. To implement this, we developed an additional method called `shouldAssignTasks(tasktracker)`, which is invoked when a tasktracker comes to ask for tasks, and it returns true if the reputation of the tasktracker meets the criteria. Actually, we can either specify new customized criteria right in this method or choose the predefined ones from Opera.

It is noteworthy that if we apply certain selection criterion to a fixed list of workers, we may decrease the system utilization. For example, among 10 workers, if our criterion is "choosing above median reputation nodes", the scheduler can never assign tasks to more than 5 workers. This is because we also include the nodes which already received tasks in the candidates list to assign new tasks. We should instead assign tasks to available candidates only. As a result, we maintained a list of available workers called `candidatesList` and applied the selected criteria to this list only. Before executing any job, `candidatesList` contains all workers in the

Please cite this article in press as: T. Nguyen, W. Shi, Improving resource efficiency in data centers using reputation-based resource selection, Sustain. Comput.: Inform. Syst. (2012), doi:10.1016/j.suscom.2012.03.001

system. In executing a job, it shrinks as the number of nodes that receive tasks increases, and grows as they finish tasks. Since the list keeps changing over time, we can always select nodes successfully. The detail algorithm is shown in Algorithm 1

Algorithm 1 *(The modified shouldAssignTasks).*

```
Require: taskTracker, candidates, reputationType
  reputationVvalue ← OPERA.getReputation(tasktracker, reputationType)
  if criteria is "above average" then
    reputationThreshold ← OPERA.computeThreshold(candidates, reputationType)
    if reputationValue < reputationThreshold then
      return false
    end if
  else
    if criteria is "mixed" then
      candidateSet1 = OPERA.computeThreshold2(candidates,dim2)
      candidateSet2 = OPERA.computeThreshold2(candidates,dim3)
      if (candidateSet1 ⋂ candidateSet2) ≠ ø then
        if tasktracker ∈ (candidateSet1 ⋂ candidateSet2) then
          return true
        else
          return false
        end if
      else
        if candidateSet2 ≠ ø and tasktracker ∈ candidateSet2 then
          return true
        else
          return false
        end if
      end if
  end if
else [criteria is "top one-third"]
  sortedMap = OPERA.ranking(candidates, reputationType
  tail = sortedMap.tailMap(tasktracker)
  if tail.size() ≤ (candidates.size()/3) then
      return true
  else
      return false
  end if
end if
return true
```

In our implementation, Opera is also used in the speculative execution of Hadoop in which slow progress tasks are speculatively executed on other nodes. We used Opera to find such nodes. Basically, the new chosen nodes should be "better" (i.e. has higher reputation) than the current hosts of the slow progress tasks.

4.4. Scalability and availability

For internal system information, we employed Ganglia, which has been widely used and can handle a cluster of 2000 nodes. Consequently, we have scalability in collecting system information. For user-provide information such as feedback, since most rating messages from raters are often small in size, Opera can handle many of them easily. As a result, it is safe to claim Opera is *scalable*.

To increase the *availability* of the system, we can have a secondary Opera server as a backup plan. This secondary server periodically backs up the reputation vector from the ordinary server. In case of failure, since the history-based reputation isn't affected much in missing short-period information, we can still use the "out-of-date" historical reputation. For the JR, since Ganglia multicasts monitor information to a channel, the secondary server can easily obtain information from it and compute the JR themselves; hence, it always has up-to-date JR. In addition, Ganglia also allows the user to define their own metric; therefore, it is extensible and fits best to our Opera design described in the previous section.

Table 3
The Opera's testbed.

Type	Machines #	CPU	Memory	OS
Server	1	4 × 2 GHz	4 GB	RHEL5 x86_64
Server	1	1 × 2.34 GHz	1 GB	Fedora 8
Workstation	21	2 × 1 GHz	512 MB	Ubuntu 8.04

5. Performance evaluation

5.1. Experiment setup

Since Opera is used to aid resource selection based on multi-property criteria, we need to create a heterogeneous systems to evaluate it. However, our available testbed is homogeneous in term of hardware because as most local institute clusters, it was built at the same time with the same hardware configuration. As a result, we created two types of heterogeneity, which are *availability* and *workload*, instead of heterogeneity in the hardware. Some nodes in the system are configured to fail during certain periods to provide the differences in availability. Since we focus on fail-stop only, we simulated a failure by disabling the network connection to that node (turn off the network interface) during predefined periods. The differences in workload were created by scripts that recompile the newest Linux kernel at a certain time to keep the CPUs busy and the memory full, and "ping" the local host heavily to pollute the network. As our previous assumptions, we configured the experiments so that they run only on half of the available resources.

5.1.1. Testbed

Our testbed cluster has a total of 23 machines including 2 servers and is described in detail in Table 3. One server is the namenode of Hadoop. The others contain both the Opera server and JobTracker. It also runs Ganglia meta data daemon *-gmetad-* to collect information from monitor daemons - gmond. All other nodes are tasktrackers/datanodes and installed *gmond* (agents that collect information). The number of replicas in Hadoop is 3. The monitoring metrics are collected, sent and updated at different rates. For example, the CPU-related metrics are collected every 20 s by gmond at each worker. They are sent to the gmetad every 90 s, and the gmetad updates its own RRD database every 15 s. In addition, these rates are different between metrics.

5.1.2. Applications

The applications we used are packaged together with the Hadoop distribution. They are all in Hadoop's example package and are MapReduce programs. We utilized three programs: *Sort*, *Grep* and *WordCount*. Their functionalities are self-described by their name. The Grep and WordCount input is a 7.7 GB text file generated from log files and a Shakespeare play. Grep finds a given pattern in a given input file. The search pattern is written as a regular expression, and the matched results are stored in an output file. WordCount counts the number of occurrences of each word in a given input set. The input of Sort is 3.9 GB and is generated randomly by the *Randomwriter* – another built-in application of Hadoop.

5.2. Metrics

We used two main metrics in this evaluation: *execution time* and *the number of failed/killed tasks* of a job. The execution time, as usual, represents the performance of the system. The number of failed/killed tasks expresses the usage efficiency of the system. Intuitively, the higher this value, the lower the efficiency is because we spend resources such as energy, computation, network bandwidth on executing these failed/killed tasks but receive nothing.

Please cite this article in press as: T. Nguyen, W. Shi, Improving resource efficiency in data centers using reputation-based resource selection, Sustain. Comput.: Inform. Syst. (2012), doi:10.1016/j.suscom.2012.03.001

Fig. 2. The effects of heterogeneity on Hadoop performance.

We should calculate the usage efficiency explicitly here, but since its computation requires information about the resource (CPU time or energy) each failed/killed task has spent before it fails or is killed, we do not have it at this moment.

5.3. Heterogeneity analysis

We first show that our two heterogeneous types, *availability* and *workload*, also affect Hadoop performance as Matei's work on [43] although we used different methods. Figs. 2 and 3 show the execution time and the total number of failed/killed tasks of three applications in Hadoop under failures and other workloads. The number of failed nodes and high workload nodes are nearly half of the system (8/21 and 9/21, respectively). In these experiments, we normalize the measurement to the healthy system which has speculative execution enabled yet has no failures or heavy workload.

The failure periods are chosen so that the whole job is still successful because as we extend these periods, the number of failed task attempts increases and so does the job execution time. At a certain level, if we keep extending them, the whole job fails as the number of task failures of any task reaches its limitation, which is 4 by default. In the systems with failures, the executed time increases 19%, 131% and 89%, and the number of failed/killed tasks increases 94.73%, 77.78% and 267% for *wordcount, grep,* and *sort*, respectively. In the systems with heavy workloads, the executed time increases 12%, 42% and 96%, and the waste increases 5.26%, 44.45% and 103% for *wordcount, grep, sort*, respectively. With these defects caused by heterogeneity, the next subsection will detail the improvement by Opera.

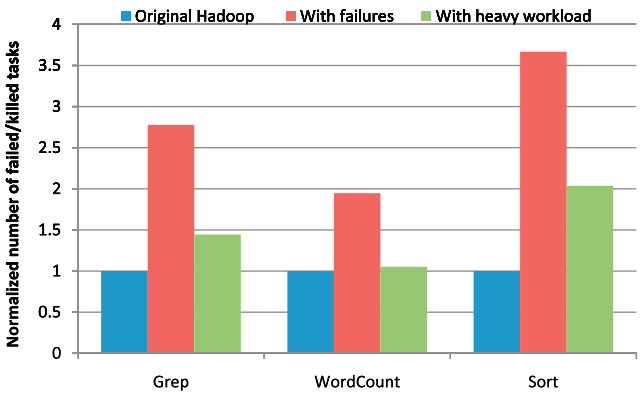


Fig. 3. The effects of heterogeneity on Hadoop number of failed/killed tasks.

5.4. Improvements

Our goal in this section is to illustrate that with the help of Opera, the scheduler can avoid nodes with either low availability or high workload, or both. The Sort benchmark is going to be used as the primary workload from now on because it is popular and also used as the main benchmark for evaluating Hadoop at Yahoo [43].

There are two factors that affect the selection process and, hence, are considered here: *the reputation element* and *the selection criteria*. While the reputation element factor represents which dimension(s) we consider, the criteria is the policy that is applied on it(them). For instance, the selection policy "choosing the nodes that have J_{sys} above the average" means the "above average" criterion is applied to the element, J_{sys}, of the reputation vector. As a result, to study the effects of each of these two factors, we run the experiments with different values for one while keeping the other fixed. The following two subsections describe in detail the effect of the reputation elements, which in fact represent many different aspects of the heterogeneity and the selection criteria. The last subsection is comparing OPERA with the LATE scheduling [43].

5.4.1. The heterogeneity

Each aspect of the heterogeneity is captured by an element in the reputation vector. In the previous section, we mentioned two types of heterogeneity: availability (failures) and load. Essentially, they are the two aspects of the heterogeneity that we consider and are captured by H_{sys} and J_{sys}, respectively. In this part, we are going to do the experiments with each single aspect of the heterogeneity first and then with both. The selection criterion factor applied in these experiments is fixed. It is the "above average". Tables 4 and 5 contain the results of running Sort with and without Opera on a system with heterogeneity in availability and load, respectively. Since the results vary greatly among executions, the tables list the worst, best and average values of the results to give an idea of their range.

From Table 4, one can see that the execution time and the number of failed/killed tasks are reduced by 23% and 41% on average, respectively, when using Opera under the presence of failure. Similarly, under the presence of other workload, the execution time and the number of failed/killed tasks can be improved on average by 11% and 59%, respectively, as shown in Table 5.

Finally, we did the experiments with four configurations as depicted in Fig. 4 under an environment that has both failure and heavyloaded nodes. It is worth noting that this environment differs from that of the two previous experiments. From the figure, we can see that the last configuration, which took into account both availability and workload, outperformed the others. Its performance was improved 26% in comparison to the original Hadoop. It demonstrates the ability to handle criteria built on more than one element of the reputation vector. This makes Opera unique, separating it from other previous work.

It is worth to note that tasks of a job in Hadoop fail independently which means that only the failed tasks need to be re-executed. If OPERA is used in the models in which task failures are highly correlated and require the whole job to be re-executed, the efficiency improvement would be much more significant.

5.4.2. The selection criteria

Now we are in a position to study the effects of the selection criteria. We run the Sort again on Hadoop with different selection criteria yet under the same, *ideal environment* (no failures, no other workload). Obviously, in such an environment, Opera brings no benefits but overhead since it is designed for heterogeneous environments. Consequently, with this experiment design, we cannot study only the effects of different selection criteria but also the overhead of Opera.

Please cite this article in press as: T. Nguyen, W. Shi, Improving resource efficiency in data centers using reputation-based resource selection, Sustain. Comput.: Inform. Syst. (2012), doi:10.1016/j.suscom.2012.03.001

Table 4
The results of using different availability.

	Original Hadoop			Opera		
	Worst	Best	Average	Worst	Best	Average
Execution time (s)	706	652	678	538	507	523
Number of failed/killed tasks	27	17	23	16	12	13

Table 5
The results of using different workloads.

	Original Hadoop			Opera		
	worst	best	average	worst	best	average
Execution time (s)	662	546	606	563	499	539
Number of failed/killed tasks	22	11	18	9	5	7

(a) Execution time (b) The number of failed/killed tasks

Fig. 4. A comparison of three different ways of using two heterogeneous types.

The Sort was run at least 5 times for each configuration. There are a total of 4 configurations. The first one is the original Hadoop. The rest three use the three criteria defined in Section 4 to select nodes.

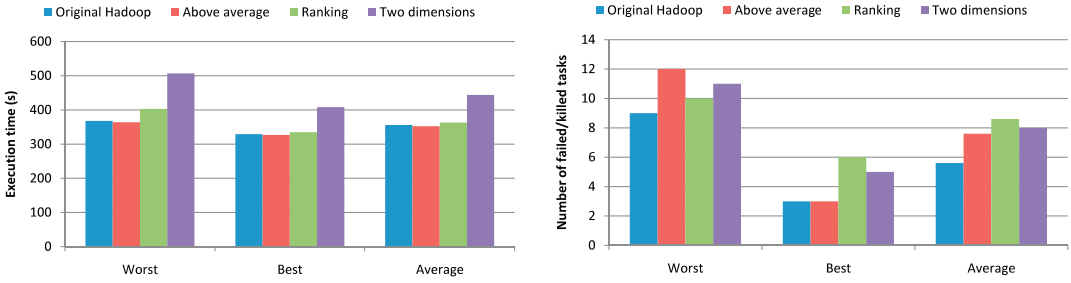
Intuitively, one can tell that the execution time increases as the criteria become more complicated. Moreover, in the ideal environment, Opera should expose its overhead over the original Hadoop. In other words, the execution time should increase from the first to the fourth configuration. Fig. 5 consolidates this intuition. On average, the execution times of the first 3 configurations are almost the same while the last one introduces a 25% additional delay. The reason is that the last (use-defined) criterion is more complicated than the other two. Furthermore, the selection code is executed every time a worker asks for tasks which is quite often. The number of killed/failed tasks varies greatly among executions. For example, the worst case in the original Hadoop is 9, and the best one is 3. On average, Opera has about 2 extra killed/failed tasks.

One may wonder why the performance of the two dimension criteria is worse in this part than the others while it is better in the previous part. This is because the previous experiments were done in a hostile environment which contains both failures and other load. To reduce this overhead, one possible approach is to use feedback control mechanism to select suitable criteria since we have a lot of monitoring information.

5.4.3. Opera and LATE

As mentioned earlier in Section 2, LATE [43] also improve the performance of Hadoop in heterogeneous environments. Although this is not exactly the same goal with us, our approaches are somewhat related. Therefore, we would like to compare the performance of Opera and LATE here.

There are four configurations in this experiment. For each of them, we ran the sort application on the same input of 1 GB of data three times. The first two configurations are LATE with and without failure. Since LATE has been integrated into Hadoop version 0.21.0, we used that version as LATE configuration in the experiment. The remaining two configurations are Opera with and without failure. The criterion we applied here for Opera is of the type "above average", and there was no other workload in the system.



Execution time Number of failed/killed tasks

Fig. 5. Study of three different selection criteria on Opera.

Please cite this article in press as: T. Nguyen, W. Shi, Improving resource efficiency in data centers using reputation-based resource selection, Sustain. Comput.: Inform. Syst. (2012), doi:10.1016/j.suscom.2012.03.001

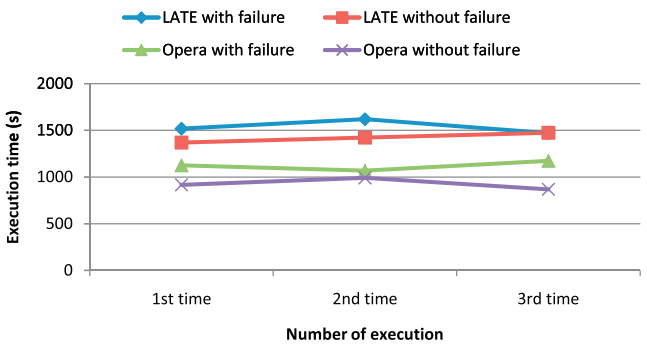


Fig. 6. The performance of Opera and LATE.

Fig. 8. The effects predictor's accuracy on Opera.

Fig. 6 shows the results of the experiment. In the figure, we can observe the outperform of Opera over LATE. This improvement can be explained as the result of the difference in the time of scheduling and the way to select the tasktrackers. While Opera is performed right at the beginning of the job execution, for every tasks, LATE focuses on the speculative tasks and is only trigger when stragglers are detected. While Opera tries to select "good" candidates for every task assignment, LATE focuses on which tasks should be speculative executed but it does not care who will take care of such tasks.

5.5. Effects of sampling rate

In this section, since we used a monitoring tool polling the clients to capture the behavior of the system, we would like to study the effects of changing the sampling rates to the performance of the system. However, we only focus on changing rate in the JR because it does not make much sense in the history-based part. Basically, the history-based reputations are used to predict the future based on the past information, which is expressed in traces such as failure trace. The past information is often measured during a long period of time, which is much longer than the execution time of a job. The sampling intervals in this situation are often long; therefore, during the execution of a job, only a few data are sampled. These data clearly cannot affect the history-based reputation severely.

Fig. 7 shows the results of sorting 10.5 GB of data under the same heavy load distribution with different sampling rates. Intuitively, the higher the rates, the more accurate the monitor is but the higher traffic in the network. As we can see, when we increase the sampling interval, the monitor loses the sensitivity of capturing the behavior of the system and, hence, produces worse performance. Further, we notice that increasing the sampling rate too high is not good because the benefit in performance improvement cannot outweigh the network overhead at some point. For example, in Fig. 7 the performances at the sampling intervals of 20 and 30 s are essentially the same.

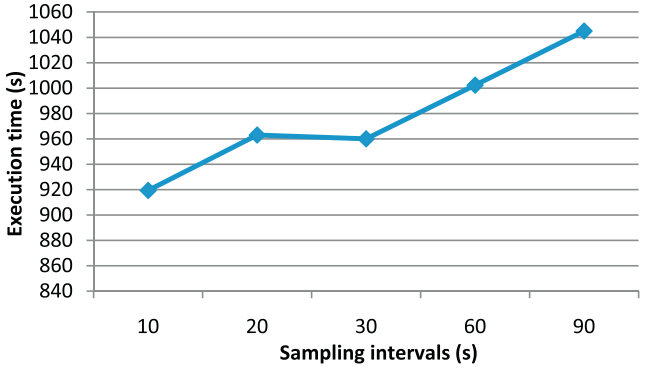


Fig. 7. The performance of Opera with different sampling rates.

5.6. Effects of the predictors

Since our system depends on a predictor to speculate bad reputations entities in the future based on the past behaviors, it is good to see how such predictor's accuracy may affect our system. To answer this question, we conducted another experiment. In this experiment, we ran the *Sort* application on our modified Hadoop with 20 reduce tasks. The selection criterion was the "above average" on the H_{sys} with the present of failures. The failure trace configuration (time, machine and duration) was the same as other experiments. The execution time and the number of failed/killed task of this test are shown in Fig. 8. The x axis is the false positive percentage of the predictor. For example, 25% means that one of every four guess from the predictor is false. A general observation from the figure is that as the percentage of false positives (the inaccuracy) of the predictor increases, our system performance is getting worse. However, it's not so bad since the performance is reduced only approximately 25% and the number of failed/killed tasks increases about 20% with the very inaccurate predictor (1 guess is wrong out of 2). It's worth to note that with this experiment, the original Hadoop did not even finish the job successfully. Also, as other experiments, the figure shows the average of several executions only.

5.7. Discussion

Although we implemented the calculation of all 4 elements reputation vector, the above evaluation only used 2 of them (J_{sys} and H_{sys}). We did not use J_{app}, which represents the job requirements and data locality, because of two reasons corresponding to its two constituents. The first one is related to the critical requirements. It is trivial to avoid nodes that do not meet them because these nodes all have the reputation of 0. It make no sense if we put, for example, some Windows or Mac machines to the systems, set the job description files, and show that Opera does not choose them. The second reason relates to the data locality. We ignored this too because Hadoop claims that it already took into account the data locality as it schedules map tasks.

For H_{app}, actually, it represents the traditional reputation systems in which the customer is the Hadoop's master node and the service providers are the workers. After each transaction (job) the customer (master) rates the service providers (workers). We introduce H_{app} to demonstrate that Opera can easily cover the traditional reputation. We skipped H_{app} because,unlike the commercial systems, our specific system does not assume the malicious behavior of nodes. All service providers are objective and try to behave correctly. Therefore, even if a node has had a high

Please cite this article in press as: T. Nguyen, W. Shi, Improving resource efficiency in data centers using reputation-based resource selection, Sustain. Comput.: Inform. Syst. (2012), doi:10.1016/j.suscom.2012.03.001

number of killed/failed tasks, we shouldn't punish it *again* because they were done unintentionally. We used "again" because the objective behaviors were already considered in other dimensions.

In these experiments, we assumed that the future behavior of nodes can be obtained exactly from their reputation vectors. In reality, this assumption is too strong. We can only predict such information with a certain probability. Fortunately, the research about such prediction is well studied [29,28,37,33], and the node behavior estimation, especially in the near future [44], is actually accurate enough.

6. Energy efficiency evaluation

In this section, we will evaluate the (energy) usage efficacy of using Opera. We are going to measure the energy of both compute machines and the network switches using wattsup meters [5]. This time, for the sake of simplicity, our Hadoop system includes only 20 slave machines. We did not measure the master node since they mostly did the same job for all configurations. In addition, the actual computation happens in the slaves, and the data is also stored in and transferred between them.

The power dissipation was measured at the finest granularity available of wattsup devices which is 1 second. At this granularity, the reading software sometimes could not read info from the devices. There are several missing values and, therefore, we only show here the average of the remaining power values over the interested time periods.

6.1. Energy of the cluster

We used 4 wattsup meters to measure the power dissipation to all compute machines in the system. Three of them were used to measure the power of 4 machines each, and one measured the rest 8 machines. The time of all machines was guaranteed to be synchronized during all experiments. Since the wattsup devices kept recording data to files during the experiment and the time is synchronized, we only need to mark the periods of interest to compute the average power in such periods later. For example, to measure the base line power (no applications but the OS), after starting all machines in the system and waiting for them to be stabilized, we recorded the time start, wait 5 min and recorded the time end of this period. Then after the whole experiment finished, we compute the average power during this recorded period for each wattsup and add them up to have the average power of the whole system.

In this experiment, we measured the power in different scenarios: there is no application except the OS (base line); running only Hadoop; and running Hadoop with three MapReduce applications namely grep, wordcount and sort (with different configurations each). The base line power of the cluster is 1713.16 W, and the power of running Hadoop (without any job execution) is 1718.59 W. We can see that running Hadoop itself raises the average power to 5.43 W in comparison to the base line.

There are three configurations for each application. This first one is running the applications with the original Hadoop without any failure. The second one is executing them with failure using original Hadoop. We did not include heavy load here because we only want to measure the energy of our job. Using our current measurement method, we can't exclude energy consumption of such load. The third configuration is using Opera with failure.

For the Hadoop with application configurations, we run the application several times, compute the average power as well as the execution time of each configuration. From these values, we can derive the average energy consumption of the whole system for each configuration and the usage efficiency (defined in Section 1). More details will be described as follows.

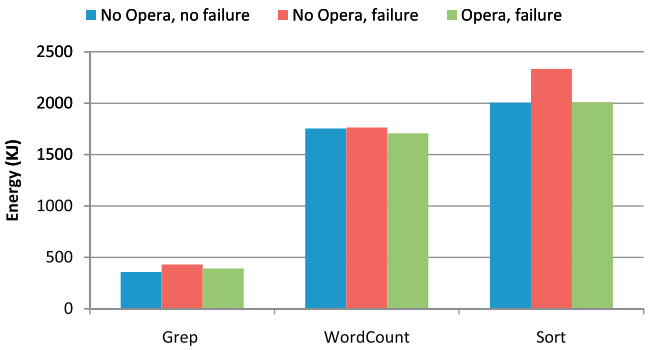


Fig. 9. Energy consumption of machines in the cluster.

Table 6
Comparison of energy usage efficiency of machines in the cluster.

	Grep	WordCount	Sort
No Opera, no failure	100%	100%	100%
No Opera, failure	83%	99.43%	85.90%
Opera, failure	91.20%	102.76%	99.79%
Improvement	9.87%	3.35%	16.17%

Fig. 9 illustrates that running applications in Hadoop increased the energy consumption significantly (in comparison to the base line). The figure also points out that, with failure, the energy consumption of the original Hadoop is increased in all cases, but it is decreased when employing Opera. The improvement is not significant in the Grep and WordCount applications because such applications has only one reduce, and the reduce phase dominates the execution time of the whole job. While the benefit of Opera comes from selecting good candidates, this selection only happens once in the reduce phase of such applications and therefore, they are not benefit much from Opera unless the original Hadoop selects a "bad" node for the reduce task.

Table 6 details the usage efficiency of the whole system with different applications and configurations. In general, Opera is always more efficient than the original Hadoop with failure. It's worth to note that although the usage efficiency of the WordCount almost does not hurt from failure (they have one time-consumed reduce task), it is still improved by Opera. Opera can even be more efficient than the ideal case (no failure) thank to the significant reduction in the number of failed/killed tasks.

6.2. Energy of the network

In this part, we did the same experiments as the previous part (the same system, applications, configurations, etc.), but we used 2 wattsup to connect to two 16-port switches to measure their power. One switch connects 13 nodes, and the other connects 7 nodes. After doing the experiments, we observed that the power dissipation of the switches, in general, did not change too much among applications as well as configurations. Particularly, the 13-node switch has the average power of 68.2 W with the standard deviation $\sigma = 0.14$, and the 7-node switch has the average power of 63.94 W with the standard deviation $\sigma = 0.07$.

Fig. 10 and Table 7 are interpreted as the previous part except that they report the energy consumptions of the network switches. Fig. 10 again supports the general trend: the failure does increase the energy consumption (of the switches), and Opera does help to soothe the negative effect of failure. However, the amount of benefit depends heavily on the application computation models. Generally, Opera is helpful for applications that need to make many

Please cite this article in press as: T. Nguyen, W. Shi, Improving resource efficiency in data centers using reputation-based resource selection, Sustain. Comput.: Inform. Syst. (2012), doi:10.1016/j.suscom.2012.03.001

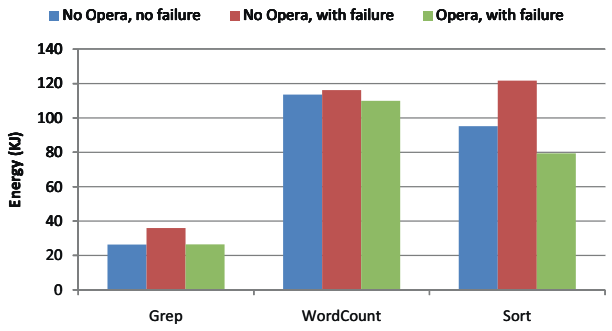


Fig. 10. Energy consumption of network switches.

Table 7
Comparison of energy usage efficiency of network switches.

	Grep	WordCount	Sort
No Opera, no failure	100%	100%	100%
No Opera, failure	73.26%	97.75%	78.22%
Opera, failure	99.60%	103.28%	119.93%
Improvement	35.96%	5.66%	53.32%

task assignments and in the cases where the probability to select bad nodes is high.

Table 7 detailed the improvement of Opera in term of the network energy usage efficiency. While the efficiency improvement is only 5.66% for the WordCount, it jumps to 53.32% for the Sort application.

7. Conclusions

To improve the efficacy in using resources, such as energy, we need to reduce the resource waste. Particularly, in this work, we would like to lower the number of task re-execution, by choosing appropriate machines to execute the tasks. We successfully designed and developed Opera, which is used to select machines from a candidate set in the systems based on their reputations. The system not only allows us to reduce the waste caused by failure but also enables users to select service providers by predefined as well as user-defined selection criteria and enables reputation researchers to develop their own reputation computation models.

We proposed using vectors to represent reputation and experiment results proved that it is a suitable approach to deal with the heterogeneity and energy efficiency. We set up a specific scenario which is using Opera to aid Hadoop scheduler to select workers. Based on the scenario, we developed four reputation computation models. Two of these models are built-in and the others are defined outside Opera to show the extendibility of Opera. The experiment results expressed both the benefits and cost of using Opera.

Acknowledgements

We would like to thank our system integrator, Andrew Murrell, for his help in measuring the energy of the whole system and anonymous reviewer who gave us invaluable comments to improve this manuscript.

References

[1] http://ganglia.info/.
[2] http://wiki.apache.org/hadoop/.
[3] http://www.ebay.com/.
[4] http://www.gartner.com.
[5] http://www.wattsupmeters.com.
[6] A.J.A. Whitby, J. Indulska, Filtering out unfair ratings in bayesian reputation systems, in: The Icfain Journal of Management Research, vol. 4, February 2003, pp. 48–64.
[7] B.K. Alunkal, I. Veljkovic, G.V. Laszewski, K. Amin, Reputation-based grid resource selection, in: Workshop on Adaptive Grid Middleware, 2003, p. 28.
[8] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, E. Harris, Reining in the outliers in Map-Reduce Clusters using Mantri, in: OSDI'10: Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation, USENIX Association, Berkeley, CA, USA, 2010.
[9] F. Azzedin, M. Maheswaran, Evolving and managing trust in grid computing systems, in: Proceedings of IEEE Canadian Conference on Electrical & Computer Engineering, May 2002, pp. 1424–1429.
[10] L.A. Barroso, U. Hölzle, The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, volume Lecture #6.
[11] L.A. Barroso, U. Hölzle, The case for energy-proportional computing, Computer 40 (12) (2007) 33–37.
[12] P. Bohrer, E.N. Elnozahy, T. Keller, M. Kistler, C. Lefurgy, C. McDowell, R. Rajamony, The Case for Power Management in Web Servers, 2002, pp. 261–289.
[13] R. Buyya, D. Abramson, J. Giddy, Nimrod-G: an architecture for a resource management and scheduling system in a global computational grid, in: HPCAsia 2000, May 2000.
[14] J. Cao, S.A. Jarvis, S. Saini, G.R. Nudd, Gridflow: workflow management for grid computing, in: CCGRID'03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid, IEEE Computer Society, Washington, DC, USA, 2003, p. 198.
[15] C. Belady, A. Rawson, J. Pfleuger, T. Cader, Green Grid Data Center Power Efficiency Metrics: Pue and Dcie, 2008, www.thegreengrid.org.
[16] J. Clark, US government to consolidate data centers? The Data Center Journal (2010).
[17] B.E. Commerce, A. Jøsang, R. Ismail, The beta reputation system, in: Proceedings of the 15th Bled Electronic Commerce Conference, 2002.
[18] D. Azevedo, J. Haas, J. Cooley, M. Monroe, P. Lembke, How to Measure and Report Pue and Dcie, 2008, www.thegreengrid.org.
[19] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Communications of the ACM 51 (1) (2008) 107–113.
[20] EPA, EPA Report to Congress on Server and Data Center Energy Efficiency, Technical Report, U.S. Environmental Protection Agency, 2007.
[21] J. Frey, T. Tannenbaum, M. Livny, I. Foster, S. Tuecke, Condor-g: a computation management agent for multi-institutional grids., in: Proc. of the 10th International Symposium on High Performance Distributed Computing (HPDC-10), August 2001.
[22] S. Ghemawat, H. Gobioff, S. Leung, The google file system, in: Proceedings of SOSP'03, Lake George, NY, October 2003.
[23] M. Hill, Amdahl's Law in the Multicore Era, January 2010.
[24] M.H. Jed Scaramella, Service-based approaches to improving data center thermal and power efficiencies, IDC White Paper, May 2007.
[25] M.A. Jøsang, E. Gray, Analysing topologies of transitive trust, in: Proceedings of the Workshop of Formal Aspects of Security and Trust (FAST), September 2003.
[26] S. Kamvar, M.T. Schlosser, H. Gacia-Molina, The eigentrust algorithm for reputation management in p2p networks, in: Proc. of the 12th International World Wide Web Conference, May 2003.
[27] E. Kotsovinos, D. Mcilwraith, replic8: location-aware data replication for high availability in ubiquitous environments, in: Wired/Wireless Internet Communications, vol. 3510, Springer, Berlin/Heidelberg, 2005, pp. 32–41.
[28] Y. Li, Z. Lan, Exploit failure prediction for adaptive fault-tolerance in cluster computing, in: IEEE International Symposium on Cluster Computing and the Grid, 2006, pp. 531–538.
[29] Y. Liang, Y. Zhang, A. Sivasubramaniam, R.K. Sahoo, J. Moreira, M. Gupta, Filtering Failure Logs for a Bluegene/l Prototype, IEEE Computer Society, Los Alamitos, CA, USA, 2005, pp. 476–485.
[30] Z. Liang, W. Shi, Analysis of recommendations on trust inference in the open environment, Technical Report MIST-TR-2005-002, Department of Computer Science, Wayne State University, February 2005.
[31] Z. Liang, W. Shi, Enforcing cooperative resource sharing in untrusted peer-to-peer environment, ACM Journal of Mobile Networks and Applications (MONET) special issue on Non-cooperative Wireless Networking and Computing 10 (6) (December 2005) 771–783.
[32] Z. Liang, W. Shi, Analysis of ratings on trust inference in open environments, Elsevier Performance Evaluation 65 (2) (February 2008) 99–128.
[33] Z. Liang, W. Shi, A reputation-driven scheduler for autonomic and sustainable resource sharing in grid computing, Journal of Parallel and Distributed Computing 70 (2) (2010) 111–125.
[34] L. Liu, H. Wang, X. Liu, X. Jin, W.B. He, Q.B. Wang, Y. Chen, Greencloud: a new architecture for green data center, in: ICAC-INDST'09: Proceedings of the 6th International Conference Industry Session on Autonomic Computing and Communications Industry Session, ACM, New York, NY, USA, 2009, pp. 29–38.

Please cite this article in press as: T. Nguyen, W. Shi, Improving resource efficiency in data centers using reputation-based resource selection, Sustain. Comput.: Inform. Syst. (2012), doi:10.1016/j.suscom.2012.03.001

[35] M. Parashar, S. Hariri, Autonomic computing: an overview, in: Proceedings of the Intl Workshop on Unconventional Programming Paradigms (UPP'04), Springer Berlin-Heidelberg, 2005, pp. 257–269.
[36] R. Raman, M. Livny, M. Solomon, Matchmaking: distributed resource management for high throughput computing, in: International Symposium on High-Performance Distributed Computing, 1998, p. 140.
[37] X. Ren, S. Lee, R. Eigenmann, S. Bagchi, Prediction of resource availability in fine-grained cycle sharing systems empirical evaluation, Journal of Grid Computing 5 (2007) 173–195, doi:10.1007/s10723-007-9077-5.
[38] B. Schroeder, G.A. Gibson, Disk failures in the real world: what does an mttf of 1,000,000 hours mean to you? in: Proc. of the 5th USENIX Conf. on File and Storage Technologies, February 2007, pp. 1–9.
[39] J. Sonnek, A. Chandra, J. Weissman, Adaptive reputation-based scheduling on unreliable distributed infrastructures IEEE Transactions on Parallel and Distributed Systems 18 (11) (2007) 1551–1564.
[40] J. Sonnek, J. Weissman, A quantitative comparison of reputation systems in the grid, in: IEEE/ACM International Workshop on Grid Computing, 2005, pp. 242–249.
[41] H. Wei, S. Ganguly, R. Izmailov, Z. Haas, Interference-aware ieee 802.16 wimax mesh networks, in: Vehicular Technology Conference, VTC 2005-Spring, 2005 IEEE 61st, vol. 5, IEEE, 2005, pp. 3102–3106.
[42] M. Zaharia, D. Borthakur, J. Sarma, K. Elmeleegy, S. Shenker, I. Stoica, Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling, in: EuroSys 2010, EuroSys, 2010.
[43] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, I. Stoica, Improving mapreduce performance in heterogeneous environments, in: Proceedings of the 8th Symposium on Operating Systems Design and Implementation, USENIX Association, OSDI, San Diego, CA, 12/2008, 2008, pp. 29–42.
[44] C.W. Zhifeng Yu, W. Shi, Failure-aware workflow scheduling in cluster environments, Cluster Computing Journal 13 (2010) 421–434.

Tung Nguyen is currently a Ph.D. candidate at Wayne State University. He received his bachelor and master degrees in computer science and engineering from HoChiMinh City University of Technology, Vietnam in 2001 and 2006, respectively. His research interests include Green Computing, Cloud Computing, Data Intensive Computing, and application of Cloud Computing to life science. He has published several papers in workshops, conferences and journal in both Computer Science and Bioinformatics such as OSDI, NPC, SUSCOM, BMC, Frontiers Genetics, etc. He has also served as a peer reviewer for many conferences such as euro-par, CollaborateCom, etc. More information can be found on his homepage at http://www.cs.wayne.edu/tung/ or with Google.

Dr. Weisong Shi is an associate professor of computer science at Wayne State University. He received his B.S. from Xidian University in 1995, and Ph.D. degree from the Chinese Academy of Sciences in 2000, both in Computer Engineering. His current research focuses on computer systems, mobile and cloud computing. Dr. Shi has published more than 100 peer reviewed journal and conference papers. He is the author of the book "Performance Optimization of Software Distributed Shared Memory Systems" (High Education Press, 2004). He has served the program chairs and technical program committee members of several international conferences. He is a recipient of the NSF CAREER award, one of 100 outstanding Ph.D. dissertations (China) in 2002, Career Development Chair Award of Wayne State University in 2009, and the "Best Paper Award" of ICWE'04 and IPDPS'05.

Please cite this article in press as: T. Nguyen, W. Shi, Improving resource efficiency in data centers using reputation-based resource selection, Sustain. Comput.: Inform. Syst. (2012), doi:10.1016/j.suscom.2012.03.001