Multi-Resource Aware Fairsharing for Heterogeneous Systems

Dalibor Klusáček[1,2] and Hana Rudová[2]

[1] CESNET z.s.p.o., Zikova 4, Prague, Czech Republic
[2] Faculty of Informatics, Masaryk University
Botanická 68a, Brno, Czech Republic
{xklusac,hanka}@fi.muni.cz

Abstract. Current production resource management and scheduling systems often use some mechanism to guarantee fair sharing of computational resources among different users of the system. For example, the user who so far consumed small amount of CPU time gets higher priority and vice versa. However, different users may have highly heterogeneous demands concerning system resources, including CPUs, RAM, HDD storage capacity or, e.g., GPU cores. Therefore, it may not be fair to prioritize them only with respect to the consumed CPU time. Still, applied mechanisms often do not reflect other consumed resources or they use rather simplified and "ad hoc" solutions to approach these issues. We show that such solutions may be (highly) unfair and unsuitable for heterogeneous systems. We provide a survey of existing works that try to deal with this situation, analyzing and evaluating their characteristics. Next, we present new enhanced approach that supports multi-resource aware user prioritization mechanism. Importantly, this approach is capable of dealing with the heterogeneity of both jobs and resources. A working implementation of this new prioritization scheme is currently applied in the Czech National Grid Infrastructure MetaCentrum.

Keywords: Multi-Resource Fairness, Fairshare, Heterogeneity

1 Introduction

This paper is inspired by our cooperation with the Czech National Grid Infrastructure MetaCentrum [18]. MetaCentrum is highly heterogeneous national Grid that provides computational resources to various users and research groups. Naturally, it is crucial to guarantee that computational resources are shared in a fair fashion with respect to different users and research groups [14, 12]. Fairness is guaranteed by the *fairshare algorithm* [11, 2], which is implemented within the applied resource manager, in this case the TORQUE [3].

For many years the fairshare algorithm considered only single resource when establishing users priorities. The fairshare algorithm measured the amount of consumed CPU time for each user and then calculated users priorities such that the user with the smallest amount of consumed CPU time obtained the highest

priority and vice versa [14]. However, with the growing heterogeneity of jobs and resources, it quickly became apparent that this solution is very unfair since it does not reflect other consumed resources [17, 16].

An intuitive solution is to apply a more complex, multi-resource aware fairshare algorithm. However, as we will demonstrate in Section 2, existing solutions are not very suitable for truly heterogeneous workloads and systems. Often, these solutions either use unrealistic system models or fail to provide fair solutions in specific, yet frequent usage scenarios.

In this paper we present a new fair sharing prioritization scheme which we have proposed, implemented and put into daily service. It represents a rather unique multi-resource aware fairshare mechanism, which was designed for truly heterogeneous workloads and systems. Based on an extensive analysis of pros and cons of several related works (see Section 2) we have carefully extended widely used *Processor Equivalent (PE)* metric which is available in Maui and Moab schedulers [11, 1, 2]. The extension guarantees that the prioritization scheme is not sensitive to job and machine parameters, i.e., it remains fair even when the jobs and resources are (highly) heterogeneous. Importantly, the scheme is insensitive to scheduler decisions, i.e., the computation of priorities is not influenced by the job-to-machine mapping process of the applied job scheduler. Also, jobs running across different nodes are supported, and the solution reflects various speeds of machines and performs corresponding walltime normalization to capture the effects of slow vs. fast machines on resulting job walltime[1].

This paper is structured as follows. In Section 2 we discuss the pros and cons of existing works covering both classical CPU-based and multi-resource aware fairness techniques using several real life-based examples. In Section 3 we describe the newly proposed multi-resource aware fairness technique. Section 4 evaluates the proposed solution using historic MetaCentrum workload. We conclude the paper and discuss the future work in Section 5.

2 Related Work

Before we start, we would like to stress out that there is no widely accepted and universal definition concerning fairness. In fact, different people and/or organizations may have different notion of "what is fair" when it comes to multiple resources [13, 9]. In our previous work [17], we have shown how different reasonable fairness-related requirements may interact together, often resulting in conflicting situations. Therefore, in the following text we present approaches and viewpoints that were established and are currently applied in MetaCentrum.

2.1 Fairshare

All resource management systems and schedulers such as TORQUE [3], PBS-Pro [19], Moab, Maui [1], Quincy [10] or Hadoop Fair and Capacity Schedulers [6,

[1] Walltime is the time a job spends executing on a machine(s). It is an important parameter used in the fairshare algorithm as we explain in Section 2.

4] support some form of fairshare mechanism. Nice explanation of Maui fairshare mechanism can be found in [11]. For many years, the solution applied in MetaCentrum TORQUE was very similar to Maui or Moab. It used the well known *max-min* approach [9], giving the highest priority to a user with the smallest amount of consumed CPU time and vice versa.

For the purpose of this paper, we assume that a user priority is established by Formula 1 [11]. Here, F_u is the resulting priority of a given user u. F_u is computed over the set J_u, which contains all jobs of user u that shall be used to establish user priority. The final value is computed as a sum of products of job penalty $P(j)$ and the job walltime ($walltime_j$). As soon as priorities are computed for all users, the user with the smallest value of F_u gets the highest priority in a job queue.

$$F_u = \sum_{j \in J_u} walltime_j \cdot P(j) \tag{1}$$

Formula 1 is a general form of a function that can be used to establish ordering of users. It represents the simplest version, that does not use so called decay algorithm [11]. Decay algorithm is typically applied to determine the value of F_u with respect to aging, i.e., it specifies how the effective fairshare usage is decreased over the time[2]. For simplicity, we will not consider the decay algorithm in the formulas as its inclusion is straightforward and can be found in [11] or [17].

When computing F_u, a proper computation of the job penalty $P(j)$ is the key problem. Commonly, fairshare algorithms only consider a single resource, typically CPUs. In such a case, the penalty function $P(j)$ for a given job j is simply $P(j) = reqCPU_j$, where $reqCPU_j$ is the number of CPUs allocated to that job[3]. Clearly, the penalty of a given user's job j is proportional to the number of CPUs it requires. To illustrate the problems related to a CPU-based penalty we provide following real life-based Example 1 [17], which is based on a workload coming from Zewura cluster, a part of MetaCentrum.

Example 1. Zewura consists of 20 nodes, each having 80 CPUs and 512 GB of RAM. Fig. 1 (left) shows the heterogeneity of CPU and RAM requirements of jobs that were executed on this cluster. Clearly, there are many jobs that use a lot of RAM while using only a fraction of CPUs. Similarly, Fig. 1 (right) shows an example of CPUs and RAM usage on a selected node within the Zewura cluster. For nearly two weeks in July 2012, jobs were using at most 10% of CPUs while consuming all available RAM memory. Those remaining 90% of CPUs were then

[2] A *fairshare usage* represents the metric of utilization measurement [11]. Typically, it is the amount of consumed CPU time of a given user.

[3] In MetaCentrum, resources allocated (i.e., reserved) to a given job cannot be used by other jobs even if those resources are not fully used. Therefore, when speaking about CPU, RAM, etc., requirements we mean the amount of a given resource that has been allocated for a job, even if actual job requirements were smaller. Similarly, a job CPU time is the number of allocated CPUs multiplied by that job walltime.

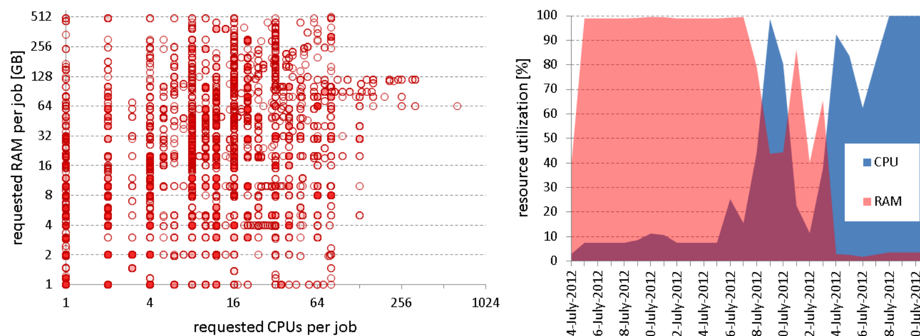

Fig. 1. Heterogeneity of jobs CPU and RAM requirements (left) and an example of CPU and RAM utilization on one Zewura node (right).

useless because no new job could have been executed there due to the lack of available RAM. More importantly, using the standard fairshare algorithm, the owner(s) of these memory-demanding jobs were only accounted for using 10% of available CPU time. However, as intuition suggests, they should have been accounted as if using 100% of machine CPU time because they effectively "disabled" whole machine by using all its RAM [17].

Apparently, the classical — single resource-based — fairshare mechanism computed according to consumed CPU time may be considered unfair as the users with high RAM requirements are not adequately penalized with respect to those users who only need (a lot of) CPUs. Of course, similar findings can be done concerning other resources such as GPUs or HDD storage. For simplicity, we only consider CPUs and RAM in the rest of the paper. The addition of additional resources is possible and it is a part of our future work (see Section 5).

Although the single resource-based fairshare algorithm may seem inadequate, many systems are still using it today [10, 5, 19]. Let us now discuss multi-resource aware solutions that are already available in several mainstream resource managers and schedulers.

2.2 Standard Job Metric

The latest documentation of PBS-Pro [19] suggests that an administrator must select exactly one resource to be tracked for fairshare purposes, therefore it is not possible to combine multiple consumed resources in fairshare. We have discussed this issue with people from PBS Works[4] and according to their advice, it is possible to use so called *standard job* metric. It works as follows. First, a system administrator defines resource requirements of so called "standard job". These

[4] PBS Works is a division of Altair which is responsible for PBS-Pro development. The meeting took place at the Supercomputing 2013 conference in Denver, CO, USA.

requirements should correspond to a typical small job, e.g., $standardCPU = 1$ CPU and $standardRAM = 1$ GB of RAM. Next, a job penalty of any actual job is computed by Formula 2, i.e., $P(j)$ is the number of standardized jobs that are needed to cover resource requirements of a considered job j.

$$P(j) = \max\left(\frac{reqCPU_j}{standardCPU}, \frac{reqRAM_j}{standardRAM}\right) \tag{2}$$

Although this metric is simple, we see an apparent problem — it is highly sensitive with respect to the used setup of "standard job". At the same time, our existing workloads indicate that there is no "standard job" as is also visible in Fig. 1 (left). Therefore, it is quite questionable to use this metric across all users and the whole system.

2.3 Processor Equivalent Metric

Moab and Maui provide different, yet still simple solution called *processor equivalent (PE)* [11, 2, 1], which allows to combine CPU and, e.g., RAM consumptions, translating multi-resource consumption requests into a scalar value. PE is based on the application of *max* function that determines the most constraining resource consumption of a job and translates it into an equivalent processor count using Formula 3, where $availCPU$ and $availRAM$ are the total amounts of CPUs and RAM in the system, respectively.

$$P(j) = PE(j) = \max\left(\frac{reqCPU_j}{availCPU}, \frac{reqRAM_j}{availRAM}\right) \cdot availCPU \tag{3}$$

Moab documentation illustrates the processor equivalent functionality using following Example 2.

Example 2. Consider a situation that a job requires 20% of all CPUs and 50% of the total memory of a 128-processor system. Only two such jobs could be supported by this system. The job is essentially using 50% of all available resources since the most constrained resource is memory in this case. The processor equivalents for this job should be 50% of the processors, or $PE = 64$ [2].

Although the documentation states that "the calculation works equally well on homogeneous or heterogeneous systems" [2], this is not true as problems may appear once the system and workload become heterogeneous. Let us demonstrate MetaCentrum-inspired Example 3 where PE fails to produce reasonable job penalties.

Example 3. Consider a heterogeneous system with 2 types of nodes. First type of nodes has 8 CPUs and 16 GB of RAM. Second type of nodes has 80 CPUs and 512 GB of RAM. The system contains 10 nodes of type 1 and 1 nodes of type 2. Together, the system has 160 CPUs ($availCPU$) and 672 GB of

RAM (*availRAM*). Now a user submits a RAM-constrained job requiring 1 CPU and 512 GB of RAM per node. This scenario emulates the situation discussed in Example 1 which is depicted in Fig. 1 (right). The resulting job processor equivalent (using Formula 3) is $PE(j) = \max(1/160, 512/672) \cdot 160 = 121.9$. Using the interpretations of PE as found in Moab documentation [2], we can say that memory is the most constrained resource for this job, thus $PE(j) = 121.9$ which means that approximately 76% of all available resources are used by this job. However, this is not entirely true. Since that job requires 512 GB of RAM per node, it can only be executed on that large (type 2) machine. At the same time, the job uses all RAM of that (type 2) machine. As a result, this job "occupies" all 80 CPUs of this machine. The question is, whether it is fair to "charge" the user as if using 121.9 CPUs, as suggests the PE-based penalty. We think that this is not fair.

Still, one may suggest that since the job is really using 76% of all available RAM, it should be penalized by $PE(j) = 121.9$ (an equivalent of 76% CPUs). As it turns out this interpretation is not correct, as we can easily construct following counter example.

Example 4. Let us consider a scenario with a CPU-constrained job requiring 80 CPUs per node and 80 GB of RAM. The resulting job processor equivalent is $PE(j) = \max(80/160, 80/672) \cdot 160 = 80$. It indicates that 50% of all available resources are used by this job. Since that job requires 80 CPUs per node, it can only be executed on that large (type 2) machine. At the same time, the job uses all CPUs of that (type 2) machine. Then, also all RAM on this machine (512 GB of RAM) must be considered as unavailable. Using the same argumentation as in case of Example 3, we must say that this job is occupying 76% of all RAM. However, in this case the $PE(j)$ is only 80 (an equivalent of 50% CPUs).

To sum up, Examples 3 and 4 show how two different jobs (RAM vs. CPU-constrained) that *occupy the same resources* (one type 2 node) may obtain highly different penalties. From our point of view, it means that the use of PE in heterogeneous environments does not solve fairly the problem observed in Fig. 1 (right) and described in Example 1.

2.4 Other Approaches

So far, we have discussed solutions that are available within several mainstream systems. However, there are also several works that propose novel multi-resource aware scheduling methods. We have provided a detailed survey of those methods in our previous work [17], so we only briefly recapitulate here. For example, *Dominant Resource Factor (DRF)* [9] suggests to perform max-min fairshare algorithm over so called dominant user's share, which is the maximum share that a user has been allocated of any resource. Recently, DRF has been included into the new *Fair Scheduler* in Hadoop Next Generation [6]. Simultaneous fair allocation of multiple continuously divisible resources called *bottleneck-based fairness (BBF)* is proposed in [8]. In BBF, an allocation of resources is considered fair

if all users either get all the resources they wished for, or else get at least their entitlement on some bottleneck resource, and therefore cannot complain about not receiving more. The tradeoffs of using multi-resource oriented fairness algorithms including newly proposed *Generalized Fairness on Jobs (GFJ)* are discussed in [13]. Especially, the overall utilization is of interest. Unlike DRF, GJF measures fairness only in terms of the number of jobs allocated to each user, disregarding the amount of requested resources [13]. From our point of view, such a notion of fairness is impractical as it allows to cheat easily by "packing" several small jobs as a one large job. Also, all these approaches make the assumption that all jobs and/or resources are continuously divisible [7]. However, for common grid and cluster environments, this is rarely the case, thus these techniques are rather impractical for our purposes. In our recent short abstract [16], we have presented a possible extension of the fairshare algorithm to cover heterogeneity of resources. However, the abstract provided neither detailed analysis of related work, neither a detailed explanation or an evaluation of the solution itself. Moreover, the proposed techniques did not support some important features, e.g., computation of penalties for multi-node jobs.

In the previous text we have illustrated several problems that complicate the design of a proper multi-resource aware job penalty function (a key part of the fairshare algorithm). Existing mainstream solutions often rely on too simplified and sensitive approaches ("standard job"-based metric) or they fail to provide reliable results for heterogeneous systems and workloads (PE metric). Other works such as DRF, GFJ or BBF then use system models that are not suitable for our purposes.

3 Proposed Multi-Resource Aware Fairshare Algorithm

In this section we describe the newly developed multi-resource aware fairshare mechanism which is currently used to prioritize users in MetaCentrum. It has several important features that we summarize in the following list:

1) **multi-resource awareness:** The solution reflects various consumed resources (CPU and RAM by default) using modified processor equivalent metric.
2) **heterogeneity awareness:** Processor equivalent metric is used in a new way, guaranteeing the same penalties for jobs that occupy the same resources. This modification solves the problems related to the heterogeneity of jobs and resources described in Section 2.3.
3) **insensitivity to scheduler decisions:** Job penalty is not sensitive to scheduler decisions, i.e., a given job penalty is not influenced by the results of the job-to-machine mapping process being performed by the job scheduler.
4) **walltime normalization:** We reflect various speeds of machines and perform so called *walltime normalization* to capture the effects of slow vs. fast machines on resulting job walltime.
5) **support for multi-node jobs:** The solution calculates proper job penalties for multi-node jobs that may have different per-node requirements.

In the following text we describe how these features are implemented, starting with the new penalty function that allows features 1-3, then proceeding to walltime normalization (feature 4). For simplicity, we first describe how the scheme works for single node jobs and then proceed to the description of multi-node job support (feature 5). Finally, we briefly describe how the solution has been implemented in TORQUE.

3.1 Proposed Penalty Function

The newly proposed penalty function is based on an extension of processor equivalent (PE) metric (see Formula 3) presented in Section 2.3. As we have shown in Examples 3 and 4, PE cannot solve the problems observed in Example 1. It may provide misleading and unfair results when measuring the usage of the system, by producing different penalties for jobs that occupy the same resources. The origin of the problem observed in Example 3 and 4 is that the system and jobs are heterogeneous, thus only a subset of nodes may be suitable to execute a job. Then, it is questionable to compute job penalty with respect to all available resources. A simple solution addressing this problem is to compute PE only with respect to a machine i that has been used to execute that particular job j as shows Formula 4. Instead of using global amounts of CPUs and RAM, here the $availCPU_i$ and the $availRAM_i$ are the amounts of CPUs and RAM on that machine i, respectively[5].

$$PE(j,i) = \max\left(\frac{reqCPU_j}{availCPU_i}, \frac{reqRAM_j}{availRAM_i}\right) \cdot availCPU_i \cdot node_cost_i \quad (4)$$

This reformulation solves the problem observed in Example 3 and 4 as those RAM and CPU-heavy jobs now obtain the same penalties ($PE(j,i) = 80$). Sadly, $PE(j,i)$ brings a new disadvantage. Now, the PE calculation is *sensitive to scheduler decisions* [16]. Consider following example.

Example 5. Let a job j requests 1 CPU and 16 GB of RAM. Clearly such job can be executed both on type 1 and type 2 nodes. However, when j is executed on a type 1 node, then $PE(j,1) = \max(1/8, 16/16) \cdot 8 = 8$ while if j is executed on a type 2 node, then $PE(j,2) = \max(1/80, 16/512) \cdot 80 = 2.5$.

Since a user has limited capabilities to influence scheduler behavior, such metric is highly unfair as it may assign highly variable penalties for identical jobs. Therefore, we use this metric in a different way. In the first step, we construct the set M_j which is the set of all machines that are suitable to execute job j. Then we compute the "local" processor equivalent $PE(j,i)$ for each machine i such that $i \in M_j$ using Formula 4. Finally, we compute the job penalty $P(j)$ using Formula 5.

[5] The additional parameter $node_cost_i$ is optional and can be used to express (real) cost and/or importance of machine i, e.g., GPU-equipped nodes are less common (i.e., more valuable) in MetaCentrum. By default, $node_cost_i = 1.0$.

$$P(j) = queue_cost_j \cdot \min_{i \in M_j} PE(j, i) \quad (5)$$

Job penalty $P(j)$ is based on the minimal $PE(j,i)$, i.e., it uses the cheapest "price" available in the system. It is important to notice, that it represents the best possible fit and $P(j)$ *remains the same disregarding the final job assignment*. Therefore, we can guarantee that $P(j)$ is *insensitive to scheduler decisions*. At the same time, we avoid the problems related to heterogeneity, since we only consider those machines (M_j) that are suitable for that job j. The use of this penalty has one major benefit — our users are satisfied as we always choose the best price for them, disregarding the final scheduler decision. Therefore, they are not tempted to fool the system by "playing" with job parameters or with job-to-machine mapping, which could otherwise degrade, e.g., the system throughput.

As can be seen in Formula 5, we also use *queue_cost* parameter. It can be used to further increase or decrease job penalty depending on the user's choice of queue. By default, all queues have the same cost (1.0). However, it is sometimes useful to increase the price for, e.g., those queues that are used for very long jobs or provide access to some specialized/expensive hardware. To sum up, the proposed penalty shown in Formula 5 is multi-resource aware, reflects heterogeneity of jobs and resources, and provides results that are not sensitive to scheduler decisions, i.e., it supports the features 1, 2 and 3 described at the beginning of Section 3.

3.2 Walltime Normalization

Job walltime is a very important parameter that is used along with the job penalty to establish the final user ordering (see Formula 1). However, in heterogeneous systems like MetaCentrum, the walltime of a job may depend on the speed of machine(s) where that job is executed. Fig. 2 illustrates this situation by showing the per-CPU-core results of the Standard Performance Evaluation Corporation's SPEC CPU2006 benchmark (CFP2006 suite/fp_rate_base2006) for major MetaCentrum clusters. In order to further illustrate the heterogeneity of resources, the figure also shows the total number of CPU cores on each cluster, as well as the number of CPU cores per node and the amount of RAM per node.

The figure demonstrates large differences in machine performance. Especially those "flexible" jobs that can be executed on many clusters can end up with highly variable walltimes. In addition, if a job ends up on a slow machine, its walltime will be higher, thus the fairshare usage of its owner will increase even more. Clearly, this scheduler-dependent job assignment results in a highly unfair behavior of the fairshare algorithm and explains the importance of walltime normalization. As far as we know, walltime normalization is not typically applied and it is not even mentioned in the documentation of PBS-Pro, TORQUE, Maui or Moab. Therefore, we have decided to apply simple walltime normalization, where the resulting walltime of a job j is multiplied by the SPEC result ($SPEC_j$) of the machine that was used to execute that job j. We assume that the resulting

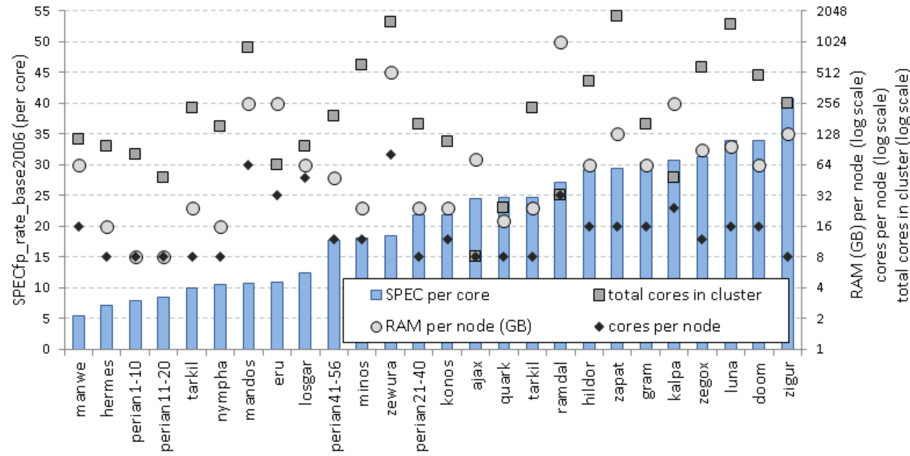

Fig. 2. Heterogeneity of SPEC CPU2006 results, CPU cores and RAM for major MetaCentrum clusters.

job walltime is inversely proportional to the $SPEC_j$. Since this is not true for some applications, our users can directly specify a desired speed of machine(s) upon each job submission, by providing the minimum and the maximum eligible SPEC. In that case, only those machines satisfying these constraints remain in the set of eligible machines M_j. Once the walltime normalization is applied, the resulting priority of a given user u is now computed using Formula 6.

$$\begin{aligned} F_u &= \sum_{j \in J_u} walltime_j \cdot SPEC_j \cdot P(j) \\ &= \sum_{j \in J_u} walltime_j \cdot SPEC_j \cdot queue_cost_j \cdot \min_{i \in M_j} PE(j,i) \end{aligned} \tag{6}$$

Beside features 1-3, this formula also supports feature 4. Still, it is only suitable for single node jobs. In practice, users may submit jobs that require several nodes to execute. Moreover, the specifications concerning each requested node may be different. Such a situation requires more complex function which we describe in the next section.

3.3 Multi-Node Jobs

The last feature 5 mentioned at the beginning of Section 3 enables us to correctly compute fairshare priority with respect to multi-node jobs, that may have heterogeneous per-node requirements. We assume that a given multi-node job j requests r nodes. For each such node, there is a separate resource specification[6].

[6] It is obtained by parsing node specification requests obtained by `qsub` command.

Resource requests concerning k-th node ($1 \leq k \leq r$) are denoted as $reqCPU_{k,j}$ and $reqRAM_{k,j}$. To compute the job penalty, following steps are performed for all requested nodes. At first, we find the set of machines that meet the k-th request and denote it as $M_{k,j}$. Then, for every suitable machine i such that $i \in M_{k,j}$ we compute the corresponding "local" processor equivalent, denoted as $PE_k(j,i)$ (see Formula 7). Then, the "price" for the k-th request is the minimal (cheapest) $PE_k(j,i)$. The resulting job penalty $P(j)$ is the sum of minimal prices, multiplied by the *queue_cost* as shown in Formula 8. In the next step, we normalize the walltime of the job. Since the job uses r different machines, we have r (possibly different) SPEC values, where the k-th value is denoted as $SPEC_{j,k}$. As the walltime is the time when the whole job completes, it is most likely influenced by the slowest machine being used, i.e, the machine with lowest SPEC result. Therefore, the walltime is normalized by the minimal $SPEC_{j,k}$. Together, the fairshare priority F_u is computed as shows Formula 9.

$$PE_k(j,i) = \max\left(\frac{reqCPU_{k,j}}{availCPU_i}, \frac{reqRAM_{k,j}}{availRAM_i}\right) \cdot availCPU_i \cdot node_cost_i \quad (7)$$

$$P(j) = queue_cost_j \cdot \sum_{k=1}^{r} \min_{i \in M_{k,j}} PE_k(j,i) \quad (8)$$

$$F_u = \sum_{j \in J_u} walltime_j \cdot \min_{1 \leq k \leq r} (SPEC_{j,k}) \cdot P(j) \quad (9)$$

3.4 Implementation in TORQUE

To conclude this section, we just briefly describe computation of the proposed multi-resource aware fairshare priority function (see Formula 9) within the TORQUE deployed in MetaCentrum[7].

The computation is done in two major steps. In the first step, a job penalty $P(j)$ is computed upon each job arrival, i.e., prior to a job execution. This computation is performed by the scheduler. $P(j)$ is refreshed during each scheduling cycle until a job starts its execution. As soon as a job starts, the TORQUE server obtains information about machine(s) being used by that job, especially those corresponding value(s) of SPEC and node cost(s). At this point, the server has all information required to recompute a fairshare priority F_u of a corresponding user. If needed, job queues are then reordered according to a newly computed fairshare priority.

It is important to notice, that a fairshare priority F_u of a user is updated immediately after his or her job starts its execution. Otherwise, a priority of that user would remain the same until at least one of his or her jobs completes, which is potentially dangerous. Since an exact walltime of a running job is not known until that job completes, the maximum walltime limit is used instead as an approximation. As soon as that job completes, its actual walltime is used

[7] This enhanced TORQUE can be obtained at: `https://github.com/CESNET/torque`.

accordingly and a fairshare priority F_u of a corresponding user is recomputed from scratch (replacing previous approximation). If needed, job queues are then reordered accordingly.

4 Experimental Analysis

In this section we describe how the new multi-resource aware fairshare works on a real workload. For the purpose of evaluation we have used workload from MetaCentrum which covers first six months of the year 2013. This log contains 726,401 jobs, and is available at: `http://www.fi.muni.cz/~xklusac/jsspp/`. We have used *Alea* [15] job scheduling simulator to demonstrate the effects of our new prioritization mechanism. Alea is commonly used in MetaCentrum to evaluate suitability of newly developed solutions. Using the simulator, we have emulated both previous (CPU-based) as well as the new multi-resource aware fairshare mechanism and then analyzed their differences. The proposed prioritization scheme consists of two main parts — the new penalty function $P(j)$ and the walltime normalization. Therefore, we have performed two major experiments that cover these main parts of the proposed solution.

First, we have analyzed which jobs are affected by the new penalty function (see Formula 8). We have plotted all jobs from the workload according to their heterogeneous CPU and RAM requirements (see Fig. 3), and we have highlighted those jobs that have different (higher) value of the new penalty function compared to the old (CPU-based) version.

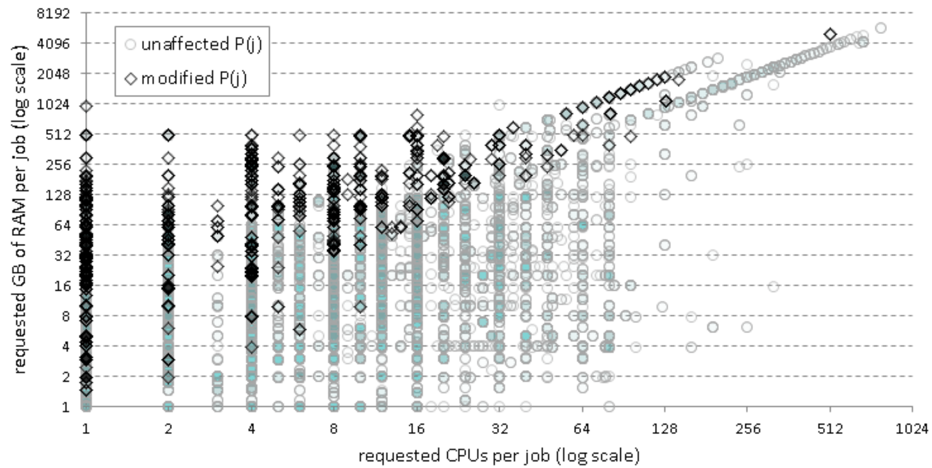

Fig. 3. The heterogeneity of CPU and RAM requirements of jobs from the workload. Dark boxes highlight those jobs that are affected by the new penalty function.

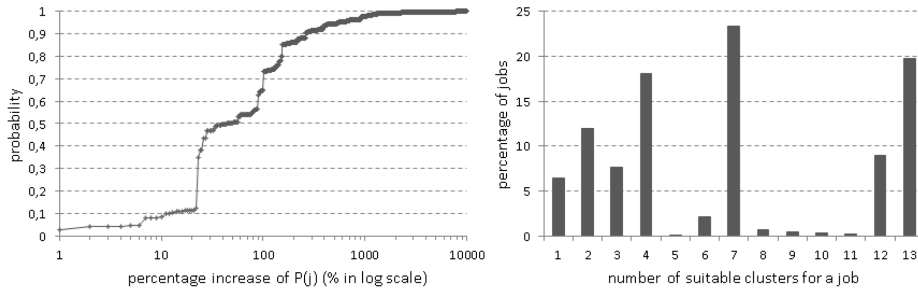

Fig. 4. CDF of resulting changes in $P(j)$ value (left) and the histogram of job-to-cluster suitability (right).

The results correspond to our expectations, i.e., the new $P(j)$ assigns higher penalties to those jobs with high RAM to CPU ratio. Such jobs represent less than 2% of all jobs, and generate more than 5% of the overall CPU utilization. Also, approximately 40% of users now have at least one job that would obtain higher penalty, compared to the original solution.

Next, we took those jobs with higher penalty (those affected by the new $P(j)$) and measured the percentage increase of the new $P(j)$ (with respect to the old, CPU-based version). Fig. 4 (left) shows the results using a cumulative distribution function (CDF). In this case, the CDF is a $f(x)$-like function showing the probability that the percentage increase of $P(j)$ for given job j is less than or equal to x. In another words, the CDF represents the fraction of jobs having their $P(j)$ less than or equal to x. As can be seen, the improvement is mostly significant. For example, for nearly 90% of considered jobs their new $P(j)$ has increased at least by 20%. Also, 40% of considered jobs have their $P(j)$ at least two times higher ($\geq 100\%$).

In the next step, we have measured the influence of the new $P(j)$ on the overall performance of the system. For this purpose we have measured the distribution of job wait times and bounded slowdowns when the original and the new $P(j)$ has been used, respectively. Jobs were scheduled from a single queue that was dynamically reordered according to continuously updated job priorities. Only the job at the head of the queue was eligible to run, i.e., we intentionally did not use backfilling. The reason is that backfilling can dilute the impact of the job prioritization algorithm [11], and therefore make it much harder to analyze the effect of new prioritization scheme. For similar reasons, walltime normalization has not been used in this experiment. Our results in Fig. 5 show that there is no danger when using the new priority function. In fact, the cumulative distribution functions (CDF) of wait times and slowdowns were slightly better for the new $P(j)$.

While the wait times and slowdowns were generally lower for the new prioritization scheme (see Fig. 5), this was not true for those jobs that — according to the new $P(j)$ — now obtain higher penalties. This is an expected and desir-

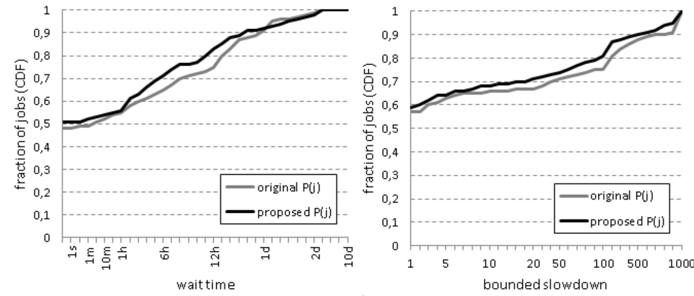

Fig. 5. Comparison of wait times (left) and bounded slowdowns (right) distributions.

able behavior. For example, in this experiment the wait times of such jobs have increased by 33 minutes on average.

In the final experiment, we have measured the possible influence of walltime normalization. As was demonstrated in Fig. 2, there are significant differences in the performance of clusters in MetaCentrum. At first, we have analyzed a job-to-cluster suitability by measuring how many clusters can be used to execute a given job. A cluster is capable to execute a job if it satisfies all job requirements as specified upon job submission. Typically, a job requires a set of CPUs, a fixed amount of RAM (per node), and data storage capacity. Moreover, it may require additional properties such as geographical locality of cluster(s), operating system, CPU architecture, etc[8]. Fig. 4 (right) shows the histogram of job-to-cluster suitability. The x-axis shows the number of suitable clusters and y-axis shows the percentage of jobs that can run on this number of clusters. There are 26 main clusters in MetaCentrum, but there are no jobs that can be executed on every cluster. Therefore, the x-axis is bounded by 13, which is the maximum number of clusters that some jobs can use (approximately 20% of jobs). Most jobs in the workload (93.6%) can execute on at least 2 clusters and more than 50% of jobs can use at least 7 clusters.

Since we have observed that many jobs are rather flexible, we have decided to measure the possible effect of walltime normalization on a job. For each job we have found the set of suitable clusters. Next, we have found the cluster(s) with the minimum and the maximum SPEC (denoted as $SPEC_{j,min}$ and $SPEC_{j,max}$), and $SPEC_j$ of the original cluster that has been used to execute that job (this information is available in the original workload log). Fig. 6 (left) shows the CDFs of $SPEC_{j,min}$, $SPEC_{j,max}$ and $SPEC_j$, respectively. It clearly demonstrates how large can be the differences among the original, the "slowest" and the "fastest" suitable cluster, i.e., how important is to perform some form of walltime normalization. Without doing so, we can significantly handicap those jobs (and users), that were assigned to slow machines. Beside poorer performance,

[8] Detailed description is available at: `https://wiki.metacentrum.cz/wiki/Running_jobs_in_scheduler`.

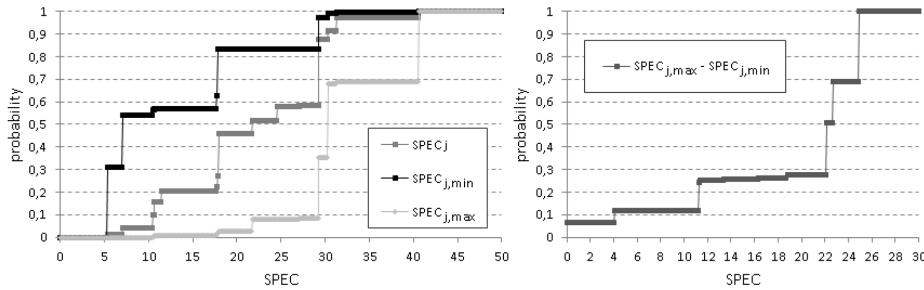

Fig. 6. CDFs of the minimal ($SPEC_{j,min}$), the maximal ($SPEC_{j,max}$) and the actual SPEC values ($SPEC_j$) as observed in the workload (left). The CDF showing the maximum possible differences in SPEC values (right).

slow machines also imply higher walltimes, thus further increasing the fairshare usage of corresponding job owners.

To further highlight this issue, we have computed the absolute difference between the "slowest" and the "fastest" suitable cluster ($SPEC_{j,max} - SPEC_{j,min}$) for every job j. The results are shown in the CDF in Fig. 6 (right). We can clearly see how large the differences are. For example, the maximal possible difference in SPEC values is greater than 11 for 89% of jobs, while the maximal possible difference is greater than 22 for 50% of jobs. Again, this example demonstrates how important is to perform some form of walltime normalization. Otherwise, the resulting priority ordering of users is likely to be (very) unfair.

5 Conclusion and Future Work

This paper addresses an urgent real life job scheduling problem, focusing on fair sharing of various resources among different users of the system. The novelty of our work is related to the fact that we consider *multiple consumed resources in heterogeneous systems* when establishing users priorities. We have discussed the pros and cons of several existing approaches, using real life-based examples. Next, we have provided the description and the analysis of the multi-resource aware fairshare technique which is currently used in the Czech National Grid Infrastructure MetaCentrum. The main features of this solutions are the ability to reflect both CPU and RAM requirements of jobs, the ability to handle heterogeneity of jobs and resources, the insensitivity to scheduler decisions, walltime normalization, and the support of multi-node jobs.

We plan to further analyze the performance and suitability of the production solution as well as possible problems that may appear in the future. The proposed solution is used for about 6 months (from November 2013) in the production system with 26 clusters and no significant comments from the users were recorded so far. It is important that the solution is capable of working in the

real production environment, even tough we still need to extend the set of implemented features (e.g., GPU-awareness, peer-to-peer fairshare synchronization) to become fully functional in larger scale. Therefore, our further development will focus on more complex usage scenarios that are based on MetaCentrum needs. For example, it is quite obvious that the PE-based metric may be too severe for jobs requiring special resources that are not needed by all jobs, e.g., GPUs. If a given job consumes all GPUs on a machine, it does not mean that such a machine cannot execute other jobs. Therefore, we will try to find some suitable relaxation of this metric for such special situations. Another example of a "problematic" resource is a storage capacity (e.g., local HDD/SSD or (hierarchical) data storages). Here the problem is that consumed capacity is rarely constrained by a job lifetime and therefore cannot be simply incorporated into the PE-based metric.

Also, we want to develop a new variant of fairshare, where selected groups of users can have their "local" and "global" priorities, that would be used depending on whether their jobs are executed on their own infrastructure or within the public pool of resources, respectively. Finally, as MetaCentrum is planning to use several TORQUE servers simultaneously using a peer-to-peer model, we will need to synchronize computations of fairshare priorities among several servers.

Acknowledgments. We highly appreciate the support of the Grant Agency of the Czech Republic under the grant No. P202/12/0306. The support provided under the programme "Projects of Large Infrastructure for Research, Development, and Innovations" LM2010005 funded by the Ministry of Education, Youth, and Sports of the Czech Republic is highly appreciated. The access to the MetaCentrum computing facilities and workloads is kindly acknowledged.

References

1. Adaptive Computing Enterprises, Inc. *Maui Scheduler Administrator's Guide, version 3.2*, January 2014. http://docs.adaptivecomputing.com.
2. Adaptive Computing Enterprises, Inc. *Moab workload manager administrator's guide, version 7.2.6*, January 2014. http://docs.adaptivecomputing.com.
3. Adaptive Computing Enterprises, Inc. *TORQUE Admininstrator Guide, version 4.2.6*, January 2014. http://docs.adaptivecomputing.com.
4. Apache.org. *Hadoop Capacity Scheduler*, January 2014. http://hadoop.apache.org/docs/r1.2.1/capacity_scheduler.html.
5. Apache.org. *Hadoop Fair Scheduler*, January 2014. http://hadoop.apache.org/docs/r1.2.1/fair_scheduler.html.
6. Apache.org. *Hadoop Next Generation Fair Scheduler*, January 2014. http://hadoop.apache.org/docs/r2.2.0/hadoop-yarn/hadoop-yarn-site/FairScheduler.html.
7. Jacek Blazewicz, Maciej Drozdowski, and Mariusz Markiewicz. Divisible task scheduling - concept and verification. *Parallel Computing*, 25(1):87–98, 1999.
8. Danny Dolev, Dror G. Feitelson, Joseph Y. Halpern, Raz Kupferman, and Nathan Linial. No justified complaints: on fair sharing of multiple resources. In *Proceedings*

of the 3rd Innovations in Theoretical Computer Science Conference, ITCS '12, pages 68–75, New York, NY, USA, 2012. ACM.
9. A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *8th USENIX Symposium on Networked Systems Design and Implementation*, 2011.
10. M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 261–276, 2009.
11. David Jackson, Quinn Snell, and Mark Clement. Core algorithms of the Maui scheduler. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 2221 of *LNCS*, pages 87–102. Springer Verlag, 2001.
12. Raj Jain, Dah-Ming Chiu, and William Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. Technical Report TR-301, Digital Equipment Corporation, 1984.
13. C. Joe-Wong, S. Sen, T. Lan, and M. Chiang. Multi-resource allocation: Fairness-efficiency tradeoffs in a unifying framework. In *31st Annual International Conference on Computer Communications (IEEE INFOCOM)*, pages 1206 – 1214, 2012.
14. Stephen D. Kleban and Scott H. Clearwater. Fair share on high performance computing systems: What does fair really mean? In *Third IEEE International Symposium on Cluster Computing and the Grid (CCGrid'03)*, pages 146 – 153. IEEE Computer Society, 2003.
15. Dalibor Klusáček and Hana Rudová. Alea 2 – job scheduling simulator. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques (SIMUTools 2010)*. ICST, 2010.
16. Dalibor Klusáček and Hana Rudová. New multi-resource fairshare prioritization mechanisms for heterogeneous computing platforms. In *Cracow Grid Workshop*, pages 89–90. ACC Cyfronet AGH, 2013.
17. Dalibor Klusáček, Hana Rudová, and Michal Jaroš. Multi resource fairness: Problems and challenges. In *Job Scheduling Strategies for Parallel Processing*, LNCS. Springer, 2014. To appear.
18. MetaCentrum, January 2014. `http://www.metacentrum.cz/`.
19. PBS Works. *PBS Professional 12.1, Administrator's Guide*, January 2014. `http://www.pbsworks.com`.