Multi Resource Fairness: Problems and Challenges

Dalibor Klusáček[1,2], Hana Rudová[1], and Michal Jaroš[3]

[1] Faculty of Informatics, Masaryk University
Botanická 68a, Brno, Czech Republic
[2] CESNET z.s.p.o., Zikova 4, Prague, Czech Republic
[3] Institute of Computer Science, Masaryk University
Botanická 68a, Brno, Czech Republic
{xklusac,hanka}@fi.muni.cz,mjaros@ics.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. The problem with such a solution is that it does not reflect other consumed resources like RAM, HDD storage capacity or GPU cores. Clearly, different users may have highly heterogeneous demands concerning aforementioned resources, yet they are all prioritized only with respect to consumed CPU time. In this paper we show that such a single resource-based approach is unfair and is no longer suitable for nowadays systems. We provide a survey of existing works that somehow try to deal with this situation and we closely analyze and evaluate their characteristics. Next, we propose new enhanced approaches that would allow the development of usable multi resource-aware user prioritization mechanisms. We demonstrate that different consumed resources can be weighted and combined together within a single formula which can be used to establish users' priorities. Moreover, we show that when it comes to multiple resources, it is not always possible to find a suitable solution that would fulfill all fairness-related requirements.

Keywords: Multi Resource Fairness, Fairshare, Penalty, Scheduling

1 Introduction

This paper is inspired by the lessons learned over the few past years when analyzing the workload of the Czech National Grid Infrastructure MetaCentrum [16]. MetaCentrum is highly heterogeneous national Grid that provides computational resources to various users and research groups. As in other systems, one of the main goal is to guarantee that computational resources are shared in a fair fashion with respect to different users and research groups [14, 11]. These requirements are typically solved using the service[4] of the applied resource manager, in

[4] This service is commonly called a *fairshare algorithm* [10, 2].

this case the TORQUE [3]. Current fairshare algorithm measures the amount of consumed CPU time for each user and then calculates users' priorities such that the user with the smallest amount of consumed CPU time gets the highest priority and vice versa [14]. While jobs typically consume several different resources (e.g., CPU time, RAM, GPUs and HDD storage) simultaneously, the whole user prioritization scheme is based only on one parameter — consumed CPU time. Clearly, it is questionable whether such a solution can guarantee fair sharing of resources [15]. Therefore, we have performed several analysis of existing workload and quickly realized that this single resource-based fairshare algorithm is (very) unfair.

To demonstrate some of the issues found in the workload we present Fig. 1 that shows the usage of CPUs and RAM on a selected node within the Zewura cluster in MetaCentrum. This particular node has 80 CPUs and 512 GB of RAM. The figure shows that for nearly two weeks in July 2012 the jobs used at most 10% of CPUs while consuming all available RAM memory. Clearly, the remaining 90% of CPUs are then useless because no new job can be executed there due to the lack of available RAM. More importantly, using the standard fairshare algorithm owner(s) of these memory-demanding jobs are only accounted for using 10% of available CPU time. However, as intuition suggests they should be accounted as if using 100% of machine's CPU time because they effectively "disabled" the whole machine by using all of its RAM.

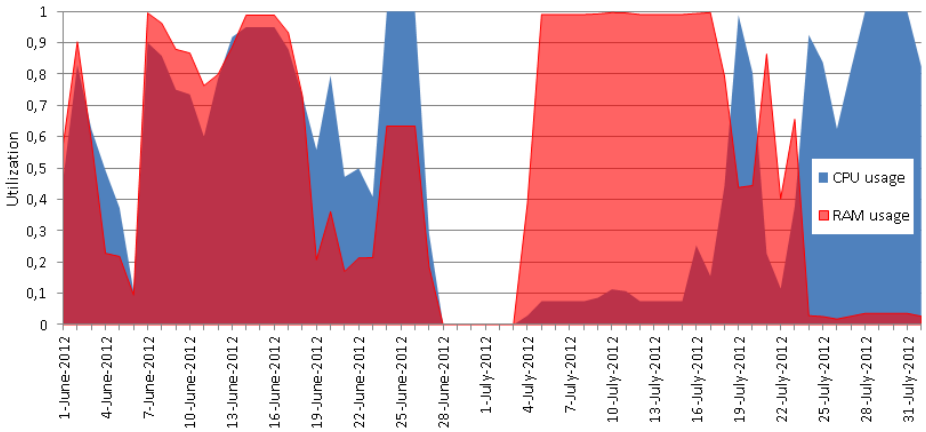

Fig. 1. An example of CPU and RAM utilization on one Zewura node.

The solution is to extend the current single resource-based fairshare algorithm and incorporate consumption of other important job-related resources, e.g., RAM, GPUs or HDD storage. For this purpose we have studied existing works that deal with similar problems and we present their survey here. We also propose new solutions that can flexibly combine several different resources with

different weights (i.e., costs) as existing works have some limitations when using several (weighted) resources together. We also define several rules that should be satisfied by considered multi resource-based fairshare formulas in order to generate fair and acceptable solutions. Based on these requirements, we analyze the suitability of considered techniques. Especially, we demonstrate weighting of different consumed resources and their combination within a single formula that is then used in the fairshare algorithm to establish priorities among users of the system. Surprisingly, we realize that — in general — it is not always possible to find a suitable solution that would fulfill all fairness-related requirements.

The structure of this paper is following. In Section 2 we discuss existing related works on single and multi resource-based fairness techniques. Especially, we closely describe current single resource-based fairshare algorithm as applied in MetaCentrum's TORQUE. In Section 3 we define several rules that should be satisfied by a prospective multi resource-based fairshare formula. Next, we present and discuss possible extensions of the fairshare algorithm that incorporate multiple resources. We also discuss whether these extensions are suitable when different resources have different weights, i.e., "cost" and/or importance. Section 4 discusses the findings of our work and suggests suitable solutions that can be applied within a multi resource-based fairshare algorithm. In Section 5 we conclude the paper and discuss the future work.

2 Related Work

All popular resource management systems and schedulers such as PBS [13], TORQUE [3], Moab, Maui [1], Quincy [9] or Hadoop's Fair and Capacity Schedulers [5, 4] support some form of fairshare mechanism. Nice explanation of Maui's fairshare mechanism can be found in [10].

The solution currently applied in MetaCentrum's TORQUE is very similar to Maui and uses the well known *max-min* approach [8], i.e., it gives 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's priority is established using a function that looks like Formula 1 [10, 15].

$$F_u = \sum_{j=1}^{n} (P_j \cdot walltime_j) \tag{1}$$

Here, the F_u is the resulting priority of a given user u that so far computed n jobs. The final value is computed as a sum of products of job penalty (P_j) and the job's walltime ($walltime_j$). Once the priorities are computed for all users, the user with the smallest value of F_u then gets the highest priority in a job queue. Such a formula 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 a so called decay algorithm [10]. 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[5]. For example, Maui's fairshare algorithm utilizes the concept of fairshare windows each covering a particular period of time. An administrator may then specify how long each window should last, how fairshare usage in each window should be weighted, and how many windows should be evaluated in obtaining the final effective fairshare usage [10]. For example, an administrator may wish to make fairshare adjustments based on the usage of resources during the previous 8 days. To do this, he or she may choose to evaluate 8 fairshare windows each consisting of 24 hour periods, with a decay, i.e., aging factor of 0.75 as seen in Fig. 2. For simplicity, we will not consider the decay algorithm in the formulas as its inclusion is straightforward.

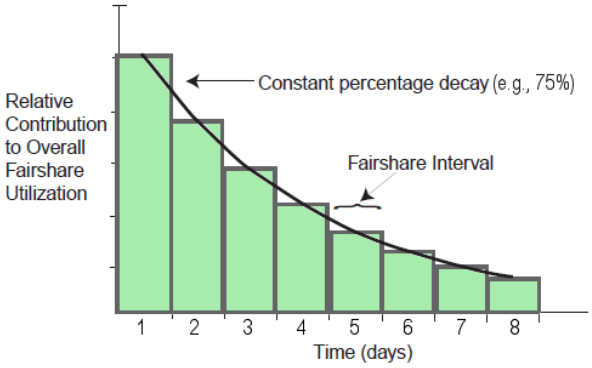

Fig. 2. Effective fairshare usage based on the decay algorithm that reflects aging. This image is adopted from [10].

When computing F_u, a proper computation of the job's penalty P_j is the key problem. In the rest of the paper we assume that the value of P_j is a real number from the interval $[0, 1]$, and we discuss several variants of P_j computation. Commonly, fairshare algorithms only consider a single resource, typically a CPU time. In such a case, the penalty function P_j for a given job j can be described by Formula 2, where $req_{CPU,j}$ is the number of CPUs allocated to a given job j and $avail_{CPU}$ is the total amount of CPUs available in the system.

$$P_j = \frac{req_{CPU,j}}{avail_{CPU}} \tag{2}$$

Clearly, the penalty of a given user's job j is proportional to the number of CPUs it requires as P_j expresses the ratio of consumed to available CPUs, i.e.,

[5] In Maui's terminology, *fairshare usage* represents the metric of utilization measurement [10]. Typically, fairshare usage expresses the amount of consumed CPU time of a given user.

the relative CPU usage[6]. The resulting distribution of such penalties is linear, and the highest penalty (1.0) is obtained when a user's job consumes all available CPUs in the system.

As we already mentioned in Section 1, the analysis of existing MetaCentrum's workloads has quickly identified that such an approach is clearly unfair. There were jobs that required few CPUs and (almost) all RAM memory (see Fig. 1). Therefore, those remaining CPUs could not be utilized by remaining users since there was no free RAM left. The classical — single resource-based — fairshare mechanism computed according to consumed CPU time is then absolutely unacceptable as the users with high RAM requirements are not adequately penalized in comparison with 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.

Although the single resource-based fairshare algorithm is inadequate, many systems are still using it today [9, 5, 12, 8]. Surprisingly, the so called multi resource fairness seems to be a rather new area of researchers' interest as there are only few works that address this problem specifically [7, 12, 8, 15]. For example, the recent *Dominant Resource Factor (DRF)* [8] suggests to perform max-min fairshare algorithm over so called dominant user's share. Dominant share is the maximum share that a user has been allocated of any resource. Such a resource is then called a *dominant resource*. Sadly, some parts of the paper are not very clear. For example, the pseudo-code of DRF algorithm does not correspond with the algorithm's textual description. Moreover, the resulting DRF allocation is formulated using a linear programming notation. However, the paper does not explain how non-integer results should be handled. As discussed in [12] which builds upon the results of [8], if a given user is allowed to execute, e.g., 0.76 jobs we cannot use such a solution unless user's jobs are continuously divisible [6]. For common grid and cluster environment, this is rarely the case. Similar situation applies for [7], which proposes new definition for the simultaneous fair allocation of multiple continuously divisible resources called *bottleneck-based fairness (BBF)*. In BBF, an allocation of resources is considered fair if every user either gets all the resources she wishes for, or else gets at least her entitlement on some bottleneck resource, and therefore cannot complain about not receiving more. Beside that, the tradeoffs of using multi resource-based fairness algorithms like DRF are discussed in [12]. Especially, the overall efficiency is of interest, e.g., the amount of unused resources is studied. Apart from DRF, the paper proposes the use of other approaches such as so called *Generalized Fairness on Jobs (GFJ)*. Unlike DRF, GJF measures fairness only in terms of the number of jobs allocated to each user. Users requiring more resources are thus treated equally [12]. 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. Last but not least, all approaches proposed in [12] or in [7] make the assumption that all jobs

[6] 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, in the whole paper we measure CPU, RAM, etc., requirements as the amount of a given resource that has been allocated for a job, even if actual job's requirements are smaller.

and resources are continuously divisible which is rather unrealistic for our purposes. In our previous work [15], we have proposed multi resource-based penalty function that uses a product of relative resources' requirements. In Section 3.2 we show that this function is less suitable than other approaches. Also, Moab or Maui schedulers allow the system administrator to combine CPU and, e.g., RAM consumptions within the fairshare function [10, 2] using so called *processor equivalent (PE)* mechanism [10]. It is based on the application of *max* function that determines a job's most constraining resource consumption and translates it into an equivalent processor count [10]. In fact, this solution uses similar idea as the DRF. Although PE mechanism is available in several production schedulers, we did not find any work that would specifically discuss its suitability. Also Moab's and Maui's documentation did not bring much insight into this solution [2, 1].

In the following section, we define several major principles that should be followed by a multi resource-based fairshare algorithm and we closely analyze selected promising multi resource-based fairshare metrics that are either based on existing works or are our own contribution.

3 Multi Resource-based Fairshare Algorithm

As discussed in previous section, the core part of the fairshare algorithm is the *penalty function*. Therefore, using the results from the literature, we now present and analyze several variants of multi resource-based job penalty functions that — beside the common CPU consumption — also consider additional consumed resources. Before we start, we first formulate several basic rules that are to be followed by an ideal multi resource-based penalty function. These rules are a result of several discussions that were held within the MetaCentrum team and reflect the specific requirements of MetaCentrum. We believe that these rules are general enough, still we are aware that for different institutions they may be either too restrictive or incomplete.

1) multiple resources: When calculating the value of penalty, the function should not consider only one type of consumed resource, e.g., CPUs.

2) nondominant resources: Penalty function should consider the consumption of nondominant resources as well. In another words, if two different jobs have the same consumption of a given dominant resource then the one having smaller consumption of nondominant resources should receive smaller penalty.

3) max-min penalty: Maximum penalty (i.e., 1) should be applied whenever a job completely utilizes at least one resource since the corresponding machine is then practically unusable for other jobs. Similarly, a job obtains minimum penalty (i.e., 0) only when it does not consume any resource at all[7].

[7] Max-min penalty rule defines when P_j reaches its minimum and maximum. Apparently, no "real" job should ever receive minimum penalty since it always consumes some resources.

4) linearity: Penalty function should be linear with respect to a given consumed resource. The linearity is important factor that guarantees that a user cannot cheat by dividing his or her (large) job into several smaller jobs that would — due to the nonlinear character of the penalty — together receive smaller penalty than the original (large) job.

5) weights: For a given resource, penalty function should allow to use weights that express the importance or the "cost" of that resource.

In the following text, we consider general formulas that allow inclusion of r different resources. The x-th resource is denoted as x where $x \in (1, .., r)$. For better readability, all figures that illustrate these formulas will however only contain the two most important resources — CPUs and RAM.

We start with an illustration of the penalties that are obtained when using classical CPU-based single resource penalty that has been shown in Formula 2. The resulting distribution of such penalties can be illustrated by the graph shown in Fig. 3 (left). Clearly, the penalty of a given user's job j has no relation to RAM consumption and is only proportional to the number of required CPUs as P_j expresses the ratio of consumed to available CPUs. This function is therefore impractical as it breaks all rules except for the "linearity" rule 4.

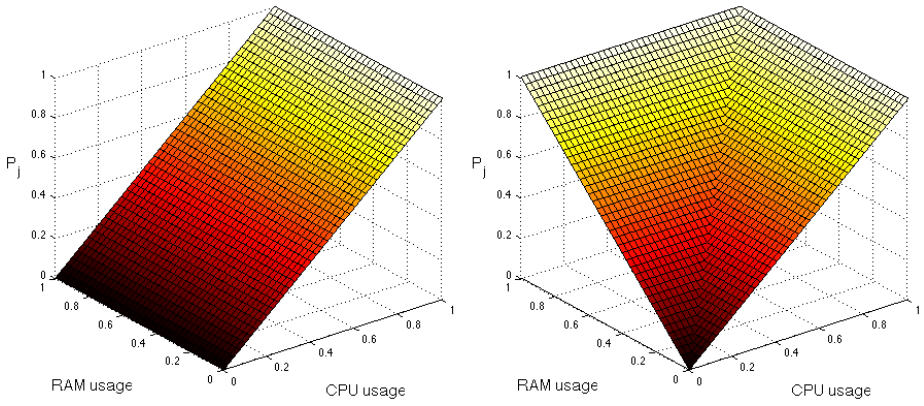

Fig. 3. Single resource CPU-based penalty (left) and *max*-based penalty function (right).

In order to resolve the unfairness of the single resource-based fairshare metric we analyze/propose several candidate penalty formulas that somehow incorporate additional resource requirements.

3.1 Dominant Resource-based Penalty

Existing works [8, 10, 1, 2] suggest to measure and apply *dominant resource*-based penalty. It means that a user is penalized according to the maximum relative

share he or she has been allocated of any resource [8]. In another words, instead of combining all resource requests together, only the maximum (most restricting) relative request is considered and penalized accordingly. The penalty is then computed using Formula 3 and the corresponding distribution of penalties is depicted in Fig. 3 (right).

$$P_j = \max\left(\frac{req_{1,j}}{avail_1}, .., \frac{req_{r,j}}{avail_r}\right) \tag{3}$$

Compared to the single resource-based penalty, this penalty function represents several benefits. First of all, it is very simple function so both users and system administrators will find it easy to understand. Second, it solves the problem described in Section 1, i.e., it adequately penalizes highly asymmetric requests, following the rule 3. Last but not least, unlike some of the functions that will be discussed in next section, this penalty is linear, fulfilling the rule 4.

Sadly, this penalty also represents several drawbacks. Although it does follow the rule 1, it does not fulfill the rule 2, i.e., it does not consider the nondominant resources at all. Therefore, users are not forced to better estimate their requests concerning nondominant resources. As a side effect, this penalty is not fair. Consider two users with equal dominant resource demands but with different nondominant resource requirements. Clearly, the one having smaller demands should be less penalized as he or she consumes less resources. However, they will both receive the same penalty, disregarding their real resource consumptions, which breaks the rule 2. We believe that this is an unfair behavior. The second problem is that we cannot apply resource weights in a reasonable manner. In reality, different resources are rarely considered as equally important. In fact, some resources are more important than others. For example, in MetaCentrum, the common sense is that CPUs are more "expensive" than, e.g., RAM. When necessary, it is often possible to increase the amount of RAM on a given machine while it is not possible to increase the number of CPUs. Therefore, the requirement is to apply resource-specific weights when computing the penalty function. As we show now, in case of Formula 3 this process is somehow tricky. There are two basic extensions of Formula 3 that involve weights and we show them in Formula 4 and 5. Both of them guarantee that the values of P_j will remain within the interval $[0, 1]$.

$$P_j = \min\left(1, \max\left(w_1\frac{req_{1,j}}{avail_1}, .., w_r\frac{req_{r,j}}{avail_r}\right)\right) \tag{4}$$

$$P_j = \frac{\max\left(w_1\frac{req_{1,j}}{avail_1}, .., w_r\frac{req_{r,j}}{avail_r}\right)}{\max\left(w_1, .., w_r\right)} \tag{5}$$

Here, the weight of a given resource x is denoted as w_x and we assume that for every resource x the weight $w_x > 0$. There are two major problems with the weighted *max*-based functions. The first problem (A) is that in some

situations we often cannot distinguish between full and partial consumption of the most "expensive" resource. The second problem (B) is that sometimes we cannot properly penalize total consumption of "cheap" resources. As stated by the rule 3, if a job fully consumes some resource on a given machine, we require full penalty for such a job as it "disabled" the whole machine that cannot be used to process other jobs. Let us consider Formula 4 first. Problem (A) appears whenever the most expensive resource has its weight $w_{most} > 1$. For example, let $w_{most} = 2$. Then every job requiring at least $1/2$ of that resource will always receive maximum penalty. Clearly, this behavior is not fair. Problem (B) can appear when $w_{most} \leq 1$. Then it can easily happen, that we cannot properly penalize full consumption of some "cheap" resource. For example, let the fully consumed "cheap" resource has weight $w_{least} = 0.1$ while the weight of the most expensive resource is, e.g., $w_{most} = 1$ and its utilization is only 50%. Then Formula 4 resolves as $P_j = \min(1, \max(0.5, 0.1)) = 0.5$. Clearly, instead of $P_j = 1$ we only get 0.5, failing to meet the requirements described by the rule 3. In case of Formula 5, the problem (A) is eliminated, however the second problem (B) can still appear. For example, let the fully consumed "cheap" resource has $w_{least} = 1$. Let the "expensive" resource be only occupied by, e.g., 10% with $w_{most} = 2.0$. Then Formula 5 resolves as $P_j = \max(0.2, 1.0)/2 = 0.5$. Clearly, instead of $P_j = 1$ we only get 0.5, failing to meet the requirements described by the rule 3. Therefore, *max*-based penalty also breaks the "weights" rule 5. Based on these findings we have decided to analyze whether there is a chance to find a new penalty function that would overcome aforementioned problems.

3.2 Penalties Based on Combination of All Resources

Following text summarizes our attempts to develop a new penalty function that would also reflect nondominant resources as required by the rule 2. Three types of penalty functions are considered and their strengths and weaknesses are discussed in the following text.

$$W = \sum_{x=1}^{r} w_x \tag{6}$$

$$P_j = \prod_{x=1}^{r} \frac{req_{x,j}}{avail_x} \tag{7}$$

$$P_j = \left(\prod_{x=1}^{r} \left(\frac{req_{x,j}}{avail_x} \right)^{w_x} \right)^{\frac{1}{W}} \tag{8}$$

The first candidate depicted by Formula 7 uses a *product* of each resource's relative requirement. Originally, this function has been used only on two resources [15] where relative CPU and RAM requirements have been multiplied. The idea behind this approach is that consumed CPUs and RAM can be represented as 2D objects, where the multiplication represents de facto a "rectangle

area" of consumed resources [15], thus reflecting consumption of both CPUs and RAM. The resulting distribution of penalties is illustrated by Fig. 4 (left).

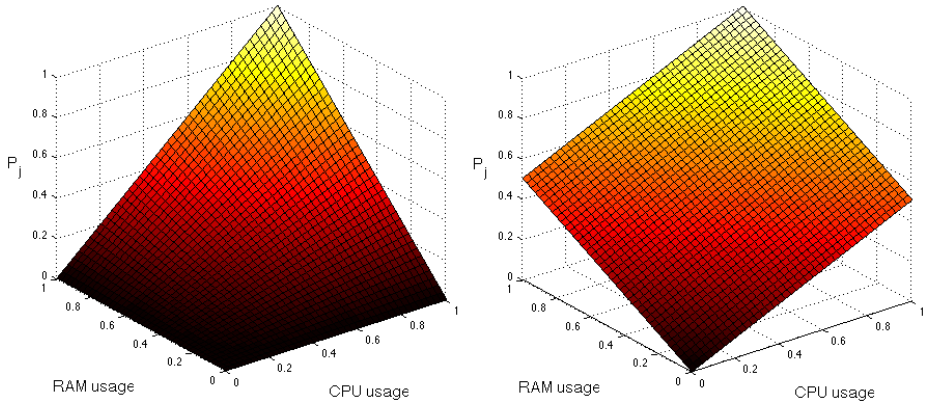

Fig. 4. Product-based penalty (left) and sum-based penalty function (right).

Sadly, this penalty function is not very suitable. As can be seen in the graph, the function assigns low penalties for highly asymmetric requests, breaking the rule 3. For example, if a user consumes all CPUs and little RAM the resulting penalty is very low compared to a scenario where "symmetric" user's job consumes all available CPUs and RAM. This appears to be unacceptable and very unfair behavior. Our analysis quickly revealed that this penalty also breaks the "linearity" rule 4. The problem lies in the adopted idea of "rectangle area", i.e., in the multiplication of CPU and RAM requests. Consider following simple scenario with two users in a system consisting of 10 CPUs and 10 GB of RAM. The first user requests 9 CPUs and 9 GB of RAM and thus gets the penalty $P_j = 0.9 \cdot 0.9 = 0.81$. The second user wants to run 9 jobs, each requiring 1 CPU and 1 GB of RAM. The total penalty for the second user is therefore $P_1 + .. + P_9 = 9 \cdot (0.1 \cdot 0.1) = 0.09$. However, both users consumed the same amount of resources. Apparently, the multiplication is a bad idea which leads to nonlinear behavior that may produce different penalties for the same amount of consumed resources. Due to the associative property of multiplication, we cannot apply weights by multiplying each resource's usage by its weight. Instead, we have to apply slightly more complicated function as is presented in Formula 8[8].

In the next attempt we have removed the multiplication and applied a *sum*-based function instead, to guarantee linear behavior. The resulting penalty function is shown in Formula 9 that summarizes all relative resource requests. Corresponding distribution of penalties is shown in Fig. 4 (right).

[8] The W parameter used in Formula 8 and lately in Formula 10 and Formula 12 is computed using Formula 6.

$$P_j = \frac{1}{r}\sum_{x=1}^{r}\frac{req_{x,j}}{avail_x} \tag{9}$$

$$P_j = \frac{1}{W}\sum_{x=1}^{r} w_x \frac{req_{x,j}}{avail_x} \tag{10}$$

This formula is linear (rule 4) and considers all resources (rules 1, 2) and can be extended to support weights as shows Formula 10. Still, it has one major drawback since it does not assign maximum penalty when a given resource is fully consumed, i.e., it breaks the important "max-min penalty" rule 3.

As a result, we propose a *root*-based penalty function that removes most of the problems mentioned for Formulas 2–10. This penalty function is shown in Formula 11 (symmetric version) and Formula 12 (weighted version), respectively. Corresponding distributions of penalties are depicted in Fig. 5.

$$P_j = 1 - \sqrt[r]{\prod_{x=1}^{r}\left(1 - \frac{req_{x,j}}{avail_x}\right)} \tag{11}$$

$$P_j = 1 - \left(\prod_{x=1}^{r}\left(1 - \frac{req_{x,j}}{avail_x}\right)^{w_x}\right)^{\frac{1}{W}} \tag{12}$$

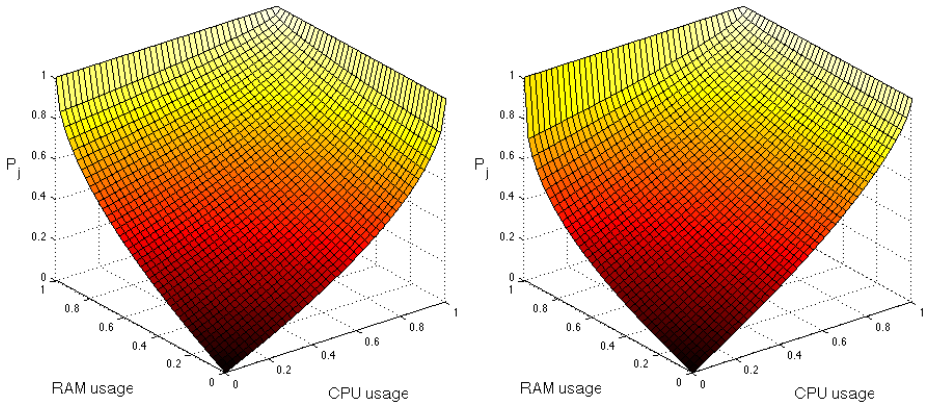

Fig. 5. Root-based penalty function (left) and its weighted version (right).

As can be seen in Fig. 5 (left) the function represents good compromise between the pure dominant resource-based *max* function and the aforementioned functions that combine all resources. More precisely, this *root*-based penalty follows the rules 1, 2, 3 and 5 as we show in the following discussion. The function

combines all consumed resources, thus it fulfills the rules 1 and 2. Notably, unlike the *max*-based function, it also reflects all nondominant resources, i.e., it motivates users to better estimate all resource-related parameters. It also follows the rule 3 as it assigns reasonably high penalties for jobs with asymmetric requests, especially total consumption of selected resource results in a full penalty. Last but not least, it can be easily extended to follow the "weights" rule 5 as depicts Formula 12. Using weights, the corresponding distribution of penalties is then adjusted as shown in Fig. 5 (right). In this case we have chosen $w_{CPU} = 2.0$ and $w_{RAM} = 1.0$ which results in a steeper shape of CPU-related curve. Also, RAM-related curve has changed, having lower initial elevation that only increases when RAM usage approaches its upper limit. Still, one problem remains — the root-based penalty function breaks the "linearity" rule 4.

4 Summary and Discussion

In this paper we have presented several problems that arise when seeking for truly fair and flexible multi resource-based penalty function. The overall results are presented in Table 1 that summarizes capabilities of considered penalty functions with respect to those five rules that were established in order to represent our requirements on a proper penalty function.

Table 1. Suitability of penalty functions with respect to required rules.

	rule 1	rule 2	rule 3	rule 4	rule 5
CPU-based penalty (Formula 2)	NO	NO	NO	**YES**	NO
Max-based penalty (Formula 3)	**YES**	NO	**YES**	**YES**	NO
Product-based penalty (Formula 8)	**YES**	**YES**	NO	NO	**YES**
Sum-based penalty (Formula 10)	**YES**	**YES**	NO	**YES**	**YES**
Root-based penalty (Formula 12)	**YES**	**YES**	**YES**	NO	**YES**

None of the presented functions fulfill all requirements at once. In fact, it is impossible to find a function that would fulfill all five rules, especially the rule 2, the rule 3 and the rule 4 cannot be fulfilled at the same time by one function. For example, as soon as the desired function follows the "max-min penalty" rule 3 it cannot fulfill the rules 2 and 4 at the same time. For simplicity, let us assume a scenario with two resources. If the rule 3 is to be followed, then the desired function must create a surface that comprises the "zero point" (no resource is consumed at all) and the two "maximum lines" (at least one resource is consumed completely) which are highlighted in black color in Fig. 6. Since the "zero point" and the "maximum lines" do not lie in a plane, full linearity of such a function is unattainable. Only partial linearity (linearity with respect to only

one resource) as prescribed by the rule 4 is attainable by the function depicted in Fig. 6 (left). However, such a function clearly fails to follow the rule 2. On the other hand, the rule 2 can be fulfilled if we allow the surface to be curved and smooth as seen for the function in Fig. 6 (right), but then the linearity is broken even in terms of the rule 4.

Thus, if we are decided to follow the rules 2 and 3, we therefore must break the rule 4, i.e., the linearity. Fortunately, it is possible to minimize the adverse effects of non-linearity by requiring that the desired function will assign linear penalties at least when the corresponding jobs have symmetric requirements concerning the relative amount of used resources. This requirement means that the desired function's surface is to comprise also the line connecting the "zero point" and the interconnection point of the two "maximum lines" (all resources consumed completely). As can be check-verified, the *root*-based function, including its weighted version, fulfills this requirement.

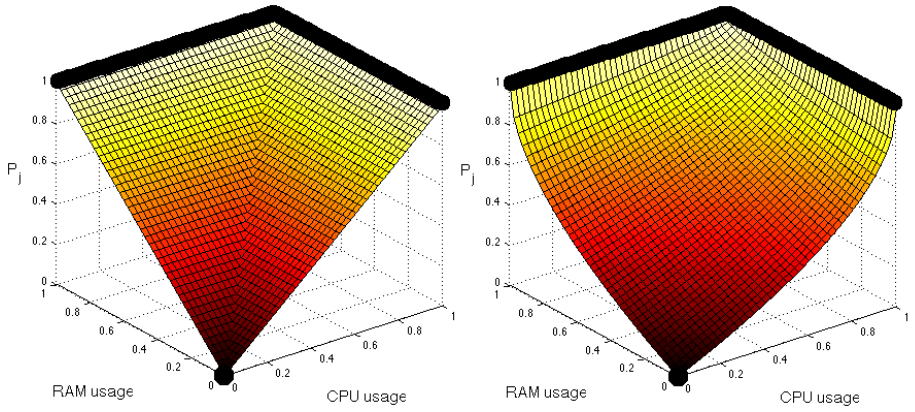

Fig. 6. Non-smooth, *max*-based penalty function vs. smooth, *root*-based penalty function (right).

Still, some of the functions mentioned above are more suitable than the others. The final decision on what penalty function should be applied is however highly individual as different people and/or organizations may have different notion of "what is fair" when it comes to multiple resources [12, 8]. From our point of view, *CPU*-based penalty as well as *product* and *sum*-based penalties are not very good candidates. Clearly, single resource *CPU*-based penalty function fails to meet all rules except for the "linearity" rule 4. As we have already shown in Section 3.2, *product*-based penalty is a very bad candidate while *sum*-based penalty function breaks the important "max-min penalty" rule 3 very heavily (see Fig. 4 (right)).

From our perspective, only two suitable candidates remain: *max*-based penalty and *root*-based penalty. *Max*-based penalty function (Formula 3) fails to fulfill

the rule 2. Moreover, once weights are applied they can cause breaking of the rule 3 (see discussion in Section 3.1). Therefore, in Table 1 we claim that *max*-based penalty function cannot fulfill the "weights" rule 5. *Root*-based penalty fulfills all rules except for the "linearity" rule 4, which is not desirable as it allows users to cheat in some situations. For example, instead of one large job a user can submit two smaller jobs. As a result, he or she will receive smaller penalty. This particular problem can be considered as serious. However, in real life users are often motivated to minimize their requirements concerning available resources. For example, in Ohio Supercomputer Center (OSC) long jobs are only allowed if a user is able to reasonably explain why he or she needs to run such a long experiment [18]. Moreover, parallel jobs have smaller maximal runtime limit compared to serial jobs in OSC. The reason is that long and/or massively parallel jobs can cause fragmentation of system resources [20, 19]. On the other hand, short jobs that are either serial or require only a small amount of CPUs are very suitable for common schedulers as they can be used for backfilling [17].

5 Conclusion and Future Work

This paper addressed an urgent real life job scheduling problem. The goal was to maintain the fairness among different users of the system. The novelty of our work is related to the fact that we consider *multiple* consumed resources when establishing users' priorities. In the area of parallel job scheduling, this problem is very urgent and seems to be rather unexplored. Therefore, we have defined several rules that — according to our knowledge and experience — define the properties that a suitable multi resource-based fairshare algorithm should satisfy. Next, we have discussed the suitability of existing approaches, focusing on the crucial penalty functions. Beside the existing *max*-based functions we have also proposed several other variants of penalty functions and show their strengths and weaknesses. The main result of this paper is the fact that it is impossible to find a penalty function that would satisfy all five rules that we have used to express the fairness-related demands.

We plan to further investigate this problem in the future. MetaCentrum will soon start to use multi resource-based fairshare algorithm. Therefore, we will further analyze the performance and suitability of the production solution as well as possible problems that may appear once the solution becomes fully operational. For example, it is quite obvious that our "max-min penalty" rule 3 is 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, in such special situations this rule is probably too severe and shall be relaxed in the future.

Acknowledgments. We highly appreciate the support of the Grant Agency of the Czech Republic under the grant No. P202/12/0306. The access to the MetaCentrum computing facilities provided under the programme LM2010005 funded by the Ministry of Education, Youth, and Sports of the Czech Republic

is highly appreciated. The Zewura workload log was kindly provided by the Czech NGI MetaCentrum. The access to the CERIT-SC computing and storage facilities provided under the programme Center CERIT Scientific Cloud, part of the Operational Program Research and Development for Innovations, reg. no. CZ. 1.05/3.2.00/08.0144 is appreciated.

References

1. Adaptive Computing Enterprises, Inc. *Maui Scheduler Administrator's Guide, version 3.2*, February 2013. `http://docs.adaptivecomputing.com`.
2. Adaptive Computing Enterprises, Inc. *Moab workload manager administrator's guide, version 7.2.1*, February 2013. `http://docs.adaptivecomputing.com`.
3. Adaptive Computing Enterprises, Inc. *TORQUE Admininstrator Guide, version 4.2.0*, February 2013. `http://docs.adaptivecomputing.com`.
4. Apache.org. *Hadoop Capacity Scheduler*, February 2013. `http://hadoop.apache.org/docs/r1.1.1/capacity_scheduler.html`.
5. Apache.org. *Hadoop Fair Scheduler*, February 2013. `http://hadoop.apache.org/docs/r1.1.1/fair_scheduler.html`.
6. Jacek Blazewicz, Maciej Drozdowski, and Mariusz Markiewicz. Divisible task scheduling - concept and verification. *Parallel Computing*, 25(1):87–98, 1999.
7. 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.
8. 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.
9. M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *SOSP'09*, 2009.
10. 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.
11. 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.
12. C. Joe-Wong, S. Sen, T. Lan, and M. Chiang. Multi-resource allocation: Fairness-efficiency tradeoffs in a unifying framework. In *INFOCOM*, 2012.
13. James Patton Jones. *PBS Professional 7, administrator guide*. Altair, April 2005.
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, Miroslav Ruda, and Hana Rudová. New fairness and performance metrics for current grids. In *Cracow Grid Workshop*, pages 73–74. ACC Cyfronet AGH, 2012.
16. MetaCentrum, February 2013. `http://www.metacentrum.cz/`.
17. Ahuva W. Mu'alem and Dror G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):529–543, 2001.

18. Ohio Supercomputer Center. *Batch Processing at OSC*, February 2013. `https://www.osc.edu/supercomputing/batch-processing-at-osc`.
19. Edi Shmueli and Dror G. Feitelson. Backfilling with lookahead to optimize the performance of parallel job scheduling. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 2862 of *LNCS*, pages 228–251. Springer Verlag, 2003.
20. Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. *IEEE Transactions on Parallel and Distributed Systems*, 18(6):789 –803, 2007.