A Fine-grained Model for Adaptive On-Demand Provisioning of CPU Shares in Data Centers

Emerson Loureiro[*][**], Paddy Nixon, and Simon Dobson

Systems Research Group
School of Computer Science and Informatics
University College Dublin, Dublin, Ireland
{emerson.loureiro,paddy.nixon,simon.dobson}@ucd.ie

Abstract. Data Centers usually host different third party applications, each of them possibly having different requirements in terms of QoS. To achieve them, sufficient resources, like CPU and memory, must be allocated to each application. However, workload fluctuations might arise, and so, resource demands will vary. Allocations based on worst/average case scenarios can lead to non-desirable results. A better approach is then to assign resources on demand. Also, due to the complexity and size of current and future systems, self-adaptive solutions are essential. In this paper, we then present *Grains*, a self-adaptive approach for resource management in Data Centers under varying workload.

1 Introduction

Data Centers usually host different third party applications (e.g., web sites), each of them possibly having different requirements in terms of Quality of Service (QoS) [1,2]. To achieve that goal, however, sufficient resources, like CPU, memory, and storage, must be allocated to each application. One way of achieving this is through a *dedicated model* [3], where sets of servers are exclusively assigned to different application classes. The issue here is to decide the number of servers to be allocated to each class. Another way is through a *shared server model* [3], where a server can host different application classes at the same time. In this case, each class will be assigned a different fraction of the server resources.

The problem is that, in both cases, workload fluctuations might arise [3,4,5], and as a consequence, resource demands across the different application classes will vary [6]. Defining the resources available to a particular application class is thus not an easy task. On the one hand, allocations based on worst-case scenarios could lead to a waste of resources, as most of the times the application class would use less resources than what it currently holds. Average-case scenarios, on the other hand, might cause an application class to be overloaded and thus not able to keep the QoS requirements within reasonable values. A more interesting

[*] This work is supported by UCD Ad Astra Scholarships
[**] This work is partially supported by Science Foundation Ireland under grant number 04/RPI/1544, "Secure and Predictable Pervasive Computing."

approach is then to assign resources on demand [1], in this case, based on current requirements and availability of resources.

As systems become larger and larger, however, it is unlikely that system administrators will be able to respond to changes in resource demands on time. Also, the frequency upon which that occurs is an issue that further complicates this matter. With this view in mind, it is clear that self-adaptive techniques for dynamically deciding the resources available to application classes are required [4]. In this case, that would mean techniques that calculate and assign resource partitions to each class, over time, based on the current state of the system and its goals.

Based on the above, in this paper we present *Grains*, an approach for adaptively managing the resources, more specifically CPU, assigned to the different application classes in a Data Center under varying workload. The rest of this paper is then organized as follows: in Section 2 we present related works in this area and from that we draw the contributions of our work. Next, in Section 3, we present how our approach contemplates the contributions we are aiming. In Section 4 we then illustrate how our work could be used for managing a system's resources through a simulated case study. Finally, in Section 5, the conclusions and future directions of this work are presented.

2 Background

Resource management has received recent attention from the research community, and consequently, a number of solutions have been proposed. Many of them, e.g., [7,8,9,6,3,10], are focused on the shared server model. More precisely, the different application classes running on each server will receive different resource shares over time. In this case, usually, a controller running on each server will decide at runtime the resource shares each class will hold at a particular time t.

Solutions focused on the dedicated model have also been developed. In [1], for example, the notion of Application Environments (AEs) is defined. In this case, AEs hold one or more Virtual Server (VS), which are deployed in different physical machines. The goal then consists in finding the optimal number of VS each AE will hold and in which machines they will be hosted. When more resources are necessary, new VSs will then have to be created and initialized. The amount of resources assigned to the new VS, however, has to be the same as the amount held by the other VSs hosted in a physical machine, thus leading to allocations which are potentially too large.

In [4] and [11], similar approaches are presented. The main difference is that each AE has a fixed number of VSs, matching the number of tiers of each AE. Therefore, the goal of this solution is to determine the resource capacity allocated to each VS of each AE. Problems will happen, though, if the resources held by a particular AE are not enough for the current demand, as it has no further sources of resources, even if there are idle resources available elsewhere.

Another approach in this context is presented in [12]. In such a work, a Cluster Reserve holds a number of Resource Containers. The goal is then find, for each Cluster Reserve, the allocation each of its Resource Containers will hold in the physical machines available. In this case, a minimum allocation has to be assigned to each Cluster Reserve in each server, even if they don't need it. This means that resources can be wasted if there are many idle service classes.

A similar approach has been proposed in [13], where the servers of the system are partitioned into as many subsets as the number of classes of users. A server can then be shared by different classes of users. However, only up to two classes can be sharing the same server. The works presented in [14] and [15] propose similar ideas. The major difference is that each server in the system is allowed to serve only one application class/tier at a particular moment. Allocating an entire server to an application class/tier, however, can cause over-provisioning.

The assignment of processors to applications is investigated in [5]. More precisely, a CPU Manager is in charge of deciding how many processors to assign to an application class so that its Service Level Agreements can be met. As some of the approaches discussed so far, over-provisioning might happen, since entire processors are assigned to an application.

We then summarize the limitations in current resource management solutions for the dedicated model in the following way:

- **Static allocations**: fixed number of resources available to an application class [4,11]. If an application class is idle, its resources can not be used by one that is under heavy load.
- **Coarse allocations**: entire servers [1,14,15] or processors [5] are assigned to application classes. Allowing servers/processors to be split between different application classes is essential (i.e., fine-grained allocation [16]), since one processor/server dedicated to an application might be too much.
- **Limitations on resource shares**: even when allocations can be fine-grained, either a minimum [12], and possibly idle, or pre-defined [1] allocation is required, or a limited number application shares are allowed to coexist on the server/processor [13].

Based on the above, this work will then extend current resource allocation methods for the dedicated model, by providing *Grains*, **GR**anular **A**llocation **I**n **N**etworked **S**ystems, a solution that aggregates the following characteristics:

- **Dynamic allocations**: resource allocations across the system will change over time to match current needs. Also, consumers (i.e., anything that can use resources) will be using resources from a time-varying set of servers, instead of having a static number of them.
- **Fine-grained allocations**: resources provided by a server can be used by any number of consumers, not being restricted to one or a few application classes.
- **On demand allocations**: allocations across the system will be created and defined on-demand, instead of having a pre-defined or minimum size.
- **Hybrid**: a fine-grained feature will imply in a model that combines characteristics from both the dedicated and shared server models. It will be dedicated in the sense that a specific, but variable, number of resource shares will be serving an application class at each moment. But it will also be shared,

in the sense that a server will be hosting requests from different application classes.

3 Grains

In this section, we describe *Grains*, the solution we are proposing for the resource management problem. We start by describing the high level details of *Grains*, i.e., definitions and architecture, and from that, we present the model that defines in more details how the different parts of the architecture interact and also the optimization problem whose solution will give us the best resource partitioning.

3.1 Concepts and Architecture

In our architecture, a system is viewed as a set of *Servers*. A Server, in this case, is anything that can offer and/or use CPU shares. The CPU resources of a Server are split into two partitions. One of these partitions, called *Dedicated Partition*[1], is to be used exclusively by the Server itself. The other partition, *Donation Partition*, is to be used not only by its own Server but also by any other Server in the system. The size of both partitions are defined in terms of percentage (e.g., the size of a Dedicated Partition might be 65% of the Server's CPU with the remaining 35% being allocated to the Donation Partition). In a multi-core architecture, for example, one of the cores could be defined as the Dedicated Partition and the remaining ones as different shares of the Donation Partition.

A Server can then act as: 1) an application class, thus using its own or another server's CPU shares to fulfill its QoS requirements; 2) a resource provider (like Virtual Servers and Resource Containers described in Section 2), thus providing resources to other Servers; or 3) both. Note that we are not making any assumption about the nature of the Server in terms of how it is deployed. It can be an actual physical server or even a virtual one running on a host operating system in a physical machine.

The CPU shares in a Donation Partition can be allocated to more than one Server at the same time. Shares are allocated in *Blocks* of a specific size, which are the atomic unit (i.e., grains) of allocation and determines the percentage of the Donation Partition that will be allocated. For example, if the size of the Block is 10%, shares in the Donation Partition will have a size which is a multiple of 10 (e.g., 20%, 30%). This way, two different Servers A and B could then hold different shares in another Server's Donation Partition, say, 20% for A and 30% for B. Besides, these shares could have their size changed at any time.

Note that, by having a Donation Partition that uses 100% of a Server's CPU resources, and blocks with sizes of 100%, we then reduce our solution to the case where Servers are allocated exclusively for an application class. This is the case with some related works in this area. As one can notice, then, we enable this

[1] The Dedicated Partition is defined only for modelling purposes, as it does not play any role in the rest of our solution.

same kind of allocation in a single model, with the advantage of still being fine-grained. It just depends on how the size of the block and the Donation Partitions are set.

In Figure 1 we illustrate a sample configuration of a system, focusing on how Servers, Donation Partitions, and shares relate to each other. In the figure, the gray rectangles on each Server indicate different shares in the Server's Donation Partition. The arrows, on the other hand, indicate which share is assigned to which Server, by pointing from the Server that is holding the share to the one that is providing it.

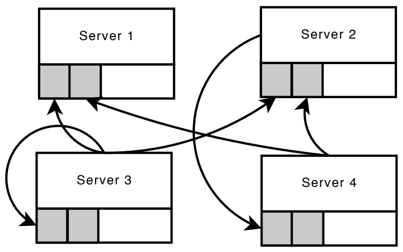


Fig. 1: Possible configuration of a system with four Servers.

The extra CPU shares a Server will receive, coming either from its own Donation Partition or from a remote Server's one, will be based on the Server's *Utility Function*. In short, the Utility Function of a Server will indicate, at any point in time, how useful it is for that Server to receive extra CPU shares.

By taking the Utility Functions of all Servers together, a *Global Controller* will then take care of deciding the shares that each Server will receive, and where they will come from. Therefore, the Utility Function of the Servers should be designed in a way that, by having the Global Controller to decide the best CPU shares for each Server over time, the system as whole is driven towards a common goal (i.e., load balancing, improving overall response time).

3.2 Model

From the concepts presented so far, a formal model of the system has been defined. In such a model, let S be the set of Servers in the system, then:

$$S = \{s_i : i \in \mathbb{N} \wedge i \in [1, n]\}$$

where n is the number of Servers in the system. As we've mentioned, the Donation Partition is a share of a Server's CPU that is available to be used by the Server itself as well as by other Servers in the system. Let p^i be the current size of Server's i Donation Partition, then:

$$\forall s_i \in S \left(p^i = n.R\right),$$

where R is the size of the allocation block, $0 \leq R \leq 1$, and n is the number of blocks available in the Donation Partition to be assigned. Consequently,

$$(n \in \mathbb{N}) \wedge \left(0 \leq n \leq \left\lfloor \frac{1}{R} \right\rfloor\right). \tag{1}$$

With that, we then model the fact that Donation Partitions have to have a size defined in terms of a block, and that such a size will be within 0% and 100% of the total server CPU capacity.

Since in each Donation Partition shares can be allocated to one or more Servers, we then have to keep track of the current shares being held in each Server's Donation Partition. Based on that, let d^i be the set of the current shares allocated in the donation partition of server i, then:

$$d^i = \{\{s_k, r^k\}^*\} \tag{2}$$

where s_k is a server having a share in s_i's Donation Partition, $s_k, s_i \in S$, and r^k is the share held by s_k. As mentioned before, shares are allocated in blocks, just like in the definition of the Donation Partition's size. Consequently, each allocation r^k in d^i is of the form $r^k = n.R$ and 1 holds. Given the total size of a Server i's Donation Partition (p^i) and the current allocations on it (d^i), we then denote the amount of free shares, f^i, in such a Donation Partition, by:

$$f^i = p^i - \sum_{j=1}^{|d^i|} d^i_{j2}$$

Since the size of the Donation Partition can be changed at runtime, we need to specify which properties we would like to hold once that happens. For example, we do not want a Donation Partition to be resized to less than the amount of shares currently in use on it, as that would imply reducing each of the shares accordingly. We then denote by $z^i(r)$ the function that will cause the Donation Partition of a Server i to be resized by the amount r, $z^i(r) = p^i + r$. Clearly, if $r > 0$, the Partition is having its size increased. If $r < 0$, on the other hand, its size is being decreased. For that, however, the following has to hold:

$$\left(p^i \geq p^i - f^i\right) \wedge \left(\forall j \in \left[0, |d^i|\right] \left(d^i_{j2} = d^i_{j2}\right)\right) \tag{3}$$

Basically, we are asserting that the size of a Donation Partition can never be less than what is currently being used on it and also that all shares from the different Servers on it will remain the same.

As we are talking about shares being allocated in a Server's Donation Partition to other Servers, we need to formalize that in terms of the system properties that have to hold after an allocation is defined. For that, then, $a^{s_i s_k}(r)$ denotes a function that allocates a share r in the Donation Partition of Server i to Server k. Clearly, if $i = k$, then Server i is receiving extra CPU shares from its own Donation Partition. For that to happen, the following has to hold:

$$\left(\exists j \in \left[0, |d^i|\right] \left(d^i_j = \{s_k, r\}\right)\right) \wedge \left(\forall w \in \left[0, |d^i|\right] \left(w \neq j \rightarrow d^i_{j1} \neq s_k\right)\right),$$

which states that one, and only one, pair <*server*, *allocation*>, where *server* is s_k, should exist in the current set of allocations of Server's i Donation Partition. In this case, all properties of 2 must hold.

Finally, we denote by $u^i(l,r)$ the utility function of Server i, where l is the new share that Server i will receive from its own Donation Partition and r is a vector of the new shares that Server i will receive from remote Servers. More formally:

$$u^i : \left[-l^i, f^i\right] \times \left[-r^{ik}, f^k\right]^{|S|} \rightarrow [0,1] \tag{4}$$

where l^i is the current share Server i holds in its own Donation Partition and r^{ik} is the current share Server i holds in the Donation Partition of Server k, $\forall k, i \in [0, |S|]\,(k \neq i)$.

From the constructs defined so far, we can then model the optimization problem the Global Controller is supposed to solve. We assume that the controller will execute continuously over time, over a fixed or varying frequency. Every time it runs, it thus has to find the best way of distributing shares in the Donated Partition of all Servers among them. That will consist of solving the following optimization problem:

$$\max_{R^l, R^m} \sum_{i=1}^{|S|} u^i(l^i, r^i) \tag{5}$$

where $R^l = \{l^{1*}, l^{2*}, ..., l^{n*}\}$, $R^m = \{r^{1*}, r^{2*}, ..., r^{n*}\}$, l^{i*} is Server i's optimal CPU share coming from its own Donation Partition, and r^{i*} is Server i's optimal CPU share coming from remote Servers' Donation Partition, given that the following constraint hold:

$$N \leq f^i,$$

where N is the sum of the new shares allocated on Server i, $N = R^l_i + \sum R^m_{k2}$, $\forall k \in [0, |R^m|]\,(R^m_{k1} = s_i)$. This constraint simply states that the sum of new shares on each Server has to be less than the current amount of free shares on it.

4 Case Study

In this section, we present a case study to illustrate the use of *Grains* in the resource management problem. In our case study, we consider a scenario with two physical machines, each of them running a Server, as defined in our model. Each server is associated with an application class (A and B). The global goal of the system is to keep the average response time of the requests in each Server as bellow as possible from a target value (e.g., 3s).

To achieve that, the CPU shares each Server holds will be reallocated on demand, based on the aggregate utility of their utility functions. We consider that the Global Controller executes continuously, at specific intervals, which we will call *iteration*. At each iteration, the Global Controller decides the best CPU shares for each Server, and performs the allocation, which will then hold until the next iteration. In this case, the best CPU shares for the Servers will be those that will cause their response time to be as less as possible from the target value.

The utility function u of a Server is then defined in the following way:

$$u^i(l,r) = \begin{cases} t^i(l,r) & \text{, if } r^i = \emptyset \\ \frac{h^i(l)+y^i(r)}{2} & \text{, if } r^i \neq \emptyset \end{cases} \tag{6}$$

where 1) r^i is the set of the current shares held by Server i that come from remote Servers' Donation Partition, and, since we have only two servers, $|r^i| = 1$ (i.e., a Server i can only have shares from up to one remote Server); 2) $t^i(l,r)$ is the utility of Server i receiving shares l and r, respectively from its own Donation Partition and from a remote Server's one, given that it does not hold any remote share yet ($r^i = \emptyset$); 3) $h^i(l)$ is the utility of Server i receiving the new share l from its own Donation Partition considering that it already holds a remote share; and finally 4) $y^i(r)$ is the utility of the remote share held by Server i receiving the new share r.

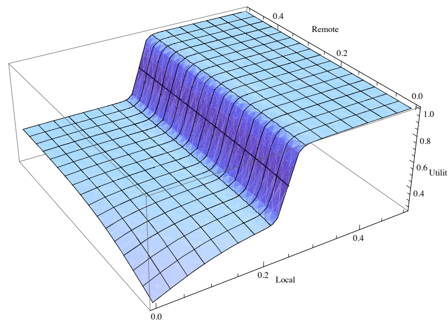
Function $t^i(l,r)$ in Equation 6 is then defined in the following way:

$$t^i(l,r) = \frac{\left(\frac{1}{\left(1+\lambda * e^{\left(\frac{-(1/b(l))}{\eta}\right)}\right)} + \frac{1}{\left(1+\theta * e^{\left(\frac{-(1/a(l,r))}{\zeta}\right)}\right)} \right)}{2},$$

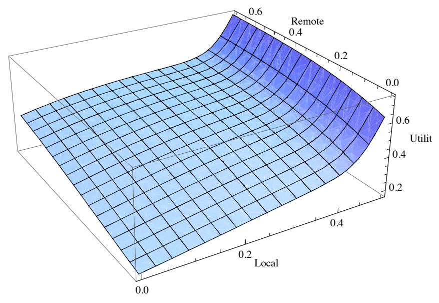
where $b(l)$ is the expected response time in Server i until the next iteration, given the extra local share l, $a(l,r)$ is the expected remaining response time in Server i given that it will receive the extra share l from its own Donation Partition and also the extra share r from a remote Server's Donation Partition, and η and ζ are specific constant values for defining the growth of $t^i(l,r)$ over the x and y axis.

The general idea of this utility function is to make it grow fast to 1 along the x axis (i.e., local share received), once the target response time is reached, indicating the fact that achieving a response time that is less than the target one with a particular local share is of high utility (requests won't have to be redirected from the Server to a remote share). This is the case illustrated in the graph of Figure 2a. The value on the x axis where this growth occurs is controlled by λ, which is calculated by finding λ such that $\frac{\partial^2 t}{\partial l^2} = 0$, with $\theta = 0$ and l being the result of solving $b(l) = 2$, with 2 being the target response time to be reached.

However, when the local share received is not enough to cause the response time to be less than the target one, the utility still increases considerably as the remote share received increases. This is the case illustrated in the graph of Figure 2b. The growth of the function in this direction is controlled by θ, calculated by finding θ such that $\frac{\partial^2 t}{\partial r^2} = 0$, with $\lambda = 0$, $l = 0$, and r being the result of solving $a(l,r) = 2$. Notice that, in this case, there is no point along the x axis to cause the utility to grow fast to 1, which would be the case if the target response time is reached. But still, the utility continues to grow as more remote shares are provided.



(a)



(b)

Fig. 2: Graph of t (a) with enough local shares (b) without enough local shares

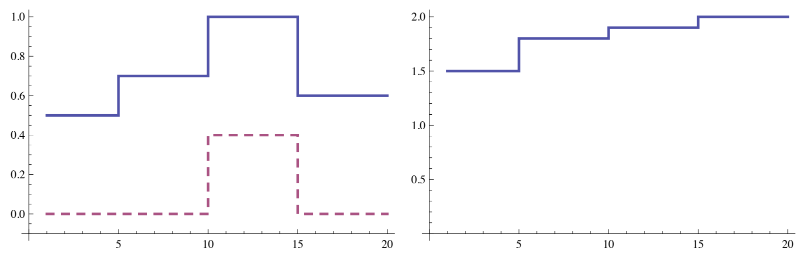
As for the utility functions $h^i(l)$ and $y^i(r)$ of a Server i, they behave similarly to $t^i(l,r)$. The difference is that each of these functions only consider either the share received from i's Donation Partition, for $h^i(l)$, or the share received from a remote Server's Donation Partition, for $y^i(r)$. The definitions for both of these functions have thus been omitted from this paper.

Now we'll illustrate a scenario where the extra CPU shares of Server 1 will be dynamically adjusted to meet current demand. In this scenario, we will be looking at the following aspects: 1) the current extra CPU share held by the Server coming from its own Donation Partition, 2) the current extra CPU share held by the Server coming from Server 2's Donation Partition, and 3) the response time in Server 1. The idea with this scenario is illustrate not only the shares of Server 1 being dynamically adjusted but also the fact that it will receive remote shares on demand, when the local shares it holds is no longer enough to keep the response time bellow or equal to the target value. Initially, Server 1's Donation Partition uses 50% of its CPU, which means that the other 50% are to be used exclusively by it. The total local share held by Server 1 is then 50%, as illustrated by the solid line in the graph of Figure 3a (in the graph, values are in between 0 and 1).

1. Server 1 initiates with a response time of 1.5s, with 2s being the target value, as illustrated in Figure 3b.
2. At iteration 5, the expected response time increases to 2.65s. The Global Controller then decides that the best allocation is to give an extra 20% of local share to Server 1, resulting in a response time around 1.8s. For that, the Global Controller then calls $a^{s_1 s_1}(0.2)$, as specified in the model.
3. At iteration 10, the expected response time increases again, now to 4s. In this case, even by providing the remaining local share (i.e., 30%) to Server

1, the response time on it would not be less than the target value. In this case, the Global Controller then decides to allocate 40% of the Donation Partition of Server 2 to Server 1 (dashed line in the graph of Figure 3a), in a way that by redirecting requests to Server 2, a better response time can be achieved. The result of this decision is then calling $a^{s_1 s_1}(0.3)$ and $a^{s_1 s_2}(0.4)$, to apply the changes in Server 1's local and remote shares. By doing that, the resulting response time of Server 1 is around 1.9, thus bellow the target value.

4. At iteration 15 then, all requests occupying the remote share used by Server 1 are finished, so such a share is revoked by the Donation Partition of Server 2, causing the remote share of Server 1 to go to 0 again. Also, the expected response time in Server 1 drops to 1s, causing the Global Controller to decrease the extra local share held by Server 1 to 10%, since the current local share it holds is too much for that response time. Server 1 then ends up with a total local CPU share of 60%, a setting that causes the response time to be exactly 2, still within the specified target value (in this case, we have chosen to keep the response time bellow the target value but without overusing the CPU resources).



(a) (b)

Fig. 3: (a) Variation of the local shares (solid line) and remote shares (dashed line) held by Server 1. (b) Variation of the response time.

Note that, in item 3, Server 1 had *no* pre-defined or minimum share in Server 2, as is the case with some approaches. Instead, its remote share was allocated at runtime and only when necessary. Also, from then on, Server 1 remote share's utility, $y^i(r)$ in Equation 6, would affect the optimization process, by specifying which size would be the most useful for it in a particular iteration. This means that if it needs more resources, its size would simply be increased, instead of creating more and more shares on different Servers, as some of the current solutions do.

Even though we have not illustrated changes in the size of the Donation Partition, it is worth stressing the fact that, as we have showed, this is supported by the model, using the function $z^i(r)$, as explained in Section 3.2. If at some point, it is decided that the Donation Partition of Server 2, for instance, should be resized to, lets say, 70% of its CPU capacity, all it would need to be done is a call to $z^i(0.2)$ (since it currently has a size of 0.5). This way, more resources would be available on this Donation Partition to be used by Server 1, with all the important properties regarding the allocated shares still holding.

5 Conclusions

In this paper we presented *Grains*, an approach for dynamically managing CPU resources in Data Centers under varying workload. A formal model for that has been proposed, thus clearly defining relationships within the system as well as properties that have to hold over its execution. As we have showed, a number of solutions within this context can be found. However, limitations in these solutions have been pointed out, such as 1) being fixed to a specific number of servers, 2) creating/initializing new Virtual Machines when remote shares are necessary, instead of just resizing their shares, or 3) creating idle shares to satisfy a minimum allocation constraint. In our case, we were able to cope with the resource management problem with a solution that overcomes such limitations (e.g., resources from a Server can be assigned to a number of other Servers and shares are dynamically allocated with no minimum/pre-defined values). Even though we have focused on CPU assignment, the *Grains* model could well be used for other kinds of resources, as long as they can be divided and shared (e.g., storage).

Whereas the approach we propose in this paper is interesting, there are issues to consider in the future. Dynamically deciding the proper size for a Server's Donation Partition, for example, is an issue to be further investigated. By that we mean finding what is the best size for the Donation Partition of a Server, considering parameters like how often it uses extra shares and how idle its Dedicated Partition is. That would allow us to make a better usage of the CPU resources, by setting the size of each Server's Donation Partition to the minimal CPU requirements it would need, and allocate the rest on demand. Another approach would be to set the size to zero, so that all resources a Server holds would be decided based on its current needs. As one can note, our approach is flexible enough for both cases, an aspect that is not found in other solutions.

Another area of future work is to decentralize the control over the system. In this case, the idea is to design algorithms that will converge to the optimal allocation by having different Servers in the system taking decisions using as much local information as possible. The best allocation, in this case, would thus emerge as the result of local interactions between the Servers of the system.

References

1. Wang, X.Y., Lan, D.J., Wang, G., Fang, X., Ye, M., Chen, Y., Wang, Q.: Appliance-based autonomic provisioning framework for virtualized outsourcing data center. In: ICAC '07: Proceedings of the Fourth International Conference on Autonomic Computing, Washington, DC, USA, IEEE Computer Society (2007) 29

2. Harada, F., Ushio, T., Nakamoto, Y.: Adaptive resource allocation control for fair qos management. IEEE Transactions on Computers **56**(3) (2007) 344–357
3. Chandra, A., Gong, W., Shenoy, P.: Dynamic resource allocation for shared data centers using online measurements. In: Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, New York, NY, USA, ACM (2003) 300–301
4. Wang, X., Du, Z., Chen, Y., Li, S.: Virtualization-based autonomic resource management for multi-tier web applications in shared data center. Journal of Systems and Software (2008) In Press.
5. Guitart, J., Carrera, D., Beltran, V., Torres, J., Ayguadé, E.: Dynamic cpu provisioning for self-managed secure web applications in smp hosting platforms. Computer Networks **52**(7) (2008) 1390–1409
6. Liu, X., Zhu, Z., Singhal, S., Arlitt, M.: Adaptive entitlement control of resource containers on shared servers. In: 9th IFIP/IEEE International Symposium on Integrated Network Management, IEEE Computer Society (2005) 163–176
7. Menasce, D.A., Bennani, M.N.: Autonomic virtualized environments. In: Proceedings of the 2006 International Conference on Autonomic and Autonomous Systems, Washington, DC, USA, IEEE Computer Society (2006) 28
8. Ionescu, D., Solomon, B., Litoiu, M., Mihaescu, M.: A robust autonomic computing architecture for server virtualization. In: Proceedings of the 2008 International Conference on Intelligent Engineering Systems, Washington, DC, USA, IEEE Computer Society (2008) 173–180
9. Garbraick, P., Naik, V.K.: Efficient resource virtualization and sharing strategies for heterogeneous grid environments. In: 9th IFIP/IEEE International Symposium on Integrated Network Management, IEEE Computer Society (May 2007) 40–49
10. Tesauro, G., Walsh, W.E., Kephart, J.O.: Utility-function-driven resource allocation in autonomic systems. In: Proceedings of the Second International Conference on Autonomic Computing, Washington, DC, USA, IEEE Computer Society (2005) 342–343
11. Padala, P., Shin, K.G., Zhu, X., Uysal, M., Wang, Z., Singhal, S., Merchant, A., Salem, K.: Adaptive control of virtualized resources in utility computing environments. In: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, New York, NY, USA, ACM (2007) 289–302
12. Aron, M., Druschel, P., Zwaenepoel, W.: Cluster reserves: a mechanism for resource management in cluster-based network servers. SIGMETRICS Perform. Eval. Rev. **28**(1) (2000) 90–101
13. Zhang, J., Hämäläinen, T., Joutsensalo, J.: A new mechanism for supporting differentiated services in cluster-based network servers. In: Proceedings of the 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'02), Washington, DC, USA, IEEE Computer Society (2002) 427
14. Urgaonkar, B., Chandra, A.: Dynamic provisioning of multi-tier internet applications. In: Proceedings of the Second International Conference on Automatic Computing, Washington, DC, USA, IEEE Computer Society (2005) 217–228
15. Sivasubramanian, S., Pierre, G., van Steen, M.: Towards autonomic hosting of multi-tier internet applications. In: Proceedings of USENIX Hot Topics in Autonomic Computing, Washington, DC, USA, IEEE Computer Society (2006)
16. Steinder, M., Whalley, I., Carrera, D., Gaweda, I., Chess, D.: Server virtualization in autonomic management of heterogeneous workloads. In: Proceedings of the 10th IFIP/IEEE International Symposium on Integrated Network Management, Washington, DC, USA, IEEE Computer Society (2007) 139–148