# A Global Scheduling Framework for Virtualization Environments

Yoav Etsion*† Tal Ben-Nun* Dror G. Feitelson*

*School of Computer Science and Engineering
The Hebrew University of Jerusalem
91904 Jerusalem, Israel
Email: {talbn,feit}@cs.huji.ac.il

†Barcelona Supercomputing Center (BSC)
08034 Barcelona, Spain
Email: yoav.etsion@bsc.es

***Abstract*—A premier goal of resource allocators in virtualization environments is to control the relative resource consumption of the different virtual machines, and moreover, to be able to change the relative allocations at will. However, it is not clear what it means to provide a certain fraction of the machine when multiple resources are involved. We suggest that a promising interpretation is to identify the system bottleneck at each instant, and to enforce the desired allocation on that device. This in turn induces an efficient allocation of the other devices.**

## I. Introduction

We are now witnessing the second wave of virtualization. The first wave occurred about forty years ago and led to the widespread use of virtualization as a means to share large scale mainframe platforms [8]. The second wave, started about ten years ago, concerns datacenter servers and desktop PCs, which are now powerful enough to support multiple virtual machines on a single physical host machine (partly due to specialized hardware support [16]). It is driven by the utility of virtualization for server consolidation, flexible provisioning of resources, and support for testing and development of new facilities.

Server consolidation is the practice of migrating distinct legacy servers from distinct physical machines that possibly use different operating systems to virtual machines on a single more powerful platform. This reduces operational expenses by saving the need to maintain and support all those legacy systems, reducing the floor footprint, and reducing power and cooling requirements. It is especially beneficial when it increases server utilization, e.g. if the legacy servers are not highly utilized, but when consolidated they lead to a reasonably high utilization of the new server.

Another important benefit of consolidation is that it promotes flexible provisioning of resources. With consolidation, the resources provided to each server are not fixed. Rather, the different servers compete for resources, which are provided by the underlying virtualization infrastructure. It is then possible to control the resources provided to each one, and assign them according to need or the relative importance of the different servers. Importantly, this partitioning of the resources can be changed easily to reflect changing conditions.

The virtualization infrastructure — usually called a hypervisor or virtual machine monitor — is therefore found to assume many of the basic responsibilities of an operating system, especially with regard to resource allocation and scheduling. Indeed, it is natural to consider the use of well known mechanisms and policies that were originally developed for operating systems and simply port them to the hypervisor or VMM. However, the situation is actually somewhat different.

One difference is that hypervisors typically operate with far less information than an operating system. An operating system mediates all interactions with hardware devices for all processes, where the processes themselves are rather simple in structure. Therefore the operating system can use a pretty simple model of operation, e.g. blocking a process that has requested an I/O operation. But a hypervisor is one level lower down, and supports a virtual machine that in turn runs a full operating system which may support many processes. When some process in the virtual machine requests an I/O operation, this then does not reflect the activity of the virtual machine as a whole — only the activity of that process, which is not even directly known by the hypervisor.

Another difference is that the goals are typically different. Operating systems attempt to optimize metrics such as response time of interactive processes, while at the same time providing equitable service to all the processes. With hypervisors, it is more typical to try and control the resource allocation, and ascertain that each virtual machine only gets the resources that it deserves. To complicate matters, this has to be done in multiple dimensions, reflecting the different devices in the system: the CPU, the disks, and the network connectivity. The question is then what does it mean to provide a certain share of multiple resources, when the processes running on each virtual machine actually require different combinations of resources.

The simplest solution is to use the desired allocation as an upper bound. For example, if a certain virtual machine is to be

allocated 30% of the resources, it will get no more than 30% of the CPU cycles, 30% of the disk bandwidth, and 30% of the network bandwidth. But this can be very inefficient if a set of virtual machines actually have complimentary requirements, e.g. if one only uses the disk while the other predominantly uses the network.

Our framework is designed to tackle this issue. In a nutshell, it is based on the combination of two basic ideas: the use of fair share scheduling to control relative resource allocation, and the identification of the bottleneck device as the locus where such control should be exercised. By controlling the allocation on the bottleneck device, we induce a schedule on other devices as well; this replaces the use of a myopic scheduler on each device that does not take the global system state into account. We claim that this approach provides a meaningful and efficient definition to the concept of predefined allocation of multiple resources.

## II. Related Work

The issue of resource allocation has been studied for many years, but mostly from different perspectives than the one we use. The closest work we know of is the following.

The requirement for control over the allocation of resources given to different users or groups of users has been addressed in several contexts. It is usually called fair-share scheduling in the literature, where "fair" should be understood as according to each user's due rather than as equitable. Early implementations were based on accounting, and simply gave priority to users who had not yet received their due share at the expense of those that had exceeded their share [12], [13]. In Unix systems it has also been suggested to manipulate each process's nice value to achieve the desired effect [11], [6]. Simpler and more direct approaches include lottery scheduling [22] or using an economic model [21], where each process's priority (and hence relative share of the resource) is expressed by its share of lottery tickets or capital.

Another approach that has been used in several implementations is based on virtual time [17], [5]. The idea is that time is simply counted at a different rate for different processes, based on their relative allocations. We use a version called RSVT, for "resource sharing virtual time", which bases scheduling decisions on the difference between the resources a process has actually received and what it would have received if the ideal resource sharing discipline (similar to the ideal processor sharing concept) had been used [7]. A similar approach is used in [3].

Focusing on virtual machine monitors, Xen uses Borrowed Virtual Time (BVT). VMware ESX server uses weighted fair queueing or lottery scheduling. The Virtuoso system uses a scheduler called VSched that treats virtual machines as real-time tasks that require a certain slice of CPU time per each period of real time [14], [15]. Controlling the slices and periods allows for adequate performance even when mixing interactive and batch jobs.

The main drawback of all the above approaches is that they focus on one resource — the CPU. The effect of CPU scheduling on I/O is discussed in [18]. It has also been suggested to temporarily prioritize VMs that do I/O so as not to cause delays and latency problems [9]. However, the interaction of such prioritization with allocations was not considered. Similarly, there has been interesting work on scheduling bottleneck devices other than the CPU, but this is then done to optimize performance of the said device and not to enforce a desired allocation [2], [10], [19].

Adapting to multiple resource constraints was addressed by [4]. However, their approach was to control the applications so that they adjust their usage, rather than enforcing an allocation from the outside. The combination between scheduling and multiple resources was discussed in [1]. However, the context is completely different as they consider targets for migration in the interest of load balancing. Interestingly, the end result is similar to our approach, as they try to avoid machines where any one of the resources will end up being highly utilized and in danger of running out.

Perhaps the closest related work is [23]. This paper also considers the interactions and degradations in service created by allocations on multiple distinct devices. However, the context is interactive applications and the need to maintain responsiveness. The approach is based on specifying the required CPU service as in Virtuoso, and subjecting the other devices to this specification. For example, virtual memory belonging to an application with any such specification will be excluded from paging considerations for a default of 30 minutes.

## III. The Linear Model of Resource Consumption

Computer applications consume multiple resources in an interdependent manner, with resources utilized as needed to perform the computational task at hand. Recently, we are witnessing the increased popularity of *Virtual Appliances*, in which a multiprogrammed application is encapsulated within a virtual machine. A virtual appliance can therefore be considered as a single application (the terms VM and appliance will thus be used interchangeably hereafter[1]).

For a single application, one may assume that the consumption rates of different resources are interdependent. Although these interdependencies are expected to change throughout the entire execution of the application, it is assumed that they remain fairly constant throughout each specific *phase* of the application.

This suggests that limiting the allocation of a virtual appliance on a certain resource, to below the appliance's base consumption, will limit its ability to perform, thereby reducing its consumption of other resources. This phenomenon has been characterized as a *linear* dependence [7], such that doubling a VMs allocation of one device will double its usage of all other devices.

Coupled with this linearity model is the *knee model*: increasing the allocations of a single appliance on all resources

[1] Although general purpose VMs may not behave as a single application, in the context of this paper we only address "appliance" type VMs, as is often the case for server consolidation scenarios.

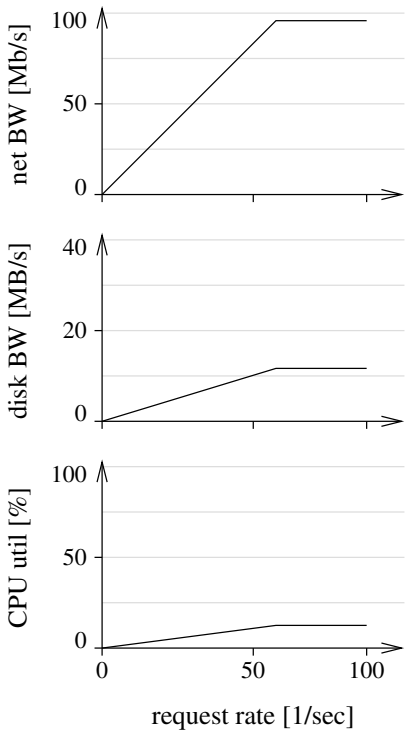


Fig. 1. *Hypothetical example demonstrating the linearity and knee models.*

may yield a threshold above which the appliance cannot utilize some resource. This means that a VM may have a maximal requirement for a device that is less than 100%; if it is given more it will not use the full allocation. Thus there is a knee in the graph of used capacity as a function of allocation.

A depiction of both linearity and knee models is given in Fig. 1. Assume a hypothetical server serving requests for files, using a 100 Mb/s network and a 40 MB/s disk. In this hypothetical scenario everything is very regular: each file is precisely 200 KB long, and processing the request takes exactly 1 ms. Thus if the server receives one request per second, its CPU utilization will be 0.1%, its disk bandwidth consumption 200 KB/s (which is 0.5% of the available 40 MB/s), and its network utilization 1.6 Mb/s (which is 1.6% of the available 100 Mb/s). The linearity assumption states that the ratios between these resource consumptions will remain essentially the same for higher rates. For example, if the server receives 10 requests per second its CPU utilization will rise to 1%, its disk bandwidth to 2 MB/s, and its network bandwidth to 16 Mb/S. The knee assumption kicks in when the request rate rises to about 58 requests per second. At this request rate, the effective network utilization will be around 93 Mb/s. Accounting for headers etc. this will saturate the network. Thus at higher request rates some of the requests will have to be dropped. The assumption is that such requests are dropped immediately upon arrival, so they do not use any resources. Thus the effective disk bandwidth will not surpass 11.6 MB/s, and the CPU utilization will not surpass 6%.

In a more realistic scenario higher request rates will actually place some load on the system, so the effective service rate might be reduced. This emphasizes the importance of controlling the resource allocation and keeping the system near the saturation point.

In practice, the linearity model offers a new approach to the multi-dimensional problem of allocating multiple physical resources to many virtualized hosts — by reducing the partitioning problem to that of throttling requests to a single bottleneck device.

## IV. Bottleneck-Based Scheduling

Our goal is to support the notion of controlled allocation of resources to virtual machines, such that we will be able to say that VM1 gets, say, 50% of the resources, VM2 gets 30%, and VM3 gets 20%. The problem is that a virtual machine may require more of one resource than of another. This raises the question of what exactly is meant by "50% of the machine". We answer this using bottleneck-based scheduling. Specifically, we make the observation that at any given time one device is the system bottleneck — this is the device that has the highest utilization. This leads to the following:

**Main Idea**

One should enforce the desired allocations on the bottleneck device. As each VM uses a different mix of the resources, this will induce different allocations of the other resources. But given that the other resources are *not* the system bottleneck, there is no need to restrict their usage.

The combination of the linearity and knee models discussed earlier suggest that bottleneck-based scheduling is stable. According to the linearity model, reducing allocations on the bottleneck device will likely reduce consumption levels on other resources. It may not reduce consumption only when a VM's allocation on a resource is above its actual consumption to begin with (the knee model). In either case, the consumption of other system's resources will not increase — therefore no other device can become saturated.

The allocations on the bottleneck device will be made using the RSVT algorithm, which uses the notion of virtual time to allocate predefined portions of the resource. The way fractional resource usage is measured depends on the resource in question:

- For the CPU, it is simply measured as a fraction of CPU time. Note that this includes both user time and system time.
- For the network(s), it is measured as a fraction of the used bandwidth, including bandwidth used for various headers. How this is partitioned into packets is of no concern, i.e. we do not count packets.
- For the disk(s), it is also measured as a fraction of the used bandwidth. The disk space used is of no concern.

As the target environment of the proposed framework is consolidated virtualized servers, we assume that all schedulable entities are VMs, and that all regular processes — mainly system daemons — are considered as helpers to the hypervisor.

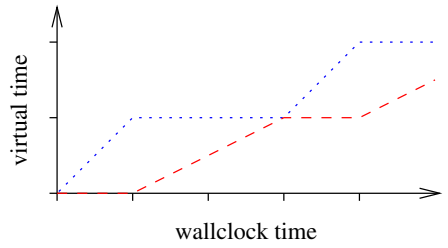


Fig. 2. *Example of differential accounting virtual time.*

As such they are given priority over the VMs, and are handled by an independent mechanism.

### A. The RSVT Algorithm

In general scheduling combines resource allocation and sequencing. RSVT, introduced[2] in [7], tackles this combination. Like other virtual time scheduling schemes it controls allocations by making time pass at a different rate for different processes (or VMs), such that the rate reflects the allocation.

An example of such differential accounting is given in Fig. 2. This shows two processes that are scheduled so as to equalize their consumed virtual time; this means that at each scheduling decision, the process that has accrued less virtual runtime is selected to run next. The process depicted by the dotted line runs first, and is charged at full rate — for every unit of runtime, its virtual time also advances by one unit. The process depicted by the dashed line, on the other hand, is charged at half rate. Thus it appears to consume less of the CPU per time unit, and therefore receives a double allocation.

Virtual time based scheduling performs the sequencing by selecting the process that is most "behind its time" to run next (e.g. [17]). RSVT is especially useful for combining fair shares with pacing. The idea is that rather than just selecting the VM with the lowest virtual time, we select the one with the biggest difference from where it was supposed to be if it was advancing continuously. This helps spread out multiple VMs with the same profile [7].

An important issue is initialization: given that we have a few processes in the system, and a new one arrives, what is its virtual time initialized to? This also relates to cases where a process is in the system but becomes inactive: should it continue to accrue delays relative to its allocated virtual time?

To answer such questions, we note that the absolute value of the virtual time is not really important — it is the difference between the accrued virtual time and the resource sharing virtual time. Thus when a new process arrives we initialize its virtual time to reflect zero delay.

In addition, we suggest to cause an exponential decay of the relative delay of inactive processes, similar to the exponential decay of accrued runtime in the Unix system. This is done for both positive and negative relative usage. On the positive side, it avoids situations where a client gains a huge credit by virtue of not using the scheduled resource, and then starves all other clients once it becomes active again. On the negative side, it avoids situations where a client stays at a disadvantage after using more than its fair share of the resource when no other clients wanted to use it.

[2]That paper used the name PSVT, as it was limited to scheduling the processor rather than general resources.

### B. Global Scheduling and System Bottlenecks

Assume the system has only one bottleneck. Having a single bottleneck implies that other devices are not fully utilized. So they don't have an allocation problem, only a sequencing problem. The idea is then to use RSVT on the bottleneck device, because this is the only place where allocations are limited. Given that RSVT is used on the bottleneck device, some VMs may fall behind their allocation due to local synchronization constraints related to other devices. These VMs can be helped by using the bottleneck RSVT priorities for sequencing on the other devices.

As an example of how allocations on the bottleneck device differ from ordinary CPU scheduling, consider the following synthetic scenario. Assume three VMs with the following maximal use capacity of the devices:

| | CPU | disk | net |
|---|---|---|---|
| VM1 | 20% | 30% | 80% |
| VM2 | 10% | 20% | 30% |
| VM3 | 30% | 50% | 30% |
| total | 60% | 100% | 140% |

In this scenario the network is the bottleneck. Assume a desired allocation of 50%, 30%, and 20% to VM1, VM2, and VM3, respectively. Imposing this on the network will lead to a final usage like this (based on the linear relationship of needs assumption):

| | CPU | disk | net |
|---|---|---|---|
| VM1 | 13% | 19% | 50% |
| VM2 | 10% | 20% | 30% |
| VM3 | 20% | 33% | 20% |
| total | 43% | 72% | 100% |

We claim that this is the desired result.

If we were to attempt to set the allocations on the CPU, the network would become saturated. This would cause feedback effects that influence the actual CPU allocation in ways that are hard to anticipate and control. It is reasonable to assume, however, that the CPU scheduling is done by giving priority to the VMs in a way that reflects the part of their allocation that is being used. Thus VM2 would have the highest priority (using no more than 10 of its allocated 30%) with VM1 close behind (using 20 of its allocated 50%), while VM3 has low priority (capable of using all its allocated 20% or more). Based on the linear relationship of needs assumption this would lead to something like[3]

[3]The numbers shown here are the solution of a set of equations that assume the scheduler actually achieves CPU allocations proportional to the desired allocations, i.e. $\frac{p1*c1}{a1} = \frac{p2*c2}{a2} = \frac{p3*c3}{a3}$, subject to saturating the network, i.e. $p1 * n1 + p2 * n2 + p3 * n3 = 100$, where $pi$ is the fraction of the desired allocation that VMi receives, and $ai$, $ci$, and $ni$ are its allocation, CPU capacity, and network capacity, respectively.

| | CPU | disk | net |
|---|---|---|---|
| VM1 | 16% | 24% | 65% |
| VM2 | 10% | 19% | 29% |
| VM3 | 6% | 11% | 6% |
| total | 32% | 54% | 100% |

Such a result reflects the desired allocations only on the CPU, which is grossly underutilized; it misses them for the network, which is the system bottleneck, and also happens to lead to lower overall system utilization.

It should be noted that our goal is not to maximize utilization, but to maximize utilization *subject to fair share allocations*. For example, if VM1 can use 100% of all devices and VM2 can use different amounts of each device, than maximum utilization is obtained by running only VM1, but if the fair shares are 50% each, then we need a compromise that necessarily reduces the utilization of some devices.

If allocations and priorities are set according to the bottleneck device, this raises the question of what happens to VMs that simply do not use the bottleneck device and therefore have the lowest priority. For example, a VM may not use the network for a prolonged period, while others saturate the network and cause it to be the system bottleneck. The answer is that as long as there is a single bottleneck, this implies that such a VM is only using devices that are utilized at less than 100%. Therefore, according to the knee assumption, this VM is actually receiving all its needs on the other devices, despite its low priority. Had it required more, it could get more (when the device is otherwise idle), and then that device would also become a bottleneck.

### C. Systems with Multiple Bottlenecks

The same principles as described above apply when multiple bottlenecks are present. Consider the following synthetic example:

| | CPU | disk | net |
|---|---|---|---|
| VM1 | 30% | 70% | 10% |
| VM2 | 30% | 70% | 10% |
| VM3 | 90% | 0% | 10% |
| total | 150% | 140% | 30% |

The most overcommitted resource is the CPU, so let's assume it is regarded as the first bottleneck by the system. Assume again that the desired allocations are 50%, 30%, and 20%, respectively. VM1 cannot use its allocated 50%, so it will actually use no more than 30% by the knee assumption. The resulting allocation will be:

| | CPU | disk | net |
|---|---|---|---|
| VM1 | 30% | 70% | 10% |
| VM2 | 30% | 70% | 10% |
| VM3 | 40% | 0% | 4% |
| total | 100% | 140% | 24% |

However, this exposes a second system bottleneck: the disk. In a real system, the over commitment of the disk means that VM1 and VM2 will not really achieve a utilization of 30% of the CPU each, and thus the system will indeed identify the disk as a bottleneck and shift its attention to it. Applying the fair-share allocation to the disk will then yield:

| | CPU | disk | net |
|---|---|---|---|
| VM1 | 19% | 63% | 6% |
| VM2 | 11% | 37% | 4% |
| VM3 | 70% | 0% | 8% |
| total | 100% | 100% | 18% |

Now the allocation on both bottleneck devices reflects the desired allocation ratios. VM3 gets much more than its fair allocation of the CPU by virtue of using cycles that would otherwise remain idle. The same result would be achieved if we considered the disk directly, without taking the detour of trying to control the CPU first.

However, there exist scenarios where the result does depend on which device is considered first. An example is provided by the following set of requirements:

| | CPU | disk | net |
|---|---|---|---|
| VM1 | 20% | 80% | 80% |
| VM2 | 20% | 80% | 0% |
| VM3 | 20% | 0% | 80% |
| total | 60% | 160% | 160% |

If allocations are made according to the disk (assuming the same 50%, 30%, 20% desired allocations as before), the network is exposed as a secondary bottleneck. But VM3 has a low priority, because it does not use the disk which is the "official" bottleneck device. As a result its use of the network is throttled, leading to an allocation of:

| | CPU | disk | net |
|---|---|---|---|
| VM1 | 16% | 63% | 63% |
| VM2 | 9% | 37% | 0% |
| VM3 | 9% | 0% | 37% |
| total | 34% | 100% | 100% |

But if the network is considered first, the relative allocations of VM1 and VM3 are the ones that dictate the results, which end up being:

| | CPU | disk | net |
|---|---|---|---|
| VM1 | 18% | 71% | 71% |
| VM2 | 7% | 29% | 0% |
| VM3 | 7% | 0% | 29% |
| total | 32% | 100% | 100% |

And if the system continuously switches from one bottleneck to the other, it will oscillate between these two allocations. However, it may be argued that this is a reasonable behavior, and that the desired allocation is in fact the average of these two states.

When there are multiple bottleneck devices, and a VM uses all of them, it might seem that its global priority should be determined by the one where it is farthest behind. However, we suggest the opposite: the global priority is set by the *minimal* lag on any of the bottleneck devices. The justification is that when a VM achieves its target allocation on any of the bottleneck devices, it should not be promoted further.

## V. Design of Possible Implementation

In this section we describe the design of a generic RSVT module that can be embedded in an operating system or VMM. Such a module handles one specific resource. The full implementation will include instances of this module for all the different resources, plus a monitoring facility that identifies the system bottleneck and determines which RSVT module to use as the main scheduler.

A given RSVT module handles clients (e.g. processes or virtual machines) that use its resource. The clients are represented by a small proxy data structure within the RSVT module. The reason for this design, other than its generality, is that a process's use of a resource may outlive the process itself — for example, a process may have network packets waiting to be sent when it is killed. The relationship of clients to resources is many to many. There are many processes in the system, several independent resources, and each process may use all the resources.

The RSVT module provides four main functions:

1) Create a new client object to represent a process that wishes to use this resource.
2) A client issues a request to use the resource. For example, in the context of controlling the disk this is an I/O request. In the context of the CPU, this is becoming ready to run.
3) Dispatch: the policy is called to select the client whose request should be satisfied next. This is done using the RSVT algorithm, based on past resource usage relative to each client's priority.
4) Record the cost of satisfying the last request on behalf of a client. This provides the module with the information needed to make the scheduling decisions.

Naturally there are all sorts of details and complications, so the actual API contains more than four functions.

The semantics of resource operation rely on the nature of the resource, with some resources operating on a request basis (network, disk), but some on a time basis (CPU). The main difference is that with request-based devices, dispatch must be followed by a sequence of calls to retrieve the requests of the selected client.

In either case, the queue of pending clients contains only the clients that are actively waiting for the resource at a given time. For example, if the RSVT module is managing a network device, those clients that have network packets waiting to be sent will be placed on the queue of pending clients. When all the pending packets of a client are sent, it is automatically removed from the queue. In case the resource is not request-based, such as a CPU, the queue of pending clients acts as the runnable tasks queue, and the module's user (the kernel) has to explicitly mark clients as *pending*, thereby inserting them to the queue, or *not-pending*, thereby removing them from the queue.

When the resource is free for processing, the dispatch function is called to select the most deserving client from the pending queue. After processing a client, the resource should update the virtual time of the client, based on the amount of processing performed. This is where the resource's usage units are converted to virtual time, by which RSVT operates.

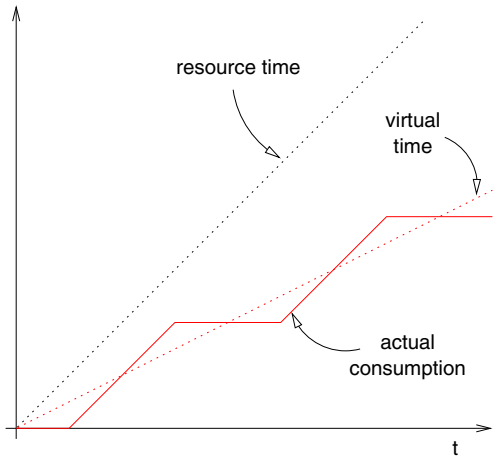


Fig. 4. *The calculation of priority based on virtual time. Priority is reflected by the slope of the virtual time calculation.*

It should be noted that in RSVT virtual time does not necessarily represent time — for example, it could represent volume of data. The RSVT module maintains counters of virtual time for each client. These should be implemented as 64-bit integers, which avoids the problem of counter overflows. For example, even if counting bits transmitted on a 10Gb/s link, such a counter will not overflow for 54 years.

Apart from the virtual time, the data structure representing a client also includes its priority, the time at which it was last dequeued (used for aging), and pointers used to link it into the queue of pending clients. This is a multi-level queue, with a separate queue for each priority level (Fig. 3). The number of priority levels is limited in order to to simplify the dispatch function as described below.

Fig. 4 depicts the calculation of virtual time and relative priorities. The client priorities reflect the desired virtual time rates, as a function of the resource time — a generalization of *wallclock time* defined to be the aggregate of the resource's basic processing units (cycles, bytes, etc.). Thus, at resource time $t$, the target virtual time of client $A$ (whose priority is $P_A$) is $P_A \times t$. But on a shared resource, the actual consumption by client $A$ is intermittent: sometimes it is using the resource, and sometimes it is being used by another client. We denote by $consumed_A(t)$ the total time in which client $A$ was using the resource up to resource time $t$. The client's lag is therefore defined as $P_A \times t - consumed_A(t)$, and the next client in line is:

$$i \in clients \quad s.t. \quad P_i \times t - consumed_i(t) \text{ is maximal}$$

At dispatch time, the RSVT module must retrieve the client with the maximal lag. But maintaining *all* clients in order is not possible because the lag is a function of both the current resource time $t$ and a client's priority $P_i$, hence the ordering of queued clients might change as the resource time advances.

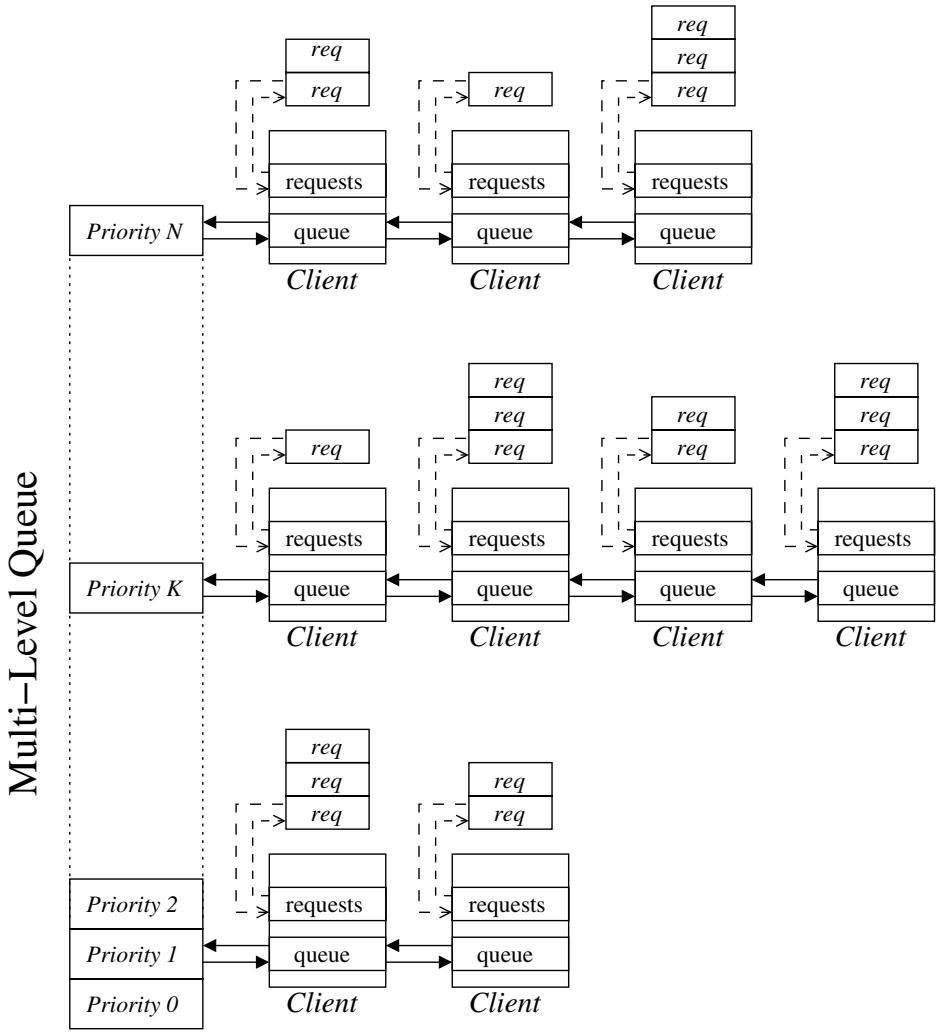


Fig. 3. *An example multi-level queue of pending clients.*

If $P$ is fixed however — i.e. given a set of clients with the same priority — the ordering depends solely on the resource time already consumed by each client.

We therefore maintain each priority's queue sorted in ascending order according to the clients' resource time consumption counters. When a dispatch decision is needed, only the lags of the first client in each queue need be considered. The client with the largest lag is guaranteed to be found in that set. The queues are maintained in-order by repositioning a client within its queue whenever its consumption counter changes (which only happens when it is scheduled). Thus the time to reach a dispatch decision is proportional to the number of different priorities $O(N_{priorities})$, rather than being proportional to the number of clients in the system $O(N_{clients})$. The time to update a client's counter is on average $O(\frac{N_{clients}}{N_{priorities}})$.

## VI. Conclusions

Scheduling research and practice can be envisioned along two axes. One is the objective, which can be either to improve performance (according to some chosen metric) or to enforce a predetermined allocation. The other is the device to which the scheduling is applied, which can be either the CPU or some other bottleneck device.

The above two axes naturally define four combinations. Most research in operating systems deals with scheduling the CPU so as to achieve performance objectives. As noted above, there has also been work on scheduling the CPU to achieve desired allocations, and scheduling other resources (such as the network) to achieve enhanced performance.

Our framework explores the fourth combination: scheduling all the resources so as to achieve a desired allocation. Our basic contribution is to suggest a meaningful definition of what predefined resource allocations mean in a multi-resource context: we claim that at each instant the system bottleneck device should be identified, and that the allocations should be performed on this bottleneck device. These allocations then induce a schedule on the other devices as well, based on the relative priorities on the bottleneck device.

The significance of the developed system lies in its comprehensive approach to the resource allocation problem. Such an

approach is urgently needed by current installations, especially those used for server consolidation. For example, by focusing on the CPU and attempting to control its allocation when it is not in fact the bottleneck one may become subject to unknown interactions among the resources, which lead to actual allocations that are far removed from the intended ones. Our approach solves such problems.

In future work we intend to implement the proposed design. This involves the association of RSVT policy modules with all the resources we want to control, namely the CPU, disks, and network. Next, we need a monitoring facility that will identify the bottleneck device and decide which RSVT module should take precedence over the others.

Given a working system we will perform a set of experiments to characterize its behavior. In particular, we intend to focus on multiple bottleneck scenarios such as those described above, and on changing workload conditions e.g. as a result of the arrival of new clients. In these contexts, we will investigate the setting of system parameters such as the aging rate for historical virtual time consumption.

Independent of this experimentation, we also intend to continue to develop the conceptual framework. In this context, additional work is needed on the following questions:

- Should the CPU be given a special status, because it is the enabler of using all other devices?
- Should non-bottleneck devices also use other specific scheduling considerations in addition or in lieu of the bottleneck RSVT priorities?
- Should memory also be considered as a schedulable resource? The above discussion assumed the system is equipped with enough memory so that VMs do not need to compete over memory. This assumption is needed because memory is a very different type of resource in terms of its interactions with other resources when it comes to controlled allocations. In principle we can also allocate memory using the same proportions, and let each VM get along using paging with what it gets. However, this leads to an *inverse* relationship between memory and I/O usage, contradicting the linear relationship assumption. More work is therefore needed to decide how to cope with this problem.

## REFERENCES


[1] Y. Amir, B. Awerbuch, A. Barak, R. S. Borgstrom, and A. Keren, "*An opportunity cost approach for job assignment in a scalable computing cluster*". *IEEE Trans. Parallel & Distributed Syst.* **11(7)**, pp. 760–768, Jul 2000.

[2] N. Bansal and M. Harchol-Balter, "*Analysis of SRPT scheduling: investigating unfairness*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 279–290, Jun 2001.

[3] A. Chandra, M. Adler, P. Goyal, and P. Shenoy, "*Surplus fair scheduling: a proportional-share CPU scheduling algorithm for symmetric multiprocessors*". In 4th *Symp. Operating Systems Design & Implementation*, pp. 45–58, Oct 2000.

[4] Y. Diao, N. Gandhi, J. L. Hellerstein, S. Parekh, and D. M. Tilbury, "*Using MIMO feedback control to enforce policies for interrelated metrics with application to the Apache web server*". In *Network Operations & Management Symp.*, pp. 219–234, 2002.

[5] K. J. Duda and D. R. Cheriton, "*Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler*". In 17th *Symp. Operating Systems Principles*, pp. 261–276, Dec 1999.

[6] D. H. J. Epema, "*Decay-usage scheduling in multiprocessors*". *ACM Trans. Comput. Syst.* **16(4)**, pp. 367–415, Nov 1998.

[7] Y. Etsion, D. Tsafrir, and D. G. Feitelson, "*Process prioritization using output production: scheduling for multimedia*". *ACM Trans. Multimedia Comput., Commun. & App.* **2(4)**, pp. 318–342, Nov 2006.

[8] R. P. Goldberg, "*Survey of virtual machine research*". *Computer* **7(6)**, pp. 34–45, Jun 1974.

[9] S. Govindan, A. R. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam, "*Xen and co.: communication-aware CPU scheduling for consolidated Xen-based hosting platforms*". In 3rd *Intl. Conf. Virtual Execution Environments*, pp. 126–136, Jun 2007.

[10] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal, "*Size-based scheduling to improve web performance*". *ACM Trans. Comput. Syst.* **21(2)**, pp. 207–233, May 2003.

[11] J. L. Hellerstein, "*Achieving service rate objectives with decay usage scheduling*". *IEEE Trans. Softw. Eng.* **19(8)**, pp. 813–825, Aug 1993.

[12] G. J. Henry, "*The fair share scheduler*". *AT&T Bell Labs Tech. J.* **63(8, part 2)**, pp. 1845–1857, Oct 1984.

[13] J. Kay and P. Lauder, "*A fair share scheduler*". *Comm. ACM* **31(1)**, pp. 44–55, Jan 1988.

[14] B. Lin and P. A. Dinda, "*VSched: mixing batch and interactive virtual machines using periodic real-time scheduling*". In *Supercomputing*, Nov 2005.

[15] B. Lin and P. A. Dinda, "*Towards scheduling virtual machines based on direct user input*". In 2nd *Intl. Workshop Virtualization Technology in Distributed Comput.*, 2006.

[16] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig, "*Intel virtualization technology: hardware support for efficient processor virtualization*". *Intel Tech. J.* **10(3)**, pp. 167–177, Aug 2006.

[17] J. Nieh, C. Vaill, and H. Zhong, "*Virtual-Time Round Robin: an O(1) proportional share scheduler*". In *USENIX Ann. Technical Conf.*, pp. 245–259, Jun 2001.

[18] D. Ongaro, A. L. Cox, and S. Rixner, "*Scheduling I/O in virtual machine monitors*". In 4th *Intl. Conf. Virtual Execution Environments*, pp. 1–10, Mar 2008.

[19] B. Schroeder and M. Harchol-Balter, "*Web servers under overload: how scheduling can help*". *ACM Trans. Internet Technology* **6(1)**, Feb 2006.

[20] A. Shah, "*Kernel-based virtualization with KVM*". *Linux Magazine* **86**, pp. 37–39, Jan 2008.

[21] I. Stoica, H. Abdel-Wahab, and A. Pothen, "*A microeconomic scheduler for parallel computers*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 200–218, Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.

[22] C. A. Waldspurger and W. E. Weihl, "*Lottery scheduling: flexible proportional-share resource management*". In 1st *Symp. Operating Systems Design & Implementation*, pp. 1–11, USENIX, Nov 1994.

[23] T. Yang, T. Liu, E. D. Berger, S. F. Kaplan, and J. E. B. Moss, "*Redline: first class support for interactivity in commodity operating systems*". In 8th *Symp. Operating Systems Design & Implementation*, Dec 2008.