Design and Implementation of a Generic Resource Sharing Virtual Time Dispatcher

Tal Ben-Nun
School of Eng. & Comp. Sci.
The Hebrew University
91904 Jerusalem, Israel
talbn@cs.huji.ac.il

Yoav Etsion
Computer Sciences Dept.
Barcelona Supercomput. Ctr.
08034 Barcelona, Spain
yoav.etsion@bsc.es

Dror G. Feitelson
School of Eng. & Comp. Sci.
The Hebrew University
91904 Jerusalem, Israel
feit@cs.huji.ac.il

Abstract

Virtual machine monitors, especially when used for server consolidation, need to enforce a predefined sharing of resources among the running virtual machines. We propose a new mechanism for doing so that provides improved pacing in the face of heterogeneous allocations and priorities. This mechanism lends from token-bucket metering and from virtual-time scheduling, and prioritizes the different clients based on the divergence between their desired allocations and the actual consumptions. The ideas are demonstrated by implementations for the CPU and networking subsystems of the Linux kernel. Notably, both use exactly the same basic module; future plans include using it for disk I/O as well.

Categories and Subject Descriptors C.2.3 [*COMPUTER-COMMUNICATION NETWORKS*]: Network Operations—Network management; D.4.1 [*OPERATING SYSTEMS*]: Process Management—Scheduling; K.6.2 [*MANAGEMENT OF COMPUTING AND INFORMATION SYSTEMS*]: Installation Management—Pricing and resource allocation

General Terms Design, management, performance

Keywords Virtual machine, fair share, resource allocation

1. Introduction

A hypervisor or virtual machine monitor (VMM) may seem to have similar responsibilities to an operating system: it abstracts the hardware and performs resource management, including scheduling the different virtual machines. However, the context is actually quite different. Operating systems' schedulers typically try to optimize performance objectives, such as response time, based on knowledge about each process's behavior. A hypervisor, on the other hand, is concerned with virtual machines that may each run a mixture of diverse processes that are unknown to the hypervisor. And its goal, especially in server consolidation scenarios, is more typically to enforce a pre-defined allocation of the resources.

Controlling the relative allocation of resources to contending processes (or virtual machines) is not new. Such "fair share" scheduling of the CPU is typically done based on variants of *virtual time*. Allocations of network bandwidth are typically done using variants of *leaky bucket* or *token bucket* approaches (all these are explained in detail below). Our approach combines these techniques into "resource sharing virtual time" scheduling (abbreviated RSVT). It includes metering of the allocation to each process on one hand, and scheduling so as to better pace the utilization of the allocated resources on the other.

The next section motivates the work by explaining the need for supporting predefined allocations, especially in virtualization scenarios. Section 3 then describes previous work and leads up to our extensions, which are delineated in Section 4. This is followed by a description of the implementation in the Linux ker-

nel in Section 5, and by an experimental evaluation in Section 6. Section 7 presents the conclusions.

2. Motivation and Context

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 [12]. 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. 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 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 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. Moreover, this partitioning of the resources can be changed easily to reflect changing conditions.

The virtualization infrastructure is therefore found to assume many of the basic responsibilities of an operating system. 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 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.

We are working on a global scheduling framework that is designed to answer this question. 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 system bottleneck device as the locus where such control should be exercised [9]. The present paper reports our progress in the first component of this work, namely the RSVT scheduler. Importantly, we wish to use the same scheduler to control all the relevant resources, as different resources may become the system bottleneck at different times. We therefore designed and implemented a generic scheduling module, which can be used as the policy component of different resource management frameworks. At this stage, the implementation in the Linux kernel supports CPU and network scheduling. In future work we intend to migrate it to a virtualization environment such as KVM, and add disk scheduling.

3. Related Work

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 scheduling 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 [16, 17]. In Unix systems it has also been suggested to manipulate each process's nice value to achieve the desired effect [8, 15]. Simpler and more direct approaches include lottery scheduling [27] or using an economic model [26], 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 [2, 7, 22]. The idea is that time is simply counted at a different rate for different processes, based on their relative allocations. Our RSVT scheduler falls in this category; it 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 had been used [11]. A similar approach was used in [4].

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 [18, 19]. Controlling the slices and periods allows for adequate performance even when mixing interactive and batch jobs.

Control over allocations of network bandwidth is usually combined with traffic shaping, i.e. the reduction of variability in bandwidth usage. One way to achieve this is the leaky bucket algorithm. The bucket figuratively represents a buffer where packets are stored when the source creates them too quickly and they cannot be transmitted immediately. Thus data flows into the bucket at a variable rate, but flows out at a steady rate. Each sender has its own bucket, where the size of the hole in the bucket represents its bandwidth allocation. Additional packets that arrive when the bucket is already full are called "nonconforming", and are typically discarded.

An alternative is to use a token bucket, which works the other way around: it stores tokens that allow packets to be sent (similar to the economic framework mentioned above). When a packet arrives, it will be sent if a token is available; otherwise it is nonconforming (and thus needs to be queued in a buffer). Tokens are added to the bucket at a steady rate, and removed whenever packets are sent. Thus a source may accumulate tokens when it is idle, and use them at a high rate (higher than their arrival rate) when it needs to transmit. As a result the momentary bandwidth of a source may surpass its average allocation, but only for a limited time.

Leaky and token buckets limit the rate of individual sources, but do not specify how they are multiplexed. Using FCFS with these algorithms may still lead to overload and is subject to manipulations. This was improved by Nagle's "fair queueing" [21], in which the requests of each source are kept in a separate queue, and these queues are served in round robin manner. However, given that packets may have different sizes, this may lead to deviations from the intended bandwidth allocation. Demers et al. therefore suggested an approximation of round-robin at the *byte* level [6]. Conceptually this uses a counter of how many byte-by-byte rounds have elapsed, which serves as a timepiece (it is actually identical to the idea of virtual time mentioned above). Using this, one can find the round at which a packet will finish transmission: it is the sum of its start time and length, where the start time is the max of its arrival time and the end of the previous packet from the same source. As the counter is monotonically increasing, the order of finish times as counted in rounds corresponds to their order in real time. The algorithm then is to calculate these finish times for all sources, and select to transmit the packet with the earliest finish time. It is also possible to provide variable allocations by modifying the count of rounds needed to transmit, leading to "weighted fair queueing".

The main drawback of all the above approaches is that they focus on one resource — either the CPU or the network. 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 [1, 14, 25]. This raises the question of the interaction between devices, e.g. the effect of CPU scheduling on I/O [23], or the prioritization of VMs that do I/O so as not to cause delays and latency problems [13]. But such interactions may naturally interfere with the desired allocations. The work presented here is part of a larger project to achieve a global scheduling scheme based on identifying the bottleneck device at each instant and using it to dictate the allocations [9].

4. The RSVT Scheduler

In general, scheduling combines resource allocation and sequencing. RSVT, introduced in [11][1], tackles this combination. Like other virtual time scheduling schemes it controls allocations by making time pass at a different rate for different clients[2], such that the rate reflects the allocation. The sequencing is done by selecting the client that is most "behind its time" to run next (e.g. [22]). RSVT is especially useful for combining fair shares with pacing. The idea is that rather than just selecting the client with the biggest lag in 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 clients with the same profile [11].

Unlike some other virtual time schemes, RSVT has a concept of allocations. This raises the question of what to do when a client becomes inactive: should its allocation continue to grow? We handle this as follows. First, allocations continue to grow for a certain "grace period" that reflects expected continuity of operation. Then they are frozen. Finally, the relative allocation is reset to zero after a long time that reflects the system's memory bound. This means that clients that have been inactive for a long time are simply treated as if they were new, and their previous history is forgotten.

4.1 Expressing Relative Priorities

There are two ways to express the desired allocations to competing clients: absolute and relative. Using the allocation of network bandwidth as a concrete example, an absolute allocation would be something like "this client should transmit at 20MB/s". Such an allocation is natural when using leaky bucket, token bucket, or economic models. A relative allocation, on the other hand, is more like "this client should transmit at double the rate of that client". This approach is natural with lottery scheduling.

RSVT uses the relative approach. Each client is given a priority, which is expressed as a rate. However, this is not an absolute rate, but rather a relative one. Thus if two clients exist and both have a rate of 1, they will each get half of the bandwidth. If a third client is added also with a rate of 1, each of the three will now get a third of the bandwidth. But if the third client has a rate of 2, it will get half of the bandwidth, and the original two clients will each get a quarter.

The reason for preferring the relative approach is that it facilitates better utilization. With absolute rates, if the total rates of active clients exceed the capacity then they cannot be satisfied, and if they fall bellow the capacity then resources are wasted. Using relative rates solves these problems [5]. Other solutions, such as using a large token bucket and allowing a client to send at a higher rate when there is no competition, lead to reduced control over the allocations and to the danger of extended monopolization of the resource by a single client. Relative rates are also more portable, in the sense that they remain equally relevant if a different network is used.

4.2 Interpretation of Virtual Time

As noted above, accounting for resource usage using virtual time is simpler than using real time [2]. Denote the consumption by client i by c_i, and its allocation rate by r_i (these and other notations are summarized below in Table 1). Then by definition

$$\frac{dc_i}{dv} = r_i \tag{1}$$

where v denotes the virtual time. The relationship between virtual time and real time is based on the available physical rate R that is available (e.g. the bandwidth of the network), and the allocation rates to the set of active clients A:

$$\frac{dv}{dt} = \frac{R}{\sum_{j \in A} r_j} \tag{2}$$

Putting this together, we find that the rate of consumption by a specific client is proportional to its relative allocation:

$$\frac{dc_i}{dt} = \frac{r_i \cdot R}{\sum_{j \in A} r_j} \tag{3}$$

Thus virtual time is indeed a weighted version of real time, where the weight reflects the relative allocation. In other words, different clients are accounted for their resource usage at different rates.

This interpretation of virtual time leads to a very simple scheduling algorithm [7]: dispatch the client with the lowest accounted resource consumption. For

[1] It was originally called PSVT, as it was envisioned in the context of a single resource: the processor.

[2] At the logical level, we will consistently call the entities handled by the RSVT scheduler "clients". These clients can actually be processes, jobs, or virtual machines, depending on context. In the Linux implementation, they are Linux tasks.

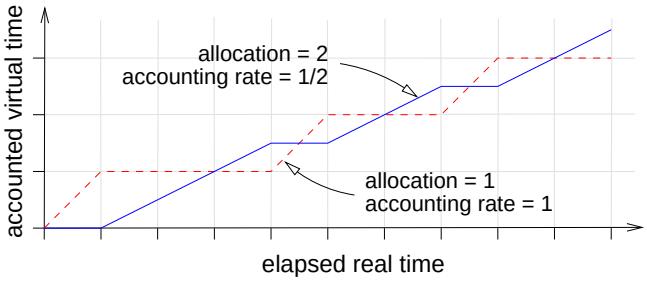


Figure 1. *Virtual time scheduling.*

example, consider two competing clients, one with an allocation $r_1 = 1$ and the other with an allocation $r_2 = 2$. When client 1 runs, it is accounted at full rate. But when client 2 runs, it is accounted at half rate, because its allocation is double. Therefore it will get to run twice as much. This is illustrated in Fig. 1.

The resource rate R is relevant for resources such as networks or disks, where it reflects the amount of data transferred per unit time (that is, the bandwidth). In the case of a CPU it can be taken as 1, since the CPU provides one second of processing for each second of real time. For simplicity, we will consider this situation in the sequel and drop R from the equations. In fact, this can also be done for any resource — it just means we schedule and account for "seconds of resource activity" rather than for "units of resource work done".

4.3 Concept of RSVT

RSVT is a variation of the virtual time approach. Virtual time scheduling algorithms can be viewed as basing the relative priority of each client on the difference between the global real time and its personal virtual time, which reflects an accounting of its consumption. However, given that all clients share the same real time, it is enough to compare their consumptions and select the lowest one.

RSVT likewise compares two values, but they are slightly different: we compare each client's actual consumption with its *ideal* consumption. The actual consumption grows at a steady rate (equal to real time) when the client is running, and stays flat when it is not. The ideal consumption grows steadily at a rate that reflects the client's relative allocation. If the consumption is ahead of the allocation, the client has a low priority and other clients should run. But if a client's consumption lags its allocation, it has a higher priority. In particular, the client for which the consumption lags the allocation by the most is the client with the highest priority, and will be selected to run next, as illustrated in Fig. 2.

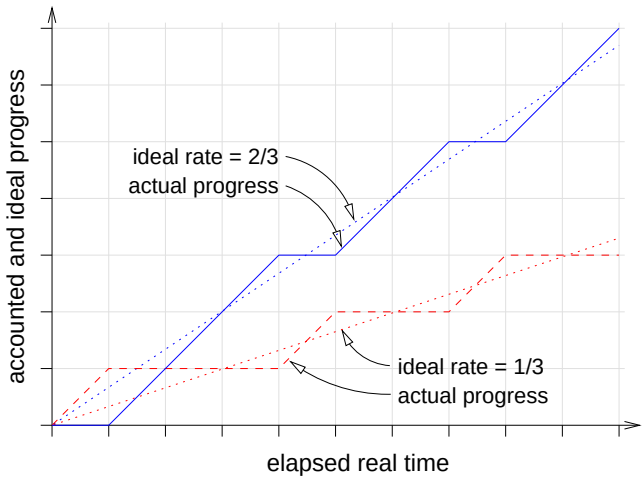


Figure 2. *RSVT scheduling.*

An examination of Figs. 1 and 2 shows that in this case the produced schedule is identical. The difference between the two approaches is only evident when several clients have similar profiles and compete with other clients that have a different one. Let's consider a concrete example where two clients have an allocation $r_1 = r_2 = 1$, and a third has $r_3 = 2$. Virtual time scheduling will settle into a pattern where the two low-allocation clients are always executed one after the other, thus excluding the high-allocation client for 2 time units in a row (top of Fig. 3). This happens because when the two low-allocation clients have a lower virtual time than the high-allocation client, scheduling one of them leaves the other with its low virtual time, so it will be scheduled next (arrows). RSVT, in contradistinction to the above, will spread them out and give the high-allocation client better-paced access to the CPU [11] (bottom of Fig. 3). The reason is that when one low-allocation client is scheduled to run (arrows from below), the ideal allocation of the high-allocation client continues to rise. Therefore by the next scheduling point the high-allocation client has gained an advantage over the second low-allocation client (arrows from above).

4.4 RSVT for a Given Set of Clients

More formally, RSVT works as follows. Assume for the moment that the set of active clients A is fixed — all clients arrived some time t_0 in the past, they do not terminate, and they are constantly active in using the

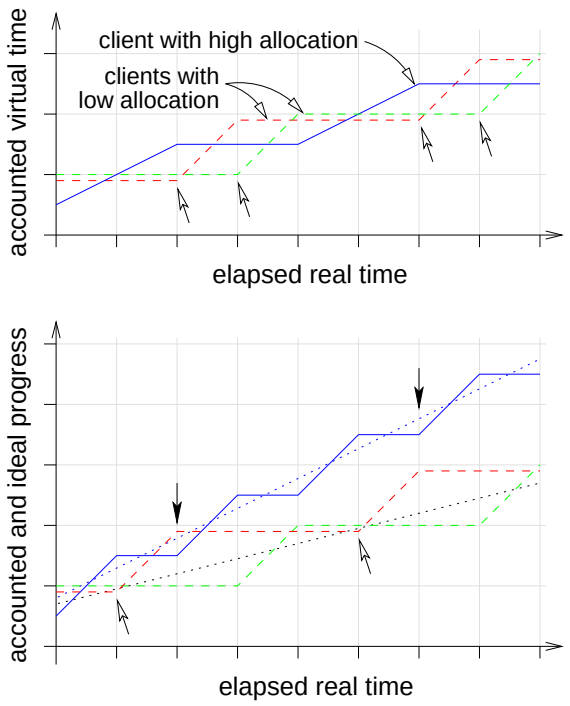


Figure 3. *The advantage of RSVT scheduling in spreading low priority clients that compete with a high-priority client.*

resource. Each client i is characterized by its ideal relative rate r_i. Assuming constant activity, its ideal allocation by time t (where obviously we are only interested in $t \geq t_0$) is then

$$a_i(t) = \frac{r_i}{\sum_{j \in A} r_j} \cdot (t - t_0) \tag{4}$$

which reflects its relative share of the capacity of the resource.

At time t, the client's actual consumption so far will be denoted by $c_i(t)$. Its priority is then simply

$$pri_i(t) = a_i(t) - c_i(t) \tag{5}$$

The scheduler will select the client with the highest priority. Note that priority may be negative, if a client happens to consume more than its fair share at some point in time.

To implement this, we need to be able to calculate $a_i(t)$ and $c_i(t)$ for all active clients. In principle, these values need to be updated upon each scheduling event, to reflect the changes since the last scheduling event. Note that there are two types of scheduling events:

- A client is selected to use the resource

A	set of active clients
r_i	relative rate of client i
a_i	cumulative allocation of client i
c_i	cumulative consumption of client i
T	tick period for allocations and timer events
g	grace period during which allocations continue when a client becomes inactive
m	rebirth period during which allocations are retained after a client becomes inactive
b	maximal allowed difference between allocation and consumption

Table 1. *Parameters used in RSVT.*

- A client stops using the resource, either because it does not need it anymore at this time or because another is selected as having higher priority

If the resource is constantly busy, pairs of such events coincide as one client stops and another is selected. In such a situation the stop events are redundant and can be ignored. But if the resource becomes idle, the stop event is important to note.

Assume that the time now is t, and that the time of the previous event was t_p. If the resource was busy during this period, assume that client i has just finished using it. As all the usage was consumed by client i, we update

$$c_i(t) = c_i(t_p) + (t - t_p) \tag{6}$$

All the other c values remain unchanged, as other clients did not consume any of the resource in this interval. All cs remain unchanged also if the resource has been idle in this period; in this case the resources capacity during the period from t to t_p was wasted.

However, the ideal allocations of *all* the clients have grown by their respective shares. Therefore we should compute

$$a_\ell(t) = a_\ell(t_p) + \frac{r_\ell}{\sum_{j \in A} r_j} (t - t_p) \tag{7}$$

for all $\ell \in A$ (including client i).

Note that in both the above equations the total increment is identical, and stands at $t - t_p$. This seems to imply a nice invariant: $\sum a_i = \sum c_i$. However, this is hard to maintain when clients have more dynamic behavior instead of being able to use the full available resources all the time.

4.5 Handling a Changing Set of Clients

Note that the above equations assume that the set of active clients A does not change with time. In a real system, of course, new clients may arrive while others depart. In addition, clients may become inactive for extended periods, i.e. they may refrain from using the resource which we are scheduling (for example, when a client is blocked doing I/O it is not contending for the CPU). This affects the above equations because the relative allocations depend on $\sum_{j \in A} r_j$. Another question is whether to retain the client's historical consumption data for when it will become active again.

One possible approach to handling this issue is to divide the timeline into segments with persistent active sets. Thus a new segment starts whenever any of the following events occurs:

- A new client arrives
- A client terminates
- An active client becomes inactive
- An inactive client becomes active again

We could then do the calculations in each such segment individually, in a way that reflects the changing conditions.

The problem with this approach is that changes may be very frequent, and it is not clear that trying to follow all their details is beneficial. In particular, a thorny issue is how to handle clients that temporarily become inactive. One option is to freeze their allocations, to avoid situations where a client gains a huge credit by virtue of not using the resource, and then starves all other clients once it starts to use it again. But this violates the notion of an ideal allocation in those cases where start/stop activity is natural, e.g. when sending network packets or performing disk I/O. For example, if a client should get an allocation of 1/3 of the bandwidth, we don't want this to be reduced due to processing that occurs between send operations.

The suggested solution is to define a grace period g which defines a time frame that reflects the natural continuity in using the resource — typically on sub-second time scales (with a 2 GHz CPU, even a mere 1 ms corresponds to 2 million cycles; with a 100 Mb/s network, it corresponds to sending 100 Kb). When a client becomes inactive, its ideal allocation will continue to grow during this grace period. After the grace period it will be frozen, to avoid over-allocations as described above.

The grace period is measured in ticks of length T from a timer. These ticks are also used to "smooth" the fluctuations in the active set in general. This is achieved by performing periodic allocations once every T time units, instead of piecemeal allocations as described by Eq. (7) above. Client arrivals and departures are also synchronized with these ticks.

To summarize, allocations are handled by a periodic timer that ticks every T time units. Handling this timer tick involves the following actions:

1. Clean up after clients who have terminated since the last tick.
2. Note clients that have become active or inactive.
3. Make new allocations for all the active clients.

Allocations are made in advance: the allocation performed on a tick reflects expected usage from this time till the next tick. However, it may happen that not all the allocated time will be used. In order to prevent the allocations from outgrowing the consumptions, it is necessary to bound their growth. Thus after calculating the allocation according to Eq. 4, we perform

$$a_i(t) = \min(\, a_i(t),\ c_i(t) + b\,) \tag{8}$$

New allocations are only given to active clients, or those in their grace period. Inactive clients retain their a_i and c_i values for a certain time (the "rebirth" period, m), but then we set $a_i = c_i$. This is done for both positive and negative relative usage. On the positive side, it avoids situations where a client may monopolize the resource based on a credit gained long ago. 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. Thus a client returning to activity after a long period of not being active will be effectively treated like a new client. In the future we plan to decay a_i exponentially towards c_i, rather than using an abrupt change.

Clients may also be added to the active set. New clients are initialized with $a_i(t_i) = c_i(t_i) = 0$. Note that this is before the new allocation, which they will receive together with all other active clients. Clients that have become active again since the last tick after a period of inactivity are also added to the active set.

5. Implementation in the Linux Kernel

RSVT is a generic proportional share scheduler. Thus a system may have many instances of RSVT controlling different resources. This is managed by the global RSVT manager.

The implementation of RSVT was started in the context of the QoS facility of the Linux networking subsystem. This facility provides a hook that can be used to select the next packet that will be sent from the queue of waiting packets. We use RSVT as the policy to guide this selection, and select the first queued packet belonging to the socket whose transmissions so far are the farthest behind what they should have been.

Rather than create a policy function that is specifically tied to the networking QoS facility, we build on the fact that the interface separates the mechanism from the policy to create a generic implementation that can be used as the policy guide for other subsystems as well. This boils down to the definition of an interface whereby the policy receives the information it requires in order to render its decisions. We then indeed used this policy module for CPU scheduling as well, and usage for I/O scheduling is planned as future work.

5.1 The RSVT Module

The RSVT module is a Linux kernel module that can provide RSVT scheduling of resources such as the CPU, disks, and networks. The scheduling algorithm is intended to provide predetermined shares of the resources to the different tasks (the Linux term for processes), which potentially embody different virtual machines.

A kernel running RSVT has one global RSVT manager and one or more instances of RSVT schedulers — one for each device. The global manager is responsible for two main functions. First, it maintains a repository of active RSVT instances; new RSVT schedulers are created (typically upon startup) using the rsvt_create function, and should be removed before shutdown using the rsvt_destroy function. Second, it activates the periodic resource allocation on all such instances, as will be described below. This is done by calling the allocate function of each instance.

Tasks are regarded as the clients of a resource. Therefore, a task is allocated a proxy client object for each resource it uses. Tasks are also the basic entities whose priority can be set, and the clients representing a task operate with the task's priority. The relationship of tasks to resources is many to many. There are many tasks in the system, several independent resources, and each task may use all the resources.

The semantics of resource operation depend on the nature of the resource, with some resources operating on a request basis (network, disk), but some on a time basis (CPU). The design of the RSVT module can account for both. It is based on a core that maintains a queue of pending clients, and two wrappers: one for direct access, and the other for request-based access (called RRSVT). The difference is that RRSVT is based on requests. Therefore when the dispatch function selects a client, it returns an abstract handle to a struct list_head, which contains all that client's requests. In addition to that, clients can enqueue and requeue requests, as demonstrated in our network implementation.

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, thereby removing them from the queue. This is done by the functions client_set_pending and client_unset_pending.

When the resource is free for processing, the dispatch function is called to select the most deserving client from the pending queue (using the RSVT algorithm, based on past resource usage relative to each client's allocation). A resource that is not request-based can then start operating on the selected client. A request-based resource will simply pull the selected client's requests one by one. After processing a client, the resource should update the resource usage of the client, based on the amount of processing performed. This provides the RSVT module with the information needed to make future scheduling decisions. As noted above, this is simply done using the time that the resource was used. The function used to note resource usage is called checkin.

5.2 Data Structures

The RSVT module completely separates the object representation of a task from that representing a client. Rather than storing each client's information as part of the corresponding task's task_struct object, it is stored as a client object that is part of the corresponding resource's RSVT management module. Client objects are created with client_create, and destroyed with client_destroy.

The reason for this design, other than its generality, is that a task's use of a resource may outlive the task — for example, a task may have network packets waiting to be sent when it is killed. Therefore, a task's proxy clients for resources must be regarded as different entities than the task itself. Moreover, this design enables future work on exchanging information among independent instances of RSVT, and support for flexibility regarding placement of virtual machines based on usage of *all* resources.

To access the client objects, a task's task_struct will contain an array of handles to the client objects for different resources. The array size limits the number of resources that can be accommodated. In the other direction, each client object contains a pointer to the corresponding task_struct. If the task has been killed this pointer is set to NULL, and when all requests have been serviced the client object is deleted.

To simplify the implementation (and especially the search for the next client to serve) only a discrete set of different rates (representing priorities) is supported. When the RSVT module is initialized, the number of distinct rates and their specific values should be specified (these are arguments to rsvt_create). These are stored in an array that maps each priority level to its associated rate.

The lowest priority is by convention set to 1. The highest priority should be set according to the desired maximal ratio of rates: should be highest rate by double the lowest rate? or 5 times higher? or 10 times? Additional intermediate levels should be set according to the desired ratios that should be supported. This would typically imply a logarithmic scale, as in 1, 2, 4, 8. As priorities should be integral numbers, it is possible to deviate from the convention of starting at 1 to accommodate smaller ratios. For example, to achieve an approximate ratio of $\sqrt{2}$ between adjacent priorities one can use 10, 14, 20, 28, 40, 57, 80.

Clients may be linked to each other in two ways. First, there is a linked list of *all* clients. This is used when all clients must be traversed, e.g. when making new allocations. In addition, active clients are linked according to their priority. This creates a multi-queue structure, with a separate queue for each discrete priority level.

5.3 Operation

In principle each resource has its own units: CPU usage is measured in cycles, network transmission is measured in bytes, and disk I/O is measured in blocks. Using these units requires us to calibrate the RSVT module so that it knows how many units to allocate or charge for a period of time. However, this is redundant if all allocations and consumptions are simply measured in time. We therefore use time as our basic unit for *all* resources, regardless of their nature. Specifically, our basic unit in the initial implementation is one microsecond. With 32 bits this limits us to usage of just over one hour. In a "real" implementation one would therefore need to use 64 bits.

Note that in some resources, notably disk and network, it may be impossible to measure the time needed to serve a specific request. The problem is that the lower-level devices may handle multiple requests concurrently and transparently. The solution is to map the request size to time using the known nominal resource rate (e.g. effective bandwidth; for example, in a network setting this would account for overhead due to physical layer headers). This is done by the "glue code" and is external to RSVT itself. In the network implementation the resource rate is obtained from the configuration information. The implication of this decision is that the rate of resource usage is constant at all times. In modern systems this is not necessarily the case, especially with regards to the CPU clock rate which may be adjusted based on power and heating considerations. We currently ignore such difficulties.

The RSVT module does not generally manage *all* of a resource's users, but only a subset — for example those tasks executing a virtual machine. Other tasks — for example system daemons — should be managed by another scheduling algorithm (probably a FIFO queue). Such tasks should have priority over the RSVT clients. For example, in our implementation for networking, traffic generated by NFS clients falls in this category.

In CPU scheduling, this may refer to some real-time tasks.

However, we do need to track the times when the resource is being used by these other tasks that are outside our control, and reduce the allocations to our clients accordingly. In general, a resource's time can be divided into three types of use:

- It is being used by RSVT's clients based on its scheduling decisions.
- It is idle.
- It is being used by some other tasks we don't know about (so called "dead time").

The RSVT scheduler should be cognizant of the first two, and use them for the purpose of calculating allocations. It should ignore the third, as if it were a gap in the timeline that does not exist. To do so, it must know when other tasks take over the resource and when they release it again.

5.3.1 Ticks and Allocations

The original concept of RSVT is based on resource sharing, as if all clients consume their allowance of the resource continuously. In reality, of course, they are multiplexed using time slicing. Allocations are likewise done at discrete instances. That is why the experimental results below show stepwise progress rather than slopes (Fig. 4 as opposed to Fig. 2).

Allocations are done periodically based on the system's clock interrupt (the same one that increments jiffies[3]). At each tick, an allocation reflecting the time since the previous allocation is made, after subtracting known dead time (that is, time when the resource was used by entities outside our control). As part of this, the time in jiffies is translated into microseconds. The total allocation is divided among all *active* clients according to their relative priorities. To make this easier, an accumulator with the total rates of all active clients is maintained at all times; it is updated when clients arrive, terminate, or change state (become active/inactive).

The total allocation reflects the total capacity available to our clients after deducting dead time. But clients may not use their whole allocation, e.g. because they pace their activity based on wallclock time. To avoid waste of resources, RSVT allows negative priorities, as may happen if a client's consumption is ahead of its allocation. In fact, it may happen that one client's allocation grows significantly more than its consumption, while another's is significantly behind its consumption. This is undesirable as it may have detrimental effects on future usage. Specifically, if the allocation is way ahead the client has a high priority and may monopolize the resource for an extended time. If it is behind the client may be blocked out for an extended time.

Our implementation solves this problem by bounding the divergence between the allocation and the consumption. This is always done by manipulating allocations, as the consumption reflects real usage by the client. Thus if a client does not use its full allocation, future allocations are bounded and will not grow too much beyond the client's consumption. Conversely, if a client's consumption grows more than its allocation, the missing allocation is made up so as to prevent too much lag. In both cases, the bound is set equal to the grace period in the initial implementation ($b = g$).

5.3.2 Dispatch and Check-in

Dispatch decisions require finding the client with the largest difference between its allocation and consumption. The problem is that allocations change at different rates for different clients. To reduce overhead we do not want to scan all the active clients each time. Therefore clients are kept in separate queues according to their priorities (rates). Each queue is sorted such that the client with the largest difference is first; as all the clients in the queue have the same rate, they cannot overtake each other [9]. Thus we just need to compare the first client in each queue.

After a client is selected its resource usage must be noted. In the networking implementation this can be done in advance. When a packet is sent, the length of the packet is translated into a time based on the network's bandwidth. This is then accounted to the clients consumption. In the CPU implementation the accounting is done when the client is de-scheduled, based on reading the processor's cycle counter, because the length of time it will run cannot be known in advance [10].

5.3.3 Handling Grace and Rebirth

Allocations should in principle be made at the finest granularity possible, and as a compromise we do them at tick granularity. But we also need two coarser timers, in order to implement the grace period and rebirth. Both of these also piggyback on the ticks used for allocation.

[3] On a 250Hz system this is $T = 4$ms.

To implement the grace period, each client has a timestamp of when it was last active. As part of handling a tick we first scan all clients to check the difference between their timestamps and the current time. A client that has been inactive for longer than the grace period then becomes inactive. The grace period in the current implementation is set to $g = 20$ ms for the network and $g = 600$ ms for the CPU. This reflects the fact that the network operates at a much finer resolution — sending of individual packets vs. scheduling time quanta.

In addition, the ticks are also used for rebirth. Rebirth works using a second chance approach. It is activated periodically, with a period that is 60 times the grace period. Each client has an activity flag, which is set whenever it performs some activity. Upon invocation of a rebirth process, all clients are checked. Those whose flag is set are retained, but the flag is reset. Those whose flags are unset have not been active since the previous check, so they have been inactive for at least a full rebirth period. Their allocation is then set to equal their consumption. When they become active again they will then be treated as if they were new clients (thus the name "rebirth").

5.4 Networking Glue Code

The generic RSVT module is responsible for the scheduling *policy* only. It exists in parallel to the mechanisms used to actually manage the resource itself. The glue code connects the policy and the mechanism.

The Linux kernel has long had a network Quality of Service (QoS) module in it, located in `net/sched` [24]. It consists of a main control and various alternative policies (called "queueing disciplines"). It is relatively easy to add new policies to the QoS module, simply by adding a file and implementing the basic functions (enqueue packet, dequeue packet, and so forth).

Our "glue-code" constitutes such a QoS policy. It implements queueing packets as client requests to RSVT. It implements de-queueing by using the RSVT `dispatch` function to find the client with the highest priority, then obtaining this client's next request, and finally checking-in with the approximate packet sending time in microseconds. Note that this is done on a packet basis. Given that the QoS module is very low level, below the TCP/IP implementation, packets do not necessarily correspond to transmissions. Long transmissions may be fragmented into multiple packets due to MTU considerations, and all these packets have to be handled consistently.

Importantly, the glue code also needs to distinguish between packets belonging to RSVT clients and packets coming from other sources. The above procedure is applied only for RSVT packets. Other packets are sent immediately, thus implementing FIFO scheduling and a higher priority for them. This includes NFS traffic. The glue code keeps track of such packets, in order to account for the dead time that should not be reflected in allocations to RSVT clients.

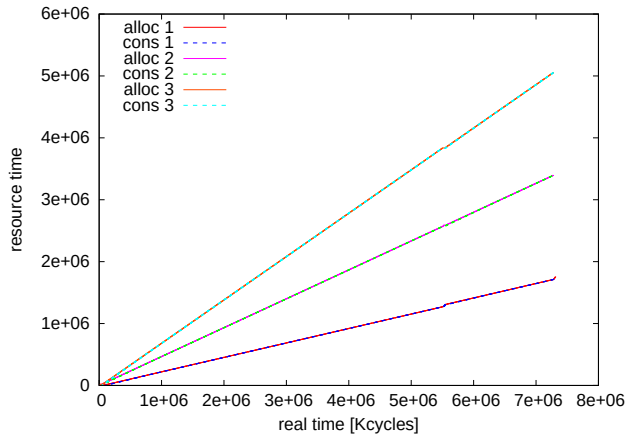
5.5 CPU Glue Code

A modular scheduling policy manager already exists in the Linux kernel (the Modular Scheduler Core [3]), containing different time-sharing techniques for tasks. For example, the Completely Fair Scheduling (CFS) policy implements fair-sharing of the CPU among all tasks [20, Sect. 2.6]. One useful feature of the modular core is custom run "queues" for each scheduling policy.

For our implementation, we created our own scheduling policy, which is essentially a wrapper for the non-request based RSVT module. The run queue in the policy is actually the RSVT object, and queueing/dequeueing tasks to the run queue causes their corresponding clients to be set as pending or not pending accordingly. "Taking out" the next task by the scheduler is implemented as dispatching a client, and "putting" the task back to the run queue is equivalent to checking in the RSVT client.

The RSVT policy is lower than the CFS policy (which is the default) in the scheduler class hierarchy, in order to allow hypervisor tasks to act as soon as possible, if necessary. It is higher than the idle class, which includes the system idle loop. By hooking into the dispatch loop, which searches for the highest priority runnable task, we keep track of time used by real-time and CFS tasks. From RSVT's perspective this is dead time that should not be allocated.

The initial implementation of RSVT CPU scheduling assumes a single processor, and is therefore unsuitable for SMP machines. This is a relatively minor technical issue concerned with keeping track of the currently scheduled client. To support SMP machines all that is needed is to allow multiple "current" clients. Also, care must be taken when calculating allocations. If there are less clients than processors, they should each get an allocation that is equivalent to 100% of one



rate	*BW*
1	15.69±0.11
2	30.81±0.03
3	46.10±0.03

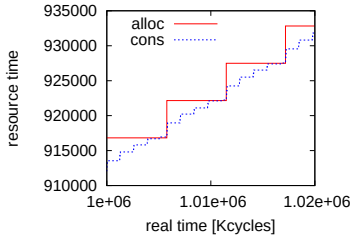


Figure 4. *relative allocations and consumption of competing netperf clients.*

processor (possibly minus dead time as appropriate). If there are more, the sum of the capacities of the processors should be divided among the clients, based on the assumption that capacity can be allocated flexibly in this case. This has the drawback of losing processor affinity characteristics; we leave the possible integration of affinity to future work.

6. Experimental Results

Using our implementation, we ran a number of experiments to demonstrate the capabilities and properties of RSVT. For network allocations, the main application we use is netperf (www.netperf.org), which is a benchmarking tool that continuously sends packets. For the CPU we use synthetic CPU-bound processes and applications like MPlayer which interleave bursts of CPU activity with periods of inaction. This allows us to demonstrate the effect of applications that do not use their full allocations.

6.1 Basic Allocations

Fig. 4 shows the results for a simple case of relative allocations with three competing netperf clients with relative priorities of 1, 2, and 3. In this and subsequent graphs the X axis is real time, and the lines show the growth of both allocations and consumptions with time for all the clients. Hence the slope of a line reflects the rate (and priority) of the corresponding client. The lines showing consumption are on top of those showing the allocation, because in this simple scenario the consumption closely tracks the allocation. The smaller graph shows a zoom into one of these lines. At this fine detail one can see the periodic allocations (big steps of allocation line) and the sending of individual packets (smaller steps of consumption line).

The actual bandwidths achieved by the competing clients are shown in the table. These are based on 10 repeated runs of 10 seconds each. The results obviously closely reflect the desired relative allocations, with extremely small variability. During our measurements we saw only a small number of results that deviated from the specified allocations. These were found to be due to variation in startup time. When one client starts before the other, it gets the full bandwidth during this time, leading to a higher average. Likewise, the lagging client gets the full bandwidth after the first one terminates, also leading to a higher average. This happens because they each run for 10 seconds.

6.2 Dynamic Scenarios

Fig. 5 shows more complicated scenarios, in which the set of clients changes with time. In the first there are initially two netperf clients, with relative priorities 1 and 3. After some time (about 2.3 million Kcycles) a third client is added, with relative priority 2. As a result the allocations of the first two clients are reduced in a way that all three receive their appropriate shares. Then, at about 4.7 million Kcycles, the first client terminates. The allocations of the remaining two are then increased, but the change is very small because the terminated client was the one with the smallest allocation. The second graph shows two competing MPlayers with relative allocations of 1 and 4 of the CPU. They start together, and when the high-priority one terminates, the other picks up the slack.

Fig. 6 shows the effect of the grace period. The scenario is two competing netperf clients, with relative priorities of 1 and 2. At two points the client with the higher priority sleeps for some time, simulating a situation that a client interleaves sending of network traffic with other non-networking activities. The first occurs at about 1.4 million Kcycles. This sleep is shorter than the grace period[4] so the allocation continues to grow.

[4] For this experiment we used an extended grace period of 60 ms.

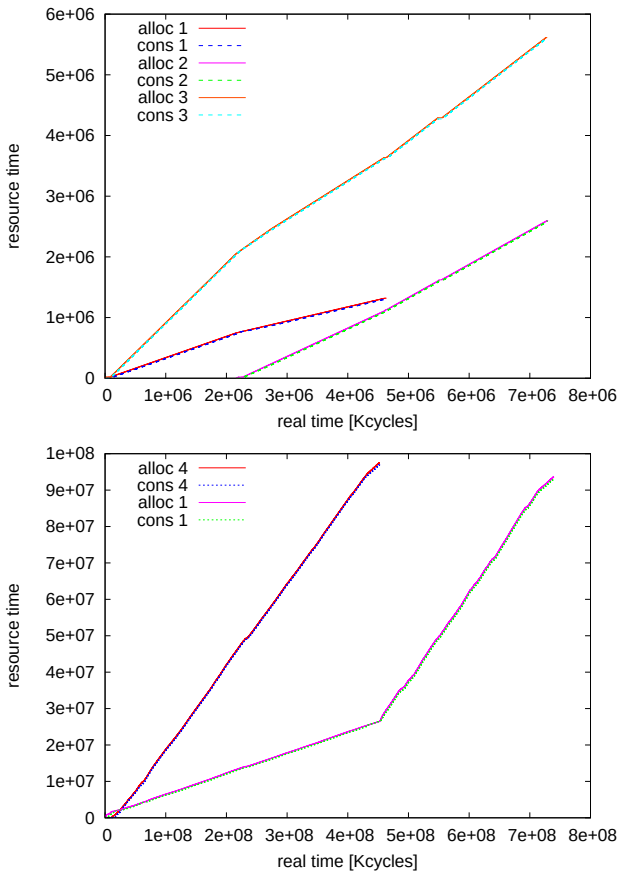


Figure 5. *Allocations change dynamically as clients are added and removed.*

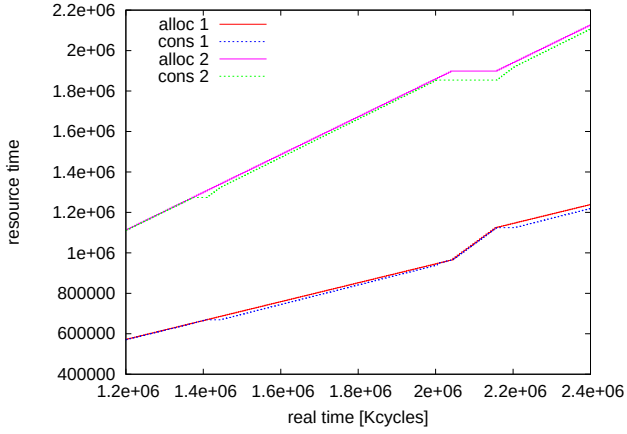


Figure 6. *Effect of the grace period.*

Thus, when this client resumes its activity, it has a relatively high priority and blocks out the other client for a short time. The second sleep occurs at about 2.1 million Kcycles. This one is much longer, so after the grace period ends the client is recognized as inactive, and it

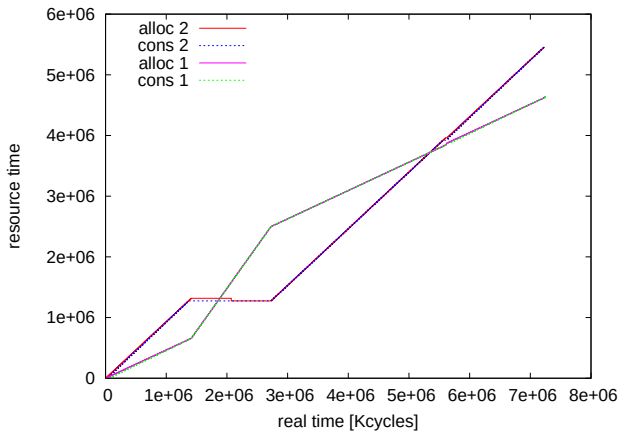


Figure 7. *Effect of the rebirth mechanism.*

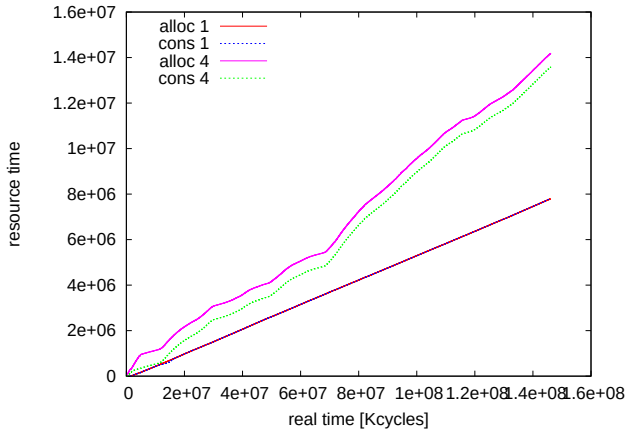


Figure 8. *Effect of self-throttling by MPlayer.*

receives no more allocations. Therefore the full bandwidth allocation is given to the other client, which uses it to increase its transmission rate. Then, when the high-priority client wakes up and resumes sending, its allocations are renewed, and the other client drops down again to its reduced allocation.

If a client refrains from action for a longer time, the rebirth mechanism kicks in. This is demonstrated in Fig. 7 The scenario is the same as in the previous case, except that when the high priority client remains inactive for too long, its extra allocation is removed (at around 2.2 million Kcycles).

Another interesting effect, using MPlayer and CPU RSVT scheduling, is shown in Fig. 8. Two MPlayer clients are decoding videos, with relative priorities of 1 and 4 controlled by RSVT. In addition an X server is displaying the resulting frames, running under CFS and thus with higher priority than the MPlayers (this saves the need to propagate usage information as in [11]). Ini-

tially (bottom left corner of the graph) the allocations indeed reflect the relative priorities. But the consumptions are similar, because it turns out that MPlayer does not really need so much CPU power in order to decode such a video. After a short time RSVT therefore starts to curb the extra allocation for the high-priority client, and it just follows the consumption. The low-priority client gets its fair allocation of CPU capacity, which is somewhat less than it needs, and indeed it also prints out a warning that the system is to slow to show the requested video at its full rate.

7. Conclusions

RSVT is a flexible proportional share scheduler. It provides adjustable allocations among competing clients, and improved pacing of allocations to high-priority clients. The basic implementation is generic, and can be used as the policy module for any schedulable resource. This is done using "glue" code that interfaces RSVT to the desired subsystem. We presented our initial implementation, which includes glue code for the networking QoS mechanism to controls network transmissions, and glue code for the modular scheduler core to control CPU scheduling. In future work we plan to also implement control of disk I/O by integrating with the Linux I/O scheduling framework.

The current implementation, while providing a proof of concept, is not complete. In particular, it is lacking support for groups of processes and for the inheritance or partitioning of both allocations and consumptions across process forks. Implementing this will obviously increase the usefulness of the system, but does not contribute much at the conceptual level.

Another interesting avenue for future research is to extend RSVT by combining relative allocations with absolute allocations. For example, it might be the case that one client should receive half the allocation of another, but not less than 20Mb/s. This can be implemented by verifying that the allocations satisfy the specification, but requires the addition of admission controls to ensure that the specified rates do not surpass the available capacity.

Finally, the whole RSVT development is part of a vision of a global scheduling framework that controls relative allocations of different resources in a coordinated manner [9]. This is to be integrated with the KVM kernel virtualization module. Realizing this vision is a major part of our future work in the coming years.

Acknowledgments

This research was supported by the Israel Science Foundation (grant no. 28/09), and by an IBM faculty award. Many thanks to Edi Shmueli for useful discussions and comments.

References

[1] 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.

[2] A. Bavier and L. Peterson, "*The power of virtual time for multimedia scheduling*". In 10th *Network & Operating Syst. Support for Digital Audio & Video*, Jun 2000.

[3] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*. O'Reilly, 2001.

[4] 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.

[5] Y. Chen, C. Qiao, M. Hamdi, and D. H. K. Tsang, "*Proportional differentiation: A scalable QoS approach*". *IEEE Comm. Mag.* **41(6)**, pp. 52–58, Jun 2003.

[6] A. Demers, S. Keshav, and S. Shenker, "*Analysis and simulation of a fair queueing algorithm*". In *ACM SIGCOMM Conf.*, pp. 1–12, Sep 1989.

[7] 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.

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

[9] Y. Etsion, T. Ben-Nun, and D. G. Feitelson, "*A global scheduling framework for virtualization environments*". In 5th *Intl. Workshop System Management Techniques, Processes, and services*, May 2009.

[10] Y. Etsion, D. Tsafrir, and D. G. Feitelson, "*Effects of clock resolution on the scheduling of interactive and soft real-time processes*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 172–183, Jun 2003.

[11] 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.

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

[13] 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.

[14] 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.

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

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

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

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

[19] 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.

[20] W. Mauerer, *Professional Linux Kernel Architecture*. Wiley Publishing Inc., 2008.

[21] J. B. Nagle, "*On packet switches with infinite storage*". *IEEE Trans. Commun.* **COM-35(4)**, pp. 435–438, Apr 1987.

[22] 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.

[23] 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.

[24] S. Radhakrishnan, "*Linux - advanced networking overview*". URL http://qos.ittc.ku.edu/howto/howto.html, 1999.

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

[26] 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.

[27] 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.