Policies and Metrics for Fair Resource Sharing

Åge Kvalnes
University of Tromsø
aage@cs.uit.no

Dmitrii Zagorodnov
University of Tromsø
dmitrii@cs.uit.no

Dag Johansen
University of Tromsø
dag@cs.uit.no

Robbert van Renesse
Cornell University
rvr@cs.cornell.edu

Abstract

Performance isolation is essential to operating systems shared by dependable services. Unfortunately, most such systems, including real-time operating systems and VMMs, only fairly divide and account for CPU cycles. We submit that dependable services require specifying and enforcing policies for all resources, and that current metrics for evaluating fair sharing are insufficient. This paper proposes new policy specifications and metrics, and illustrates these with the help of a new operating system that supports holistic resource sharing.

1 Introduction

Performance isolation is essential in shared systems used by dependable services. Given an accurate estimate of the maximum resource requirements of a particular service – the amount of CPU, memory, network bandwidth, etc., that it needs for adequate performance under the expected usage scenarios – one can provision it by deploying the service on hardware that matches those requirements. When multiple services share hardware, however, underprovisioning of one can degrade the performance of others, unless the operating system enforces fair sharing of all resources.

There is much evidence that contemporary operating systems fail to enforce fair sharing of all resources. While fair sharing of application-level CPU cycles, disk bandwidth, or network bandwidth have been achieved in isolation, guaranteeing to an application some fraction of all resources, including kernel-level CPU cycles needed for various types of I/O operations, is an unsolved problem. As a consequence, services with strict performance requirements tend to be deployed on isolated hardware, leaving it generally underutilized; other services are increasingly co-located, but without any performance guarantees.

This paper argues that fair sharing of resources, which we call *performance isolation* of services from each other, is a dependability issue that is critical, particularly in the light of unexpected load spikes that network services experience nowadays. To the future discussions of this issue, this paper contributes the following: a model for specifying resource sharing policies that can range from strict partitioning to proportional sharing; two metrics for quantitatively evaluating the fairness of sharing under the chosen policy on a particular system; examples of applying these metrics to evaluate a real system.

The next section, which constitutes the bulk of the paper, is followed by the discussion of related work in Section 3 and the conclusion in Section 4.

2 Performance isolation

A system with performance isolation is characterized by the inability of one service to cause the performance of another co-located service to degrade below a certain level.

2.1 Sharing Policy

The policy for such a system to enforce is two fold: the specification of resource shares that are guaranteed to each service, which we call *minimal guarantee policy*, and the specification of how spare capacity is shared among the services using a resource, which we call *spare capacity policy*. Spare capacity may arise when the minimal guarantees for all services do not add up to full capacity of a resource or when some services do not use their guaranteed share.

Enforcing the minimal guarantees prevents uncontrollable performance degradation because each service receives at least their guaranteed share of each resource. Allocating spare capacity allows the system to utilize resources better and it allows the services to achieve performance above the guaranteed minimum. However, since spare capacity varies with demand, services using it may experience fluctuations in performance.

Through the two policies, the system operator can achieve the desired tradeoff between better system utilization and more consistent performance for a service. At one extreme, spare capacity can be left unallocated for more consistent response times; on the other extreme, all spare capacity can be shared among services.

2.2 Ideal Share

Given a performance isolation policy and a pattern of resource requests from services, it is possible to compute the *ideal share* of a resource that a service should be receiving at any time. We define the ideal share as the minimum of the demanded share and the share that the service is entitled to. The latter is the sum of its guaranteed share, as prescribed by the minimal guarantee policy, and any spare capacity that is available to it under the current request load, as prescribed by the spare capacity policy.

Consider the following **example**: three services, each able to use 100% of the resource when running alone and guaranteed 50%, 33%, and 17%, respectively; spare capacity is allocated in proportion to these guaranteed shares. For any service running alone, the ideal share is 100%, while the ideal shares for other services are zero. If the 50% service and the 33% service run together, their ideal shares are 60% and 40%, respectively. (83% is reserved for their combined minimums and the remaining 17% is split into 50/83 and 33/83.) If the 33% service and the 17% service run together, their ideal shares are 66% and 34%. With all three services running, the ideal share for each is its guaranteed share, since there is no spare capacity left. If the policy were to *not* allocate spare capacity, then the ideal share for any running service would also be its guarantee.

2.3 Accuracy

To evaluate experimentally how well a system enforces performance isolation, we propose to compare the *obtained shares* of a resource received by the services during a run to their ideal shares under those conditions. The difference between the two values is the *error*. Since some resources, such as CPU and I/O bandwidth, are timeshared by services, comparison of obtained and ideal shares of such resources is only meaningful over a time *interval*.

Consider a timeshared resource and an interval during which the ideal shares are fixed. With an interval shorter than the scheduling timeslice, the error will be large, since at any point in time one service has 100% of the resource while the ideal share is smaller. However, as longer intervals are chosen, spanning enough time to give each resource at least one timeslice, the error will decrease.

Formally, given an interval t (of duration $|t|$) during a run, the error $E_{SR}[t]$ during that interval for each service S and resource R is the difference between the obtained share $O_{SR}[t]$ and the ideal share $I_{SR}[t]$, expressed as the percentage of the ideal one:

$$E_{SR}[t] = \frac{O_{SR}[t] - I_{SR}[t]}{I_{SR}[t]} \times 100\%.$$

The units in which the shares are expressed are not important: they can be percentages of maximum capacity of a resource, such as percent of CPU time, or they can be resource-specific units, such as bytes, bytes per second, etc. Given $E_{SR}[t]$ for every interval during a run T (of duration $|T|$), it is possible to compute the average per-service error for a resource as

$$E_{SR} = \frac{\sum_{t \in T} |E_{SR}[t]|}{|T|}. \quad (1)$$

By averaging these values over all services, we can calculate the overall error E_R associated with sharing a resource of a particular type. These per-service and per-resource error values allow one to quantify the *accuracy* of performance isolation under a specific load on a given system. Accuracy is the key metric for evaluating how well a system isolates services.

Measuring Error

Measuring error in practice is challenging, particularly due to the difficulty of computing the ideal share $I_{SR}[t]$.

Recall that the ideal share for a service is the minimum of the demanded share and the entitled share. If spare capacity is shared among services then the entitled share of one service depends on the ideal shares of others, which results in a circular definition. This problem can be solved iteratively, by repeatedly assigning spare capacity to services in proportion to their minimal shares until either all demand is satisfied or until the resource is exhausted. The harder part is measuring demand.

It may seem that demand is reflected in the number of outstanding requests for resources from a service. That is true at any particular instant: all services waiting for a CPU have 100% demand for it, all services with packets in network queues demand the share of the network bandwidth necessary to send those packets, etc. However, the demand of a service over an interval may depend on how loaded the resource is. For a hypothetical example, imagine a process that runs for 1 second no matter how much CPU time it gets: depending on how loaded the CPU is, its demand is different (and always satisfied). To put it simply, the demand may vary with supply.

To measure the error, we configure our experiments so that the demand is known for their duration. Thus, we can compute the ideal share without having to measure it.

CPU sharing example

Consider three services, guaranteed the same 50-33-17 minimal shares as in the example above. Spare capacity is allocated in proportion to those shares. The services are CPU-bound, which means their demand is 100%

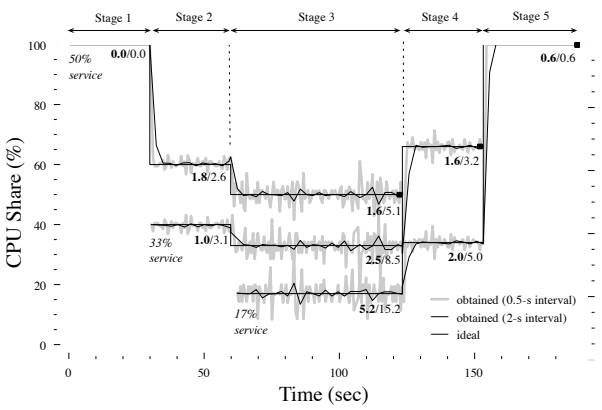

Figure 1: *Obtained and ideal shares for a* CPU*-bound load with three services given minimal* CPU *shares of 50%, 33%, and 17%. Thick grey lines and per-stage error values in bold font are for a 2-s sampling interval. Thick black lines and error values in regular font are for a 0.5-s interval. Boxes show termination.*

when they are running and 0% when they are not. Figure 1 shows a run of these services on an operating system called Vortex[1], which has been designed from the ground up by the authors to support performance isolation of services.

The services were started and stopped at different times, resulting in changes in ideal shares, which are shown as thin straight lines (to avoid clutter, we do not show 0% ideal shares). We refer to each of the time periods with stable ideal share as a *stage*. In the first and last stages, a service runs alone with 100% ideal share. In the intermediate stages, the ideal shares for concurrent services are the same as in the example above.

We sampled the CPU usage of each service twice a second and plotted it with thick grey lines. The errors computed from those measurements are thus at a 0.5-second interval. We also re-sampled the values in groups of four to demonstrate how the error changes when the interval is increased to 2 seconds. Those values are shown as thick black lines.

Using the errors in each interval, we computed per-service error E_{SR} for each stage (shown on the plot) and for the entire run (shown in the table below, together with the overall, per-resource error E_R).

	% error for interval	
Type of error	0.5 sec	2 sec
50% service E_{SR}	3.3	1.2
33% service E_{SR}	5.9	1.9
17% service E_{SR}	8.7	3.2
E_R	**6.0**	**2.1**

[1]In this, and in following experiments, we ran Vortex on a 2-GHz AMD Opteron machine with 2 GB of memory, a PCI Express bus, a 1-Gbps network interface (with Jumbo frames disabled).

Although the obtained shares generally follow the ideal ones, there is fluctuation due to imperfections of scheduling. The error is 3 times larger with an interval quarter as long. Two more observations can be made: accuracy decreases as more services are added to the system and, generally, the accuracy of services with smaller shares is lower. This behavior can be explained by the timeshared nature of the CPU.

In the experiment, Vortex was set up with a weighted fair queuing CPU scheduler and a timeslice size of 16 ms, resulting in scheduler invocations at a rate of 63.5 times per second. Thus, a service with 17% of the CPU should ideally receive 10.6 timeslices per second. However, since the scheduler does not hand out fractions of a timeslice, shares obtained within short time intervals will be inaccurate, and the smaller the share, the greater the error.

2.4 Agility

Although the overall error E_R quantifies how fairly a system schedules a resource, it gives no insight into the underlying causes of unfair scheduling. One important characteristic of scheduling, though, can be gleaned from the individual error values: By analyzing their trend within a stage (period with unchanging ideal share), one may be able to separate the error due to the system's inability to quickly adjust to changes in demand from the error due to the system's inability to enforce performance isolation with fixed demand. If either type of "inability" dominates, the best gains will come from fixing it.

Specifically, the goal of the analysis is to divide each stage into the initial *adjustment period*, when the obtained share is adjusting from the previous ideal share to the new one, and the following *stable period*, when the obtained share has stabilized. (Either period can be of length 0 if no adjustment or no stability are discernible.) By recomputing the errors E_{SR} using only the error values within adjustment periods or within the stable periods, we can obtain two new error values – E_{SR}^{adj} and E_{SR}^{sta} – which show how much each period contributes to the per-service error. Likewise, the two values can be computed for the overall, per-resource error.

Another way to quantify the ability of the system to adjust to changes in demand is to compute the average length of the adjustment period. We call this metric *agility*. As with error, agility can be computed either for each service using a resource (A_{SR}) or for a resource as a whole (A_R).

Adjustment-stable boundary

To compute either the agility or the two error components requires one to locate the *boundary* between the adjustment and the stable period inside each stage. Intuitively, in the intervals that follow the boundary the error val-

ues are contained within a small range, with a stable running average; the intervals preceding the boundary, however, exhibit error values from a much larger range, with a changing running average.

In our preliminary experiments we found the boundary using the following method: First, we found the maximum and the median error for a stage. Then, starting at the beginning of a stage and moving forward in time, we included all intervals into the adjustment period until we reached the one with the maximum error; then, we continued to include the intervals for as long as their error was larger than the median error. This method works, assuming that the instability period consists of a single cluster of error values that are all higher than the error values during the stable period, and that this cluster is fully contained in the first half of the stage. We ensured that in our experiments the stages were long enough to satisfy those two conditions.

Although this approach was sufficient in our case, we acknowledge that a more general definition, known to work with all patterns of error values, still remains to be devised. We suspect that techniques such as "change-point detection" for time-series analysis or "transient removal" for analysis of simulation data can be applied here.

Network sharing example

Consider Figure 2 for an illustration of agility analysis. The graph shows the bandwidths achieved by a network-bound load with three services, which were given the same 50-33-17 minimal shares as in the CPU experiment. In addition to the three lines corresponding to each service (and their ideal shares shown as straight lines in the middle), there is also a thick dark-gray line showing the overall network bandwidth utilization. The nature of the load is such that one service is not able to saturate the network card, two services together are sometimes saturating it, and with three services the NIC is always at its full capacity.

Due to the inherent inability of weighted fair queuing algorithms to schedule fairly in the presence of drastic changes in demand, such as when services start and stop, large bandwidth fluctuations lasting 10–30 seconds occur. These adjustment periods are shown with light-gray lines. Agility analysis of the 3rd stage, when three principals are sharing the network card, indicates that the adjustment period of that stage lasts 15.8 seconds for each service. Here is the summary, for stage 3, of the values computed using the metrics that we proposed:

Service	A_{SR}	E_{SR}^{adj}	E_{SR}^{sta}	E_{SR}
50%	15.8	43.4	5.4	9.8
33%	15.8	45.3	5.2	9.8
17%	15.8	205.3	5.2	28.3

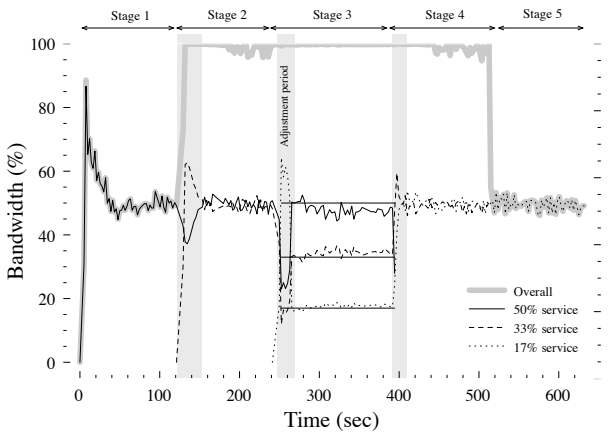

Figure 2: *Obtained bandwidth for a network-bound load with three services, given minimal bandwidth shares of 50%, 33%, and 17% and sampled in 2-s intervals. Light-gray shows adjustment periods. The dark-gray line shows overall bandwidth use. The thin straight lines show ideal shares during the 3rd stage.*

In these results, the error for the stable period is significantly lower than the error for the adjustment period. Furthermore, the adjustment period errors differ among services, while the stable period errors are similar (indicating a scheduler trait that is unaffected by the size of the share). The combined E_{SR} is less revealing since it lacks such information. As with CPU, the higher error is associated with the smaller share.

By relating the obtained values to the ideal values, which are visible as straight horizontal lines in stage 3, we can conclude that the 50% service is under-provisioned while the other two are over-provisioned. This can be verified quantitatively by computing the per-service average error using the sum of errors instead of their absolute values, as in (1). Positive values show the magnitude of over-provisioning and negative values show the magnitude of under-provisioning.

3 Related Work

We complement and extend previous work by introducing a model and metrics for evaluating how well an approach or a system enforces performance isolation. For example, systems such as Eclipse [2], Resource Containers [1], and Software Performance Units [12], which were designed to do performance isolation, can be evaluated and compared using our approach.

Our notion of error and ideal share is similar to service error and perfect fairness from work in the area of proportional share scheduling [11, 3]. However, our definition of ideal share encompasses scheduling approaches that are both work conserving and non work conserving. Previous work has identified agility as a key attribute of

mobile systems [9]. We argue that agility is an equally important attribute of dependable systems.

Accurate scheduling of resources has traditionally been an important concern in real-time operating systems. These systems host services with time constraints and must multiplex resources to ensure that all deadlines are met. However, real-time systems are typically 'CPU-centric' in that they only provide guarantees on the availability of CPU cycles. If guarantees on other types of resources are supported, these are typically probabilistic and based on techniques such as progress monitoring [7, 10]. For example, Rialto [6, 4] relies on collected profiles to produce CPU schedules. Real-time mechanisms tend to be too rigid for modern applications that have a rich mix of real-time and non-real-time considerations. The *capacity reservation* paradigm in RT-MACH [8] is designed to support such a mix, but only manages user-level CPU cycles. Lottery Scheduling [13], on the other hand, uses a probabilistic strategy that can handle a variety of resources, both in kernel and user space.

Most recently the issue of performance isolation has been explored in the context of virtual machines [5].

4 Conclusion

In this paper we argue that performance isolation is essential in shared systems used by dependable services and that current operating systems do not allow for specification and enforcement of sharing policies for all resources. Moreover, we argue that there exists no satisfactory set of metrics to measure if isolation requirements are met.

We define performance isolation as enforcement of two policies: minimal guarantee policy and a spare capacity policy. The former prevents uncontrollable performance degradations, while the latter allows one to make tradeoffs between utilization and predictability of performance. This definition covers policies ranging from strict partitioning to proportional sharing of resources.

We put forth two intuitive metrics for evaluating how well a system enforces performance isolation: accuracy and agility. Accuracy captures how close actual allotment of resources is to an ideal allotment under a given sharing policy, while agility captures how quickly a system adjusts to changes in resource demand.

Running workloads on Vortex, a new operating system that supports holistic resource sharing, we demonstrate and confirm the effectiveness of these metrics.

References

[1] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 45–58, New Orleans, LA, February 1999.

[2] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz. The Eclipse operating system: Providing quality of service via reservation domains. In *Proceedings of USENIX Annual Technical Conference*, pages 235–246, New Orleans, Louisiana, June 1998.

[3] A. Demers, S. Keshav, and S. Shenker. Analysis and simulations of a fair queuing algorithm. In *Proceedings of Special Interest Group on Data Communication*, pages 3–12, Austin, Texas, September 1989.

[4] R. P. Draves, G. Odinak, and S. M. Cutshall. The Rialto virtual memory systems. Technical Report MSR-TR-97-04, Microsoft Research, Advanced Technology Division, February 1997.

[5] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing performance isolation across virtual machines in xen. Technical Report HPL-2006-77, Hewlet Packard, May 2006.

[6] M. B. Jones, P. J. Leach, R. Draves, and J. S. Barrera. Modular real-time resource management in the Rialto operating system. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems*, pages 12–17, Orcas Island, WA, May 1995.

[7] H. Massalin and C. Pu. Fine-grain adaptive scheduling using feedback. *Computing Systems*, 3(1):139–173, Winter 1990.

[8] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pages 90–99, Boston, MA, May 1994.

[9] B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, and K. Walker. Agile application-aware adaption for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Saint Malo, France, October 1997.

[10] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 145–158, New Orleans, Louisiana, February 1999.

[11] R. Tidjeman. The Chairman assignment problem. *Discrete Mathematics*, 32:323–330, 1980.

[12] B. Verghese, A. Gupta, and M. Rosenblum. Performance isolation: Sharing and isolation in shared-memory multiprocessors. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 181–192, San Jose, CA, October 1998.

[13] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1th Symposium on Operating Systems Design and Implementation*, pages 1–11, Monterey, CA, november 1994.