Enforcing Performance Isolation Across Virtual Machines in Xen

Diwaker Gupta[1], Ludmila Cherkasova[2], Rob Gardner[2], and Amin Vahdat[1]

[1] University of California, San Diego, CA 92122, USA
{dgupta, vahdat}@cs.ucsd.edu
[2] Hewlett-Packard Laboratories
{lucy.cherkasova,rob.gardner}@hp.com

Abstract. Virtual machines (VMs) have recently emerged as the basis for allocating resources in enterprise settings and hosting centers. One benefit of VMs in these environments is the ability to multiplex several operating systems on hardware based on dynamically changing system characteristics. However, such multiplexing must often be done while observing per-VM performance guarantees or service level agreements. Thus, one important requirement in this environment is effective performance isolation among VMs. In this paper, we address performance isolation across virtual machines in Xen [1]. For instance, while Xen can allocate fixed shares of CPU among competing VMs, it does not currently account for work done on behalf of individual VM's in device drivers. Thus, the behavior of one VM can negatively impact resources available to other VMs even if appropriate per-VM resource limits are in place.
In this paper, we present the design and evaluation of a set of primitives implemented in Xen to address this issue. First, *XenMon* accurately measures per-VM resource consumption, including work done on behalf of a particular VM in Xen's driver domains. Next, our *SEDF-DC* scheduler accounts for aggregate VM resource consumption in allocating CPU. Finally, *ShareGuard* limits the total amount of resources consumed in privileged and driver domains based on administrator-specified limits. Our performance evaluation indicates that our mechanisms effectively enforce performance isolation for a variety of workloads and configurations.

1 Introduction

Virtual Machine Monitors (VMMs)[3] are gaining popularity for building more agile and dynamic hardware/software infrastructures. In large enterprises for example, VMMs enable server and application consolidation in emerging on-demand utility computing models [2, 3]. Virtualization holds the promise of achieving greater system utilization while lowering total cost of ownership and responding more effectively to changing business conditions.

Virtual machines enable *fault isolation*—"encapsulating" different applications in self-contained execution environments so that a failure in one virtual machine does not affect other VMs hosted on the same physical hardware. *performance isolation* is another important goal. Individual VMs are often configured with performance guarantees and expectations, e.g., based on service level agreements. Thus, the resource consumption of one virtual machine should not impact the promised guarantees to other VMs on the same hardware.

[3] We use the terms *hypervisor* and *domain* interchangeably with VMM and VM respectively.

In this paper, we focus on performance isolation mechanisms in Xen [1], a popular open source VMM. Xen supports per-VM CPU allocation mechanisms. However, it — like many other VMMs — does not accurately account for resource consumption in the hypervisor on behalf of individual VMs, e.g., for I/O processing. Xen's I/O model has evolved considerably over time. In the initial design [1] shown in Figure 1a, the Xen hypervisor itself contained device driver code and provided shared device access. To reduce the risk of device driver failure/misbehavior and to address problems of dependability, maintainability, and manageability of I/O devices, Xen moved to the architecture shown in Figure 1b [4]. Here, "isolated driver domains" (IDDs) host unmodified (legacy OS code) device drivers. Domain-0 is a privileged control domain used to manage other domains and resource allocation policies.

This new I/O model results in a more complex CPU usage model. For I/O intensive applications, CPU usage has two components: CPU consumed by the guest domain, where the application resides, and CPU consumed by the IDD that incorporates the device driver and performs I/O processing on behalf of the guest domain. However, the work done for I/O processing in an IDD is not charged to the initiating domain. Consider a guest domain limited to 30% CPU consumption. If the work done on its behalf within an IDD to perform packet processing consumes 20% of the CPU, then that domain may consume 50% of overall CPU resources. Such unaccounted CPU overhead is significant for I/O intensive applications, reaching 20%-45% for a web server [5].

The key contribution of this paper is the design of a set of cooperating mechanisms to effectively control total CPU consumption across virtual machines in Xen. There are a number of requirements for such a system. First, we must accurately measure the resources consumed within individual guest domains. Next, we must attribute the CPU consumption within IDDs to the appropriate guest domain. The VMM scheduler must be modified to incorporate the aggregate resource consumption in the guest domain and work done on its behalf in IDDs. Finally, we must limit the total amount of work done on behalf of a particular domain in IDDs based on past consumption history and target resource limits. For instance, if a particular domain is already consuming nearly its full resource limits, then the amount of resources available to it in the IDDs must be scaled appropriately.

The analog of accounting resources consumed on behalf of a guest domain have come up in scheduling operating system resources across individual tasks [6–12], e.g., in accounting for resources consumed in the kernel on behalf of individual processes. Our work builds upon these earlier efforts, exploring the key challenges associated with constructing appropriate abstractions and mechanisms in the context of modern VM architectures. One of the interesting problems in this space is developing minimally intrusive mechanisms that can: i) account for significant asynchrony in the hypervisor

(a) I/O Model in Xen 1.0 (b) I/O Model in Xen 3.0

Fig. 1: Evolution of Xen's I/O Architecture

and OS and ii) generalize to a variety of individual operating systems and device drivers (performance isolation will quickly become ineffective if even a relatively small number of devices or operations are unaccounted for). To this end, we have completed a full implementation and detailed performance evaluation of the necessary system components to enable effective VM performance isolation:

- XenMon: a performance monitoring and profiling tool that reports (among other things) CPU usage of different VMs at programmable time scales. XenMon includes mechanisms to measure CPU for network processing in net-IDDs (IDDs responsible for network devices) on behalf of guest domains.
- SEDF-DC: a new VM scheduler with feedback that effectively allocates CPU among competing domains while accounting for consumption both within the domain and in net-IDDs.
- ShareGuard: a control mechanism that enforces a specified limit on CPU time consumed by a net-IDD on behalf of a particular guest domain.

2 XenMon

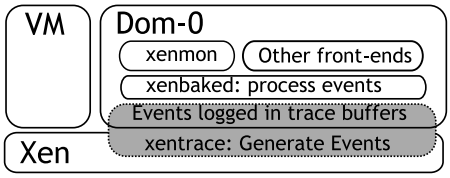


Fig. 2: XenMon Architecture

To support resource allocation and management, we implemented an accurate monitoring and performance profiling infrastructure, called XenMon.[4] There are three main components in XenMon (Figure 2):

- `xentrace`: This is a lightweight event logging facility present in Xen. XenTrace can log events at arbitrary control points in the hypervisor. Each event can have some associated attributes (for instance, for a "domain scheduled" event, the associated attributes might be the ID of the scheduled domain and the event's time stamp). Events are logged into "trace buffers": shared memory pages that can be read by user-level Domain-0 tools. Note that `xentrace` was already implemented in Xen — our contribution here was to determine the right set of events to monitor.
- `xenbaked`: The events generated by XenTrace are not very useful on their own. `xenbaked` is a user-space process that polls [5] the trace buffers for new events and processes them into meaningful information. For instance, we collate domain sleep and wake events to determine the time for which a domain was blocked in a given interval.
- `xenmon`: This is the front-end for displaying and logging the data.

[4] Our implementation of XenMon has been integrated into the official Xen 3.0 code base.

[5] In the current implementation, events are posted via a virtual interrupt instead of periodic polling.

XenMon aggregates a variety of metrics across all VMs periodically (configurable with a default of 100 ms). For this paper, we only use the CPU utilization and network accounting facilities (Section 3) of XenMon. Details on all the metrics available from XenMon and some examples of using XenMon for analyzing CPU schedulers in Xen are available separately [13].

3 Network I/O Accounting

Recall that one of the challenges posed by the new I/O model in Xen is to classify IDD CPU consumption across guest domains. This work is focused on network I/O, so we summarize network I/O processing in Xen. As mentioned earlier, in the IDD model a designated driver domain is responsible for each hardware device and all guests wishing to use the device have to share it via the corresponding IDD. The IDD has a "back-end" driver that multiplexes I/O for multiple "front-end" drivers in guest VMs over the real device driver. Figure 3 shows this I/O architecture in more detail. Note that for the experiments reported in this paper, we use Domain-0 as the driver domain.

We briefly describe the sequence of events involved in receiving a packet — the numbers correspond to those marked in Figure 3.When the hardware receives the packet (1), it raises an interrupt trapped by Xen (2). Xen then determines the domain responsible for the device and posts a *virtual* interrupt to the corresponding driver domain via the *event channel* (3). When the driver domain is scheduled next, it sees a pending interrupt and invokes the appropriate interrupt handler. The interrupt handler in the driver domain only serves to remove the packet from the real device driver (4) and hand it over to the "back-end" driver (5), *netback* in Figure 3. Note that no TCP/IP protocol processing is involved in this step (except perhaps the inspection of the IP header).

It is netback's responsibility to forward the packet to the correct "front-end" driver (*netfront* in Figure 3). The driver domain transfers the ownership of the memory page containing the packet to the target guest domain, and then notifies it with a "virtual interrupt" (6). Note that this involves no data movement/copying. When the target guest is next scheduled, it will field the pending interrupt (7). The netfront driver in the guest will then pass on the packet to higher layers of the networking stack for further processing (8). The transmit path of a packet is similar, except that no explicit memory page exchange is involved (see [1] for details).

Thus, I/O processing in a net-IDD primarily involves two components: the real device driver and the back-end (virtual) device driver. One natural approach for more accurate accounting is to instrument these components for detailed measurements of

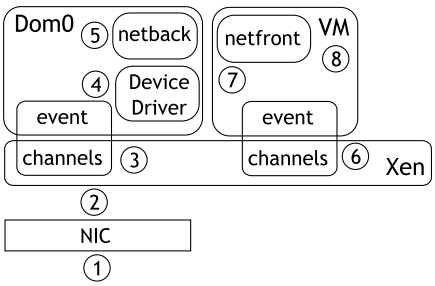


Fig. 3: I/O processing path in Xen.

all the delays on the I/O path. However, this approach does not scale in Xen for two reasons: (1) since Xen uses legacy Linux drivers, this would require instrumenting *all* network device drivers, and (2) network drivers involve significant asynchronous processing, making it difficult to isolate the time consumed in the driver in the context of a given operation.

We therefore need some alternate heuristics to estimate the per-guest CPU consumption. Intuitively, each guest should be charged in proportion to the amount of I/O operations it generates. In [5], we used the number of memory page exchanges as an estimator. However, we found this method to be a rather coarse approximation that does not take into account what fraction of these page exchanges correspond to sent versus received packets, and that does not take into account the size of the packets.

Thus, we propose using the *number of packets* sent/received per guest domain for distributing the net-IDD CPU consumption among guests. Note that netback is an ideal observation point: all of the packets (both on the send and receive paths between driver domain and guest domain) *must* pass through it. We instrumented netback to provide detailed measurements on the number of packets processed by the corresponding net-IDD in both directions for each guest domain. In particular, we added XenTrace events for each packet transmission/reception, with the appropriate guest domain as an attribute. We then extended XenMon to report this information.

Of course, knowing the number of packets sent and received on a per-domain basis does not by itself enable accurate CPU isolation. We need a mechanism to map these values to per-domain CPU consumption in the IDD. In particular, we want to know the dependence of packet size on CPU processing overhead and the breakdown of send versus receive packet processing. To answer these questions, we perform the following two part study.

The impact of packet size on CPU overhead in net-IDD: We performed controlled experiments involving sending packets of different sizes at a *fixed* rate to a guest VM. In particular, we fixed the rate at 10,000 pkts/sec and varied the packet size from 100 to 1200 bytes. Each run lasted 20 seconds and we averaged the results over 10 runs. We repeated the experiments to exercise the reverse I/O path as well – so the VM was *sending* packets instead of receiving them. To prevent "pollution" of results due to ACKS going in the opposite direction, we wrote a custom tool for these benchmarks using UDP instead of TCP. The other end point for these experiments was a separate

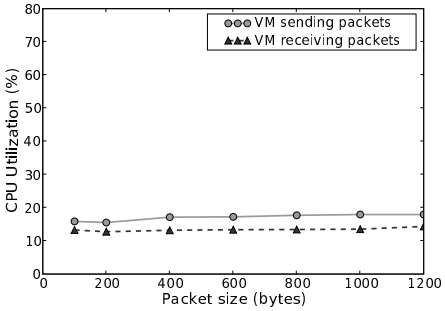


Fig. 4: CPU overhead in Domain-0 for processing packets at a fixed rate under different packet sizes.

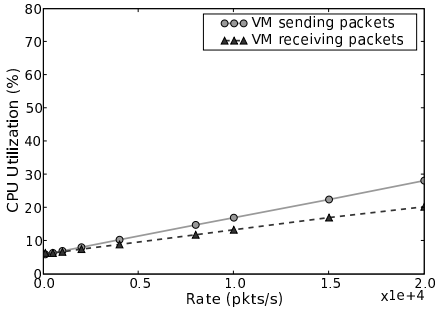


Fig. 5: CPU overhead in Domain-0 for processing packets of a fixed size under varying rates.

machine on our LAN. Recall that in all of our experiments, we use Domain-0 to host the network device driver.

Our results show that CPU consumption in net-IDD does not depend on packet size as presented in Figure 4. The explanation is as follows: during driver packet processing there is no payload processing or copying; the driver largely deals with the packet header. For the rest of the I/O path within the net-IDD, there is no data copying (where CPU processing can depend on packet size) — only the ownership of memory pages changes to reflect data transfer.

CPU overhead in net-IDD for Send vs. Receive I/O paths: In this experiment, we fixed the packet size at 256 bytes and varied the rate at which a VM sends or receives packets. We could thus selectively exercise the send and receive I/O paths within Xen and measure the resulting CPU overhead in net-IDD. We denote these as *Send Benchmark* and *Receive Benchmark*, respectively. As before, each run lasted 20 seconds and we averaged results over 10 runs.

Figure 5 presents our experimental results. An interesting outcome of this study is that the ratio of CPU consumption in net-IDD between send and receive paths is consistently the same for different packet rates. We denote this measured ratio as $weight$.

To validate the generality of presented results we repeated all of the experiments presented above for two different hardware configurations: a single CPU Intel Pentium-IV machine running at 2.66-GHz with a 1-Gbit Intel NIC (SYSTEM-1) and a dual processor Intel Xen 2.8-GHz with a 1-Gbit Broadcom NIC (SYSTEM-2). For both systems under test, the CPU consumption in net-IDD does not depend on packet size. Further, for both system under test, the ratio of CPU consumption in net-IDD between send and receive paths is consistent for different packet rates:

- SYSTEM-1: $weight = 1.1$ (standard deviation 0.07);
- SYSTEM-2: $weight = 1.16$ (standard deviation 0.15).

These results show that the number of packets in conjunction with the direction of traffic can be reasonably used to split CPU consumption among guests. Concretely, let $Send/Recv(Dom_i)$ denote packets sent/received by net-IDD to/from Dom_i and $Send/Recv(netIDD)$ denote the total packets sent/received by net-IDD. Then, we define the *weighted* packet count per domain as $weight \times Send(Dom_i) + Recv(Dom_i)$, where $weight$ is the ratio of CPU consumption in net-IDD for send versus receive paths. Similarly, we compute the weighted packet count for net-IDD: $wCount(netIDD)$. Then we can use the fraction $wCount(Dom_i)/wCount(netIDD)$ to charge CPU usage to Dom_i.

In the remainder of this paper, we use this weighted count to compute the CPU overhead in net-IDD for network processing on behalf of different guest domains. This approach is also attractive because it comes with a compact, portable benchmark that derives the weight coefficient between send/receive paths automatically for different systems and different network device drivers. It has the further advantage of being general to a variety of device drivers and operating systems (e.g., individual device drivers may be hosted on a variety of operating systems) without requiring error-prone instrumentation. Of course, it has the disadvantage of not explicitly measuring CPU consumption but rather deriving it based on benchmarks of a particular hardware configuration. We feel that this tradeoff is inherent and that instrumenting all possible device driver/OS configurations is untenable for resource isolation. A variety of middleware

tools face similar challenges, i.e., the inability to modify or directly instrument lower layers, making our approach attractive for alternate settings as well.

With this estimation of CPU utilization per guest, we now turn our attention to SEDF-DC and ShareGuard.

4 SEDF-DC: CPU Scheduler with Feedback

Xen's reservation based CPU scheduler — SEDF (Simple Earliest Deadline First) — takes its roots in the Atropos scheduler [8]. In SEDF, an administrator can specify the CPU share to be allocated per VM. However, there is no way to restrict the aggregate CPU consumed by a domain and by driver domains acting on its behalf. We have extended SEDF to accomplish this goal.

4.1 Overview

Our modified scheduler, SEDF-DC for SEDF-*Debt Collector*, periodically receives feedback from XenMon about the CPU consumed by IDDs for I/O processing on behalf of guest domains. Using this information, SEDF-DC constrains the CPU allocation to guest domains to meet the specified combined CPU usage limit.

For each domain Dom_i, SEDF takes as input a tuple (s_i, p_i), where the *slice* s_i and the *period* p_i together represent the CPU share of Dom_i: Dom_i will receive at least s_i units of time in each period of length p_i. Such specifications are particularly convenient for dynamically adjusting CPU allocations: we can directly charge the CPU time consumed by IDDs for Dom_i by decreasing s_i appropriately. In CPU schedulers based on weights, one would need to continuously re-calculate weights of domains to achieve the same result.

We now describe SEDF-DC's operation, but limit our description only to places where SEDF-DC differs from SEDF. SEDF-DC maintains 3 queues:

- Q_r: the queue of runnable domains;
- Q_w: the queue of domains that have exhausted their slice and are awaiting the next period;
- Q_b: the queue of blocked domains.

A key concept in SEDF is *deadlines*. Intuitively, a deadline denotes the absolute time that a domain *should have* received its specified share of the CPU. Both Q_r and Q_w are sorted by deadlines, making the selection of the next domain to schedule a constant time operation.

Each domain D_i's deadline is initialized to $NOW + p_i$, where NOW denotes the current time. Let t denote the *feedback interval* (set to $500\ ms$ in our current implementation). Let net-IDD be a driver domain with a networking device that is shared by $Dom_1, \ldots, Dom_n$. We will simplify the algorithm description (without loss of generality) by considering a single net-IDD. Using XenMon, we compute the CPU consumption $used_i^{IDD}$ of net-IDD for network I/O processing on behalf of Dom_i during the latest t-ms interval and provide this information (for all domains) to SEDF-DC.

For each domain Dom_i, the scheduler tracks three values $(d_i, r_i, debt_i^{IDD})$:

– d_i: domain's current *deadline* for CPU allocation, the time when the current period ends for domain Dom_i.
– r_i: domain's current *remaining time* for CPU allocation, the CPU time remaining to domain Dom_i within its current period.
– $debt_i^{IDD}$: CPU time consumed by Dom_i via the net-IDD's networking processing performed on behalf of Dom_i. We call this the *CPU debt* for Dom_i. At each feedback interval, this value is incremented by $used_i^{IDD}$ for the latest t-ms.

Note that the original SEDF scheduler only tracks (d_i, r_i). The introduction of $debt_i^{IDD}$ in the algorithm allows us to observe and enforce aggregate limits on Dom_i's CPU utilization.

Let a and b be integer numbers and let us introduce the following function $a \hat{-} b$ as follows:

$$a \hat{-} b = \begin{cases} 0 & \text{if } a \leq b \\ a - b & \text{otherwise} \end{cases}$$

We now describe the modified procedure for updating the queues (Q_r, Q_w, and Q_b) on each invocation of SEDF-DC.

1. The time $gotten_i$ for which the current Dom_i has been running is deducted from r_i: $r_i = r_i - gotten_i$.
   If $debt_i^{IDD} > 0$ then we attempt to charge Dom_i for its CPU debt by decreasing the remaining time of its CPU slice:
   – if $debt_i^{IDD} \leq r_i$ then $r_i = r_i - debt_i^{IDD}$ and $debt_i^{IDD} = 0$;
   – if $debt_i^{IDD} > r_i$ then $debt_i^{IDD} = debt_i^{IDD} - r_i$ and $r_i = 0$.
2. If $r_i = 0$, then Dom_i is moved from Q_r to Q_w, since Dom_i has received its required CPU time in the current period.
3. For each domain Dom_k in Q_w, if $NOW \geq d_k$ then we perform the following updates:
   – r_k is reset to $s_k \hat{-} debt_k^{IDD}$;
   – $debt_k^{IDD}$ is decreased by $min(s_k, debt_k)$;
   – the new deadline is set to $d_k + p_k$;
   – If $r_k > 0$ then Dom_k is moved from Q_w to Q_r.
4. The next timer interrupt is scheduled for $\min(d_w^h + p_w^h, d_r^h)$, where (d_w^h, p_w^h) and (d_r^h, p_r^h) denote the deadline and period of the domains that are respective heads of the Q_r and Q_w queues.
5. On an interrupt, the scheduler runs the head of Q_r. If Q_r is empty, it selects the head of Q_w.
6. When domain Dom_k in Q_b is unblocked, we make the following updates:
   – if $NOW < d_k$ then
     • if $debt_k^{IDD} \leq r_k$ then $r_k = r_k - debt_k^{IDD}$, and $debt_k^{IDD} = 0$, and Dom_k is moved from Q_b to Q_r;
     • if $debt_k^{IDD} > r_k$ then $debt_k^{IDD} = debt_k^{IDD} - r_k$ and $r_k = 0$.
   – if $NOW \geq d_k$ then we compute for how many periods Dom_k was blocked. Since Dom_k was not runnable, this unused CPU time can be charged against its CPU debt:

$$bl\_periods = int\left\{\frac{(NOW - d_k)}{p_k}\right\}$$
$$debt_k^{IDD} = debt_k^{IDD} - r_k - (bl\_periods \times s_k)$$

– r_k is reset to $s_k \hat{-} debt_k^{IDD}$. If $r_k > 0$, then Dom_k is moved from Q_b to Q_r and can be scheduled to receive the remaining r_k;
– $debt_k^{IDD}$ is adjusted by s_k: $debt_k^{IDD} = debt_k^{IDD} \hat{-} s_k$;
– the new deadline is set to $d_k + p_k$

The SEDF-DC implementation described above might have bursty CPU allocation for domains hosting network-intensive applications, especially when a coarser granularity time interval t is used for the scheduler feedback, e.g., $t = 2s$. It might happen that domain Dom_i will get zero allocation of CPU shares for several consecutive periods until the CPU debt time $debt_i^{IDD}$ is "repaid". To avoid this, we implemented an optimization to SEDF-DC that attempts to spread the CPU debt across multiple execution periods.

We compute the number of times period p_i fits within a feedback interval — the intent is to spread the CPU debt of Dom_i across periods that happen during the feedback interval. We call this the *CPU period frequency* of domain Dom_i and denote it as $period\_freq_i$:

$$period\_freq_i = int\left(\frac{t}{p_i}\right)$$

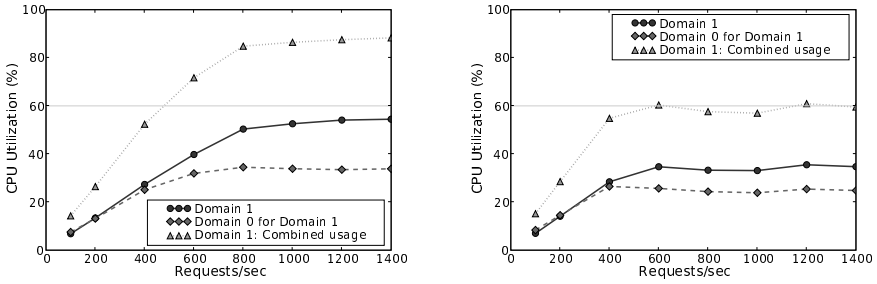
If $period\_freq_i > 1$, then we can spread $debt_i^{IDD}$ across $period\_freq_i$ number of periods, where at each period the domain is charged for a fraction of its overall CPU debt:

$$spread\_debt_i = int\left(\frac{debt_i^{IDD}}{period\_freq_i}\right)$$

This optimized SEDF-DC algorithm supports more consistent and smoother CPU allocation to domains with network-intensive applications.

4.2 Evaluation

In this section we evaluate SEDF-DC beginning with a simple setup to demonstrate the correctness of the scheduler and continue with a more complex scenario to illustrate SEDF-DC's feasibility for realistic workloads. All tests were conducted on single processor Pentium-IV machines running at 2.8-GHz.



(a) With original SEDF. (b) With SEDF-DC.

Fig. 6: Simple SEDF Benchmark

In the first experiment, we have a single VM (Domain-1) configured to receive a maximum of 60% of the CPU; Domain-0 is entitled to the remaining 40%. Domain-1 hosts a Web server, loaded using `httperf` [14] from another machine. We gradually increase the load and measure the resulting CPU utilizations.

Figure 6a shows the results with the unmodified SEDF scheduler. We see that as the load increases, Domain-1 consumes almost all of its share of the CPU. Additionally, Domain-0 incurs an overhead of almost 35% at peak loads to serve Domain-1's traffic. Hence, while Domain-1 was entitled to receive 60% of the CPU, it had received a *combined* CPU share of 90% via additional I/O processing in Domain-0. We repeated the same experiment with SEDF-DC, with the results shown in Figure 6b. We can see that SEDF-DC is able to enforce the desired behavior, keeping the combined CPU usage of Domain-1 bounded to 60%.

In practice, system configurations are likely to be more complicated: multiple VMs, each running a different service with different requirements; some VMs may be I/O intensive, others might be CPU intensive and so on. Our next experiment tries to evaluate SEDF and SEDF-DC under a more realistic setup.

For this experiment, we have two VMs (Domain-1 and Domain-2), each hosting a web-server. We configure both VMs and Domain-0 to receive a maximum of 22% of the CPU. Any slack time in the system is consumed by CPU intensive tasks running in a third VM. Domain-1's web-server is served with requests for files of size 10 KB at 400 requests/second, while Domain-2's web-server is served with requests for files of size 100 KB at 200 requests/second. We chose these rates because they completely saturate Domain-0 and demonstrate how CPU usage in Domain-0 may be divided between guest domains with different workload requirements. As before, we use `httperf` to generate client requests. Each run lasts 60 seconds.

We first conduct the experiment with unmodified SEDF to establish the baseline. Figure 7a shows the throughput of the two web-servers as a function of time. We also measure the CPU utilizations of all the VMs, shown in 7b. Note that Domain-1 consumes all of its 22% available CPU cycles, while Domain-2 consumes only about 15% of the CPU. Even more interesting is the split of Domain-0 CPU utilization across Domain-1 and Domain-2 as shown in Figure 7c. For clarify, we summarize the experiment in Table 1. The first column shows the average values for the metrics over the entire run. Domain-1 uses an additional 9.6% of CPU for I/O processing in Domain-0 (42% of overall Domain-0 usage) while Domain-2 uses an additional 13.6% of CPU via Domain-0 (58% of overall Domain-0 usage). Thus, the combined CPU utilization

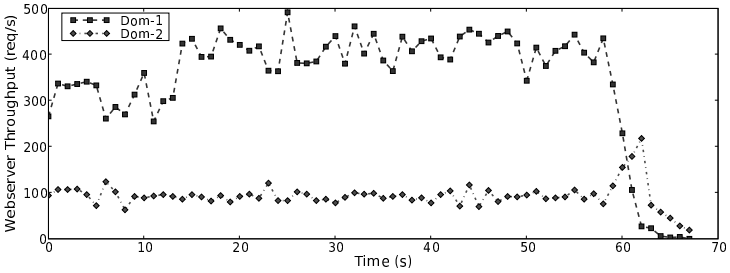
Metric	**SEDF**	**SEDF-DC**
Dom-1 web-server Throughput	348.06 req/s	225.20 req/s
Dom-2 web-server Throughput	93.12 req/s	69.53 req/s
Dom-1 CPU	19.6%	13.7%
Dom-0 for Dom-1	9.6%	7.7%
Dom-1 Combined	29.2%	21.4%
Dom-2 CPU	14.5%	10.9%
Dom-0 for Dom-2	13.2%	10.6%
Dom-2 Combined	27.7%	21.5%

Table 1: SEDF-DC in action: metric values averaged over the duration of the run

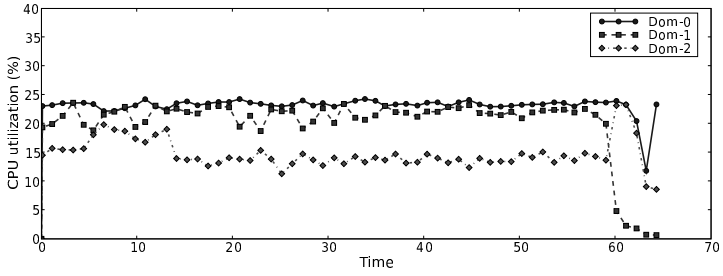
of Domains 1 and 2 (the sum of their individual CPU utilization and CPU overhead in Domain-0 on their behalf) is 29.2% and 27.7% respectively.

We then investigate whether we can limit the system-wide CPU usage of Domain-1 and Domain-2 to their 22% CPU share using SEDF-DC. Figure 8 shows the results of this experiment. Recall the operation of SEDF-DC: it computes the *debt* of a VM (work done by the IDD – in this case Domain-0 – on its behalf), and incrementally charges it back to the appropriate VM. This is clearly visible in Figure 8c: the *combined* utilizations of both Domain-1 and Domain-2 hover around 22% for the duration of the experiment. The oscillations result from discretization in the way we charge back debt.

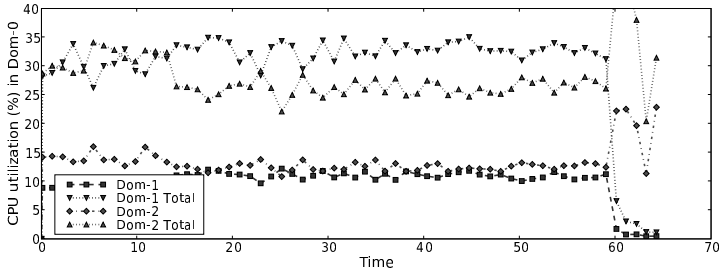
Controlling combined CPU utilization for Domain-1 and 2 does impact the web servers' achievable throughput. Since the combined CPU usage of Domain-1 and 2 is limited to 22% under SEDF-DC—versus the uncontrolled values of 29.2% and 27.7%



(a) Web-server Throughput



(b) CPU Utilization

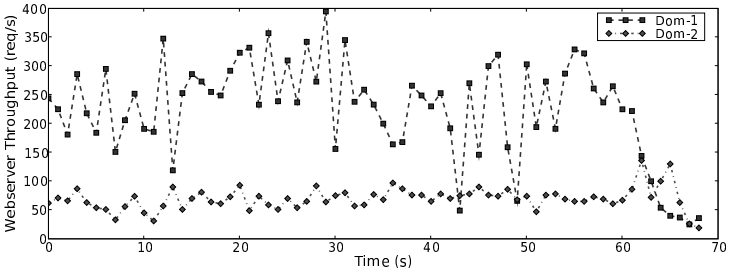


(c) CPU Utilization in Dom-0

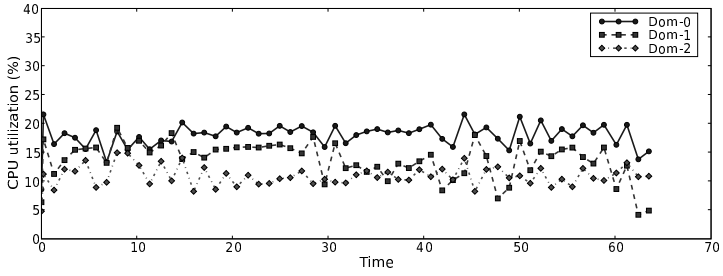
Fig. 7: With original SEDF.

under the original SEDF scheduler—there is a drop in throughput as shown in Figure 8a. The second column of Table 1 gives the average throughput values over the run for a more concise comparison.

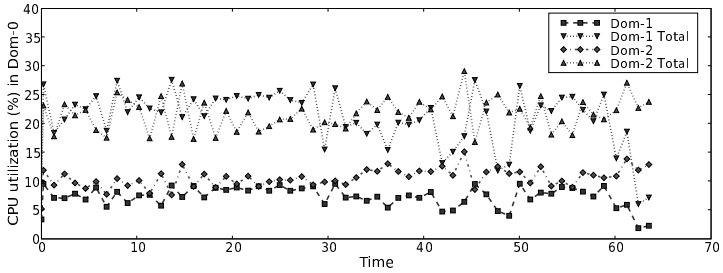
While SEDF-DC is capable of limiting the combined CPU usage across guest domains, it does not explicitly control CPU usage in a driver domain. Note that the split of the CPU utilization in Domain-0 for Domain-1 and Domain-2 is still unequal. Domain-1 is using 7.7% of CPU via Domain-0 (42% of overall Domain-0 usage) while Domain-2 is using 10.6% of CPU via Domain-0 (58% of overall Domain-0 usage). We turn our attention to controlling per-domain IDD utilization using ShareGuard in the next section.



(a) Web-server Throughput



(b) CPU Utilization



(c) CPU Utilization in Dom-0

Fig. 8: With SEDF-DC.

5 ShareGuard

In the current Xen implementation, a driver domain does not control the amount of CPU it consumes for I/O processing on behalf of different guest domains. This lack of control may significantly impact the performance of network services. Such control is also required to enable SEDF-DC to enforce aggregate CPU usage limits. In this section we describe ShareGuard: a control mechanism to solve this problem.

5.1 Overview

ShareGuard is a control mechanism to enforce a specified limit on CPU time consumed by an IDD for I/O processing on behalf of a particular guest domain. ShareGuard periodically polls XenMon for CPU time consumed by IDDs, and if a guest domain's CPU usage is above the specified limit, then ShareGuard stops network traffic to/from the corresponding guest domain.

Let the CPU requirement of net-IDD be specified by a pair (s^{IDD}, p^{IDD}), meaning that net-IDD will receive a CPU allocation of at least s^{IDD} time units in each period of length p^{IDD} units (the time unit is typically milli-seconds). In other words, this specification is bounding CPU consumption of net-IDD over time to $CPUshare^{IDD} = \frac{s^{IDD}}{p^{IDD}}$. Let $limit_i^{IDD}$ specify a fraction of CPU time in net-IDD available for network processing on behalf of Dom_i such that $limit_i^{IDD} < CPUshare^{IDD}$. If such a limit is not set then Dom_i is entitled to unlimited I/O processing in net-IDD. Let t be the time period ShareGuard uses to evaluate current CPU usage in net-IDD and perform decision making. In the current implementation of ShareGuard, we use $t = 500\ ms$.

Using XenMon, ShareGuard collects information on CPU usage by net-IDD at every feedback interval, and computes the fraction of overall CPU time used by net-IDD for networking processing on behalf of Dom_i ($1 \leq i \leq n$) during the latest t interval. Let us denote this fraction as $used_i^{IDD}$. In each time interval t, ShareGuard determines the validity of the condition: $used_i^{IDD} \leq limit_i^{IDD}$. If this condition is violated, then Dom_i has exhausted its CPU share for network traffic processing in net-IDD. At this point, ShareGuard applies appropriate defensive actions for the next time interval t^{def}, where

$$t^{def} = t \times int\left(\frac{used_i^{IDD} + 1}{limit_i^{IDD}}\right)$$

ShareGuard performs the following defensive actions:

- **Stop accepting incoming traffic to a domain:** Since our net-IDDs run Linux, we use Linux's routing and traffic control mechanisms [15] to drop/reject traffic destined for a particular domain. In particular, we use `iptables` [16] — they are easily scriptable and configurable from user space. Similar techniques can be applied in other operating systems that may serve as wrappers for other legacy device drivers.
- **Stop processing outgoing traffic from a domain:** As in the previous case, we can use `iptables` to drop packets being transmitted *from* a domain. However, this will still incur substantial overhead in the IDD because `iptables` will only process the packet once it has traversed the network stack of the IDD. Ideally we want to drop the packet before it even enters the IDD to limit processing overhead.

One approach would be to enforce `iptables` filtering *within* the guest domain. However, ShareGuard does not assume any cooperation from guests so we reject this option. However, we still have an attractive control point within the net-IDD where packets can be dropped before entering the net-IDDs network stack: the *net-back* driver (see Figure 3). ShareGuard sends a notification to netback identifying the target domain and the required action (drop or forward). This is akin to setting `iptables` rules, except that these rules will be applied within netback.
Whenever netback receives an outbound packet from a domain, it will determine if there are any rules applicable to this domain. If so, it will take the specified action. This is both lightweight (in terms of overhead incurred by IDD) and flexible (in terms of control exercised by IDD).

After time interval t^{def}, ShareGuard restores normal functionality in net-IDD with respect to network traffic to/from domain Dom_i.

5.2 Evaluation

To evaluate the effectiveness of ShareGuard in isolating total domain CPU consumption, we ran the following experimental configuration. Three virtual machines run on the same physical hardware. Domain-1 and Domain-2 host web servers that support business critical services. These services have well-defined expectations for their throughput and response time. The CPU shares for these domains are set to meet these expectations. Domain-3 hosts a batch application that does some computation and performs occasional bulk data transfers. This VM supports a less important application that is not time sensitive, but needs to complete its job eventually.

In our first experiment, we observe overall performance of these three services to quantify the degree of performance isolation ShareGuard can deliver. We configure a dual-processor machine as follows: Domain-0 runs on a separate processor and set to consume at most 60% of the CPU. The second CPU hosts three VMs: Domain-1 and Domain-2 run web servers (serving 10 KB and 100 KB files respectively), and Domain-3 occasionally does a bulk file transfer. All these VMs have equal share of the second CPU, 33% each. In this initial experiment, we do not enable ShareGuard to demonstrate baseline performance characteristics. The experiments were conducted over a gigabit network, so our experiments are not network limited. In this experiment, we start a benchmark that loads web servers in Domain-1 and Domain-2 from two separate machines using `httperf` for two minutes. Forty seconds into the benchmark, Domain-3 initiates a bulk-file transfer that lasts for 40 seconds.

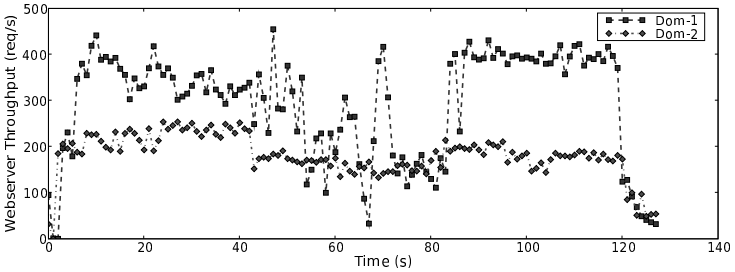
Figure 9 shows the results as a function of time. We can clearly see the adverse impact of Domain-3's workload on both web servers' throughput (Figure 9a). Considering the split of CPU utilization in Domain-0 for the corresponding interval (Figure 9c), we find that Domain-3 uses between 20% to 30% of CPU for I/O processing in Domain-0 leaving insufficient CPU resources for I/O processing on behalf of Domain-1 and Domain-2.

The first column in Table 2 provides a summary of average metric values for the baseline case where Domain-1 and Domain-2 meet their performance expectations and deliver expected web server throughput. These metrics reflect Domain-1 and Domain-2 performance when there is no competing I/O traffic issued by Domain-3 in the experiment. Note that in this case the combined CPU utilization in Domain-0 for I/O processing by Domain-1 and Domain-2 is about 50%. Since Domain-0 is entitled to

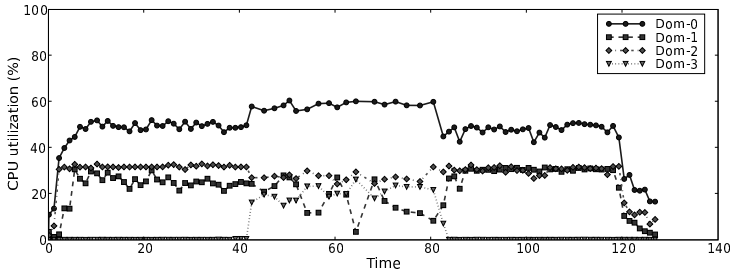
60% of the CPU, this means that there is about 10% CPU available for additional I/O processing in Domain-0.

The average metric values for this experiment (without ShareGuard) over the middle 40 second segment (where Domain-1, Domain-2, and Domain-3 all compete for CPU processing in Domain-0) are summarized in the second column of Table 2. Domain-3 gets 23.92% of CPU for I/O processing in Domain-0, squeezing in the CPU share available for Domain-1's and Domain-2's I/O processing. As a result, there is a significant decrease in achievable web server throughput: both web servers are delivering only 72% of their expected baseline capacity.

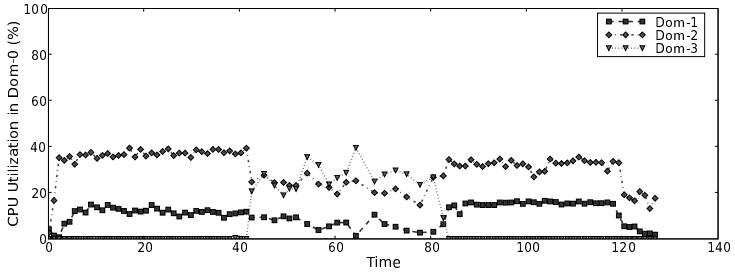
This example clearly indicates the impact of not controlling IDD CPU consumption by different guest domains. The question is whether ShareGuard can alleviate this problem. We repeat the experiment with ShareGuard enabled, and configure ShareGuard to



(a) Webserver Throughput



(b) CPU Utilization



(c) CPU Utilization in Dom-0

Fig. 9: Without ShareGuard

Metric	Baseline	Without ShareGuard	With ShareGuard
Dom-1 Webserver	329.85	236.8	321.13
Dom-2 Webserver	231.49	166.67	211.88
Dom-0 for Dom-1	11.55	7.26	11.9
Dom-0 for Dom-2	37.41	23.9	34.1
Dom-0 for Dom-3	N/A	23.92	4.42

Table 2: ShareGuard at work: metric values are averaged over the middle 40 second segment of the runs.

limit the CPU consumption for Domain-3 in Domain-0 to 5%. Figure 10 shows the results.

Recall ShareGuard's operation: every 500 ms it evaluates CPU usage in the IDD; if a VM is violating its CPU share, it turns off all traffic processing for that VM for some time. We compute this duration such that over that interval, the average CPU utilization of the VM within the IDD will comply with the specification. This mode of operation is clearly visible in Figure 10c. We had directed ShareGuard to restrict Domain-3's consumption in Domain-0 to 5%. At $t = 40s$, ShareGuard detected that Domain-3 had consumed almost 30% CPU in Domain-0. Accordingly, it disables traffic processing for Domain-3 for the next 2.5 seconds, such that the average utilization over this 3 second window drops to 5%. This pattern subsequently repeats ensuring that the isolation guarantee is maintained through the entire run.

Comparing Figure 9c and 10c, we see that with ShareGuard, Domain-1 and Domain-2 obtain more uniform service in Domain-0 even in the presence of Domain-3's workload. This is also visible in the CPU utilizations (see Figure 10b). Finally, observe that the web-server throughput for Domain-1 and Domain-2 improve significantly under ShareGuard: both web servers deliver the expected throughput.

The third column in Table 2 provides a summary of average metric values over the middle 40 second segment with ShareGuard enabled. As we can see, CPU consumption by Domain-1 and Domain-2, as well as web server throughput are similar to the baseline case. Web server performance does not degrade in presence of the bulk data transfer in Domain-3 because CPU processing in the IDD on behalf of Domain-3 is controlled by ShareGuard.

6 Discussion

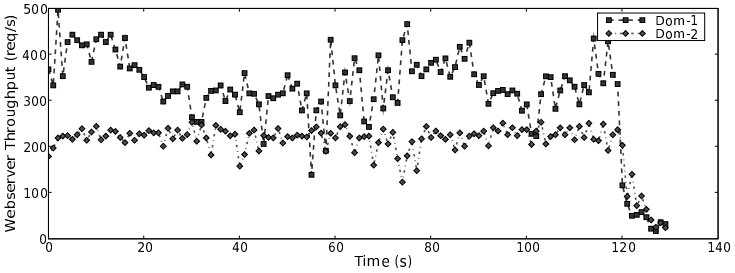
All three of the components discussed in this play important, complementary tasks in enforcing performance isolation. Both SEDF-DC and ShareGuard depend on XenMon for detailed CPU utilization information. While ShareGuard is only relevant for workloads involving network I/O, SEDF-DC is agnostic to the choice of workloads — it only depends on accurate feedback on CPU utilization from XenMon.

However, SEDF-DC can only enforce guarantees on the aggregate CPU consumption of a guest and its IDD — it does not consider fair allocation of the driver domain's finite CPU resources. ShareGuard can be used to enforce such limits for networking workloads. Further, ShareGuard works irrespective of the choice of CPU scheduler. An artifact of Xen's current CPU schedulers in Xen is that SEDF-DC only works for single processor systems. ShareGuard, however, supports multi-processor systems as well. We expect that this limitation will be removed with future releases of Xen.

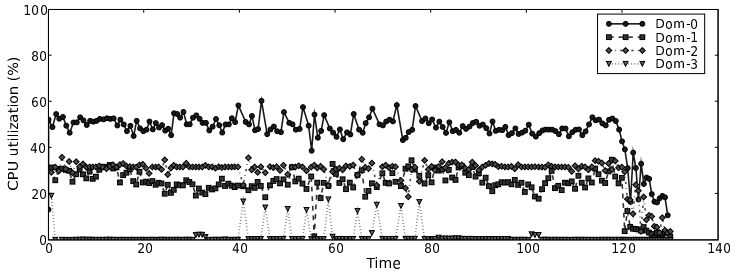
Finally, ShareGuard is more intrusive in the sense that it actively blocks a VM's traffic. In comparison, SEDF-DC is more passive and transparent. Also, as shown in Section 5, CPU allocation in ShareGuard is more bursty than in SEDF-DC (compare Figures 8c and 10c). All this underscores the fact that while on its own no single mechanism is perfect, working together they form a complete system.

7 Related Work

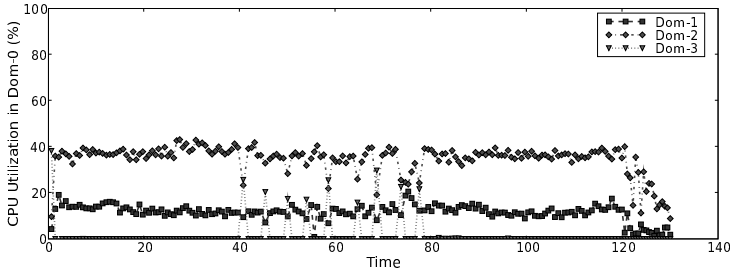
The problem of resource isolation is as old as time sharing systems. Most of the previous work in this area has focused on resource isolation between processes in an operating system or users on a single machine. In these systems, scheduling and resource management primitives do not extend to the execution of significant parts of kernel code.



(a) Webserver Throughput



(b) CPU Utilization



(c) CPU Utilization in Dom-0

Fig. 10: With ShareGuard

An application has no control over the consumption of many system resources that the kernel consumes on behalf of the application.

Consider network-intensive applications: most of the processing is typically done in the kernel and the kernel generally does not control or properly account for resources consumed during the processing of network traffic. The techniques used in ShareGuard have been inspired by earlier work addressing this problem with respect to receive livelocks in interrupt based networking subsystems. Mogul et al. [17] restrict the amount of I/O processing that the kernel does on behalf of user processes. In Lazy Receiver Processing [9] (LRP), the system uses better accounting information (such as hardware support for identifying which process an incoming packet is destined to) to improve resource isolation, e.g., such that packet processing on behalf of one process does not adversely affect the resource available to other processes.

Some of the ideas motivating LRP were extended to Resource Containers [12]. A resource container is an operating system abstraction to account for all system resources consumed by an *activity*, where an activity might span multiple processes. Resource Containers separate the notion of resource principal from threads or processes and provide support for fine-grained resource management in operating systems. This distinction between a *protection domain* and a *resource principal* is also visible in Xen's new I/O model: a VM (the protection domain) may request service from several different IDDs, therefore the tracking of its resource usage needs to span across executions of all these domains.

One limitation of Resource Containers is that they only work for single processor systems. There does not seem to be any straightforward way of extending the notion of an *activity* to span multiple processors. This is further complicated by the fact that in most operating systems, each CPU is scheduled independently. SEDF-DC scheduler suffers from the same limitation. However, ShareGuard is both scheduler agnostic and it fully supports multi-processor systems.

The problem of performance isolation has been actively addressed by multimedia systems [8, 18]. The Nemesis operating system [8] was designed to provide guaranteed quality of service (QoS) to applications. Nemesis aims to prevent *QoS crosstalk* that can occur when the operating system kernel (or a shared server) performs a significant amount of work on behalf of a number of applications. One key way in which Nemesis supports this isolation is by having applications execute as many of their own tasks as possible. Since a large proportion of the code executed on behalf of an application in a traditional operating system requires no additional privileges and does not, therefore, need to execute in a separate protection domain, the Nemesis operating system moves the majority of operating system services into the application itself, leading to a vertically structured operating system. QoS crosstalk can also occur when there is contention for physical resources, and applications do not have guaranteed access to the resources. Nemesis provides explicit low-level resource guarantees or reservations to applications. This is not limited simply to CPU: all resources including disks [19], network interfaces [20], and physical memory [21] – are treated in the same way.

The networking architecture of Nemesis still has some problems related to the charging of CPU time to applications. When the device driver transmits packets for an application, used CPU time is not charged to the application but to the device driver. Also, the handling of incoming packets before de-multiplexing it to the receiving application is charged to the device driver. We observe the same problem in the context of Xen VMM and the network driver domains, and suggest possible solution to this problem.

Exokernel [22] and Denali [23] provide resource management systems similar to vertically structured operating systems. The design goal for Exokernel was to separate protection from management. In this architecture, a minimal kernel — called Exokernel — securely multiplexes available hardware resources. It differs from the VMM approach in that it *exports* hardware resources rather than emulates them. VMMs have served as the foundation of several "security kernels" [24–27]. Denali differs from these efforts in that it aims to provide scalability as well as isolation for untrusted code, but it does not provide any specialized for performance isolation.

Most of the earlier work on VMMs focused on pursuing OS support for isolating untrusted code as a primary goal. While there is significant work on resource management in traditional operating systems, relatively less work has been performed in the context of virtual machines. Waldspurger [28] considers the problem of allocating memory across virtual machines; other systems such as Denali [23], HP SoftUDC [2] and Planetlab vServers [29] have also touched on some of these issues. Our work takes another step towards a general framework for strict resource isolation in virtual machines by considering the auxiliary work done on behalf of a guest in privileged or driver domains.

8 Conclusion and Future Work

Virtualization is fast becoming a commercially viable alternative for increasing system utilization. But from a customer perspective, virtualization cannot succeed without providing appropriate resource and performance isolation guarantees. In this work, we have proposed two mechanisms – SEDF-DC and ShareGuard – that improve CPU and network resource isolation in Xen. We demonstrated how these mechanisms enable new policies to ensure performance isolation under a variety of configurations and workloads.

For future work, we plan to extend these mechanisms to support other resources such as disk I/O and memory. Work is also underway on a hierarchical CPU scheduler for Xen: currently Xen ships with two CPU schedulers, but the choice of scheduler has to be fixed at boot time. We expect that in the future, many more CPU schedulers will become available (SEDF-DC being among the first), and that having a hierarchical scheduler that allows the use of different schedulers for different domains depending on the kinds of applications and workloads that need to be supported will enable more efficient resource utilization.

We believe that performance isolation requires appropriate resource allocation policies. Thus, another area for future investigation is policies for efficient capacity planning and workload management.

References

1. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: Proc. of the 19th ACM SOSP, New York, NY (2003)
2. Kallahalla, M., Uysal, M., Swaminathan, R., Lowell, D.E., Wray, M., Christian, T., Edwards, N., Dalton, C.I., Gittler, F.: SoftUDC: A software based data center for utility computing. IEEE Computer (2004)
3. The Oceano Project. `http://www.research.ibm.com/oceanoproject/index.html`: Last accessed 1/17/2006.

4. Fraser, K., Hand, S., Neugebauer, R., Pratt, I., Warfield, A., Williamson, M.: Reconstructing I/O. Technical Report UCAM-CL-TR-596, University of Cambridge (2005)
5. Cherkasova, L., Gardner, R.: Measuring CPU Overhead for I/O Processing in the Xen Virtual Machine Monitor. In: Proc. of USENIX Annual Technical Conference. (2005)
6. Chase, J.S., Levy, H.M., Feeley, M.J., Lazowska, E.D.: Sharing and protection in a single-address-space operating system. ACM Trans. Comput. Syst. **12**(4) (1994) 271–307
7. Jones, M.B., Leach, P.J., Draves, R.P., J. S., .I.B.: Modular real-time resource management in the Rialto operating system. In: Proc. of the 5th HotOS, Washington, DC, USA, IEEE Computer Society (1995) 12
8. Leslie, I.M., McAuley, D., Black, R., Roscoe, T., Barham, P.T., Evers, D., Fairbairns, R., Hyden, E.: The design and implementation of an operating system to support distributed multimedia applications. IEEE Journal of Selected Areas in Communications **14**(7) (1996)
9. Druschel, P., Banga, G.: Lazy receiver processing (LRP): a network subsystem architecture for server systems. In: Proc. of the second USENIX OSDI. (1996) 261–275
10. Bruno, J., Gabber, E., Ozden, B., Silberschatz, A.: The Eclipse Operating System: Providing Quality of Service via Reservation Domains. USENIX Annual Technical Conference (1998)
11. Verghese, B., Gupta, A., Rosenblum, M.: Performance isolation: sharing and isolation in shared-memory multiprocessors. In: Proc. of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, New York, NY, USA, ACM Press (1998) 181–192
12. Banga, G., Druschel, P., Mogul, J.C.: Resource Containers: a New Facility for Resource Management in Server Systems. In: Proc. of the third USENIX OSDI, New Orleans, Louisiana (1999)
13. Gupta, D., Gardner, R., Cherkasova, L.: XenMon: QoS Monitoring and Performance Profiling Tool. Technical report, HPL-2005-187 (2005)
14. Httperf. `http://www.hpl.hp.com/research/linux/httperf/`
15. `http://www.lartc.org/howto/`: Last accessed 04/02/2006.
16. `http://www.netfilter.org`: Last accessed 04/02/2006.
17. Mogul, J.C., Ramakrishnan, K.K.: Eliminating receive livelock in an interrupt-driven kernel. ACM Trans. Comput. Syst. **15**(3) (1997)
18. Yuan, W., Nahrstedt, K.: Energy-efficient soft real-time cpu scheduling for mobile multimedia systems. In: Proc. of the 19th SOSP, New York, NY, USA, ACM Press (2003) 149–163
19. Barham, P.: A Fresh Approach to File System Quality of Service. In Proc. of NOSSDAV (1998)
20. Black, R., Barham, P., Donnelly, A., Stratford, N.: Protocol Implementation in a Vertically Structured Operating System. In: Proc. of IEEE Conference on Computer Networks. (1997)
21. Hand, S.M.: Self-paging in the Nemesis operating system. In: Proc. of the third USENIX OSDI, Berkeley, CA, USA, USENIX Association (1999) 73–86
22. Engler, D.R., Kaashoek, M.F., J. O'Toole, J.: Exokernel: an operating system architecture for application-level resource management. In: Proc. of the 15th ACM SOSP, New York, NY, USA, ACM Press (1995) 251–266
23. Whitaker, A., Shaw, M., Gribble, S.D.: Scale and performance in the Denali isolation kernel. In: Proc. of the 5th USENIX OSDI, Boston, MA (2002)
24. Karger, P.A.: A retrospective of the VAX VMM security kernel. IEEE Trans, on Software Engineering (1991)
25. Meushaw, R., Simard, D.: NetTop: Commercial Technology in high assurance applications. (2005)
26. Bugnion, E., Devine, S., Rosenblum, M.: Disco: running commodity operating systems on scalable multiprocessors. In: Proc. of the 16th ACM SOSP, New York, NY, USA, ACM Press (1997) 143–156
27. Creasy, R.J.: The origin of the VM/370 time-sharing system. IBM Journal of Research and Development (1982)
28. Waldspurger, C.A.: Memory resource management in VMware ESX server. In: Proc. of the 5th USENIX OSDI. (2002)
29. Chun, B., Culler, D., Roscoe, T., Bavier, A., Peterson, L., Wawrzoniak, M., Bowman, M.: PlanetLab: an overlay testbed for broad-coverage services. SIGCOMM Comput. Commun. Rev. **33**(3) (2003) 3–12