

# Supporting Soft Real-Time Tasks in the Xen Hypervisor

Min Lee<sup>\*1</sup>, A. S. Krishnakumar<sup>2</sup>, P. Krishnan<sup>2</sup>, Navjot Singh<sup>2</sup>, Shalini Yajnik<sup>2</sup>

<sup>1</sup>Georgia Institute of Technology  
Atlanta, GA, USA  
minlee@cc.gatech.edu

<sup>2</sup>Avaya Labs  
Basking Ridge, NJ, USA  
{ask, pk, singh, shalini}@avaya.com

## ABSTRACT

Virtualization technology enables server consolidation and has given an impetus to low-cost green data centers. However, current hypervisors do not provide adequate support for real-time applications, and this has limited the adoption of virtualization in some domains. Soft real-time applications, such as media-based ones, are impeded by components of virtualization including low-performance virtualization I/O, increased scheduling latency, and shared-cache contention. The virtual machine scheduler is central to all these issues. The goal in this paper is to adapt the virtual machine scheduler to be more soft-real-time friendly.

We improve two aspects of the VMM scheduler – managing scheduling latency as a first-class resource and managing shared caches. We use enterprise IP telephony as an illustrative soft real-time workload and design a scheduler *S* that incorporates the knowledge of soft real-time applications in *all* aspects of the scheduler to support responsiveness. For this we first define a *laxity value* that can be interpreted as the target scheduling latency that the workload desires. The load balancer is also designed to minimize the latency for real-time tasks. For cache management, we take cache-affinity into account for real time tasks and load-balance accordingly to prevent cache thrashing. We measured cache misses and demonstrated that cache management is essential for soft real time tasks. Although our scheduler *S* employs a different design philosophy, interestingly enough it can be implemented with simple modifications to the Xen hypervisor’s credit scheduler. Our experiments demonstrate that the Xen scheduler with our modifications can support soft real-time guests well, without penalizing non-real-time domains.

**Categories and Subject Descriptors** D.4.1 [OPERATING SYSTEMS]: Process Management – Scheduling

**General Terms** Performance, Design, Experimentation

**Keywords** Virtualization, Xen, Enterprise telephony workloads, Server consolidation, Laxity

## 1. INTRODUCTION

Virtualization-enabled server consolidation helps in the effective management of resources and the deployment of green data centers. The availability of increased computing power with the introduction of more powerful multi-core processors is accelerating this trend. Virtualization allows the sharing of the underlying physical machine resources between different virtual machines or domains, each running its own operating system. The software layer providing the virtualization is called the hypervisor or the virtual machine monitor (VMM). The VMM ensures isolation between the virtual machines and is responsible for scheduling them on the available processors. There are many popular virtualization implementations, and a popular open-source platform is Xen [1]. In the rest of this paper, our discussion will be centered around the Xen virtualization platform, although the concepts can be applied more generally to other systems also.

Virtualization has been successfully used with many different application classes. However, some applications like IP telephony media servers and audio and video servers demonstrate inadequate performance when run on virtualized platforms [2][3]. While traditionally considered real-time, such applications have enough intelligence to tolerate the lack of hard guarantees by the underlying server and network, and can be more accurately categorized as *soft real-time* applications. Although such applications do not require hard guarantees, they are unique in that they require the underlying platform to support both low latency and provide adequate computational resources *in a timely fashion* for completion of their tasks. Both these aspects are intimately intertwined with the logic of the virtual machine scheduler. While techniques have been introduced into schedulers to boost blocked domains upon receipt of an I/O event to improve latency [4] and I/O bypass techniques involving hardware support could mitigate the latency bottleneck [5], the twin requirements of low latency and adequate computational resources imply that the virtual machine scheduler must have the ability to support domains hosting such applications.

Our goal in this paper is to distill the requirements of such soft real-time application domains and design an SMP (symmetric multi-processor) scheduler *S* that can support them. We found that our requirements for soft real-time application domains could be encapsulated using a single parameter that we call *laxity* which captures the target scheduling latency desired by the domain. The concept of laxity not only provides an explicit target latency hint to the scheduler in its decision making process, but also allows it to better prioritize and load balance the domains in an SMP environment. Xen’s existing default *credit scheduler* [6] is a proportional fair share CPU scheduler that is work conserving on SMP hosts. Although our scheduler employs a different design

---

\* This work was done when Min Lee was an intern at Avaya Labs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

VEE’10, March 17–19, 2010, Pittsburgh, Pennsylvania, USA.

Copyright 2010 ACM 978-1-60558-910-7/10/03...\$10.00.

philosophy, interestingly enough it can be implemented with simple modifications to the Xen hypervisor’s credit scheduler. We mapped laxity, prioritization, and our load balancing logic onto Xen’s credit scheduler. We conducted experiments using an IP telephony workload and media server and determined that while the default credit scheduler was unable to deliver good quality audio support, the modified credit scheduler with knowledge of laxity *not only supported the soft real-time domains well, it also did not penalize the non real-time domains.*

The rest of the paper is organized as follows. In Section 2 we describe Xen’s credit scheduler, its performance with an IP telephony media workload, and establish the need for a conceptually new scheduling approach. In Section 3 we present the main components of our scheduler design and describe how they can be implemented within the framework of the credit scheduler. In Section 4 we present our experimental setup and describe our experimental results in detail in Sections 5 and 6. We present related work in Section 7 and conclude in Section 8.

## 2. THE CASE FOR A NEW SCHEDULER

We briefly describe some key components of Xen, its scheduler and our observations in running a media server virtual machine with Xen.

### 2.1 Xen: A Brief Background

At the heart of the Xen virtualization platform is the Xen hypervisor. It runs in the highest privilege level and controls the hardware. The domain (virtual machine) called Dom0 and other guest domains called DomU run above the hypervisor like an application runs on an OS. Dom0 is a special purpose domain that is privileged by Xen to directly access the hardware and create/terminate other domains

The hypervisor virtualizes the physical resources such as CPUs and memory for the guest domains. Most of the non-privileged instructions can be executed by the guest domains natively without the intervention of the hypervisor. However, the privileged instructions will generate a trap into the hypervisor. The hypervisor validates the request and allows it to continue. The guest can also use *hypercalls* to invoke functions in the hypervisor. For this, the guest OS needs to be ported to use the functionality and this porting is called *para-virtualization*. The performance of applications in the guest domain is dependent on the methods used for I/O and CPU scheduling. We briefly describe these below.

As the I/O devices are shared across all the guest domains, the hypervisor controls access to them. Xen provides a delegation approach for I/O via a split device driver model where each I/O device driver called the *backend* driver runs in Dom0. The guest domain has a *frontend* driver that communicates with the backend driver via shared memory. The split I/O model requires implementation in Dom0 and DomU. The I/O processing requires Dom0 and the guest domain to be scheduled on the CPU. Another recent approach that is getting some traction with support provided in newer I/O (especially networking) hardware is direct I/O [5]. This approach bypasses Dom0 for I/O and provides better performance. However, both approaches are sensitive to the scheduler used – the main theme of this paper – and we outline Xen’s credit scheduler below.

### 2.2 Xen’s Default Credit Scheduler

The Xen credit scheduler is designed to ensure proportionate sharing of CPUs. In the credit scheduler, each domain is assigned a parameter called the weight, and CPU resources (or *credits*) are distributed to the virtual CPUs (VCPUs) of the domains in proportion to their weight at the end of each accounting period (default 30ms). The VCPU priority (or *task* priority) is also recalculated at that time based on the credits the task has. When a task exhausts its credits, its priority is set to *over*. When it still has credits, it is set to *under*. To accommodate low latency, the scheduler supports a *boost* priority, where *blocked tasks waiting for an I/O event* are boosted upon receiving an event/interrupt [6]. A task gets boosted for a very short time (less than one tick, typically, 10ms) after which it reverts to normal priority. This simple optimization of task boosting is good in the delegation model for I/O-bound domains such as Dom0 which handles I/O for all domains, wakes up a lot and finishes its work within a very short time. The scheduler is invoked (tickled) when a task enters the boost state, and it will pre-empt the current running task (unless it is running in boost priority) and run the boosted task immediately [6]. The priorities are implemented as different regions of the same queue, with one queue of task per physical processor. The front part of the queue is *boost* priority; the next part is *under* priority, followed by *over* and *idle* priority. When some task enters the run queue, it is simply inserted into the end of the corresponding region of the queue, and the scheduler picks up the task from the head of the queue to execute.

The credit scheduler also performs load balancing between cores in an SMP environment. When a physical CPU becomes idle or has no boosted or *under* priority tasks, it checks its peer CPUs’ queues to see if there is a strictly higher priority task. If so, it steals that higher priority task.

### 2.3 A Credit Crisis: Media Server Workload

We experimentally demonstrate how an IP telephony media server application is unable to meet its performance requirements with the default credit scheduler. A detailed description of our experimental testbed and performance metrics are given in Section 4; here, we give a brief introduction for the purposes of this section. Our Xen-based system hosted several virtual machines to provide call signaling, media processing, and when needed, a virtual machine with 4 cpu-intensive tasks. Our emphasis is on the performance of the media server which takes an RTP stream from the caller, uses a standard jitter buffer, and re-encodes the stream for the callee. We exercised the system at 4 calls per second with a call hold time of 30s for a maximum of 240 streams (120 callers and 120 callees) incoming into the media server. We sample 1 in 4 calls and measure call quality with the ITU-T PESQ metric (see Section 4). The PESQ value ranges from a minimum of 0 (bad) to 4.5 (best), with a value above 4 considered to be toll-quality voice. We then plot a PESQ graph, which is a cumulative density function of PESQ values for the sampled streams. We also show a box plot of PESQ values. The box in the plot shows the median, 25<sup>th</sup> and 75<sup>th</sup> percentile, and the whiskers show the minimum and maximum values of the distribution.

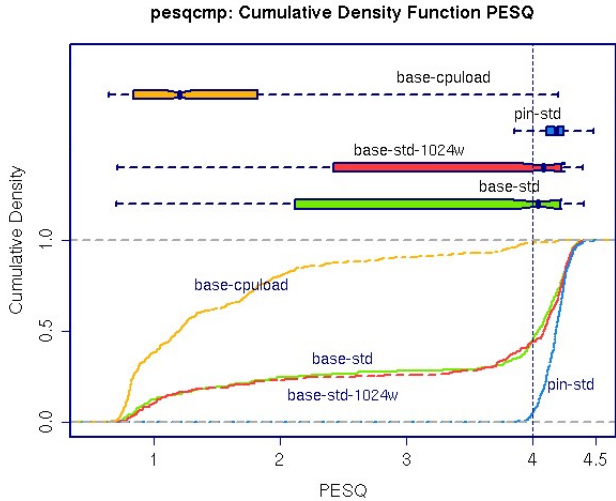


Figure 1: Voice quality (PESQ)

Figure 1 shows voice quality numbers for various configurations. We see that for the default configuration (*base-std* in Figure 1), almost 50% of the streams see poor quality (PESQ below 4). The performance is further degraded in the presence of other CPU-intensive tasks (*base-cpload*). Increasing the weight of the media server domain (*base-std-1024w*) has minimal benefit. However, by pinning the media and signaling servers to dedicated physical CPUs (and pinning other domains except Dom0 away from those physical CPUs) results in good performance (*pin-std*). Unfortunately, this method of pinning effectively “disables” the scheduler for the domain in question and has the potential to underutilize CPU resources.

## 2.4 The Reason for the Shortfall

To better understand the reasons for the media server performance issue, we instrumented Xen using *xenoprof* [7] and *xentrace* to collect statistics of interest. We observed several interesting properties of the media server domain. Firstly, the media and signaling servers do require significant amount of CPU (more than one core out of a four core machine). Second, a lot of time by the media server is spent in the *under* priority; i.e., almost 90% of the time the media server is scheduled on the CPU, it is in the *under* priority. Interestingly, Dom0 spends almost all its time in the boost priority. This clearly demonstrates that while Dom0 is mostly waiting for events, the media server is unable to get enough CPU resources to complete its task at hand in a timely fashion. Recall from Section 2.2 that a task is boosted only from blocked (or sleeping) state, not from the running state, which also accounts for the media server not getting boosted. Third, although the media server does enter the *over* queue, it does not happen significantly enough to explain the performance degradation – a fact also borne out by the lack of performance improvement with increased weight. The above observations vividly demonstrate the requirements of such soft real-time applications: they need CPU *in a timely fashion*.

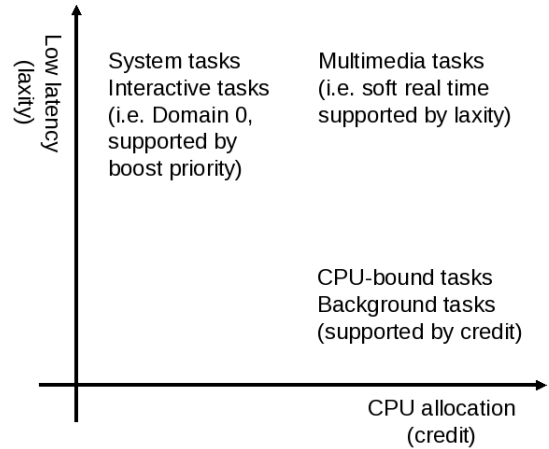


Figure 2 Scheduler Optimization: CPU vs. Latency

These observations are not surprising. Soft real-time workloads such as IP telephony media servers consume a significant number of CPU cycles while also handling many I/O events. Such a media server not only requires low latency but also a fair amount of CPU (Figure 2). Boosting the media server for short durations might, therefore, be ineffectual. Interestingly, while the soft real-time applications require CPU in a timely fashion, they do not require it immediately upon receipt of a packet. These observations provide the motivation for our soft real-time scheduler.

## 3. Design of a Soft Real-Time Scheduler, S

As discussed in Section 2, our aim in this work is to provide a framework for supporting near real-time virtual machines natively in the Xen scheduler. A multi-core SMP scheduler usually performs three main functions – (a) distributing jobs across cores for efficient use of cores, (b) within a core, prioritizing the job in such a way that some criteria (e.g. low-latency, fairness etc..) is met, and (c) cache management. Usually, the first and third functions are performed periodically by a load balancer that moves jobs between cores based on some criteria. The balancer usually takes cache affinity/cache locality into account, so that the performance of an application is optimized. The second function is performed by a per-core scheduling operation. Real-time applications like multimedia need some scheduling guarantees from the operating system and the hypervisor. Our contention is that any scheduler that aims to meet the requirements of a near real-time application needs to incorporate the notion of real-timeness in all the functions of the scheduler. Other approaches (e.g., the SEDF scheduler in Xen or the boost credit notion in [3]) and their issues are discussed in Section 7.

Our concepts in this section are explained in the context of implementation within the existing credit scheduler framework. However, the same concepts can be implemented in a new scheduler independent of the credit scheduler. We first define our *baseline scheduler* and then introduce the concepts that go into designing the new scheduler that we call S.

As discussed above, we use Xen’s default credit scheduler as the base of our implementation. However, the current implementation of the credit scheduler does not provide a very accurate accounting of credits. All the credits for a particular accounting interval get deducted from the VCPU that occupied the CPU at the end of an accounting interval. This leads to unfairness in CPU

utilization. Nishiguchi [3] modified the credit scheduler to correct this inaccuracy in the scheduler. They also modified the scheduler to balance the credits given to a domain uniformly across all VCPUs of a domain. We incorporated the above patches into the basic credit scheduler and we define this as our *baseline scheduler*.

The next few sections summarize the changes that we have made to improve the performance of the above baseline scheduler. From the user point of view, we allow a virtual machine to be categorized as “near real time” and the user specifies a single value called *laxity*. This value of laxity provides an estimate of when the task needs to be scheduled next. Most real time applications do periodic processing, and it is intuitive for a user to specify this value.

### 3.1 The Concept of Laxity – A

The basic idea behind *laxity* of a domain is to allow the real-time domain to be inserted in the middle of the scheduler’s run queue so that it can be scheduled within its desired deadline. For this, a new parameter called ‘laxity’ is defined. Soft real-time domains are given a finite positive value for laxity. In contrast, non-real-time domains are not given a laxity value and conceptually have a laxity value of infinity. When a VCPU of a real-time domain is inserted into the run queue, it is inserted where its deadline is expected to be met.

When a VCPU of a real-time domain enters the run queue we need to determine which position to insert it in the queue such that it can get scheduled before its deadline. This means that we need to predict the expected wait times of all the VCPUs in the queue. For this purpose, we use time slice length (we refer to it as the expected run time). Each VCPU maintains an expected run time which is the amount of CPU time it utilized in its previous run cycle (previous time slice length). Note that other policies for estimating the expected run time of a VCPU can also be used: e.g., average or moving average of last few run times, with exponential or uniform weights for these run times. We use this expected run time information to predict the wait time of any task in the run queue. Since tasks tend to repeat their CPU usage, this simple approach of estimating wait times works reasonably well, as we will see from Figure 3.

The box plot in Figure 3 shows the difference between the expected and the actual wait times for each VCPU of all domains. The Y-axis gives the difference in microseconds and the X-axis gives the VCPUs for all domains. (The names of the domains are explained later in Section 4.) As can be seen from the box plot, the difference between expected and actual wait times for each VCPU is approximately within 100µs.

Figure 4 gives an example of how this insertion happens. If a new real-time VCPU is inserted and its laxity value is 5µs, it would be placed between VCPU4 and VCPU2. If its laxity value is 20µs, it would be placed between VCPU7 and VCPU1 where its expected wait time would be 12µs. So the expected wait time of a VCPU in the run queue is the sum of the expected run times of each VCPU ahead of this VCPU in the run queue.

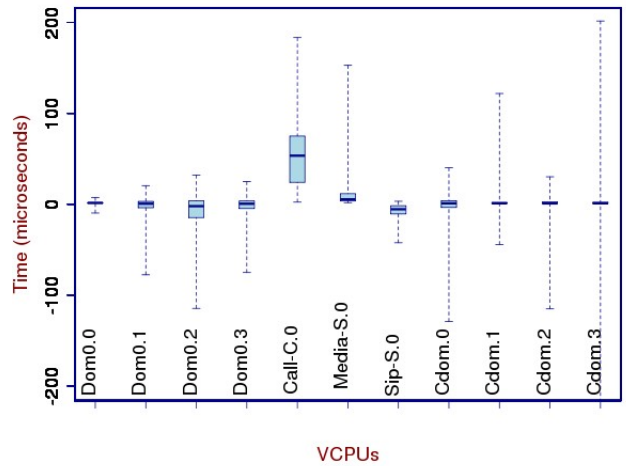


Figure 3: Average difference between expected and actual wait times for each domain

However, if the position in the queue where the VCPU would get inserted without taking laxity into account is ahead of the target position based on the laxity, then the VCPU is inserted at the earlier position. As an example, if the laxity value is 20µs but the VCPU has a boost priority, it is placed between VCPU4 and VCPU2 (at the end of the boost portion of the queue), instead of being placed between VCPU7 and VCPU1 based on the laxity calculations.

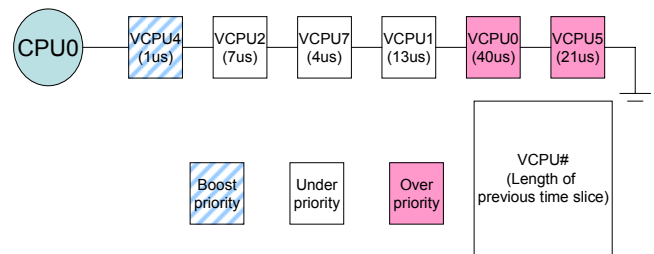


Figure 4: Run queue for CPU 0

It is important to observe that the scheduler *S does not change the credit distribution mechanism* of Xen’s credit scheduler. This automatically prevents starvation, because at every accounting interval (typically 30ms), credits are distributed to tasks and the real time tasks would also eventually consume its credits. Since the credit scheduler has an upper bound on the total credits it distributes, the system remains stable.

### 3.2 Boost with event – B

The credit scheduler provides a mechanism to achieve low I/O response latency through the use of a boost priority. The VCPUs that receive external events are assigned a boost priority that is higher than *under* and *over* priorities. However, the current implementation of the credit scheduler assumes that if a VCPU of a domain is waiting for an event it will be in blocked (or sleeping) state. Hence, upon the arrival of an external event, the scheduler only boosts VCPUs that are in blocked state. This is sufficient for

a domain that is mostly I/O-intensive (such as Dom0). This allows Dom0 to respond quickly to external interrupts since it is usually in blocked state and gets boosted upon arrival of an I/O event. However, if in addition to being I/O intensive, a domain also needs to consume significant amount of CPU for processing each event, it will go back into the run queue after the event is processed. If another event for the same VCPU arrives, the VCPU will not be boosted since it may reside in the *under* queue rather than in a blocked state. The event will get delivered to the VCPU only when it gets scheduled next thereby increasing the latency of response time. To overcome this problem, our scheduler boosts real-time VCPUs even when they reside with *under* priority in the run queue as shown in Figure 5.

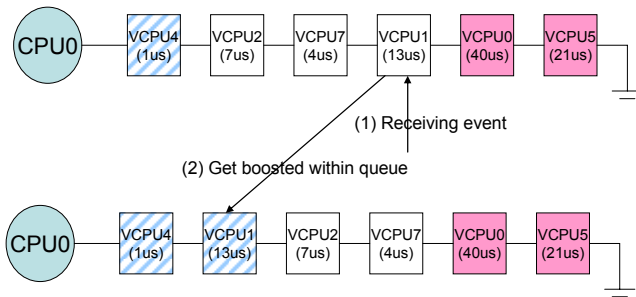


Figure 5: Boosting a VCPU upon event arrival

### 3.3 Multi-core Load Balancing

In a multi-core platform, load balancing tasks across cores is crucial for efficient utilization of all cores. Most schedulers also follow affinity scheduling, where tasks prefer to remain on the same cores to improve cache locality. The algorithms that we provide in this section aim to incorporate the support for real-time applications in load balancing and cache management. We propose first a simple load balancing algorithm *L* that does not do cache management and then introduce an enhanced algorithm *LL* that takes into account cache affinity and its impact on performance of real-time tasks.

#### 3.3.1 Basic Load Balancing *L*

The initial load balancing algorithm *L* is a simple modification to the current load-balancer implemented in the credit scheduler and aims to improve the overall wait-time of all the tasks in the run queues. In the current incarnation of the credit scheduler, if the next task in a CPU's run queue is of *over* or *idle* priority, the scheduler running on the CPU checks a peer CPU's queue to see if the queue has a strictly higher priority job. If there is such a job in the queue, it steals that higher priority task. It only steals work when its queue does not have any boost or *under* priority jobs. This may result in limiting the responsiveness of boosted or real-time tasks.

In our simple load balancing algorithm *L*, a CPU may also steal work from other queues even if its next task is in the *under* queue or the two tasks being compared are of same priority. This is because a peer CPU's queue may have some real-time or boosted tasks waiting. However, when both tasks are of the same priority and the next task in the current CPU's queue is a real-time task, algorithm *L* does not steal the peer's task. On the other hand, when both tasks are of same priority and the peer's task is a real-time task, the algorithm steals it. This decreases the wait time of

boosted tasks and real-time tasks. In the case when both tasks are non-real-time and of the same priority, the algorithm compares when they entered the run queue and steals only if the peer's task has been waiting longer in the queue. The VCPUs in the run queue are time-stamped when they enter the queue. However, to prevent excessive work stealing (which may thrash the caches), we add a threshold delay value. Our measured wait-times in the queue suggest 2ms-5ms wait times and we use 2 ms as the threshold delay value. If the peer's next task entered the run queue at least 2ms before next task of a given CPU, the CPU steals the peer's next task.

This basic load balancing results in an overall decrease in the wait time in *under* queue. In addition, the above strategy plays an important role in preventing starvation and reducing the wait times for non-realtime tasks. Because the real time tasks tend to run periodically, when a number of real time tasks are running on one CPU, they may always come back to the run queue ahead of the non-realtime tasks, thus causing starvation of the non-realtime tasks. The above load balancing strategy allows other peers to steal such long-waiting non-real-time tasks and limiting the wait times of such tasks.

#### 3.3.2 Cache-Aware Real-time Load Balancing *LL*

The basic load-balancing algorithm *L* described above, effectively distributes real-time tasks over CPUs and decreases the average wait time. However, it has a major drawback. It does not take a task's cache affinity into account and may result in low performance for real-time applications due to cache trashing. In this section, we enhance the basic load balancing algorithm with cache management for real-time applications.

Cache affinity of a task is best if it is bound to a CPU and allowed to stay on the CPU for long periods of time. If the CPU usage of applications is known apriori, one method used in the literature is to compute upfront the load caused by each task and allocate the tasks to CPUs such that the CPU resources are most efficiently used. This is possible only if the behavior of all applications in a VM is known apriori. However, since characteristics of application running inside a VM may vary over time, the initial allocation may not be good enough and it may become necessary to move them across cores for more efficient use of CPU resources. The solution presented here tries to balance the two requirements of efficiency and cache affinity. The algorithm *LL*, load balances real-time tasks at a coarse granularity to achieve maximum benefits from both balancing the load and maintaining cache affinity. The load balancing algorithm *LL* binds the real-time tasks to a CPU, such that they do not get migrated by the simple load balancer *L*. To achieve this, *L* is modified not to steal peer's task if it is real-time task. In addition, for real-time tasks, we also disable the default strategy used by the scheduler to check at every tick (10ms) if the task has a better place to run.

The new real-time load balancer for our scheduler *S* is *Algorithm LL*, which is activated every *x* seconds and determines the best CPU allocation for each real-time task. So the real-time tasks move across CPUs at a coarser granularity than the non-realtime tasks. Note that only real time tasks are bound to a CPU in *LL* while migration of non-realtime tasks is still controlled by the basic load-balancer *L*.

In our experiments, we found *x* as 1 second to be an appropriate choice, although the value of *x* could also be chosen dynamically.

The aim of the one second load balancer is to achieve maximum CPU utilization while not impacting the real-time responsiveness negatively. The policy used for load balancing real-time applications tries to balance the real-time workload based on their CPU usage history. The policy aims to uniformly distribute the total CPU utilization incurred by *real-time tasks* over all CPUs. If the number of real-time tasks is less than the number of physical CPUs, use of the load balancer will cause each real-time task to migrate to a CPU and affinitize itself to that CPU such that there is no more than one real-time task per CPU. However, when there are more real-time tasks than physical CPUs, there could be an imbalance across all CPU's real-time task allocation. The load balancing algorithm aims to correct this imbalance.

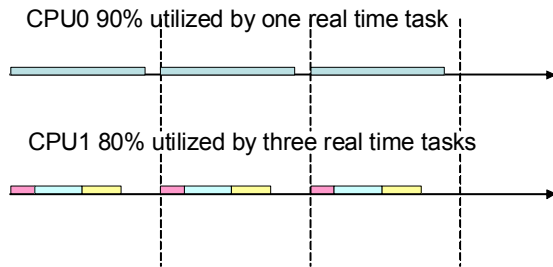


Figure 6: Load balancing based on CPU utilization

Figure 6 shows the target load balance that we would like to achieve. The load balancing algorithm tracks each VCPU's CPU utilization over time. In addition, for each physical CPU it also tracks the total CPU utilization and the corresponding breakdown for real-time and non-realtime tasks. The load balancer then uses a bin-packing algorithm to distribute the real-time tasks based on their respective CPU utilizations. The load balancing is done incrementally by looking for candidate real-time jobs whose migration would result in a more balanced situation. For simplicity we choose only one candidate at a time and move the job from the most utilized CPU (in terms of the real-time tasks) to least utilized CPU (again, in terms of the real-time tasks). Note that the granularity of load balancing does not have to be one second and can also be modified dynamically to meet the requirements of the system.

Before we evaluate the impact of all the above components on the performance of the media server, we introduce below the experimental setup used for the evaluation.

#### 4. Experimental Setup

Our experimental platform consisted of a Dell 2950 server with 2 quad-core Xeon processors and 4 GB of RAM running Xen. In addition, we had a set of lower-end Linux desktops that could generate traffic directed at the server. At any point of time in our experiments, we only used 4 processor cores from the available 8 cores to exercise two cache configurations: the *non-shared cache* configuration and the *shared cache* configuration. In the first configuration, the 4 chosen cores (2 from each socket) did not share any cache, and in the second configuration, the four cores were from one CPU socket and a pair of cores shared a cache.

The workload that we ran on the Xen server consisted of several virtual machines. (See Figure 7.)

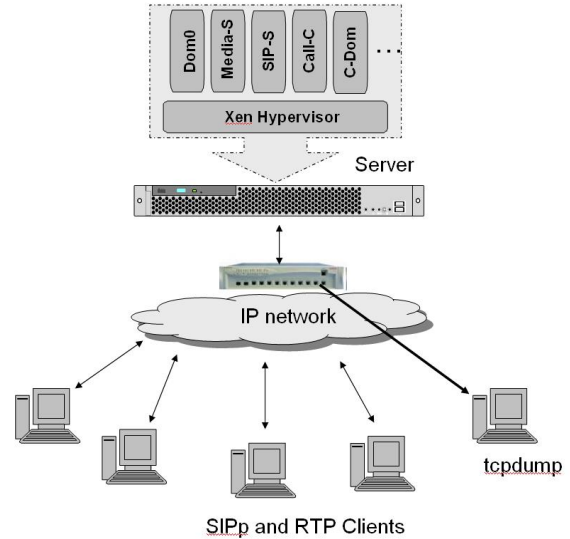


Figure 7: Experimental Setup

The main virtual machine was the media server, Media-S, that takes a voice stream, processes it (including transcoding, if needed), and then sends the stream to the recipient. The media server is a near-real-time server and requires low latency to process its voice streams. At the same time it also consumes CPU resources to process the streams. The two other main servers are the SIP server (Sip-S) and the Call Control server (Call-C). Together, the Sip-S and Call-C servers deal with end-point registration, call setup and teardown, and media port allocation/deallocation on Media-S. The signaling traffic goes through Sip-S and Call-C, while the media (RTP) traffic flows through Media-S. We call this the *standard* configuration or workload. In addition, we have a Computational Domain (C-Dom) that hosts up to 4 CPU intensive tasks. When the CPU tasks are activated, we call the resulting configuration as the *cpuload* configuration or workload. (In practice, there are two more domains responsible for licensing and management on the system; however, in our experiments, these two domains do not have any execution or traffic of note and we ignore them for the rest of this presentation.)

Our experiments were run by exercising the system at 4 calls per second with RTP traffic at 20ms packetization using the G.711 codec. The signaling was initiated from one of the Linux desktops using SIPp [8], and a 30-second reference waveform was sent to the media server after call establishment. The call hold time was 30 seconds. The parameters above imply that there were at most 240 RTP streams flowing into the media server at any point in time (120 streams from the caller and 120 streams from the callee). We ran each of experiments for 5 minutes, and repeated each experiment three times. We sampled 1 in 4 calls (from the caller side) for our metric of interest: the quality of the audio stream. The sampling was done by tapping an ethernet switch and recording packets via tcpdump on a dedicated Linux workstation.

Our main metric of measurement was the call audio quality and for this we use the ITU-T specified PESQ standard measure [9]. PESQ is a metric to measure the quality of voice, and it does so by comparing the original wave form from the caller to that received by the callee. The PESQ value ranges from a minimum of 0 (bad) to 4.5 (best). Typically, a value above 4 is considered

toll-quality. For each sampled stream, we measure its PESQ value. We then plotted a PESQ graph, which is a cumulative density function of PESQ values for the sampled streams. Where relevant, we also show a box plot of the PESQ values obtained, which more clearly reveals the spread of PESQ values. The box plot shows the median, and the box around it shows the 25 and 75 percentile values. The whiskers show the minimum and maximum values.

When run with the CPU intensive tasks (cpuload configuration), we computed the amount of work done within the CPU intensive tasks to evaluate how the scheduler behaved with non real-time domains.

### 5. Results: Non-Shared Cache Architecture

First we discuss the overall summary of the results and then go into the impact of each component of the scheduler (discussed in Section 3) on our primary metric of interest – voice quality as measured by PESQ.

Figure 8 shows the overall performance of the scheduler (referred to as ALL with laxity of 10µs for real-time domains) with respect to the baseline scheduler (referred to as base) and the best results obtained by pinning the real-time workload (referred to as pin). Figure 8 shows that, in the standard workload case, the scheduler matches the good performance achieved by the pinned case. Adding more CPU-intensive workload to the system yields poor performance for the baseline scheduler.

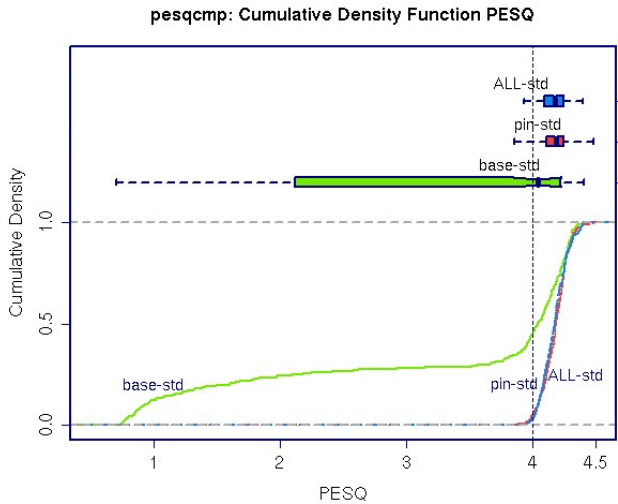


Figure 8: PESQ standard configuration - summary

We also studied the impact of the scheduler on non-real-time workloads. The results show that the scheduler is good with respect to improving overall CPU utilization as compared to the case where resources are pinned /reserved for the real-time tasks.

Figure 9 gives the amount of work completed by four non-real-time CPU-intensive jobs added to the system in C-Dom. The four histograms in the figure give the number of iterations of computations performed by each job. As can be seen from the figure, scheduler S shows almost 100% improvement in workload completion over the pinned case and matches the default credit scheduler performance.

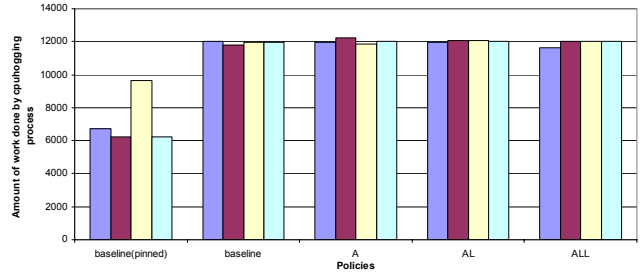


Figure 9: Work completion for the CPU-intensive non-realtime jobs

The results above show that the overall performance of the scheduler is good with respect to both (a) giving enough resources to real-time tasks and (b) at the same time preventing starvation of non-realtime tasks. Next, we will study how individual components of the scheduler impact the performance of real-time and non-realtime applications.

#### 5.1 Effect of Laxity

Figure 10 shows the improvement obtained in voice quality from the laxity component of the scheduler for the standard workload. As shown in the figure, the A-std (baseline + laxity-based queuing) scheduler improves the voice quality over the baseline case. This improvement is significant in the case when the media server runs along with the CPU-intensive workloads (A-cpload vs. base-cpload).

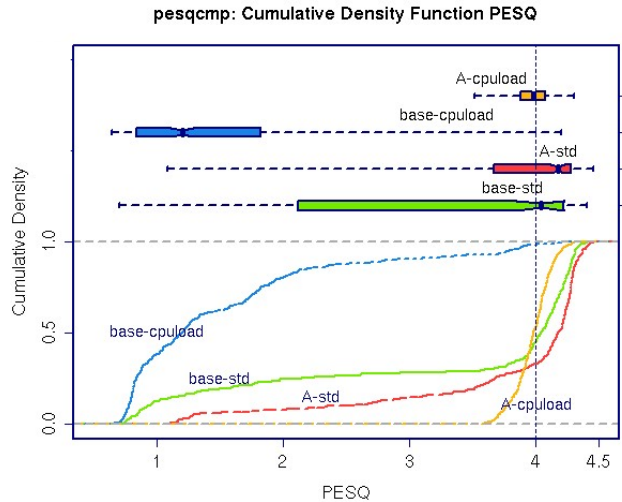


Figure 10: PESQ comparison for Laxity

In order to understand the behavior of scheduler in the two cases, we instrumented the scheduler to collect metrics like the (a) the number of times that a VCPU entered a particular queue, (b) average wait time in the queue., (c) the amount of time that the VCPU ran after being scheduled from a particular queue. The graphs below show two of these metrics for the base-cpload case and the laxity scheduler (A-cpload) case.

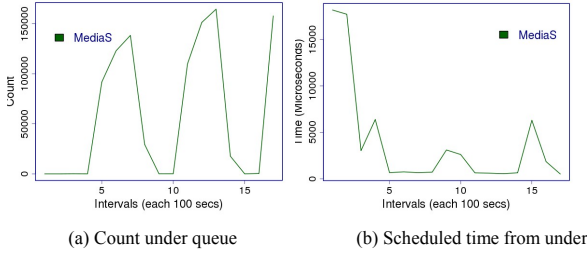


Figure 11: Base-Cpload: Detailed queue metrics

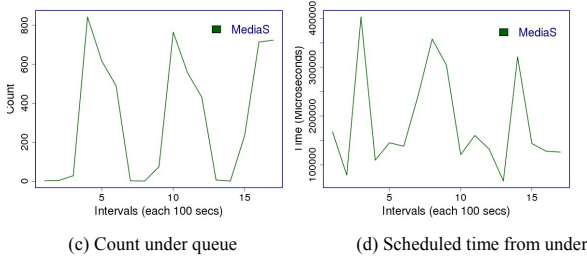


Figure 12: A-Cpload: Detailed queue metrics

In the graphs in Figure 11 and Figure 12 *count under queue* (UC) is the number of times a VCPU entered the run queue with *under* priority. The *scheduled time from under* (UT) is the average number of CPU time slices that the VCPU got scheduled for while it had a priority of *under*. Figure 11 shows the value of the metrics across time for the base-cpload case and Figure 12 shows the values for the laxity component of scheduler S. The X-axis in all these graphs shows the time for the experiments and the Y-axis shows the value of the corresponding metric. Figure 11(a), (b) show that in the base-cpload case, the media server VCPU enters the run queue with *under* priority very often, but gets scheduled for only small time slices. Figure 12(a),(b) show that the media server enters into the run queue with *under* priority less often (note the scale difference), but the time slice it gets every time that it is scheduled is much more. In the two cases, the total CPU utilization in *under* priority queue (product of UC and UT) was roughly the same; however, the ability to get more CPU with each scheduling cycle allowed the media server to process its workload in a more timely manner and hence achieve good voice quality.

We notice a similar trend across all our results. If the value of UT is higher (longer time slice allowed at each scheduling cycle), in general the voice quality was better.

In Figure 13 we evaluate the dependence of performance on the value of laxity. We observe that for small laxity values, the performance of the real-time domain is good. When the laxity value is large (e.g., 40ms), as expected, the effect of laxity is diminished and no effect is noticed.

## 5.2 Effect of Boost

We study the effect of the updated boosting technique described in Section 3.2. Here, the real-time domain is boosted even when it is in the *under* priority. We observe from Figure 14 that the boost technique has some impact, but it is not very significant. We

conjecture that the most significant performance effect is due to the laxity concept.

While boosting the domain does help, providing enough CPU resources to it is essential for improved performance.

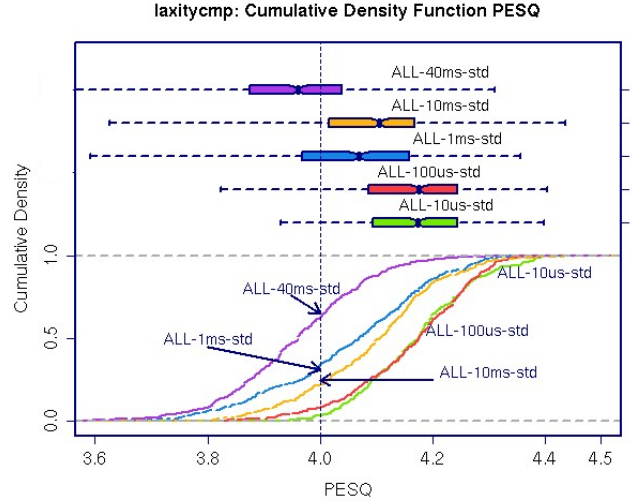


Figure 13: Comparison of PESQ with different laxity values - standard configuration

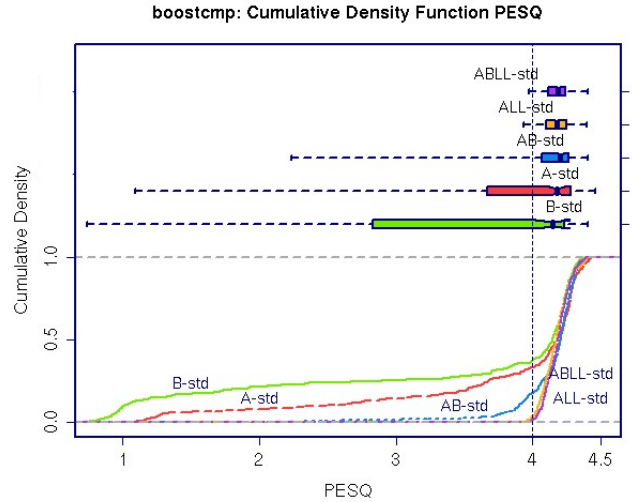
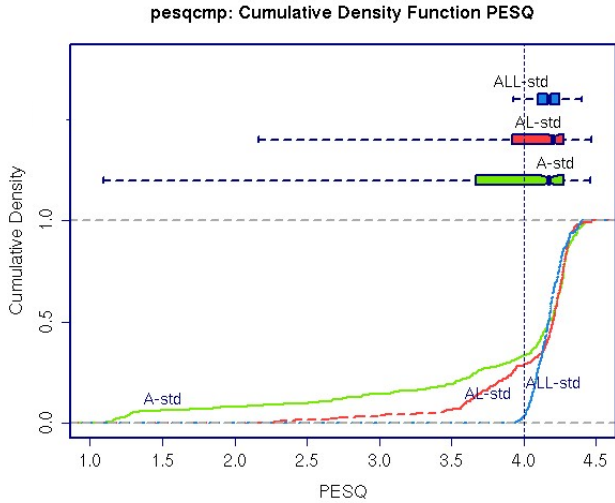


Figure 14: Adding Boost to scheduler S

## 5.3 Effect of Multi-core Load Balancing

In this section, we evaluate the impact of the load balancing component of scheduler S. As discussed in Section 3.3.1, we proposed a simple load balancing algorithm  $L$  for balancing load across CPUs based on the job priority and enhanced that simple scheme in Section 3.3.2 with  $LL$  by adding cache awareness and CPU utilization criteria for real-time tasks to the load balancer. In this section, we first evaluate the impact of algorithm  $L$  and show how  $L$  is not sufficient for meeting the requirements of real-time tasks. We then show how  $LL$  meets the needs of the real-time tasks.

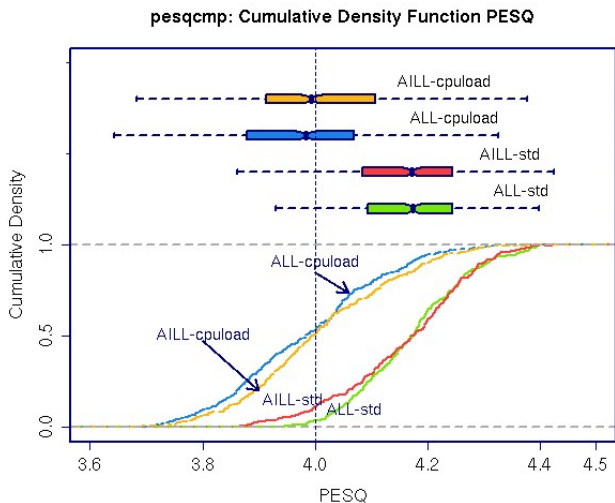


**Figure 15: PESQ improvement through load balancer in standard configuration**

Figure 15 shows the PESQ numbers for laxity component ( $A$ ) and compares the numbers when the simple load balancer  $L$  and the enhanced load balancer  $LL$  are added to the laxity component of the scheduler. This experiment was run for the standard configuration. Similar results were observed for the cpuload configuration also.

#### 5.4 Effect of Weights

In order to determine if there is room for further improvement of voice quality by giving more compute power to real-time domains, we experimented with a strategy in scheduler  $S$  that gave “infinite” credit to the real-time domains (i.e. if the domains were runnable, we let them run even if they had a priority of *over*).



**Figure 16: Impact of credits (weights) on PESQ**

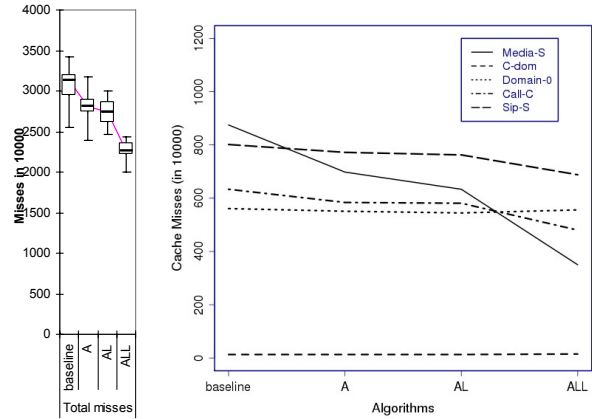
Figure 16 above shows the comparison of PESQ numbers for scheduler  $S$  ( $ALL$ ) and scheduler  $S$  with infinite credit for real-time tasks ( $AILL$ ). The numbers show no significant improvement in voice quality. We believe that this is because scheduler  $S$  already provides the media server with all the CPU needed for its

computation, and hence the ability to get even more CPU does not show any further improvement.

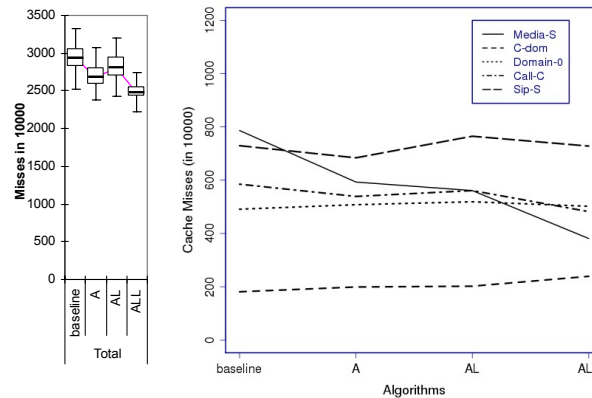
#### 5.5 Impact of S on Cache Behavior

The load balancer component  $LL$  of scheduler  $S$  brings in cache-awareness into the scheduling policies and aims to minimize the impact of migrating the real-time tasks across CPUs. In this section, we evaluate the effectiveness of the solution by measuring the cache misses for each configuration.

Figure 17 and Figure 18 show the L2 cache misses for the standard configuration and the cpuload configuration.



**Figure 17: Cache misses for standard configuration**



**Figure 18: Cache misses for cpuload configuration**

The box plots on the left show the total cache misses in units of 10000 for each component ( $A$ ,  $AL$ ,  $ALL$ ) as compared to the baseline scheduler. The  $A$ ,  $AL$  and  $ALL$  show reduced cache misses equaling 86%, 89% and 76%, respectively. The graph on the right shows the breakdown of the cache misses on a per-domain basis. As can be seen from the graph, the maximum benefit of keeping the cache hot is seen by the media server (which is a real-time domain). Cache misses for  $A$ ,  $AL$  and  $ALL$  in the media server case amount to about 77%, 73% and 37% of the baseline case. This shows that the cache-aware load balancer  $LL$  is doing a good job of maintaining the cache affinity while at the same time balancing the load. As expected, the non-real-time domains like Domain-0 do not see any significant change in cache misses.

### 5.6 Impact of S on scheduling latency

We also evaluated the impact of scheduler S on the worst and average wait times (scheduling latency) of tasks in the queue. As expected, the wait times for real time tasks were all minimized. In case of *AL*, the average wait time of real time tasks (Media-S and Call-C) were less than 17µs, mostly less than 1µs, while that of non-real time tasks ranged from 40µs to 1.3ms. The worst wait time of real time tasks were less than 2.8ms, mostly less than 200µs, while that of non-real time tasks ranged from 200µs to 30ms, mostly between 1 and 10ms. (These were measured every 100 seconds)

We could observe the same trend across all the experiments except for the *ALL* case. There was a significant increase in wait time for the *ALL* case. This is because, in order to achieve hot-cache, the *LL* component of the scheduler effectively binds the real-time tasks to a single CPU for the load balancing interval of 1 second. This means that other CPUs cannot steal the real-time task and it has to wait on its own CPU while the current task is running. Even with the increased scheduling latency, *ALL* gives better voice quality than the other cases. This shows the importance of maintaining a hot cache for achieving good performance for the media server workload.

### 6. Results: Shared Cache Architecture

As discussed in Section 3.3.2, scheduler S performs cache-aware load balancing for real-time tasks to get the full benefits of cache affinity as well as efficient CPU utilization. In order to evaluate the scheduler’s impact on different cache architectures, in addition to non-shared cache architecture above, we also studied a shared cache architecture where physical CPUs 0 and 2 share a 4 MB L2 cache and physical CPUs 1 and 3 share another 4MB L2 cache. So effectively, this halves the total cache capacity of the platform.

Figure 19 and Figure 20 show the overall voice quality results for the various components of the algorithm for the standard case and the cpuload case. As shown in the figures, scheduler S (*ALL*) improves the voice quality significantly as compared to the baseline credit scheduler in both the configurations.

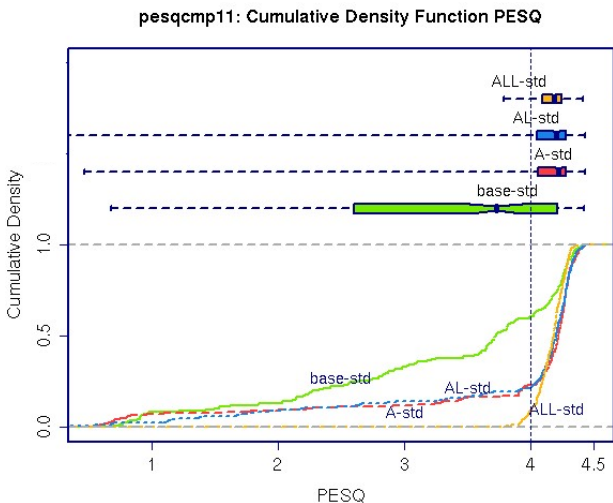


Figure 19: PESQ Shared Cache - standard configuration

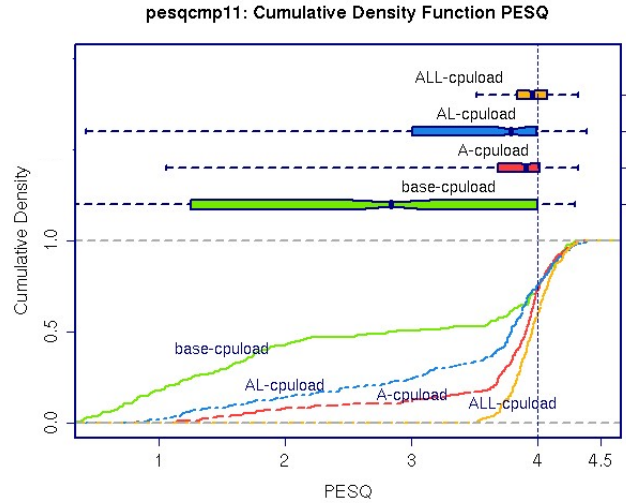


Figure 20: PESQ Shared cache: cpuload configuration

### 7. Related Work

Even though significant amount of effort has been dedicated to optimizing the performance of virtual environments [7][10][11], I/O virtualization still remains an area that poses challenges to the adoption of the technology in certain domains, like enterprise telephony [2]. The key factors affecting the performance of enterprise telephony workloads are scheduling and network I/O performance.

Research efforts such as [4][5][12] study the impact of VM scheduling on I/O virtualization performance and highlight the need for improved scheduling techniques that help I/O bound VMs while maintaining fairness. The Xen credit scheduler does a good job of fairly sharing the CPU resources among the domains that have compute intensive jobs. However, it can lead to sub-optimal results for bandwidth and latency sensitive domains. Ongaro et al. studied the impact of VMM on I/O performance [4]. To improve the I/O support of the credit scheduler, they suggested sorting the VCPUs in the run queue based on the remaining credits in addition to the existing boost credits. Govindan et al. [12] introduced an enhancement to the SEDF scheduler that gave preference to I/O domains over the CPU intensive domains. They looked at the total number of packets flowing in and out of each domain and picked the one with the highest count that had not yet consumed its entire slice in the current period.

There is comparatively lesser work in real-time scheduling for Xen. The earlier SEDF scheduler [13] in Xen allowed specifications of time slices and frequency of scheduling desired by a domain. However, SEDF suffered from several deficiencies including insufficient support for SMP. The credit scheduler is the current default scheduler for Xen. Recently, there has been work in the Xen community [3] on incorporating the support for real-time virtual machines in the credit scheduler. The premise behind the approach in [3] is to provide a boost credit to a domain in proportion to its weight. This would allow the domain to get CPU resources in the boosted state. From this perspective, the approach is not as much for real time but for boosting-up I/O-bound tasks. However, what we have noticed in our experimental results

presented here is that our real-time domain is *not even getting boosted since it is in the under priority*, and so additional boost credits would not help it. Furthermore, the boost credits tackles one aspect of the scheduling problem, namely, the per-core scheduling operation and hence does not go all the way in meeting the demands of a multimedia application. Our work aims to bridge the gap by providing a framework that builds in soft real-time support in *all aspects* of scheduling. More recently, Dunlap et al. [14] are looking into updating the Xen scheduler. There are interesting options being proposed (e.g., a queue per L2 cache, balancing run queues at fixed intervals, etc.). We hope some of our results in this paper will help in that effort.

The approach used in our paper is philosophically different from that used in other RT-schedulers in that we try to fit soft real-time applications in a generic environment with minimal administrative overhead. In contrast, most other proportional-share schedulers (BVT [15], EEVDF [16], SMART [17], SFQ [18] etc.), require some notion of virtual time and deadlines as compared to our concept of laxity which is a hint for scheduling latency. Unlike other real-time schedulers which focus mainly on RT aspects of scheduling, the notions of credit and laxity help us balance the needs of both real-time and best-effort workloads.

Liao et al. proposed an efficient I/O virtualization for the high end systems [5]. The performance improvement was achieved by offloading the virtualization functionality from the guest domain onto the device. The work in [19] proposed a task-aware VMM scheduler that took into account the I/O bound nature of guest-level tasks and correlated incoming events with I/O-bound tasks while making scheduling decisions.

There has been a significant amount of work done in improving the performance of the I/O path in Xen. Menon et al. [7] built a framework for monitoring called *xenoprof*, and using this framework identified that the network performance of a para-virtualized Linux VM running under Xen was significantly lower when compared to the native Linux execution performance. Zhang et al. [20] observed that system performance bottlenecks in hardware-assisted virtualized environment are mainly introduced by memory and I/O virtualization overheads. Several past papers [10][11][21][22][23] have addressed the problems of network I/O virtualization. The approach taken by Menon et al. [11] optimizes the I/O channel between the driver domain and guest domains and utilizes high-level network offload features present in modern network cards. Their optimizations improved the receive performance but it still did not match the native Linux performance. Oi et al. [22] analyzed and improved network throughput of a Xen virtualized system with schemes like Large Receive Offload in the virtual interfaces for packets with 4KB MTU. The work by Santos et al. [24] on improving network I/O performance uses netperf as the workload generator and studies the impact of their optimizations on large packets. There has also been recent work on improving other aspects of virtualized I/O performance [25][26]. Recently the Xen development community has introduced stub-domains [27] which aim at overcoming the problems in hardware-assisted VMs (HVMs) that arise due to Dom0 being a bottleneck for I/O intensive environments.

Apparao et al. [10] studied the performance of a server consolidation benchmark (specifically, the java, web and mail benchmarks) on a multi-core platform running the Xen virtual environment. They looked at the architectural characteristics such as CPI (cycles per instruction) and L2 cache misses per

instruction and analyzed the benefits of larger caches for these benchmarks.

## 8. Conclusion

In this paper, we have identified the area of soft real-time application domains and the performance problems they encounter in virtualized environments. The solution to these problems lies in the hypervisor's scheduler. We have proposed a new scheduler that manages scheduling latency as a first-class resource and is also intelligent in its scheduling based on preserving cache coherency. The scheduler incorporates the knowledge of soft real-time applications in all aspects of the scheduler to support responsiveness. The scheduler would remove the current practice of achieving performance by pinning domains to cores, which underutilizes available CPU capacity. We have demonstrated how our scheduler can be implemented with simple modifications to the Xen hypervisor's credit scheduler. We have implemented our scheduler and conducted experiments with an enterprise IP telephony workload. Our experiments have demonstrated that the Xen scheduler with our modifications can support soft real-time guests well, without penalizing non-real-time domains.

**Acknowledgments:** We thank the anonymous reviewers for their comments. Min Lee thanks Avaya Labs for their support and also thanks his lovely wife, Hyunsuk Kim, for her support.

## REFERENCES

- [1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt and A. Warfield, "Xen and the art of virtualization," in ACM SOSP 2003.
- [2] D. Patnaik, A.S. Krishnakumar, P. Krishnan, N. Singh, S. Yajnik, "Performance implications of hosting enterprise telephony applications on virtualized multi-core platforms," in IPTComm 2009.
- [3] N. Nishiguchi, "Evaluation and consideration of the credit scheduler for client virtualization," Xen Summit Asia 2008.
- [4] D. Ongaro, A. L. Cox, and S. Rixner, "Scheduling I/O in virtual machine monitors," in VEE '08: Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, pp. 1-10, 2008.
- [5] G. Liao, D. Guo, L. Bhuyan, and S. R. King, "Software techniques to improve virtualized IO performance on multi-core systems," in ANCS '08: Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, pp. 161-170, 2008.
- [6] E. Ackaouy, "New CPU scheduler with SMP load balancer," <http://lists.xen-source.com/archives/html/xen-devel/2006-05/msg01315.htm>
- [7] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel, "Diagnosing performance overheads in the Xen virtual machine environment," in VEE '05: Proceedings of the 1st ACM/USENIX International Conference on Virtual execution environments, pp. 13-23, 2005.
- [8] SIPp, <http://sipp.sourceforge.net/>
- [9] PESQ, <http://www.itu.int/rec/T-REC-P.862/en>, ITU-T Recommendation P.862, "Perceptual evaluation of speech

quality (PESQ): An objective method for end-to-end speech quality assessment of narrow-band telephone networks and speech codecs”.

- [10] P. Apparao, R. Iyer, X. Zhang, D. Newell, and T. Adelmeyer, “Characterization & analysis of a server consolidation benchmark,” in VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual Execution Environments, pp. 21-30, 2008.
- [11] A. Menon, A. L. Cox, and W. Zwaenepoel, “Optimizing network virtualization in Xen,” in ATEC '06: Proceedings of the USENIX '06 Annual Technical Conference, 2006.
- [12] 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 VEE '07: Proceedings of the 3rd International Conference on Virtual Execution Environments, pp.126-136, 2007.
- [13] SEDF Scheduler: Xen wikpage on Xen scheduling, <http://wiki.xensource.com/xenwiki/Scheduling>.
- [14] G. Dunlap, Planned csched improvements and credit2, Xen mailing list, <http://lists.xensource.com/archives/html/xen-devel/2009-10/msg00442.html>
- [15] K. Duda and D. Cheriton, “Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general purpose scheduler,” in ACM SIGOPS Operating Systems Review, 33 (5), December 1999.
- [16] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, G. Plaxton, “A proportional share resource allocation algorithm for real-time, time-shared systems,” Proc. 17<sup>th</sup> IEEE Real Time System Symposium, December 1996.
- [17] J. Nieh and M. Lam, “A SMART scheduler for multimedia applications,” in ACM Transactions on Computer Systems, vol. 21, no. 2, May 2003, pp. 117–163.
- [18] P. Goyal, H. Vin, H. Chen, “Start-time fair queuing: A scheduling algorithm for integrated service packet switching networks,” in Proc. of the ACM SIGCOMM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications, August 1996.
- [19] H. Kim, H. Lim, J. Jeong, H. Jo, J. Lee, “Task-aware virtual machine scheduling for I/O performance,” VEE 2009, pp. 101-110.
- [20] X. Zhang and Y. Dong, “Optimizing Xen VMM based on Intel virtualization technology,” in ICICSE '08: Proceedings of the 2008 International Conference on Internet Computing in Science and Engineering, pp. 367-374, 2008.
- [21] P. Apparao, S. Makineni, and D. Newell, “Characterization of network processing overheads in Xen,” in VTDC '06: Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing, p.2, 2006.
- [22] H. Oi and F. Nakajima, “Performance analysis of large receive offload in a Xen virtualized system,” International Conference on Computer Engineering and Technology, vol. 1, pp. 375-480, 2009.
- [23] P. Willmann, J. Shafer, D. Carr, A. Menon, S. Rixner, A. L. Cox, and W. Zwaenepoel, “Concurrent direct network access for virtual monitors,” in HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture, pp. 306-317, 2007.
- [24] J. R. Santos, Y. Turner, G. Janakiraman, and I. Pratt, “Bridging the gap between software and hardware techniques for IO virtualization,” in ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference, pp. 29-42, 2008.
- [25] H. Raj and K. Schwan, “High performance and scalable IO virtualization via self-virtualized devices,” in HPDC '07: Proceedings of the 16<sup>th</sup> International Symposium on High Performance Distributed Computing, pp. 179-188, ACM, 2007.
- [26] J. Liu, W. Huang, B. Abali, and D. K. Panda, “High performance VMM-bypass I/O in virtual machines,” in ATEC '06: Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference, 2006.
- [27] S. Thibault, “Stub domains,” in Xen Summit, June 2008.