# Friendly Virtual Machines

## Leveraging a Feedback-Control Model for Application Adaptation *

Yuting Zhang
danazh@cs.bu.edu

Azer Bestavros
best@cs.bu.edu

Mina Guirguis
msg@cs.bu.edu

Ibrahim Matta
matta@cs.bu.edu

Richard West
richwest@cs.bu.edu

Computer Science Department
Boston University
Boston, MA 02215

## ABSTRACT

With the increased use of "Virtual Machines" (VMs) as vehicles that isolate applications running on the same host, it is necessary to devise techniques that enable multiple VMs to share underlying resources both fairly and efficiently. To that end, one common approach is to deploy complex resource management techniques in the hosting infrastructure. Alternately, in this paper, we advocate the use of self-adaptation in the VMs themselves based on feedback about resource usage and availability. Consequently, we define a "Friendly" VM (FVM) to be a virtual machine that adjusts its demand for system resources, so that they are both efficiently and fairly allocated to competing FVMs. Such properties are ensured using one of many provably convergent control rules, such as Additive-Increase/Multiplicative-Decrease (AIMD). By adopting this distributed application-based approach to resource management, it is not necessary to make assumptions about the underlying resources nor about the requirements of FVMs competing for these resources. To demonstrate the elegance and simplicity of our approach, we present a prototype implementation of our FVM framework in User-Mode Linux (UML)—an implementation that consists of less than 500 lines of code changes to UML. We present an analytic, control-theoretic model of FVM adaptation, which establishes convergence and fairness properties. These properties are also backed up with experimental results using our prototype FVM implementation.

*This work was supported in part by NSF grants ANI-0095988, ANI-9986397, EIA-0202067 and ITR ANI-0205294.


*VEE'05,* June 11-12, 2005, Chicago, Illinois, USA.


## Categories and Subject Descriptors

D.4.10.a [**Operating Systems**]: Support for Adaptation—*Application-aware adaptation*

## General Terms

Management, Measurement, Performance, Design, Experimentation

## Keywords

Friendy Virtual Machines, Feedback Control, Resource Management

## 1. INTRODUCTION

The emerging trend of enabling the dynamic dispatch of "guest" applications on powerful end-host systems necessitates a proper way for guest applications to share the resources in the hosting environment. Specifically, two common challenges must be addressed: (1) how could the execution of such applications be isolated from one another for safety and security purposes, and (2) how could the resources of the underlying system be managed fairly and efficiently. Commonly, the first challenge is handled by relying on compile-time checks and on operating system capabilities that ensure the integrity of address spaces. The second challenge concerning resource management is often ignored under the assumption that the end-host is over-provisioned with resources, or else that traditional OS resource arbitration across applications (e.g., using CPU scheduling, memory management and disk scheduling) will mitigate that challenge.

Relying on compiler and OS constructs for resource isolation in general, and memory isolation in particular has proven to be problematic [14, 20]. Moreover, the diversity of platforms on which guest applications are developed as well as those hosting them makes it harder to verify isolation properties. This has fueled research in the area of virtualization [5, 9, 23, 29, 31, 34, 35, 36, 39], (or group of applications) could run on a different "Virtual Machine" (VM). Since VMs can be trusted, in the sense of "sandboxing" the untrusted applications and services they contain, virtualization is seen as a promising approach for providing security and protection. Indeed, the use of VMs has enabled services, possibly deployed by untrusted sources, to be injected into a third party hosting infrastructure.

**Motivation:** The use of VMs as an architecture that promotes safety and security through isolation does not mitigate the second challenge—that of ensuring efficient and fair use of underlying resources. Due to the complexities of the interactions between VMs, applications running on these VMs, and services running on the hosting kernel, it is generally accepted that the most practical approach to deal with VM resource sharing issues is to ensure that the system is "over provisioned". In open systems, in which services may be deployed on-demand, over provisioning is an expensive proposition, and may not even be possible to guarantee. This in turn would result in unpredictable performance degradation as a result of increased contention for resources by VMs and the applications they support. This degradation could be in terms of efficiency (wasteful use of resources, *e.g.*, thrashing and excessive page faults) or fairness (*e.g.*, VMs being denied service).

**Application Adaptation for Efficiency and Fairness:** In one sense, VMs are nothing but instances of applications that need to share underlying resources fairly and efficiently. Traditionally, efficiency is achieved by instrumenting the underlying system to avoid being overloaded (by over-stressing resources and hence operating in inefficient regions) through admission control, for example, whereas fairness is achieved through the use of potentially complex policies for scheduling/allocating resources. Delegating resource allocation decisions to an underlying system has a number of disadvantages, including: (1) complicating the design of the underlying system which must include functionalities to deal with contingencies that may or may not be needed for the particular mix of applications currently running on the system, and (2) depriving the application of meaningfully adapting its behavior to match available resources. A radically different approach would be the adoption of a minimalistic end-host system design, with all the complexities necessary for resource management pushed into the application layer. Such an approach would be reminiscent of the end-to-end argument [32], which states that a certain functionality should be pushed to higher layers (if possible) unless implementing it at the lower layer achieves large performance benefits that outweigh the cost of additional complexity at lower layers.[1]

**Paper Scope and Contributions:** In this paper we adopt this end-to-end philosophy in the management of resources underlying/supporting the execution of multiple applications. While this philosophy (of application adaptation as opposed to traditional resource arbitration by an underlying system) is quite general, we believe that it is most appropriate for the design of VMs. Thus, in this paper, we focus exclusively on the design of VMs as the application of choice, noting that our models, analysis, as well as much of our design and implementation techniques are quite applicable to *any* application whose execution needs to be "friendly" to other applications sharing the same underlying resources.[2] We argue that a Virtual Machine Monitor (VMM) or host OS kernel (as the hosting environment) should be kept quite simple, and that a VM (as the hosted application) should increase or decrease its resource demands in a friendly way. To that end, in Section 3, we develop control-theoretic models for application adaptation, using simple Additive-Increase/Multiplicative-Decrease (AIMD) rules.

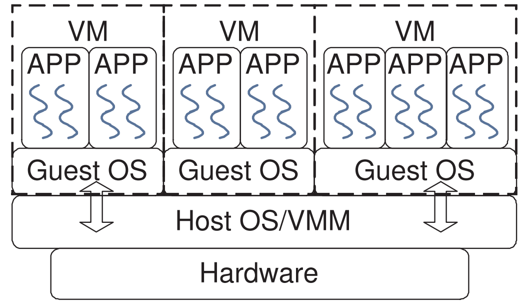



**Figure 1: Virtual Machine Model**

We refer to a VM that dynamically adapts its resource usage in this manner as a "Friendly" VM (FVM). Using our FVM framework, which we present in Section 2, instead of a complicated central multi-resource provisioning system implemented in the VMM, each FVM actively adjusts its demands of resources using some feedback-based control rules (such as AIMD) to avoid the inefficiencies resulting from overloading or underloading of resources, while at the same time ensuring fairness of use across FVMs. In our FVM prototype implementation, we treat a VM as a resource constraint unit. A VM-specific performance metric—the virtual clock time—is used as the sole feedback signal that enables a VM to "infer" the status of underlying resources.[3] We use a common thread model as well as a simple sleep scheme to adjust the resource consumption of each VM. By dynamically adjusting the number of threads in the VM,[4] and if that is insufficient the sleep rate[5] for the whole VM, we can adapt the resource demand of each VM to efficiently and fairly share the resources.

Our FVM framework is a general resource management framework for VMs on a shared machine. It can be applied to different types of existing VMs with minor code modifications. To demonstrate the elegance and simplicity of the FVM concept, we have implemented a prototype FVM in User-Mode Linux (UML), which is described in Section 4. In Section 5 we show the advantages gained by this simple mechanism in a number of experimental settings.

## 2. FVM: FRAMEWORK

In this section, we focus on general considerations and high-level design decisions of our FVM framework. Figure 1 illustrates a typical VM architecture, whereby multiple VMs share the same "physical machine", or host. At the lowest level, right above the hardware layer, the host OS kernel or VMM provides resource allocation to VMs. Within each VM, several applications (or services) run on top of the "guest" OS which in turn provides the customary set of high-level abstractions such as file access and network support to applications running on the VM.

The main goal of our FVM framework is to allow an

[1]In IP networks, for example, the IP layer implements the minimal functionality of packet forwarding, whereas error and congestion control is left to the end system [12].

[2]Throughout this paper, we refer to the terms "applications" and "virtual machines" interchangeably, where appropriate.

[3]This is akin to TCP's use of packet losses (or marking) at the end system to infer whether congestion exists or not.

[4]This is akin to the use of the number of packets en route as the control variable for TCP's demand.

[5]This is akin to the use of timeouts to further reduce TCP's demand in case of severe congestion.

efficient and fair sharing of system resources across VMs, especially under moderate and heavy load conditions. Efficiency implies that underlying system resources are neither *overloaded* nor *unnecessarily underutilized*. Overload conditions typically involve inefficiencies (*e.g.*, significant paging due to an excessive number of active VMs), whereas underutilization wastes system resources. Fairness implies that each VM is allocated a proportionate share [6] of the bottleneck resource for that VM.

**Host-Level (Centralized) Resource Management:** To a hosting system—as well as to other VMs on that host—a VM is not only a protection domain entity, but also a resource consumption entity. Moreover, the demand for a given resource by a VM depends not only on the applications executing on that VM but also on the demand and availability of other resources as well. Traditionally, resource management involves a one-way allocation, whereby a central authority (*e.g.*, host kernel or VMM) allocates resources such as CPU and memory to entities competing for such resources. For CPU allocation, for example, many schedulers have been proposed, from simple FCFS and Round Robin algorithms to more complicated real-time and progress-based schedulers. Due to the diverse nature (*e.g.*, CPU- vs memory- vs I/O-bound) and often dynamic characteristics (*e.g.*, lifetimes and interaction with user or other processes) of applications, it is hard for a centralized authority such as a host OS or a VMM to achieve fairness without taking these factors into consideration.

Essentially, host OS or VMM resource allocation decisions are made complicated by the unpredictable interactions involving the demands of all VMs on the various resources, and also the allocation policies for these resources. Moreover, notice that VMs, in turn, have to support their own applications through whatever resources they are allocated by the host OS or VMM. For example, the host OS may delay the delivery of a packet from the network card to the intended VM for scheduling purposes. However the real-time application running on top of the VM may discard the packet if it arrives later than expected. Other examples of inappropriate decisions taken by the host OS are abound.

**Application-Level (Distributed) Resource Management:** Rather than devising a complicated centralized solution (*e.g.*, in the VMM) to the problem of multi-resource allocation across VMs, our approach is to delegate the responsibility of ensuring efficiency and fairness to the VMs themselves, effectively deploying a distributed resource management strategy using feedback-based adaptive control in each VM. Such an approach has many desirable features and advantages. First, it is transparent to the applications running on top of the VMs as well as to the OS hosting the VMs. Second, it is resource agnostic in the sense that the VM adapts its demand to meet efficiency and fairness requirements for whatever resource happens to be the bottleneck for that VM. Third, it allows for multiple adaptation strategies to be used in different VMs as long as these adaptation strategies are compatible.[6] This enables a VM's adaptation strategy to be tailored to the requirements of the applications they are likely to support.

To enable a VM to adaptively adjust its demand of underlying resources, we must devise (i) a mechanism via which feedback is relayed to the VM concerning the state of the bottleneck resource for that VM, (ii) a mechanism via which the VM could scale up or down its demand, and (iii) an adaptation strategy that sets the demand of a VM based on the feedback signal in a friendly way. In the remainder of this section we discuss choices for the design of these three mechanisms.

[6]Different adaptation strategies are said to be "compatible", if they converge (at steady state) to a fair allocation of the bottlenecked resource despite their different reaction to sudden changes in resource availability [4].

## 2.1 Feedback Signal: Overload Detection

An application may use a variety of performance metrics to infer the state of underlying resources in general, and whether these resources are overloaded in particular. Examples include system-centric metrics such as CPU utilization, network utilization, and page fault rate, or process-centric metrics such as response time, jitter, and throughput.

CPU utilization, network utilization, and page fault rate are examples of resource-specific metrics, which are not particularly useful in predicting overload conditions *unless* the corresponding resources (CPU cycles, network bandwidth, and virtual memory, respectively) happen to represent the "bottleneck" of the system. More importantly, detecting overload using any one such metric will trigger an application to reduce its demand even if this application is not consuming much of that resource's capacity. For example, mitigating excessive CPU utilization by throttling down an I/O-bound application would be counter productive. Besides, these metrics can only be captured with special support from the host system—*e.g.*, through the use of special APIs or programming style interface such as the /proc interface in Linux [7]. Requiring such support from underlying layers is not in the spirit of the end-to-end argument that application adaptation seeks to advance.

Process-centric metrics such as throughput and response time do not suffer from the above disadvantages. For instance, a significant increase in response time measured by an application is symptomatic of an overload condition for the underlying resource that happens to be the bottleneck for that application. Clearly, measuring such process-centric metrics can be done in an "end-to-end" fashion, requiring no support from the underlying system.

**Virtual Clock Time:** We now turn our attention to how a VM—our application of choice—could detect overload condition using response time. To that end, a natural choice would be the *Virtual Clock Time* (VCT), which is common to all VM implementations. VCT is defined as the *real-time interval* between two consecutive virtual clock cycles (on the VM). Thus, one can think of VCT as the response time of the hosting system to a request by a VM for advancing its clock by a tick (or equivalently, it is the inverse of the virtual clock rate).

When a VM contends for an overloaded resource, the value of VCT will increase to reflect the waiting time for that resource. While the relationship between VCT and resource contention is monotonic, the manner in which this relationship manifests itself is highly dependent on the resource in question—how it is shared, apportioned, or scheduled. For example, if CPU cycles are the bottleneck, then an increase in the number of VMs will ideally[7] result in a linear increase in VCT. However, if other resources such as memory become the bottleneck, then an increase in the number of VMs (or in the number of threads within each VM) will result in a super-linear (or even "exponential") increase in VCT. This is so because the increased multiprogramming level will result in increased paging activities since the same main memory

[7]In practice, the relationship is super-linear due to various overheads related to the increased level of concurrency.

size on the host system must now cater to a much larger working set. The impact of an increase in the rate of page faults (even if slight) will thus translate to a huge increase in VCT, which can be used by the VM as an indicator of overload conditions.

It is important to note that a VM may well demand the heavy use of multiple resources (*e.g.*, memory and I/O). However, only one of these resources will be *first* to trigger a significant increase in VCT, making that the "bottlenecked" resource for that VM. Thus, we argue that large changes in the VCT of a VM will be associated with a single "congested" resource, as opposed to other resources that may be used by the VM.[8] This allows VMs to be naturally partitioned into "bottleneck-equivalent classes." As we will briefly discuss later in this section, and analytically show in the next section, with proper adaptation strategies, all VMs in a bottleneck-equivalent class will share that bottlenecked resource fairly.

**Congestion Signal Generation:** To detect whether there is resource congestion, a VM could measure the ratio between the current VCT and the minimum VCT observed over a longer time scale. To smooth out short-term variabilities, current VCT can be estimated as an average value $\overline{VCT}(t)$ at time $t$ using an Exponentially-Weighted Moving Average (EWMA) with parameter $\gamma$ $(0 < \gamma < 1)$ on the instantaneous values of $VCT(t)$ as follows:

$$\overline{VCT}(t) \quad = \quad (1-\gamma) \cdot \overline{VCT}(t-1) + \gamma \cdot VCT(t) \qquad (1)$$

The minimum VCT could be seen as a baseline value against which the current VCT is evaluated. Given an appropriate window of time $w$, the minimum VCT could be estimated as follows:

$$VCT_{\min}(t) \quad = \quad min\{\overline{VCT(t-i)} : i = 1, \ldots, w\} \qquad (2)$$

The congestion signal is calculated by computing the ratio $R$ between $\overline{VCT}(t)$ and $VCT_{\min}(t)$ and comparing that to some chosen threshold $H$ (*e.g.*, $H = 2$). This ratio could be seen as a measure of the "slowdown" caused by contention for underlying resources. If $R > H$, then the resource congestion signal is set to 1, otherwise, it is set to 0.[9]

## 2.2 Control Signal: Resource Consumption

For an application in general (and a VM in particular) to adapt its demand of underlying resources, a mechanism must be devised that maps a "control signal" to actual demand. In this section we present two complementary approaches: (1) The imposition of an upper bound on the Multi Programming Level (MPL), *e.g.*, maximum number of processes or threads in an application, and (2) the imposition of an upper bound on the rate of execution, or equivalently a lower bound on periodic timeout or sleep time for an application.

**MPL Control:** Most applications and services use processes and threads to scale up their performance. Clearly, a thread can be viewed not only as a unit of execution, but also as a unit of resource consumption, whereby each thread of execution consumes additional resources such as CPU cycles, memory, I/O bandwidth, *etc.* By virtue of its structure, a VM can be considered as a multi-threaded application as illustrated in Figure 1. Within each VM, a number of threads are available to execute the various applications running within that VM. Hence, the resource demand of each VM could be bound by a limit on the maximum number of threads that may be active at any given time.

MPL control in a VM can be implemented by suspending or resuming threads within a VM. While it is true that each thread within the VM does not necessarily consume the exact same amount of resources,[10] we expect that such heterogeneity will not be problematic unless the MPL is quite small. In such cases (when MPL is too small and certainly when MPL=1), a VM must resort to a different mechanism for adapting its consumption as we will see next.

**Rate Control:** Another mechanism that could be used to control demand for resources is to throttle the execution of a given VM by forcing it to periodically sleep. By forcing a VM to sleep for a minimum "timeout" $T_s$, we effectively cap the maximum execution rate for that VM to be proportional to $1/T_s$. Rate control is an effective mechanism to offload underlying resources, especially when reducing the number of threads is not feasible (*e.g.*, an application has only one thread of execution) or else the various threads in an application perform radically different functionalities, or have very different resource consumption profiles.

## 2.3 Controller: Adaptation Strategy

By dynamically adjusting the number of active threads in it, or by adjusting its periodic sleep interval, an application such as a VM could adapt its resource demand. Clearly, this adaptation will follow some prescribed increase/decrease rules that guarantee desirable properties of efficiency and fairness. An example of such rules would be the additive-increase/multiplicative-decrease (AIMD) rule, which would allow a VM to probe the capacity of underlying resources by incrementing demand (*e.g.*, number of threads or rate of execution) in a linear fashion, but would force this demand to decrease exponentially when a feedback signal indicates an overload condition.

While AIMD is one example of an adaptation rule that is known to converge to fairness and efficiency, a whole spectrum of such increase/decrease rules have been explored in the literature—see [4, 21] for examples. In this paper, while we focus on AIMD control, it should be clear that other adaptation strategies could well be adopted in our framework, especially when they result in other favorable properties (*e.g.*, smoother adaptation, or aggressive probing of available underlying resources, faster convergence, *etc.*)

# 3. FVM: MODEL AND ANALYSIS

In this section, we present a non-linear, dynamic model of our proposed FVM framework. Specifically, we present a model whereby a number of VMs adapt their demand to match the capacity of underlying system resources. We then present a linearized model and investigate steady-state and convergence/stability properties.

## 3.1 Model Derivation

We consider a dynamic model of $V$ applications—namely, VMs—sharing a single resource. Let $n_v(t)$ denote the demand from each application at any instant of time $t$. For

[8]For example, a CPU-bound VM will not experience a large increase in its VCT (on average) if the hosting system is only experiencing significant I/O contention.

[9]Such an on-off signal makes analysis of the control system quite complex, thus in Section 3, we use a probabilistic function to map $R$ and $H$ to a congestion signal.

[10]In practice, if a VM is hosting a multi-threaded service (*e.g.*, a web server or an end-system multicast agent, *etc.*) then it may well be the case that threads within the VM will be comparable in terms of their resource consumption needs—both in terms of the nature of resources they need (CPU vs I/O bound) as well as the amount of use.

example, such demand can be expressed as the number of threads each VM owns. The total demand, $\sum_{v=1}^{V} n_v(t)$, determines the total resource usage, $m(t)$. One possible mapping from demand to usage could be expressed using the following linear relationship:

$$m(t) \quad = \quad K_1 \times \sum_{v=1}^{V} n_v(t) \tag{3}$$

where $K_1$ is the number of resource units consumed by each thread. If we consider memory to be the bottlenecked resource, then $m(t)$ reflects the total demand on memory usage from all threads. An increase in total demand $m(t)$ will typically translate to additional overhead, which in turn will translate to an increase in service time (*e.g.*, VCT slowdown) or equivalently a decrease in service rate. Such an overhead could be viewed as the *price* that a VM must pay for acquiring the resource when total demand is $m(t)$. As we hinted earlier in the paper, the relationship between price (VCT slowdown in our case) and demand is likely to follow a super-linear function. For example, as the demand for memory increases, paging activity increases at a faster rate, which may result in a super-linear increase in VCT. In general, we can depict the VCT slowdown $p(t)$ at time $t$ as a function of the total demand $m(t)$. While there could be many possible instantiations of this function, for simplicity we use the following equation:

$$p(t) \quad = \quad m^2(t) \tag{4}$$

In a real setting, the price, measured as the slowdown in VCT, may exhibit high variability over short time scales, due to the granularity of overhead events, such as delays caused by page faults. Such variability could be smoothed out over longer time scales using EWMA averaging, similar to equation (1). Hence, the average VCT slowdown (average price) $g(t)$ at time $t$ evolves according to the following equation:[11]

$$\dot{g}(t) \quad = \quad -\gamma(g(t) - p(t)), \qquad 0 < \gamma < 1 \tag{5}$$

where $\gamma$ reflects the weight given to the instantaneous VCT slowdown $p(t)$. Based on the average VCT slowdown, a congestion signal, $c(t)$, is generated. This congestion signal can be expressed as a linear function of the average VCT slowdown according to the following equation:

$$c(t) \quad = \quad \sigma \times g(t) \tag{6}$$

The value of $\sigma$ will typically bound the congestion signal at any time instant below 1. By choosing $\sigma$ to be the reciprocal of the maximum VCT slowdown (maximum price), such condition is ensured.

Under AIMD, each application adjusts its demand based on the congestion signal $c(t)$ according to the equation:

$$\dot{n}_v(t) \quad = \quad (1 - c(t)) - \frac{n_v(t)}{2} c(t) \qquad v = 1, 2, .., V \tag{7}$$

where the first term represents the additive increase of one unit of demand and the second term, represents the multiplicative decrease by halving the demand. Figure 2 depicts the general control block diagram for the model presented above.[12]

[11]In general, we denote that change in a variable $x(t)$ by $\dot{x}(t)$

[12]Since VMs are running on the same physical machine, the effect of feedback delay should be negligible, and hence is not accounted for in our model.

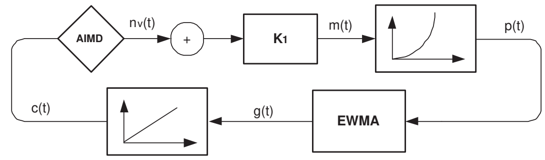
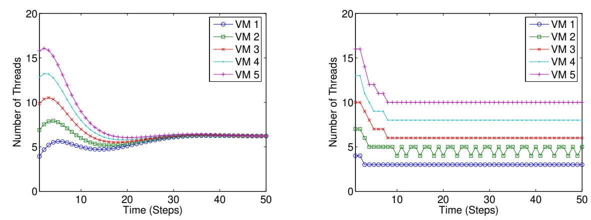



**Figure 2: Block diagram for application adaptation**

## 3.2 Impact on Efficiency and Fairness

We instantiate the above model for 5 VMs. We choose $K_1$ to be 10 units. The averaging weight, $\gamma$, in equation (5) is chosen to be 0.1 and $\sigma$, in equation (6) is chosen to be $2.5 \times 10^{-6}$. We solve the above fluid model numerically for a careful and continuous tracking of the model variables even through nonlinear regions. We assume that each one of the five VMs starts with an initial number of threads.[13] Let the initial number of threads for the five VMs be 4, 7, 10, 13 and 16 threads, respectively. Figure 3(a) shows how the various




(a) Application Adaptation (b) Admission Control

**Figure 3: Convergence to Fairness (a) With Application Adaptation; (b) With Admission Control**

VMs converge to a fair and efficient allocation of threads using the above model and prescribed AIMD adaptation. During the first 5 steps, each VM increases its demand since $g(t)$ is still lagging behind $p(t)$. Once $g(t)$ catches up with $p(t)$, the appropriate congestion signal, $c(t)$ will ensure a fair convergence across the VMs. Clearly, the convergence time depends on our choice of parameters. In particular, a small value of $\gamma$ will provide good stability, but the response of the system to sudden changes in resource availability (*e.g.*, due to sudden increase/decrease in number of VMs) maybe be sluggish. Having a larger $\gamma$ will decrease convergence time, but may result in a more oscillatory transient behavior (*e.g.*, larger overshooting).

We compare this model to a centralized overload protection approach whereby the system is protected against congestion through an admission controller. The admission controller decides the admission ratio, $a(t)$, of new threads using a Proportional-Integral (PI) controller. A PI controller is one whose control signal is proportional to the error signal and its integral, where the error signal is computed as the difference between $p(t)$ and some target level, $P_T$. Thus the change in the admission ratio, $a(t)$, can be described by the equation:

$$\dot{a}(t) \quad = \quad K_2 \times (P_T - p(t)) \tag{8}$$

where $a(t)$ is bounded from above by 1 and from below by 0 and $K_2$ is the PI constant that controls how fast the con-

[13]In a real setting, VMs may not have started at the same time. Thus, this initial value should be viewed as reflecting an unbalanced allocation of resources (threads) across VMs at some arbitrary point in time.

troller reacts to the error signal. Higher values of $K_2$ induce faster convergence, but with the drawback of possible oscillations; lower values provide better stability, with the possibility of a more sluggish response to sudden changes in resource availability.

We choose $K_2$ to be $1.0 \times 10^{-6}$ and $P_T$ to be 100000. Figure 3 (b) represents the convergence of the number of threads per VM to steady-state values, when $p(t)$ reaches $P_T$. One can easily observe that such convergence does not lend itself to any fair allocation of the underlying resource. As for efficiency, both models maintain a fixed number of threads over time, ensuring that efficiency is not compromised.

## 3.3 Model Linearization and Stability

Setting equation (7) to zero, gives the steady-state relationship between the demand of each application and the congestion signal. We let $x^*$ denote the steady-state value for variable $x$.

$$n_v^* = \frac{2}{c^*} - 2 \tag{9}$$

$$p^* = g^* = m^{*2} \tag{10}$$

We analyze the stability of the above model by linearizing the system around its steady-state operating region. Equation (7) becomes:

$$\dot{\bar{n}}_v(t) = (-1 - \frac{{n_v}^*}{2})\bar{c}(t) + \frac{-c^*}{2}\bar{n}_v(t) \tag{11}$$

where $\bar{x}$ denotes the linearized form of the variable $x$ (*i.e.*, perturbations around the steady-state value). Equation (4) becomes:

$$\bar{p}(t) = 2m^*\bar{m}(t) \tag{12}$$

Taking the Laplace transforms for the above closed-loop model, we get the open-loop transfer function $L(s)$:

$$L(s) = \frac{\alpha V K_1 2m^* \gamma\sigma}{(s+\gamma)(s+\beta)} \tag{13}$$

where $\alpha$ and $\beta$ are given by $(1 + \frac{n_v^*}{2})$ and $\frac{c^*}{2}$, respectively.

Let $G$ be equal to $\sqrt[3]{4\sigma(K_1V)^2}$ and assuming that $n_v^* \gg 2$ and that $\sigma$ is chosen so that $\gamma G \ll 0.5$, then the closed loop characteristic equation can be derived to be:

$$s^2 + \frac{1}{G}s + 0.5 = 0 \tag{14}$$

The location of the roots of the characteristic equation in the $s$-plane determines the stability of the overall system and the nature of its transient behavior [28]. The roots of characteristic equation $r_1$ and $r_2$ are given by:

$$r_1, r_2 = \frac{-1}{2G} \pm \frac{\sqrt{\frac{1}{G^2} - 2}}{2} \tag{15}$$

As the value of $G$ in equation (15) varies from 0 to $\sqrt{0.5}$, both roots are real and inside the left half of the $s$-plane, ensuring exponential convergence. As $G$ increases beyond $\sqrt{0.5}$, the roots become imaginary, but their real components are still on the left half of the $s$-plane, ensuring oscillatory convergence. Notice that there is no positive value for $G$ that would cause the system to be unstable, since the positive value of $\sqrt{\frac{1}{G^2} - 2}$ is always less than $\frac{1}{G}$. Notice also that our assumption about $n_v^* \gg 2$ ensures that $c^*$ is relatively much smaller than 1 (cf. equation 9).

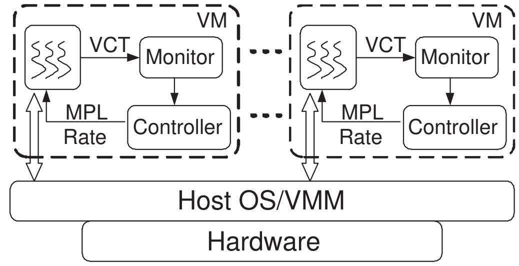



**Figure 4: FVM Control Model**

# 4. FVM: IMPLEMENTATION

Our prototype FVM implementation is based on a modified version of User Mode Linux (UML). UML's open source code is one of the reasons that motivated the choice of this platform. Moreover, UML's implementation is relatively simple and easy to modify.

**UML Overview:** UML [35] is a virtual machine abstraction that allows *guest* Linux systems and processes to run at user-level on top of a *host* Linux system. All of UML's devices are virtual, being constructed from software resources provided by the underlying host. Essentially, UML supports any application that is able to run on the host, including itself. Unlike other VMs such as VMware [36], UML is a port of the Linux kernel to the system call interface rather than to a hardware interface.

Currently UML has two modes: the `tt` (Tracing Thread) mode and the `skas` (Separate Kernel Address Space) mode. In `skas` mode, guest processes are seen as threads within a single host process. For performance reasons, our FVM implementation is based on `skas` mode of UML.

**FVM Prototype Implementation Details:** Our implementation of FVM requires only a few minor modifications to UML code (located in the `arch/um` directory in the Linux source tree). The entire changes add about 500 lines of new code to UML. Based on our presentation in Section 2, Figure 4 shows the basic components of our FVM framework. Multiple FVMs share underlying resources, with each FVM supporting a number of threads for its own applications. Each FVM has a "Monitor" that measures the feedback signal (namely, the slowdown $R$ in VCT) and a "Controller" that adjusts the level of demand offered by the FVM using either MPL or rate control (*i.e.*, the number of threads within the FVM or the FVM sleep time, respectively). The FVM "Monitor" is implemented by measuring the real-time interval between two consecutive virtual clock interrupts, using the Pentium-based Timestamp Counter (TSC) for high precision. In UML, clock interrupts are implemented using the `SIGALRM` and `SIGVTALRM` timers. Periodically, every period $T_c$ (or equivalently every $N_{clk}$ virtual clock interrupts), the value of $\overline{VCT}(t)$ is calculated from an EWMA of the intervals between successive clock ticks during the period $T_c$. The minimum value, $VCT_{\min}(t)$, is then calculated over a fixed window of $w$ non-overlapping values of $\overline{VCT}(t)$. This enables the FVM "Monitor" to generate the feedback signal, namely, whether or not overload conditions exist.

The FVM "Controller" is invoked periodically every $T_c$ interval. Based on whether or not the FVM "Monitor" signals an overload condition, the "Controller" adjusts the maximum number of threads (and possibly the sleep time, if the MPL value reaches 1) using an "Additive Increase Multiplicative Decrease" (AIMD) scheme. Based on the newly calculated limit on the number of threads allowed in each

FVM, the "Controller" may need to suspend currently active threads (in case of a multiplicative decrease), or it may resume currently suspended threads (in case of an additive increase). `SIGSTOP` and `SIGCONT` job-control signals are used to suspend and resume randomly selected threads.

To ensure that the FVM does not fork threads in excess of the limit calculated by the FVM "Controller", a check is performed on each thread creation system call (*e.g.*, through a call to `sys_fork(sys_clone(), sys_vfork())`. As we mentioned before, rate control (if enabled and needed, *e.g.*, when MPL=1) is implemented in the "Controller" simply by forcing the entire virtual machine to sleep for a period of time $T_s$ using the `sleep` function. Rate control follows an AIMD adaptation whereby $1/T_s$ is increased linearly and decreased multiplicatively.

The operation of the "Monitor" and "Controller" in the FVM is parameterized (and can be changed) using a script. The settings include the timescales for monitoring, control, and measurement, as well as the initial values for the maximum number of threads allowed for a given FVM and the constants of our adaptation strategies. Since our FVM implementation in UML allows for two adaptation strategies (MPL control and rate control), we must specify "constants" for the AIMD rules of both control.

For the prototype used in this paper, Table 1 shows the baseline settings of our FVM implementation in UML. Unless otherwise specified, these settings are used in our experimental section.

| **Parameter** | **Setting** |
|---|---|
| Monitoring/control Period ($T_c$) | 5 sec |
| Window of $VCT_{\min}$ ($T_w = w \times T_c$) | 60 sec |
| EWMA constant for $\overline{VCT}$ ($\gamma$) | 0.3 |
| Initial limit on number of threads | 10 |
| VCT slowdown threshold ($H$) | 2.5 |
| AIMD additive constant (MPL ctrl) | 1 thread |
| AIMD additive constant (Rate ctrl) | 0.1 hz (=$1/T_s$) |
| AIMD multiplicative constant | 1.5 |

**Table 1: Baseline Settings of FVM Prototype**

**Underlying System Requirements and Functionality:** The application-level adaptation we advocate in this paper in general, as well as its specific FVM instantiation presented in this section, makes some inherent assumptions about the manner in which the underlying system (*e.g.*, host OS/VMM) allocates/schedules its resources to competing processes (*e.g.*, VMs). Specifically, there is an inherent assumption that underlying resource allocators are *not* biased in that each resource is shared across competing processes in proportion to corresponding process demands. For instance, a simple round-robin scheduler is "unbiased" whereas a multi-level feedback scheduler is not.[14] If underlying resource managers base their decision on other considerations, then application-level adaption may not converge to efficiency and fairness due to the unpredictability of the interference between the application and underlying system control planes.

Clearly, the resource management strategies deployed in Linux (*i.e.*, the underlying system in our FVM prototype implementation) are not entirely "unbiased". For instance, when a Linux system is faced with extensive paging (*i.e.*, thrashing), the kernel swaps out least frequently used pages first. Furthermore, when the system is out of memory, Linux may terminate a subset of processes, which is clearly unfair to processes (especially those with minimal memory requirements). Luckily, Linux resorts to such draconian measures only under overload conditions. Such conditions should not materialize by virtue of the application adaptation framework we have advocated and implemented in this paper. That said, one can readily see that our framework could be implemented (and is likely to be far more effective) on a much "thinner" underlying system with minimalistic resource allocation strategies.

The above discussion begs the question of what "functionality" (other than minimalistic unbiased resource management) should an underlying system provide. The framework we advocate in this paper guarantees efficiency and fairness by relying on the "friendliness" of competing applications. If such a framework is to be used in an open environment in which such friendliness could not be expected of all applications, then an important functionality of the underlying system would be to provide incentives for applications to be friendly. To that end, one effective mechanism would be the implementation of resource *policing* functionalities. By "policing" we mean the ability of the system to identify misbehaving applications, *i.e.*, those which do not adapt in a manner compatible with prescribed rules, such as AIMD. It is important to note that policing could be implemented through random sampling of resource usage by various applications over an appropriate timescale. While such policing functionality will incur overhead, it should be evident that such an overhead would pale in comparison to the overhead of requiring the system to perform clever (and certainly complex) resource management on behalf of applications.

## 5. FVM: PERFORMANCE EVALUATION

Using our prototype FVM implementation, we conducted a series of experiments to evaluate the performance of our framework in terms of efficiency and fairness. Two sets of experiments were conducted. In the first, our FVM framework was used to host a set of memory-intensive "benchmark" applications. In the second, our FVM framework was used to host a set of "real" applications—namely the Apache web server. Before presenting results from these two sets of experiments, we discuss briefly the various performance metrics used in our evaluation.

The first metric we consider is the *Virtual Clock Time* (VCT), as defined earlier in the paper. The value of VCT could be seen as a gauge for responsiveness. The second metric is *throughput*, which we define to be the total number of completed work units (*e.g.*, a single execution of a hosted application, or a response from a web server) per unit time. The third metric we consider is the *fairness index*. To capture both the fairness and efficiency of an adaptation strategy, we define the Fairness Index (FI) using the following equation:

$$\mathrm{FI} \quad = \quad 1 - \frac{\sqrt{\sum_{i=1}^{V}(x_i - o_i)^2}}{\sqrt{\sum_{i=1}^{V} o_i^2}} \tag{16}$$

where $V$ is the total number of VMs, $x_i$ is the performance metric for $VM_i$ for which the index is to be calculated, $o_i$ is the optimal value of the metric under a perfectly fair and efficient allocation. Notice that FI=1.0 implies optimal performance with respect to both efficiency and fairness. FI will decrease if *either* efficiency or fairness are degraded. In our

[14]Notice that our requirement for unbiased underlying resource scheduling does not necessarily imply fairness. Indeed, an unbiased scheduler such as round robin may well be unfair to I/O bound processes, for instance.

experiments, we use throughput as the underlying metric for calculating FI.

## 5.1 Benchmark Application Experiments

The application benchmark we used was developed to emulate a memory-intensive application as follows. First, the application grabs 1MB of buffer, reads data from a file into this buffer, performs some computations, writes back the content of the buffer to another file, and then frees memory. Each VM has a number of threads, each executing the above operations repetitively. To generate different workloads, we vary the initial number of threads in each VM or the number of VMs on the host. For each such workload, the system is allowed to run for 10 minutes before results are collected.

The underlying physical machine used in this set of experiments is a 2.4GHz Pentium IV with 512KB of cache and 1.2GB of RAM. To ensure that memory is the bottlenecked resource, a background application is used to lock 800MB of memory, leaving only 400MB of memory for the VMs to share.

Figure 5 shows the values of VCT, throughput, and FI when the number $V$ of VMs is varied from 1 to 6, with each one of the VMs hosting a total of 50 benchmark application threads. For each metric, we show results obtained with a standard UML VM (the dashed line) as well as with our FVM prototype implementation in UML (the solid line).

The results show clearly the effect of overload on both UML and FVM. Without adaptation, as the number of VMs increases the performance reflected by VCT and throughput values degrades significantly. With FVM adaptation, the degradation is quite graceful.[15] Notice that with three or less VMs running on the host, the available memory is large enough, allowing for an efficient operation, making the three metrics almost the same, whether or not adaptation is employed. However, beyond three VMs, the performance of non-adaptive VMs deteriorates rapidly (superlinearly) compared to that of adaptive VMs. With six VMs sharing the host, the total throughput with adaptation is more than 400% that without adaptation. Linux' suspension policies designed to thwart the effects of thrashing (due to extensive paging) can be seen clearly in the sudden drop in FI when the number of VMs reaches and exceeds four.

Figure 6 shows the values of VCT, throughput, and FI when a total of two VMs ($V = 2$) are running on the host, with each VM supporting the execution of a number of threads, which we vary from 1 to 300 per VM. One can see a very similar behavior to that observed in Figure 5. An interesting observation from this experiment is that under very light loads—namely when the number of threads per VM is below 20—our FVM approach lags (albeit very slightly) in terms of total bandwidth. Under such circumstances, unnecessary adaptation results in a slight degradation in performance.

## 5.2 Web Server Experiments

To evaluate the performance of our FVM framework in more realistic environments, we used our FVM prototype (as well as an unmodified UML VM) to host a real web server application. For that purpose, we used the popular multi-threaded Apache 2.0 web server. A total of four VMs are used in each experiment, with each VM hosting an Apache server.

To vary the workload on the four VMs, we used `httperf` clients [27] running on Linux 2.4.20 to generate HTTP session-based workloads. Session-based workloads are commonly used in the literature to evaluate various performance characteristics [38, 15, 11]. In our experiments, we vary the number of sessions, with each session repeatedly generating new CGI requests. To respond to a CGI request, the Apache server forks a new CGI thread to execute a CGI program, which reads and writes a 1MB chunk of memory, and then sends out a 4KB html file to the `httperf` client. Four client machines are used to run the `httperf` synthetic workload generator, with each machine targeting one of the VMs. The `httperf` client machines are 2.4GHz Pentium IV with 1.2GB of RAM, whereas the server machine shared by the four VMs is a 1.4GHz Pentium IV with 512MB of RAM.

Figure 7 shows the values of VCT, throughput (number of successful HTTP transfers per second), and FI when the four VMs are running on the host, with each VM subjected to a synthetic workload that we vary between 20 and 140 sessions. For each metric, we show results obtained with a standard UML VM (the dashed line) as well as with our FVM prototype implementation in UML (the solid line). For each load setting, we run the system for 20 minutes, with data collected every 5 seconds. The results are quite similar to those obtained in the previous set of experiments (for the memory intensive benchmarking application)—highlighting the efficiency and fairness of application-level adaptation, especially under heavy loads.

Figure 8 and Figure 9 show the observed values of VCT and achievable throughput (measured in HTTP gets per second) under a low load (of 60 sessions per VM), a moderate load (of 100 sessions per VM), and a heavy load (of 120 sessions per VM). In both figures, the values of VCT and throughput are shown for each one of the four VMs (*i.e.*, Apache servers) over time, averaged over 50 second intervals. The top row in each figure corresponds to the results obtained using vanilla UML, whereas the bottom row corresponds to the results obtained using our FVM prototype.

The results in Figure 8 and Figure 9 show that the advantage of adaptation is less pronounced under light loads. Under moderate loads, without FVM control, the system oscillates for a long while before eventually converging, whereas using FVM control, the system converges much faster. Under heavy loads (when the system is effectively in overload), the lack of FVM control results in a very poor behavior, whereby only one VM (the web server on $VM_1$ ) able to respond adequately to its workload, with all three other VMs effectively shut out (as indicated by the wild variation in their VCT and the significantly low throughput). Under the same load conditions, when FVM control is used, the four VMs adapt their demand, resulting in fast convergence to a fair and efficient sharing of the host's capacity.

# 6. RELATED WORK

Our FVM framework spans a number of active research areas. In this section, we briefly summarize how prior work in these areas relates to ours. Given the huge literature in these areas, our citations are only meant to be representative.

**Application-level Resource Management:** A significant number of OS architectures have adopted the "end-to-end argument" [32] by providing user-level resource management APIs (as is done in microkernels, using user-level services [24, 2], and in exokernels [16] using library operating systems). Nemesis [18] is an OS designed to deliver Quality-of-Service (QoS) guarantees to applications, and uses ideas similar to the exokernel in terms of application-specific mul-

[15] The small, graceful performance degradation is due to expected overheads due to increased context switching when the number of FVMs increases.

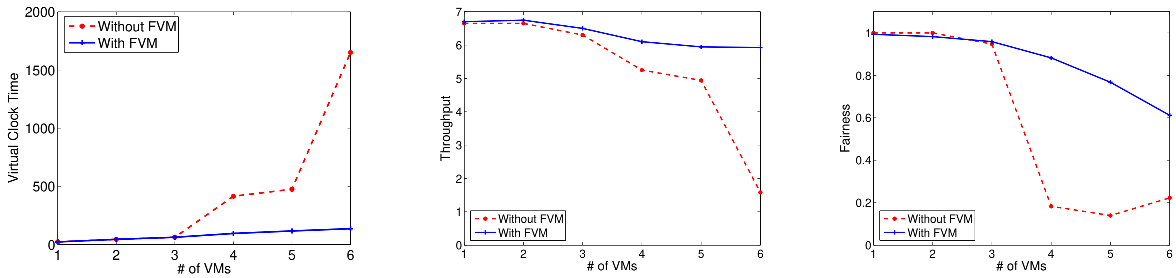



**Figure 5: Benchmarking results showing performance metrics vs number of VMs (# threads=50)**

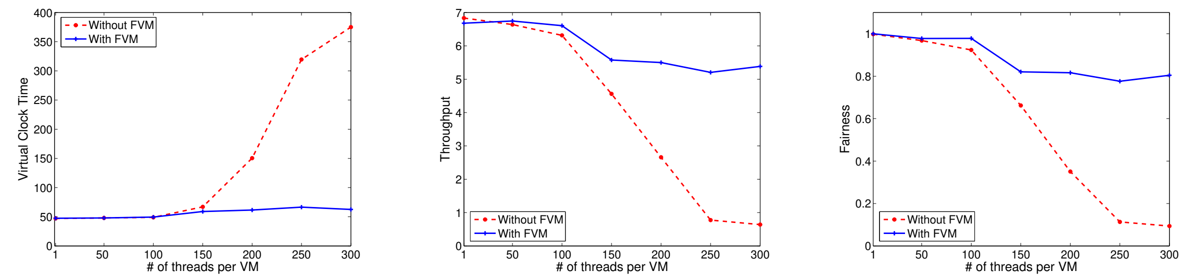



**Figure 6: Benchmarking results showing performance metrics vs number of threads per VM (# VMs=2)**

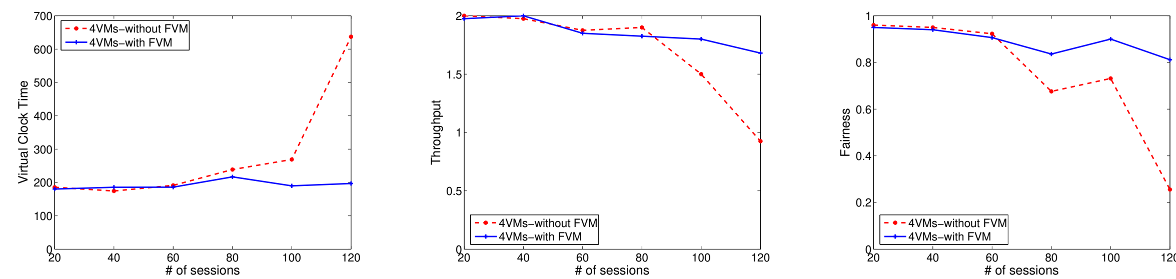
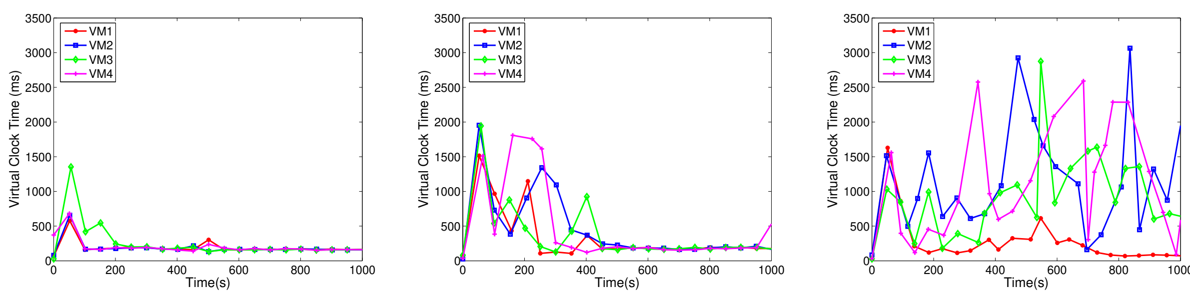
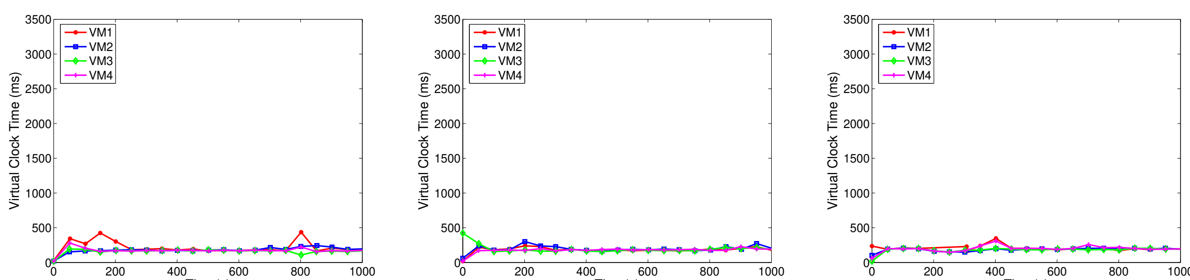



**Figure 7: Performance of VM-hosted Web servers under varying number of `httperf` sessions (load).**

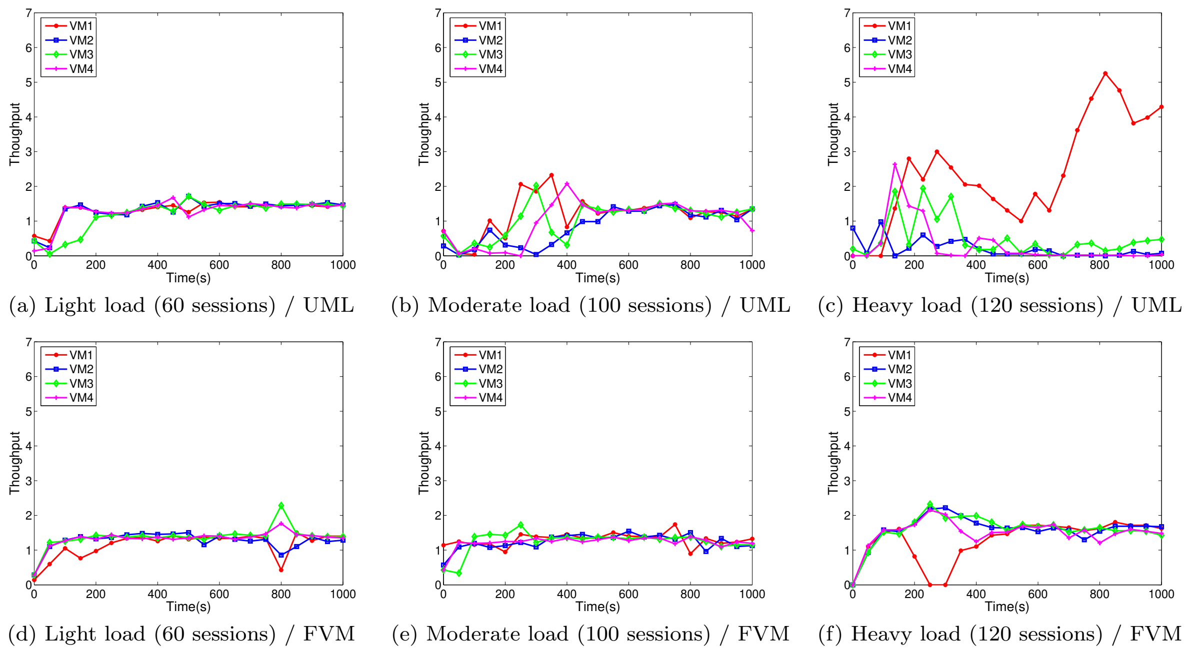



(a) Light load (60 sessions) / UML (b) Moderate load (100 sessions) / UML (c) Heavy load (120 sessions) / UML




(d) Light load (60 sessions) / FVM (e) Moderate load (100 sessions) / FVM (f) Heavy load (120 sessions) / FVM

**Figure 8: Evolution of Virtual Clock Time (VCT) for each VM over time**




(a) Light load (60 sessions) / UML (b) Moderate load (100 sessions) / UML (c) Heavy load (120 sessions) / UML

(d) Light load (60 sessions) / FVM (e) Moderate load (100 sessions) / FVM (f) Heavy load (120 sessions) / FVM

**Figure 9: Achievable throughput per VM-hosted Web server over time.**

tiplexing of system resources such as memory. All these system designs allow resource-management policies of the OS, including scheduling, memory management, and I/O, to be tailored for a specific application. While safety and efficiency are central to these systems, there is no explicit support for applications to friendly manage resources in an adaptive manner. Specifically, the notion of a feedback signal from the underlying system, with which applications can make informed decisions about their resource demands, is not integral to these system designs.

**Resource Management in VMs:** Recent research on VMs, such as Denali [39] and Xen [5], focus on efficient techniques to enforce resource isolation. Techniques such as *paravirtualization* are used to expose part of the hardware interface, thereby reducing the overheads of total machine virtualization. This enables a larger number of VMs to execute concurrently on a single host, which would be beneficial for secure (virtual) webservers supporting many thousands of clients. Denali adopts a static allocation scheme to partition resources amongst VMs, which makes it difficult to fully utilize all resources, especially in an open system in which VMs may be deployed (and their workloads changed dynamically). In Xen[5], admission control and resource reservation are used when starting new VMs. To reduce overall memory pressure on the system, Xen is capable of dynamically reclaiming pages of memory from some of its hosted VMs, for allocation to others.

Our FVM work complements the above work on machine virtualization. In fact, it is possible for our feedback-based resource adaptation methods to be incorporated into these VM architectures–a subject of some of our future work.

**Adaptive Feedback Control:** Marshaling techniques from control and optimization theory has been a fruitful direction as exemplified in many systems-related [1, 3, 10, 15, 25, 30] as well as networking-related [8, 17, 19, 22, 26] studies. By adopting a control-theoretic model, we provide meaningful feedback signals to applications for fair and efficient resource management. While many "fair" resource allocation policies exist, including those for allocating network bandwidth [13], CPU cycles [37], and disk bandwidth [33], these approaches do not adapt to dynamic changes in resource requirements and usage. Typically, applications change their resource demands over time, so making resource allocation decisions using fixed parameters (e.g., weights in weighted fair queuing [13]) is inadequate or requires more complex policy decisions to be considered.

We believe our work to be the first to combine aspects of control theory, VMs and application-level resource management to alleviate the need for complex, centralized system-level management of multiple resources.

# 7. CONCLUSION

In this paper, we advanced the concept of *"virtual machine friendliness"* which applies the classical end-to-end argument to the problem of multi-resource allocation across a set of applications sharing the same infrastructure. We have shown through modeling and analysis, as well as through a prototype implementation and performance evaluation of a Friendly Virtual Machine (FVM) that this approach is not only feasible but also desirable.

**(1)** Our approach enables applications sharing a common host to be naturally partitioned into different congestion equivalence classes based on the particular source of congestion for each application. This "performance isolation" property is much harder to guarantee using traditional centralized multi-resource allocation approaches, which cannot easily infer such a partition.

**(2)** Our approach delegates the regulation of resource usage to the application itself. This enables different FVM-compatible adaptation strategies to coexist, thus allowing applications to select the most appropriate manner in which resource allocations are allowed to vary over time. The evaluation of this feature of our framework is currently underway.

**(3)** Our approach lends itself well to emerging open systems whereby "guest applications" must be executed on demand on a shared hosting infrastructure. In this paper, we have considered the case of an infrastructure that consists of a single (centralized) host, albeit with multiple resources. We are currently investigating the applicability of this framework to infrastructures in which resources are distributed.

**(4)** Our approach enables the design complexity of underlying hosting systems to be significantly reduced. While our current prototype implementation was built on top of a fairly complex host OS (Linux), the concepts and mechanisms in this paper should be readily applicable to other hosting environments.

# 8. REFERENCES


[1] T. Abdelzaher and C. Lu. Modeling and performance control of internet servers. In *Proceedings of the 39th IEEE Conference on Decision and Control (ICDC)*, Sydney, Australia, December 2000.

[2] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Summer USENIX Conference*, Atlanta, GA, USA, July 1986.

[3] M. Andersson, M. Kihl, and A. Robertsson. Modelling and design of admission control mechanisms for web servers using non-linear control theory. In *Proceedings of ITCom*, 2003.

[4] D. Bansal and H. Balakrishnan. Binomial congestion control algorithms. In *Proceedings of IEEE INFOCOM*, 2001.

[5] 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 *Proceedings of SOSP*, 2003.

[6] D. Bertsekas and R. Gallager. *Data Networks*. Prentice-Hall, 1987.

[7] D. Bovet, M. Cesati, and A. Oram. *Understanding the Linux Kernel, 2nd Ed*. O'Reilly & Associates, Inc., 2002.

[8] T. Bu and D. Towsley. Fixed point approximations for tcp behavior in an aqm network. In *ACM SIGMETRICS*, Boston, MA, June 2001.

[9] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running commodity operatingsystems on scalable multiprocessors. *ACM Transactions on Computer Systems*, volume 15(number 4):pp 412–447, 1997.

[10] J. Carlstrom and R. Rom. Application-aware admission control and scheduling in web servers. In *Proceedings of IEEE INFOCOM*, June 2002.

[11] H. Chen and P. Mohapatra. Session-based overload control in qos-aware web servers. In *Proceedings of IEEE INFOCOM*, June 2002.

[12] D. D. Clark. The design philosophy of the DARPA internet protocols. In *Proceedings of ACM SIGCOMM*, 1988.

[13] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Proceedings of the ACM SIGCOMM*, Austin, TX, September 1989.

[14] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without runtime checks or garbage collection. In *Proc. Languages Compilers and Tools for Embedded Systems 2003*, San Diego, CA, June 2003.

[15] Y. Diao, N. Gandhi, S. Parekh, J. Hellerstein, and D. Tilbury. Using mimo feedback control to enforce policies for interrelated metrics with application to the apache web server. In *Proceedings of the Network Operations and Management Symposium 2002*, Florence, Italy, April 2002.

[16] D. R. Engler, F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of SOSP*, pages 251–266, Copper Mountain Resort, Colorado, USA, December 1995.

[17] R. Gibbens and F. Kelly. Resource pricing and the evolution of congestion control. *Automatica*, 35:1969–1985, 1999.

[18] S. M. Hand. Self-paging in the nemesis operating system. In *Proceedings of OSDI*, pages 73–86. USENIX Association, 1999.

[19] C. Hollot, V. Misra, D. Towsley, and W. Gong. A control theoretic analysis of red. In *Proceedings of IEEE INFOCOM*, April 2001.

[20] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of c. In *USENIX Annual Technical Conference*, 2002.

[21] S. Jin, L. Guo, I. Matta, and A. Bestavros. A spectrum of tcp-friendly window-based congestion control algorithms. *IEEE/ACM Transactions on Networking, 11(3)*, June 2003.

[22] F. Kelly, A. Maulloo, and D. Tan. Rate control for communication networks: Shadow prices, proporti onal fairness and stability. *Journal of Operations Research Society*, 1998.

[23] S. King, G. W. Dunlap, and P. M. Chen. Operating system support for virtual machines. In *USENIX Annual Technical Conference*, 2003.

[24] J. Liedtke. On $\mu$-kernel construction. In *Proceedings of SOSP*, December 1995.

[25] S. Lim, C. Lee, C. Ahn, C. Lee, and K. Park. An adaptive admission control mechanism for a cluster-based web server system. In *Proceedings of IPDPS'02*, Fort Lauderdale, Florida, April 2002.

[26] S. Low and D. Lapsley. Optimization flow control, I:basic algorithm and convergence. *IEEE/ACM Transactions on Networking*, 1999.

[27] D. Mosberger and T. Jin. httperf- a tool for measuring web server performance. In *Proceedings of the First workshop on Internet Server Performance*, Madison, WI, June 1998.

[28] K. Ogata. Modern control engineering,. Prentice Hall, 2002.

[29] G. Popek and R. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):pp 413–421, July 1974.

[30] A. Robertsson, B. Wittenmark, and M. Kihl. Analysis and design of admission control systems in web-server systems. In *Proceedings of American Control Conference (ACC)*, June 2003.

[31] J. S. Robin and C. IrvineK. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *USENIX Security Symposium*, 2000.

[32] J. Saltzer, D. Reed, and D. Clark. End-to-end arguments in system design. In *ACM Transactions on Computer Systems (TOCS)*, pages Vol.2, No.4 195–206, 1984.

[33] P. J. Shenoy and H. Vin. Cello: A disk scheduling framework for next generation operating systems. In *Proceedings of ACM SIGMETRICS*, Madison, Wisconsin, June 1998.

[34] J. Sugerman, G. Venkitachalam, and B. H. Lim. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference*, pages 1–14, 2001.

[35] The user-mode linux kernel home page: http://user-mode-linux.sourceforge.net/.

[36] Vmware: http://www.vmware.com/.

[37] C. Waldspurger and W. Weihl. Stride scheduling: Deterministic proportional share resource management. In *Technical Memorandum MIT/LCS/TM-528*, June 1995.

[38] M. Welsh and D. Culler. Adaptive overload control for busy internet servers. In *Proceedings of the 4th USENIX Conference on Internet Technologies and Systems*, March 2003.

[39] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the denali isolation kernel. In *Proceedings of OSDI*, Boston, MA, USA, December 2002.