Energy Management for Hypervisor-Based Virtual Machines

Jan Stoess Christian Lang Frank Bellosa
System Architecture Group, University of Karlsruhe, Germany
{stoess, chlang, bellosa}@ira.uka.de

Abstract

Current approaches to power management are based on operating systems with full knowledge of and full control over the underlying hardware; the distributed nature of multi-layered virtual machine environments renders such approaches insufficient. In this paper, we present a novel framework for energy management in modular, multi-layered operating system structures. The framework provides a unified model to partition and distribute energy, and mechanisms for energy-aware resource accounting and allocation. As a key property, the framework explicitly takes the recursive energy consumption into account, which is spent, e.g., in the virtualization layer or subsequent driver components.

Our prototypical implementation targets hypervisor-based virtual machine systems and comprises two components: a host-level subsystem, which controls machine-wide energy constraints and enforces them among all guest OSes and service components, and, complementary, an energy-aware guest operating system, capable of fine-grained application-specific energy management. Guest level energy management thereby relies on effective virtualization of physical energy effects provided by the virtual machine monitor. Experiments with CPU and disk devices and an external data acquisition system demonstrate that our framework accurately controls and stipulates the power consumption of individual hardware devices, both for energy-aware and energy-unaware guest operating systems.

1 Introduction

Over the past few years, virtualization technology has regained considerable attention in the design of computer systems. Virtual machines (VMs) establish a development path for incorporating new functionality – server consolidation, transparent migration, secure computing, to name a few – into a system that still retains compatibility to existing operating systems (OSes) and applications. At the very same time, the ever increasing power density and dissipation of modern servers has turned energy management into a key concern in the design of OSes. Research has proposed several approaches to OS directed control over a computer's energy consumption, including user- and service-centric management schemes. However, most current approaches to energy management are developed for standard, legacy OSes with a monolithic kernel. A monolithic kernel has full control over all hardware devices and their modes of operation; it can directly regulate device activity or energy consumption to meet thermal or energy constraints. A monolithic kernel also controls the whole execution flow in the system. It can easily track the power consumption at the level of individual applications and leverage its application-specific knowledge during device allocation to achieve dynamic and comprehensive energy management.

Modern VM environments, in contrast, consist of a distributed and multi-layered software stack including a hypervisor, multiple VMs and guest OSes, device driver modules, and other service infrastructure (Figure 1). In such an environment, direct and centralized energy management is unfeasible, as device control and accounting information are distributed across the whole system.

At the lowest-level of the virtual environment, the privileged hypervisor and host driver modules have direct control over hardware devices and their energy consumption. By inspecting internal data structures, they can obtain coarse-grained per-VM information on how energy is spent on the hardware. However, the host level does not possess any knowledge of the energy consumption of individual applications. Moreover, with the ongoing trend to restrict the hypervisor's support to a minimal set of hardware and to perform most of the device control in unprivileged driver domains [8,15], hypervisor and driver modules each have direct control over a small set of devices; but they are oblivious to the ones not managed by themselves.

The guest OSes, in turn, have intrinsic knowledge of their own applications. However, guest OSes operate on deprivileged virtualized devices, without direct access to the physical hardware, and are unaware that the hardware may be shared with other VMs. Guest OSes are also unaware of the *side-effects* on power consumption caused by the vir-

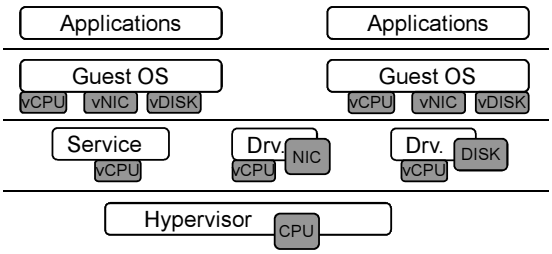

Figure 1: Increasing number of layers and components in today's virtualization-based OSes.

tual device logic: since virtualization is transparent, the "hidden", or recursive power consumption, which the virtualization layer itself causes when requiring the CPU or other resources simply vanishes unaccounted in the software stack. Depending on the complexity of the interposition, resource requirements can be substantial: a recent study shows that the virtualization layer requires a considerable amount of CPU processing time for I/O virtualization [5].

The whole situation is even worsened by the non-partitionability of some of the physical effects of power dissipation: the temperature of a power consuming device, for example, cannot simply be partitioned among different VMs in a way that each one gets alloted its own share on the temperature. Beyond the lack of comprehensive control over and knowledge of the power consumption in the system, we can thus identify the lack of a model to comprehensively express physical effects of energy consumption in distributed OS environments.

To summarize, current power management schemes are limited to legacy OSes and unsuitable for VM environments. Current virtualization solutions disregard most energy-related aspects of the hardware platform; they usually virtualize a set of standard hardware devices only, without any special power management capabilities or support for energy management. Up to now, power management for VMs is limited to the capabilities of the host OS in hosted solutions and mostly dispelled from the server-oriented hypervisor solutions.

Observing these problems, we present a novel framework for managing energy in distributed, multi-layered OS environments, as they are common in today's computer systems. Our framework makes three contributions. The first contribution is a model for partitioning and distributing energy effects; our model solely relies on the notion of energy as the base abstraction. Energy quantifies the physical effects of power consumption in a distributable way and can be partitioned and translated from a global, system-wide notion into a local, component- or user-specific one. The second contribution is a distributed energy accounting approach, which accurately tracks back the energy spent in the system to originating activities. In particular, the presented approach incorporates both the direct and the side-effectual energy consumption spent in the virtualization layers or subsequent driver components. As the third contribution, our framework exposes all resource allocation mechanisms from drivers and other resource managers to the respective energy management subsystems. Exposed allocation enables dynamic and remote regulation of energy consumption in a way that the overall consumption matches the desired constraints.

We have implemented a prototype that targets hypervisor-based systems. We argue that virtual server environments benefit from energy management within and across VMs; hence the prototype employs management software both at host-level and at guest-level. A host-level management subsystem enforces system-wide energy constraints among all guest OSes and driver or service components. It accounts direct and hidden power consumption of VMs and regulates the allocation of physical devices to ensure that each VM does not consume more than a given power allotment. Naturally, the host-level subsystem performs independent of the guest operating system; on the downside, it operates at low level and in coarse-grained manner. To benefit from fine-grained, application-level knowledge, we have complemented the host-level part with an optional energy-aware guest OS, which redistributes the VM-wide power allotments among its own, subordinate applications. In analogy to the host-level, where physical devices are allocated to VMs, the guest OS regulates the allocation of virtual devices to ensure that its applications do not spend more energy than their alloted budget.

Our experiments with CPU and disk devices demonstrate that the prototype effectively accounts and regulates the power consumption of individual physical and virtual devices, both for energy-aware and energy-unaware guest OSes.

The rest of the paper is structured as follows: In Section 2, we present a generic model to energy management in distributed, multi-layered OS environments. We then detail our prototypical implementation for hypervisor-based systems in Section 3. We present experiments and results in Section 4. We then discuss related approaches in Section 5, and finally draw a conclusion and outline future work in Section 6.

2 Distributed Energy Management

The following section presents the design principles we consider to be essential for distributed energy management. We begin with formulating the goals of our work. We then describe the unified energy model that serves as a foundation for the rest of our approach. We finally describe the overall structure of our distributed energy management framework.

2.1 Design Goals

The increasing number of layers, components, and subsystems in modern OS structures demands for a distributed approach to control the energy spent in the system. The approach must perform effectively across protection boundaries, and it must comprise different types of activities, software abstractions, and hardware resources. Furthermore, the approach must be flexible enough to support diversity in energy management paradigms. The desire to control power and energy effects of a computer system stems from a variety of objectives: Failure rates typically increase with the temperature of a computer node or device; reliability requirements or limited cooling capacities thus directly translate into *temperature constraints*, which are to be obeyed for the hardware to operate correctly. Specific *power limits*, in turn, are typically imposed by battery or backup generators, or by contracts with the power supplier. Controlling power consumption on a *per-user base* finally enables accountable computing, where customers are billed for the energy consumed by their applications, but also receive a guaranteed level or quality of service. However, not only the objectives for power management are diverse; there also exists a variety of algorithms to achieve those objectives. Some of them use real temperature sensors, whereas others rely on estimation models [3, 12]. To reach their goals, the algorithms employ different mechanisms, like throttling resource usage, request batching, or migrating of execution [4, 9, 17]. Hence, a valid solution must be flexible and extensible enough to suit a diversity of goals and algorithms.

2.2 Unified Energy Model

To encompass the diverse demands on energy management, we propose to use the notion of energy as the base abstraction in our system, an approach which is similar to the currentcy model in [28]. The key advantage of using energy is that it quantifies power consumption in a partitionable way – unlike other physical effects of power consumption such as the temperature of a device. Such effects can easily be expressed as energy constraints, by means of a thermal model [3, 12]. The energy constraints can then be partitioned from global notions into local, component-wise ones. Energy constraints also serve as a coherent base metric to unify and integrate management schemes for different hardware devices.

2.3 Distributed Management

Current approaches to OS power management are tailored to single building-block OS design, where one kernel instance manages all software and hardware resources. We instead model the OS as a set of components, each responsible for controlling a hardware device, exporting a service library, or providing a software resource for use by applications.

Our design is guided by the familiar concept of separating policy and mechanism. We formulate the procedure of energy management as a simple feedback loop: the first step is to determine the current power consumption and to account it to the originating activities. The next step is to analyze the accounting data and to make a decision based on a given policy or goal. The final step is to respond with allocation or de-allocation of energy consuming resources to the activities, with the goal to align the energy consumption with the desired constraints.

We observe that mainly the second step is associated with policy, whereas the two other steps are mechanisms, bound to the respective providers of the resource, which we hence call *resource drivers*. We thus model the second step as an *energy manager module*, which may, but need not reside in a separate software component or protection domain. Multiple such managers may exist concurrently the system, at different position in the hierarchy and with different scopes.

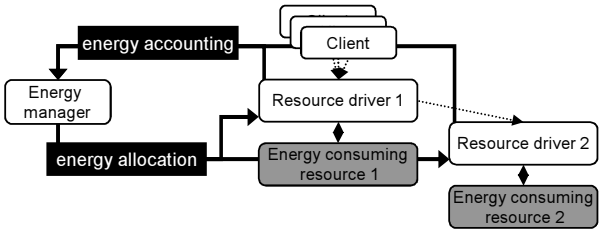

Figure 2: Distributed energy management. Energy managers may reside in different components or protection domains. Resource drivers consume resources themselves, for which the energy is accounted back to the original clients.

Each energy manager is responsible for a set of subordinate resources and their energy consump-

tion. Since the system is distributed, the resource manager cannot assume direct control or access over the resource; it requires remote mechanisms to account and allocate the energy (see Figure 2). Hence, by separating policy from mechanism, we translate our general goal of distributed energy management into the two specific aspects of *distributed energy accounting* and *dynamic, exposed resource allocation*; these are the subject of the following paragraphs.

Distributed energy accounting Estimating and accounting the energy of a physical device usually requires detailed knowledge of the particular device. Our framework therefore requires each driver of an energy consuming device or resource to be capable of determining (or estimating) the energy consumption of its resources. Likewise, it must be capable to account the consumption to its consumers. If the energy management software resides outside the resource driver, it must propagate the accounting information to the manager.

Since the framework does not assume a single kernel comprising all resource subsystems, it has to track energy consumptions across module boundaries. In particular, it must incorporate the recursive energy consumption: that is, the driver of a given resource such as a disk typically requires other resources, like the CPU, in order to provide its service successfully. Depending on the complexity, such recursive resource consumption may be substantial; consider, as examples, a disk driver that transparently encrypts and decrypts its client requests, or a driver that forwards client requests to a network attached storage server via a network interface card. Recursive resource consumption requires energy, which must be accounted back to the clients. In our example, it would be the responsibility of disk driver to calculate its clients' shares of the disk and on its own CPU energy. To determine its CPU energy, the driver must recursively query the driver of the CPU resource, which is the hypervisor in our case.

Dynamic and exposed resource allocation To regulate the energy spent on a device or resource, each driver must expose its allocation mechanisms to energy manager subsystems. The manager leverages the allocation mechanisms to ensure that energy consumption matches the desired constraints. Allocation mechanisms relevant for energy management can be roughly distinguished into hardware and software mechanisms. Hardware-provided power saving features typically provide a means to change power consumption of a device, by offering several modes of operation with different efficiency and energy coefficients (e.g., halt cycles or different active and sleep modes). The ultimate goal is to achieve the optimal level of efficiency with respect to the current resource utilization, and to reduce the wasted power consumption. Software-based mechanisms, in turn, rely on the assumption that energy consumption depends on the level of utilization, which is ultimately dictated by the number of device requests. The rate of served requests can thus be adapted by software to control the power consumption.

3 A Prototype for Hypervisor-Based Systems

Based on the design principles presented above, we have developed a distributed, two-level energy management framework for hypervisor-based VM systems. The prototype employs management software both at host-level and at guest-level. It currently supports management of two main energy consumers, CPU and disk. CPU services are directly provided by the hypervisor, while the disk is managed by a special device driver VM. In the following section, we first describe the basic architecture of our prototype. We then present the energy model for CPU and disk devices. We then describe the host-level part, and finally the guest-level part of our energy management prototype.

3.1 Prototype Architecture

Our prototype uses the L4 micro-kernel as the privileged hypervisor, and para-virtualized Linux kernel instances running on top of it. L4 provides core abstractions for user level resource management: virtual processors (kernel threads), synchronous communication, and mechanisms to recursively construct virtual address spaces. I/O devices are managed at user-level; L4 only deals with exposing interrupts and providing mechanisms to protect device memory.

The guest OSes are adaptions of the Linux 2.6 kernel, modified to run on top of L4 instead of on bare hardware [11]. For managing guest OS instances, the prototype includes a user-level VM monitor (VMM), which provides the virtualization service based on L4's core abstractions. To provide user-level device driver functionality, the framework dedicates a special device driver VM to each device, which exports a virtual device interface to client VMs and multiplexes virtual device requests onto the physical device. The driver VMs are Linux guest OS instances

themselves, which encapsulate and reuse standard Linux device driver logic for hardware control [15].

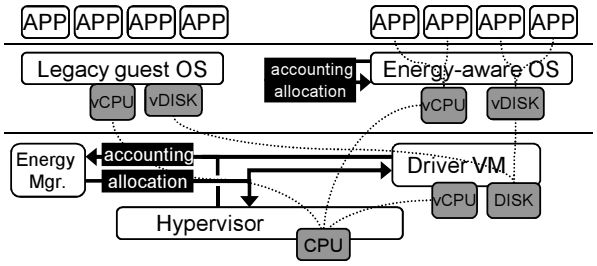

Figure 3: Prototype architecture. The host-level subsystem controls system-wide energy constraints and enforces them among all guests. A complementary energy-aware guest OS is capable of performing its own, application-specific energy management.

The prototype features a host-level energy manager module responsible for controlling the energy consumption of VMs on CPUs and disk drives. The energy manager periodically obtains the per-VM CPU and disk energy consumption from the hypervisor and driver VM, and matches them against a given power limit. To bring both in line, it responds by invoking the exposed throttling mechanisms for the CPU and disk devices. Our energy-aware guest OS is a modified version of L4Linux that implements the resource container abstraction [1] for resource management and scheduling. We enhanced the resource containers to support energy management of virtual CPUs and disks. Since the energy-aware guest OS requires virtualization of the energy effects of CPU and disk, the hypervisor and driver VM propagate their accounting records to the user-level VM monitor. The monitor then creates, for each VM, a local view on the current energy consumption, and thereby enables the guest to pursue its own energy-aware resource management. Note, that our energy-aware guest OS is an optional part of the prototype: it provides the benefit of fine-grained energy management for Linux-compatible applications. For all energy-unaware guests, our prototype resorts to the coarser-grained host-level management, which achieves the constraints regardless whether the guest-level subsystem is present or not.

Figure 3 gives a schematic overview of the basic architecture. Our prototype currently runs on IA-32 microprocessors. Certain parts, like the device driver VMs, are presently limited to single processor systems; we are working on multi-processor support and will integrate it into future versions.

3.2 Device Energy Models

In the following section, we present the device energy models that serve as a base for CPU and disk accounting. We generally break down the energy consumption into *access* and *idle consumption*. Access consumption consists of the energy spent when using the device. This portion of the energy consumption can be reduced by controlling device allocation, e.g., in terms of the client request rate. Idle consumption, in turn, is the minimum power consumption of the device, which it needs even when it does not serve requests. Many current microprocessors support multiple sleep and active modes, e.g., via frequency scaling or clock gating. A similar technology, though not yet available on current standard servers, can be found in multi-speed disks, which allow lowering the spinning speed during phases of low disk utilization [10]. To retain fairness, we propose to decouple the power state of a multi-speed device from the accounting of its idle costs. Clients that do not use the device are charged for the lowest active power state. Higher idle consumptions are only charged to the clients are actively using the device.

3.2.1 CPU Energy Model

Our prototype leverages previous work [3, 13] and bases CPU energy estimation on the rich set of performance counters featured by modern IA-32 microprocessors. For each performance counter event, the approach assigns a weight representing its contribution to the processor energy. The weights are the result of a calibration procedure that employs test applications with constant and known power consumptions and physical instrumentation of the microprocessors [3]. Previous experiments have demonstrated that this approach is fairly accurate for integer applications, with an error of at most 10 percent. To obtain the processor energy consumption during a certain period of time, e.g., during execution of a VM, the prototype sums up the number of events that occurred during that period, multiplied with their weights. The time stamp counter, which counts clock cycles regardless whether the processor is halted or not, yields an accurate estimation of the CPU's idle consumption.

3.2.2 Disk Energy Model

Our disk energy model differs from the CPU model in that it uses a time-based approach rather than event sampling. Instead of attributing energy consumption to events, we attribute power consumption to different device states, and calculate the time the

device requires to transfer requests of a given size. There is no conceptual limit to the number of power states. However, we consider suspending the disk to be an unrealistic approach for hypervisor systems; for lack of availability, we do not consider multi-speed disks as well. We thus distinguish two different power states: active and idle.

To determine the transfer time of a request – which is equal to the time the device must remain in active state to handle it –, we divide the size of the request by the disk's transfer rate in bytes per second. We calculate the disk transfer rate dynamically, in intervals of 50 requests. Although we ignore several parameters that affect the energy consumption of requests, (e.g., seek time or the rotational delays), our evaluation shows that our simple approach is sufficiently accurate. Our observation is substantiated by the study in [26], which indicates that such a 2-parameter model is inaccurate only because of sleep-modes, which we can safely disregard for our approach.

3.3 Host-Level Energy Management

Our framework requires each driver of a physical device to determine the device's energy consumption and to account the consumption to the client VMs. The accounting infrastructure uses the device energy model presented above: Access consumption is charged directly to each request, after the request has been fulfilled. The idle consumption, in turn, cannot be attributed to specific requests; rather, it is alloted to all client VMs in proportion to their respective utilization. For use by the energy manager and others, the driver grants access to its accounting records via shared memory and updates the records regularly.

In addition to providing accounting records, each resource driver exposes its allocation mechanisms to energy managers and other resource management subsystems. At host-level, our framework currently supports two allocation mechanisms: CPU throttling and disk request shaping. CPU throttling can be considered as a combined software-hardware approach, which throttles activities in software and spends the unused time in halt cycles. Our disk request shaping algorithm is implemented in software.

In the remainder of this section, we first explain how we implemented runtime energy accounting and allocation for CPU and the disk devices. We then detail how the these mechanisms enable our energy management software module to keep the VMs' energy consumption within constrained limits.

3.3.1 CPU Energy Accounting

To accurately account the CPU energy consumption, we trace the performance counter events within the hypervisor and propagate them to the user-space energy manager module. Our approach extends our previous work to support resource management via event logging [20] to the context of energy management. The tracing mechanism instruments context switches between VMs within the hypervisor; at each switch, it records the current values of the performance counters into an in-memory log buffer. The hypervisor memory-maps the buffers into the address space of the energy manager. The energy manager periodically analyzes the log buffer and calculates the energy consumption of each VM (Figure 4).

By design, our tracing mechanism is asynchronous and separates performance counter accumulation from their analysis and the derivation of the energy consumption. It is up to the energy manager to perform the analysis often enough to ensure timeliness and accuracy. Since the performance counter logs are relatively small, we consider this to be easy to fulfil; our experience shows that the performance counter records cover a few hundred or thousand bytes, if the periodical analysis is performed about every 20th millisecond.

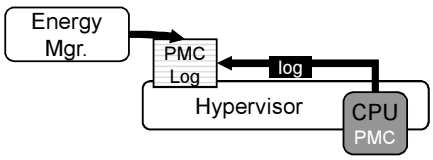

Figure 4: The hypervisor collects performance counter traces and propagates the trace logs to user-space energy managers.

The main advantage of using tracing for recording CPU performance counters is that it separates policy from mechanism. The hypervisor is extended by a simple and cheap mechanism to record performance counter events. All aspects relevant to energy estimation and policy are kept outside the hypervisor, within the energy manager module. A further advantage of in-memory performance counter records is that they can easily be shared – propagating them to other guest-level energy accountants is a simple matter of leveraging the hypervisor's memory-management primitives.

In the current prototype, the energy manager is invoked every 20 ms to check the performance counter logs for new records. The log records contain the performance counter values relevant for en-

ergy accounting, sampled at each context switch together with an identifier of the VM that was active on the CPU. For each period between subsequent context switches, the manager calculates the energy consumption during that period, by multiplying the advance of the performance counters with their weights. Rather than charging the complete energy consumption to the active VM, the energy manager subtracts the idle cost and splits it between all VMs running on that processor. The time stamp counter, which is included in the recorded performance counters, provides an accurate estimation of the processor's idle cost. Thus the energy estimation looks as follows:

```
/* per-VM idle energy based on TSC advance (pmc0) */
for (id = 0; id < max_vms; id++)
   vm[id].cpu_idle += weight[0] * pmc[0] / max_vms;

/* calculate and charge access energy (pmc1..pmc8) */
for (p=1; p < 8; p++)
  vm[cur_id].cpu_access += weight[p] * pmc[p];
```

3.3.2 Disk Energy Accounting

To virtualize physical disks drives, our framework reuses legacy Linux disk driver code by executing it inside VMs. The driver functionality is exported via a translation module that mediates requests between the device driver and external client VMs. The translation module runs in the same address space as the device driver and handles all requests sent to and from the driver. It receives disk requests from other VMs, translates them to basic Linux block I/O requests, and passes them to the original device driver. When the device driver has finalized the request, the module again translates the result and returns it to the client VM.

The translation module has access to all information relevant for accounting the energy dissipated by the associated disk device. We implemented accounting completely in this translation module, without changing the original device driver. The module estimates the energy consumption of the disk using the energy model presented above. When the device driver has completed a request, the translation module estimates the energy consumption of the request, depending on the number of transferred bytes:

```
/* estimate transfer cost for size bytes */
vm[cur_id].disk_access += (size / transfer_rate)
          * (active_disk_power - idle_disk_power);
```

Because the idle consumption is independent of the requests, it does not have to be calculated for each request. However, the driver must recalculate it periodically, to provide the energy manager with up-to-date accounting records power consumption of the disk. For that purpose, the driver invokes the following procedure periodically every 50 ms:

```
/* estimate idle energy since last time */
idle_disk_energy = idle_disk_power * (now - last)
                   / max_client_vms;
for (id = 0; v < max_client_vms; id++)
   vm[id].disk_idle += idle_disk_energy;
```

3.3.3 Recursive Energy Accounting

Fulfilling a virtual device request issued by a guest VM may involve interacting with several different physical devices. Thus, with respect to host-level energy accounting, it is not sufficient to focus on single physical devices; rather, accounting must incorporate the energy spent recursively in the virtualization layer or subsequent service.

We therefore perform a recursive, request-based accounting of the energy spent in the system, according to the design principles presented in Section 2. In particular, each driver of a physical device determines the energy spent for fulfilling a given request and passes the cost information back to its client. If the driver requires other devices to fulfill a request, it charges the additional energy to its clients as well. Since idle consumption of a device cannot be attributed directly to requests, each driver additionally provides an "electricity meter" for each client. It indicates the client's share in the total energy consumption of the device, including the cost already charged with the requests. A client can query the meter each time it determines the energy consumption of its respective clients.

As a result, recursive accounting yields a distributed matrix of virtual-to-physical transactions, consisting of the idle and the active energy consumption of each physical device required to provide a given virtual device (see Figure 5). Each device driver is responsible for reporting its own vector of the physical device energy it consumes to provide its virtual device abstraction.

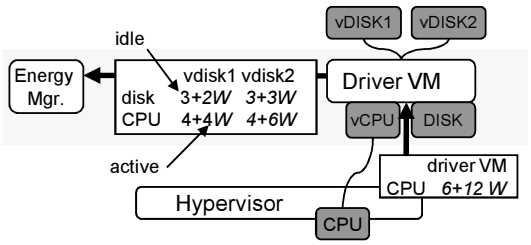

Figure 5: Recursive accounting of disk energy consumption; for each client VM and physical device, the driver reports idle and active energy to the energy manager. The driver is assumed to consume 8W CPU idle power, which is apportioned equally to the two clients.

Since our framework currently supports CPU and disk energy accounting, the only case where recursive accounting is required occurs in the virtual disk driver located in the driver VM. The cost for the virtualized disk consists of the energy consumed by the disk and the energy consumed by the CPU while processing the requests. Hence, our disk driver also determines the processing energy for each request in addition to the disk energy as presented above.

As with disk energy accounting, we instrumented the translation module in the disk driver VM to determine the active and idle CPU energy per client VM. The Linux disk driver combines requests to get better performance and delays part of the processing in work-queues and tasklets. When determining the active CPU energy, it would be infeasible to track the CPU energy consumption of each individual request. Instead, we retrieve the CPU energy consumption at times and apportion it between the requests. Since the driver runs in a VM, it relies on the energy virtualization capabilities of our framework to retrieve a local view on the CPU energy consumption (details on energy virtualization are presented in Section 3.4).

The Linux kernel constantly consumes a certain amount of energy, even if it does not handle disk requests. According to our energy model, we do not charge idle consumption with the request. To be able to distinguish the idle driver consumption from the access consumption, we approximate the idle consumption of the Linux kernel when no client VM uses the disk.

To account active CPU consumption, we assume constant values per request, and predict the energy consumption of future requests based on the past. Every 50th request, we estimate the driver's CPU energy consumption by means of virtualized performance monitoring counters and adjust the expected cost for the next 50 requests. The following code illustrates how we calculate the cost per request. In the code fragment, the static variable `unaccounted_cpu_energy` keeps track of the deviation between the consumed energy and the energy consumption already charged to the clients. The function `get_cpu_energy()` returns the guest-local view of the current idle and active CPU energy since the last query.

```
/* subtract idle CPU consumption of driver VM */
unaccounted_cpu_energy -= drv_idle_cpu_power
                          * (now - last);

/* calculate cost per request */
num_req = 50;
unaccounted_cpu_energy += get_cpu_energy();
unaccounted_cpu_energy -= cpu_req_energy * num_req;
cpu_req_energy = unaccounted_cpu_energy / num_req;
```

3.3.4 CPU Resource Allocation

To regulate the CPU energy consumption of individual machines, our hypervisor provides a mechanism to throttle the CPU allocation at runtime, from user-level. The hypervisor employs a stride scheduling algorithm [21, 23] that allots proportional CPU shares to virtual processors; it exposes control over the shares to selected, privileged user-level components. The host-level energy manager dynamically throttles a virtual CPU's energy consumption by adjusting the alloted share accordingly. A key feature of stride scheduling is that it does not impose fixed upper bounds on CPU utilization: the shares have only relative meaning, and if one virtual processor does not fully utilize its share, the scheduler allows other, competing virtual processors to steal the unused remainder. An obvious consequence of dynamic upper bounds is that energy consumption will not be constrained either, at least not with a straight-forward implementation of stride scheduling. We solved this problem by creating a distinct and privileged *idle virtual processor* per CPU, which is guaranteed to spend all alloted time with issuing halt instructions (we modified our hypervisor to translate the idle processor's virtual halt instructions directly into real ones). Initially, each idle processor is alloted only a minuscule CPU share, thus all other virtual processors will be favored on the CPU if they require it. However, to constrain energy consumption, the energy manager will decrease the CPU shares of those virtual processors, and idle virtual processor will directly translate the remaining CPU time into halt cycles. Our approach guarantees that energy limits are effectively imposed; but it still preserves the advantageous processor stealing behavior for all other virtual processors. It also keeps the energy-policy out of the hypervisor and allows, for instance, to improve the scheduling policy with little effort, or to exchange it with a more throughput-oriented one for those environments where CPU energy management is not required.

3.3.5 Disk Request Shaping

To reduce disk power consumption, we pursue a similar approach and enable a energy manager to throttle disk requests of individual VMs. Throttling the request rate not only reduces the direct access consumption of the disk; it also reduces the recursive CPU consumption which the disk driver requires to process, recompute, and issue requests. We implemented the algorithm as follows: the disk driver processes a client VM's disk requests only to a specific request budget, and it delays all pending requests.

The driver periodically refreshes the budgets according to the specific throttling level set by the energy manager. The algorithm is illustrated by the following code snippet:

```
void process_io(client_t *client)
{
   ring = &client->ring;

   for (i=0; i < client->budget; i++)
   {
      desc = &client->desc[ ring->start ];
      ring->start = (ring->start+1) % ring->cnt;
      initiate_io(conn, desc, ring);
   }
}
```

3.3.6 Host-level Energy Manager

Our host-level energy manager manager relies on the accounting and allocation mechanisms described previously, and implements a simple policy that enforces given device power limits on a per-VM base. The manager consists of an initialization procedure and a subsequent feedback loop. During initialization, the manager determines a power limit for each VM and device type, which may not be exceeded during runtime. The CPU power limit reflects the active CPU power a VM is allowed to consume directly. The disk power limit reflects the overall active power consumption the disk driver VM is allowed to spend in servicing a particular VM, including the CPU energy spent for processing (Nevertheless, the driver's CPU and disk energy are accounted separately, as depicted by the matrix in Figure 5). Finding an optimal policy for allotment of power budgets is not the focus of our work; at present, the limits are set to static values.

The feedback loop is invoked periodically, every 100 ms for the CPU and every 200 ms for the disk. It first obtains the CPU and disk energy consumption of the past interval by querying the accounting infrastructure. The current consumptions are used to predict future consumptions. For each device, the manager compares the VM's current energy consumption with the desired power limit multiplied with the time between subsequent invocations. If they do not match for a given VM, the manager regulates the device consumption by recomputing and propagating the CPU strides and disk throttle factors respectively. To compute a new CPU stride, the manager adds or subtracts a constant offset from the current value. When computing the disk throttle factor, the manager takes the past period into consideration, and calculates the offset Δt according to the following formula. In this formula, e_c denotes the energy consumed, e_l the energy limit per period, and t and t_l and denote the present and past disk throttle factors; viable throttle factors range from 0 to a few thousand:

$$\Delta t = \begin{cases} \frac{1}{4}(t_l - t) + \frac{e_l - e_c}{|e_l - e_c|} & : \begin{cases} e_c > e_l, t > t_l \\ e_c < e_l, t < t_l \end{cases} \\ \frac{4}{3}(t - t_l) + \frac{e_l - e_c}{|e_l - e_c|} & : \quad else \end{cases}$$

3.4 Virtualized Energy

To enable application-specific energy management, our framework supports accounting and control not only for physical but also of virtual devices. In fact, the advantage of having guest-level support for energy accounting is actually twofold: first, it enables guest resource management subsystems to leverage their application-specific knowledge; second, it allows drivers and other components to recursively determine the energy for their services.

The main difference between a virtual device and other software services and abstractions lies in its interface: a virtual device closely resembles its physical counterpart. Unfortunately, most current hardware devices offer no direct way to query energy or power consumption. The most common approach to determine the energy consumption is to estimate it based on certain device characteristics, which are assumed to correlate with the power or energy consumption of the device. By emulating the according behavior for the virtual devices, we support energy estimation in the guest without major modifications to the guest's energy accounting. Our ultimate goal is to enable the guest to use the same driver for virtual and for real hardware. In the remainder of this section, we first describe how we support energy accounting of virtual CPU and disk. We then present the implementation of our energy-aware guest OS, which provides the support for application-specific energy management.

3.4.1 Virtual CPU Energy Accounting

For virtualization of physical energy effects of the CPU, we provide a virtual performance counter model that gives guest OSes a private view of their current energy consumption. The virtual model relies on the tracing of performance counters within the hypervisor, which we presented in Section 3.3.1. As mentioned, not only an energy-aware guest OS requires the virtual performance counters; the specialized device driver VM uses them as well, when recursively determining the CPU energy for its disk services.

Like their physical counterparts, each virtual CPU has a set of virtual performance counters, which

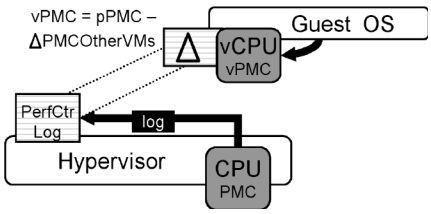

Figure 6: Virtualizing performance counters via hypervisor-collected performance counter traces.

factor out the events of other, simultaneously running VMs. If a guest OS determines the current value of a virtual performance counter, an emulation routine in the in-place monitor obtains the current hardware performance counter and subtracts all advances of the performance counters that occurred when other VMs were running. The hardware performance counters are made read-accessible to user-level software by setting a control register flag in the physical processors. The advances of other VMs are derived from the performance counter log buffers. To be accessible by the in-place VMM, the log buffers are mapped read-only into the address space of the guest OS.

3.4.2 Virtual Disk Energy Accounting

In contrast to the CPU, the disk energy estimation schemes does not rely on on-line measurements of sensors or counters; rather, it is based on known parameters such as the disk's power consumption in idle and active mode and the time it remains in active mode to handle a request. Directly translating the energy consumption of physical devices from our run-time energy model to the parameter-based model of the respective guest OS would yield only inaccurate results. The VMM would have to calibrate the energy consumption of the devices to calculate the energy parameters of the virtual devices. Furthermore, parameters of shared devices may change with the number of VMs, which contradicts the original estimation model. To ensure accuracy in the long run, the guest would have to query the virtual devices regularly for updated parameters.

For our current virtual disk energy model, we therefore use a para-virtual device extension. We expose each disk energy meter as an extension of the virtual disk device; energy-aware guest operating systems can take advantage of them by customizing the standard device driver appropriately.

3.4.3 An Energy-aware Guest OS

For application-specific energy management, we have incorporated the familiar resource container concept into a standard version of our para-virtualized Linux 2.6 adoption. Our implementation relies on a previous approach to use resource containers in the context of CPU energy management [3,24]. We extended the original version with support for disk energy. No further efforts were needed to manage virtual CPU energy; we only had to virtualize the performance counters to get the original version to run.

Similar to the host-level subsystem, the energy-aware guest operating system performs scheduling based on energy criteria. In contrast to standard schedulers, it uses resource containers as the base abstraction rather than threads or processes. Each application is assigned to a resource container, which then accounts all energy spent on its behalf. To account virtual CPU energy, the resource container implementation retrieves the (virtual) performance counter values on container switches, and charges the resulting energy to the previously active container. A container switch occurs on every context switch between processes residing in different containers.

To account virtual disk energy, we enhanced the client-side of the virtual device driver, which forwards disk requests to the device driver VM. Originally, the custom device driver received single disk requests from the Linux kernel, which contained no information about the user-level application that caused it. We added a pointer to a resource container to every data structure involved in a read or write operation. When an application starts a disk operation, we bind the current resource container to the according page in the page cache. When the kernel writes the pages to the virtual disk, we pass the resource container on to the respective data structures (i.e., buffer heads and bio objects). The custom device driver in the client accepts requests in form of bio objects and translates them to a request for the device driver VM. When it receives the reply together with the cost for processing the request, it charges the cost to the resource container bound to the bio structure.

To control the energy consumption of virtual devices, the guest kernel redistributes its own, VM-wide power limits to subordinate resource containers, and enforces them by means of preemption. Whenever a container exhausts the energy budget of the current period (presently set to 50 ms), it is preempted until a refresh occurs in the next period. A

simple user-level application retrieves the VM wide budgets from host-level energy-manager and passes them onto the guest kernel via special system calls.

4 Experiments and Results

In the following section, we present experimental results we obtained from our prototype. Our main goal is to demonstrate that our infrastructure provides an effective solution to manage energy in distributed, multi-layered OSes. We consider two aspects as relevant: At first, we validate the benefits of distributed energy accounting. We then present experiments that aim to show the advantages of multi-layered resource allocation to enforce energy constraints.

For CPU measurements, we used a Pentium D 830 with two cores at 3GHz. Since our implementation is currently limited to single processor systems, we enabled only on one core, which always ran at its maximum frequency. When idle, the core consumes about 42W; under full load, power consumption may be 100W and more. We performed disk measurements on a Maxtor DiamondMax Plus 9 IDE hard disk with 160GB size, for which we took the active power (about 5.6W) and idle power (about 3.6W) characteristics from the data sheet [16]. We validated our internal, estimation-based accounting mechanisms by means of an external high-performance data acquisition (DAQ) system, which measured the real disk and CPU power consumption with a sampling frequency of 1KHz.

4.1 Energy Accounting

To evaluate our approach of distributed energy accounting, we measured the overall energy required for using a virtual disk. For that purpose, we ran a synthetic disk stress test within a Linux guest OS. The test runs on a virtual hard drive, which is multiplexed on the physical disk by the disk driver VM. The test performs almost no computation, but generates heavy disk load. By opening the virtual disk in *raw* access mode, the test bypasses most of the guest OS's caching effects, and causes the file I/O to be performed directly to and from user space buffers. Afterwards, the test permanently reads (writes) consecutive disk blocks of a given size from (to) the disk, until a maximum size has been reached. We performed the test for block sizes from 0.5 KByte up to 32 KByte. We obtained the required energy per block size to run the benchmark from our accounting infrastructure.

The results for the read case are shown in Figure 7. The write case yields virtually the same energy distribution; for reasons of space, we do not show it here. For each size, the figure shows disk and CPU power consumption of the client and the device driver VM. The lowermost part of each bar shows the base CPU power consumption required by core components such as the hypervisor and the user-level VMM (36W); this part is consumed independently of any disk load. The upper parts of each bar show the active and idle power consumption caused by the stress test, broken into CPU and disk consumption. Since the client VM is the only consumer of the hard disk, it is accounted the complete idle disk power (3.5) and CPU power (8W) consumed by the driver VM. Since the benchmark saturates the disk, the active disk power consumption of the disk driver mostly stays at its maximum (2W), which is again accounted to the client VM as the only consumer. Active CPU power consumption in the driver VM heavily depends on the block size and ranges from 9W for small block sizes down to 1W for large ones. Note that the CPU costs for processing a virtual disk request may even surpass the costs for handling the request on the physical disk. Finally, active CPU power consumption in the client VM varies with the block sizes as well, but at a substantially lower level; the lower level comes unsurprising, as the benchmark bypasses most parts of the disk driver in the client OS. The thin bar on the right of each energy bar shows the real power consumption of the CPU and disk, measured with the external DAQ system.

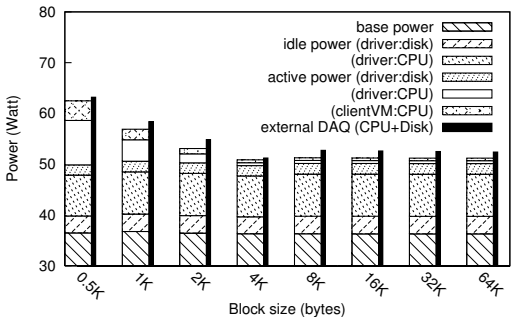

Figure 7: Energy distribution for CPU and disk during the disk stress test. The thin bar shows the real CPU and disk power consumption, measured with an external DAQ system.

4.2 Enforcing Power Constraints

To demonstrate the capabilities of VM-based energy allocation, and to evaluate the behavior of our disk throttling algorithm over time, we performed a second experiment with *two* clients that simultaneously

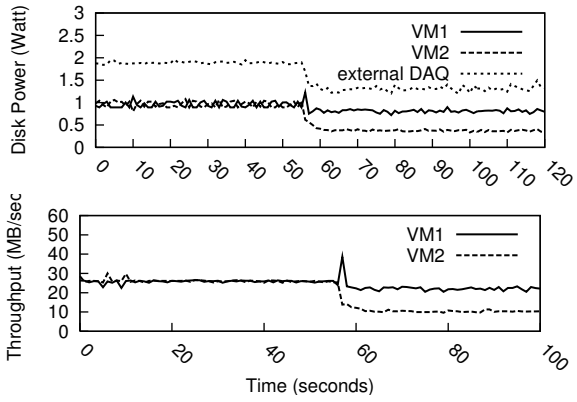

Figure 8: Disk power consumption and throughput of two constrained disk test simultaneously running in two different guest VMs.

require disk service from the driver. The clients interface with a single disk driver VM, but operate on distinct hard disk partitions. We set the active driver power limit of client VM 1 to 1W and the limit of client VM 2 to 0.5W, and periodically obtained driver energy and disk throughput over a period of about 2 minutes. Figure 8 shows both distributions; we set the limit about 45 seconds after having started the measurements. Our experiment demonstrates the driver's capabilities to VM-specific control over power consumption. The internal accounting and control furthermore corresponds with the external measurements.

4.2.1 Guest-Level Energy Allocation

In the next experiment, we compared the effects of enforcing power limits at the host-level against the effects of guest-level enforcement. In the first part of the experiment, we ran two instances of the compute-intensive bzip2 application within an energy-unaware guest OS. In the unconstrained case, a single bzip2 instance causes an active CPU power consumption of more than 50W. The guest, in turn, is alloted an overall CPU active power of only 40W. As the guest is not energy-aware, the limit is enforced by the host-level subsystem. In the second part, we used an energy-aware guest, which complies with the alloted power itself. It redistributes the budget among the two bzip2 instances using the resource container facility. Within the guest, we set the application-level power limits to 10W for the first, and to 30W for the second bzip2 instance. Note that the power limits are effective limits; strictly spoken, both bzip2 still consume each 50 Joules per second when running; however, the resource container implementation reduces the each task time accordingly, with the result that over time, the limits are effectively obeyed.

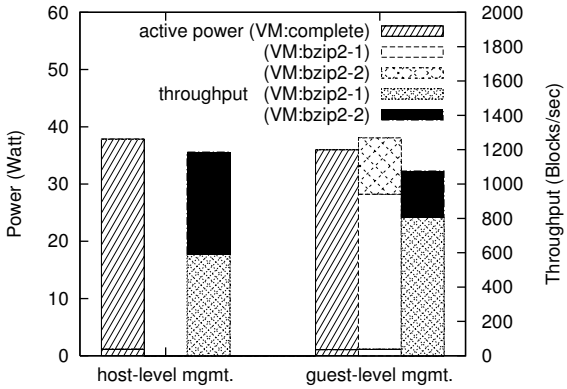

Figure 9: Guest-Level energy redistribution.

The results are given in Figure 9. For both cases, the figure shows overall active CPU power of the guest VM in the leftmost bar, and the throughput broken down to each bzip2 instance in the rightmost bar. For the energy-aware VM, we additionally obtained the power consumption per bzip2 instance as seen by the guest's energy management subsystem itself; it is drawn as the bar in the middle.

Note that the guest's view of the power consumption is slightly higher than the view of the host-level energy manager. Hence, the guest imposes somewhat harsher power limits, and causes the overall throughput of both bzip2 instances to drop compared to host-level control. We attribute the differences in estimation to the clock drift and rounding errors in the client.

However, the results are still as expected: host-level control enforces the budgets independent of the guest's particular capabilities – but the enforcement treats all guest's applications as equal and thus reduces the throughput of both bzip2 instances proportionally. In contrast, guest-level management allows the guest to respect its own user priorities and preferences: it allots a higher power budget to the first bzip2 instance, resulting in a higher throughput compared to the second instance.

5 Related Work

There has been a considerable research interest in involving operating systems and applications in the management of energy and power of a computer system [6, 7, 9, 10, 14, 25, 27]. Except for the approach of vertically structured OSes, which will be discussed here, none of them has addressed the prob-

lems that arise if the OS consists of several layers and is distributed across multiple components, as current virtualization environments do. To our knowledge, neither the popular Xen hypervisor [2, 19] nor VMware's most recent hypervisor-based ESX Server [22] support distributed energy accounting or allocation across module boundaries or software layers.

Achieving accurate and easy accounting of energy by vertically structuring an OS was proposed by the designers of *Nemesis* [14,18]. Their approach is very similar to our work in that it addresses accountability issues within multi-layered OSes. A vertically structured system multiplexes all resources at a low level, and moves protocol stacks and most parts of device drivers into user-level libraries. As a result, shared services are abandoned, and the activities typically performed by the kernel are executed within each application itself. Thus, most resource and energy consumption can be accounted to individual applications, and there is no significant anonymous consumption anymore.

Our general observation is that hypervisor-based VM environments are structured similarly to some extent: a hypervisor also multiplexes the system resources at a low level, and lets each VM use its own protocol stack and services. Unfortunately, a big limitation of vertical structuring is that it is hard to achieve with I/O device drivers. As only one driver can use the device exclusively, all applications share a common driver provided by the low-level subsystem. To process I/O requests, a shared driver consumes CPU resources, which recent experiments demonstrate to be substantial in multi-layered systems that are used in practice [5]. In a completely vertically structured system, the processing costs and energy can not be accounted to the applications. In contrast, it was one of the key goals of our work to explicitly account the energy spent in service or driver components.

The *ECOSystem* [27] approach resembles our work in that it proposes to use energy as the base abstraction for power management and to treat it as a first-class OS resource. ECOSystem presents a *currentcy* model that allows to manage the energy consumption of all devices in a uniform way. Apart from its focus on a monolithic OS design, ECOSystem differs from our work in several further aspects. The main goal of ECOSystem is to control energy consumption of mobile systems, in order to extend their battery lifetime. To estimate the energy consumption of individual tasks, ECOSystem attributes power consumptions to different states of each device (e.g. standby, idle, and active states) and charges applications if they cause a device switch to a higher power state. ECOSystem does not distinguish between the fractions contributed by different devices; all cost that a task causes is accumulated to one value. This allows the OS to control the overall energy consumption without considering the currently installed devices. However, it renders the approach too inflexible for other energy management schemes such as thermal management, for which energy consumption must be managed individually per device.

In previous work [3], Bellosa et al. proposed to estimate the energy consumption of the CPU for the purpose of thermal management. The approach leverages the performance monitoring counters present in modern processors to accurately estimate the energy consumption caused by individual tasks. Like the ECOSystem approach, this work uses a monolithic operating system kernel. Also, the estimated energy consumption is just a means to the end of a specific management goal, i.e., thermal management. Based on the energy consumption and a thermal model, the kernel estimates the temperature of the CPU and throttles the execution of individual tasks according to their energy characteristics if the temperature reaches a predefined limit.

6 Conclusion

In this work, we have presented a novel framework for managing energy in multi-layered OS environments. Based on a unified energy model and mechanisms for energy-aware resource accounting and allocation, the framework provides an effective infrastructure to account, distribute, and control the power consumption at different software layers. In particular, the framework explicitly accounts the recursive energy consumption spent in the virtualization layer or subsequent driver components. Our prototypical implementation encompasses a host-level subsystem controlling global power constraints and, optionally, an energy-aware guest OS for local, application-specific power management. Experiments show that our prototype is capable of enforcing power limits for energy-aware and energy-unaware guest OSes.

We see our work as a support infrastructure to develop and evaluate power management strategies for VM-based systems. We consider three areas to be important and prevalent for future work: devices with multiple power states, processors with support for hardware-assisted virtualization, and multi-core architectures. There is no design limit with respect to the integration into our framework, and we are actively developing support for them.

Acknowledgements

We would like to thank Simon Kellner, Andreas Merkel, Raphael Neider, and the anonymous reviewers for their comments and helpful suggestions. This work was in part supported by the Intel Corporation.

References

[1] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, pages 45–58, Berkeley, CA, Feb. 1999.

[2] 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 the 19th Symposium on Operating System Principles*, pages 164–177, Bolton Landing, NY, Oct. 2003.

[3] F. Bellosa, A. Weissel, M. Waitz, and S. Kellner. Event-driven energy accounting for dynamic thermal management. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power*, pages 1–10, New Orleans, LA, Sept. 2003.

[4] R. Bianchini and R. Rajamony. Power and energy management for server systems. *IEEE Computer*, 37(11):68–74, 2004.

[5] L. Cherkasova and R. Gardner. Measuring CPU overhead for I/O processing in the Xen virtual machine monitor. In *Proceedings of the USENIX Annual Technical Conference*, pages 387–390, Anaheim, CA, Apr. 2005.

[6] K. Flautner and T. N. Mudge. Vertigo: automatic performance-setting for Linux. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 105–116, Boston, MA, Dec. 2002.

[7] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proceedings of the 17th Symposium on Operating System Principles*, pages 48–63, Charleston, SC, Dec. 1999.

[8] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, Boston, MA, Oct. 2004.

[9] M. Gomaa, M. D. Powell, and T. N. Vijaykumar. Heat-and-run: leveraging SMT and CMP to manage power density through the operating system. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 260–270, Boston, MA, Sept. 2004.

[10] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke. DRPM: dynamic speed control for power management in server class disks. In *Proceedings of the 30th annual international symposium on Computer architecture (ISCA)*, pages 169–181, New York, NY, June 2003.

[11] H. Härtig, M. Hohmuth, J. Liedtke, and S. Schönberg. The performance of μ-kernel based systems. In *Proceedings of the 16th Symposium on Operating System Principles*, pages 66–77, Saint Malo, France, Oct. 1997.

[12] T. Heath, A. P. Centeno, P. George, L. Ramos, Y. Jaluria, and R. Bianchini. Mercury and Freon: temperature emulation and management in server systems. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 106–116, San Jose, CA, Oct. 2006.

[13] R. Joseph and M. Martonosi. Run-time power estimation in high performance microprocessors. In *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, pages 135–140, Huntington Beach, CA, Aug. 2001.

[14] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. T. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal of Selected Areas in Communications*, 14(7):1280–1297, Sept. 1996.

[15] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 17–30, San Fransisco, CA, Dec. 2004.

[16] Maxtor Corporation. *DiamondMax Plus 9 Data Sheet*, 2003.

[17] A. Merkel and F. Bellosa. Balancing power consumption in multiprocessor systems. In *Proceedings of the 1st EuroSys conference*, pages 403–414, Leuven, Belgium, Apr. 2006.

[18] R. Neugebauer and D. McAuley. Energy is just another resource: Energy accounting and energy pricing in the nemesis OS. In *Proceedings of 8th Workshop on Hot Topics in Operating Systems*, pages 67–74, Schloß Elmau, Oberbayern, Germany, May 2001.

[19] I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima, and A. Malick. Xen 3.0 and the art of virtualization. In *Proceedings of the 2005 Ottawa Linux Symposium*, pages 65–85, Ottawa, Canada, July 2005.

[20] J. Stoess and V. Uhlig. Flexible, low-overhead event logging to support resource scheduling. In *Proceedings of the Twelfth International Conference on Parallel and Distributed Systems*, volume 2, pages 115–120, Minneapolis, MN, July 2006.

[21] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards scalable multiprocessor virtual machines. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium*, pages 43–56, San Jose, CA, May 2004.

[22] VMware Inc. *ESX Server Data Sheet*, 2006.

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

[24] A. Weissel and F. Bellosa. Dynamic thermal management for distributed systems. In *Proceedings of the 1st Workshop on Temperature-Aware Computer Systems*, Munich, Germany, May 2004.

[25] A. Weissel, B. Beutel, and F. Bellosa. Cooperative IO - a novel IO semantics for energy-aware applications. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 117–130, Boston, MA, Dec. 2002.

[26] J. Zedlewski, S. Sobti, N. Garg, F. Zheng, A. Krishnamurthy, and R. Wang. Modeling hard-disk power consumption. In *Proceedings of the Second Conference on File and Storage Technologies*, pages 217–230, San Francisco, CA, Mar. 2003.

[27] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. ECOSystem: managing energy as a first class operating system resource. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 123–132, San Jose, CA, Oct. 2002.

[28] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. Currentcy: unifying policies for resource management. In *Proceedings of the USENIX 2003 Annual Technical Conference*, pages 43–56, San Antonio, TX, June 2003.