Multi-Aspect Hardware Management in Enterprise Server Consolidation

James E. Hanson, Ian Whalley, Malgorzata Steinder and Jeffrey O. Kephart
IBM Thomas J. Watson Research Center, Hawthorne, NY 10532
Email: {jehanson, inw, steinder, kephart}@us.ibm.com

ABSTRACT
An autonomic manager for enterprise server hardware management, called AMP, is described. AMP is designed to handle multiple aspects of hardware management and to work in conjunction with other management components, in particular application managers, in a way that reduces energy waste, protects server health, and preserves a high degree of autonomy both for itself and for the managers with which it works. AMP interacts with other managers in two ways: (1) exchange of nominal control over individual servers; and (2) provision of a synthetic cost function giving AMP’s assessment of relative desirability of using different servers. The high-level architecture of AMP is discussed, with particular focus on the way it effects a natural decomposition of the combined hardware-and-application management problem, and on initial versions of the algorithms it uses to manage server power states and determine the cost function. AMP’s viability in practice is demonstrated via prototype implementation in which it operates on real servers in collaboration with a state-of-the-art application manager. The overall system behavior it can achieve is investigated via simulation.

1. INTRODUCTION
Energy consumption in enterprise data centers has become a major concern in recent years as skyrocketing demand for computation has been accompanied by impressive advances in the ability of enterprise servers to meet that demand. One estimate puts the growth rate of spending on energy at four times that of spending on new enterprise servers to meet that demand. One estimate puts the cost of running energy spending to reach $44.5 billion, 70% of overall new server spending, by the year 2010. [1]

One well-studied technique for increasing datacenter energy efficiency is dynamic server consolidation: migrating application workload onto a minimal number of servers and putting the unused servers into low-power states. [2, 3, 4, 5] The highly variable workloads often seen in enterprise datacenters, coupled with the relatively high levels of power used by idle servers, means that significant energy savings may be achieved: estimates vary greatly, but are generally in excess of 25%. Furthermore, the “nuts-and-bolts” of dynamic server consolidation are already widely available. Remote monitoring of power-related metrics and effectors for managing server power states are generally available in recent generations of enterprise servers; and for migrating application workload between servers, two general approaches are presently in wide use: (1) running actively-managed distributed applications such as clustered web applications, where workload may be dynamically routed to different servers; and (2) running applications in a virtualized environment in which virtual machines may be migrated between hypervisors.

The primary challenge in dynamic server consolidation therefore lies in automating the decision-making aspects, which in real-world enterprise datacenters can be extremely complex. Consider a datacenter in which a large number of applications—all with potentially different operational requirements, policy-based constraints and behavioral characteristics—are distributed across a large number of servers—all with potentially different resource capacities, energy efficiencies, expected lifetimes, and hardware properties. The concerns relating to application management that must be addressed by any automated scheme for server consolidation may be summarized as: how to allocate server resources to applications such that the applications’ requirements are met, their various constraints are not violated, and the distribution of computational resources is optimal. Of similar complexity are concerns relating to we here call hardware management: how to improve energy efficiency, how to ensure safe operating temperatures, how to avoid undue risk of premature server failure, etc. And as a final, overarching challenge, the appropriate tradeoffs must be found among all these concerns.

In previous work [6] we addressed the first of these challenges and part of the third: namely, enterprise-quality application management and the tradeoffs between application performance and energy efficiency. In the present article, we turn to hardware management per se and to a fuller consideration of the remaining tradeoffs. We present a management architecture that, we believe, can cope with the complexities of server consolidation in real-world enterprise datacenters. Notably, the architecture prescribes a particular decomposition of the overall problem by dividing the solution among two separate decision-making components with complementary roles: one for application management and the other for hardware management. The hardware manager is a new component called AMP (Autonomic Manager for Power), and is the primary focus of this paper. The application manager we used is a slightly enhanced version of an existing component called APC (section 3).

In much of the work on server consolidation, hardware management has been limited to power management [3, 7, 5]. Other considerations, such as server wear-and-tear, have been addressed to some extent [2, 4], but to our knowledge, no practical, general approach for handling the real-world complexities of the problem has
been proposed heretofore (section 6).

The remainder of the paper is organized as follows. Section 2 presents AMP and describes the principal features of its design. Section 3 describes the prototype implementation, and Section 4 discusses a real experiment that shows the prototype in action. Section 5 turns to detailed simulations in order to investigate long-term behavior and parameter dependencies. Subsequent sections discuss related work, offer a brief assessment of AMP, and conclude with an indication of planned improvements.

2. ARCHITECTURE

AMP exists as a relatively autonomous component in a self-managing data center. Its concern is with the hardware, including both power consumption, wear-and-tear and, in future, additional factors. It is designed to form one part of a multi-agent solution to the problem of datacenter self-management, working in cooperation with other managers, such as application managers concerned with application performance.

In this paper we demonstrate how AMP works with the state-of-the-art application manager called APC (Application Placement Controller) that is embedded in the IBM WebSphere Virtual Enterprise product [8]. Previous work (see [6] and references therein) has described the high-level architecture of the APC and discussed its operation in conjunction with a precursor of AMP. Here, we focus on the architecture of the current version of AMP and the internal structure of its implementation.

AMP has no awareness of application management. It interacts with other managers in essentially two ways: (1) it exchanges control over machines using a “request/release” protocol discussed in Section 2.2; and (2) it provides measures of the relative desirability of using machines, to help its collaborators select which machines to request and/or to release (Section 2.3).

2.1 High Level Design

Figure 1 summarizes the high-level design of the current version of AMP. Solid arrows represent interactions which conform to an API specified by the architecture. Dashed arrows represent interactions whose form is not specified by the architecture, and hence is free to vary from implementation to implementation.

Inside AMP are subcomponents for calculating cost functions, for determining power state operations to be performed on the machines under its control, for caching sensor data, and for triggering and executing a control cycle. The two first-named subcomponents are discussed in sections 2.3 and 2.4 below. Control cycles are triggered periodically, either as a result of interactions with other managers or upon the discovery of a change in the power state of one of the machines. Each control cycle comprises the following steps: (1) gathering sensor data from the sensor-effector and updating the data cache; (2) obtaining a list of power state changes from the power state calculator; and (3) executing those changes by invoking operations on the sensor-effector.

Sensor data is gathered by the sensor-effector for each machine, and includes the following:

- current power state (on vs. off);
- current power usage;
- time since last power-on (if currently powered on);
- total lifetime power-on count. If this number is not available from the server itself, a running estimate is kept by the sensor-effector;
- power-efficiency curve (see below);

On the effector side, AMP at present can perform just two operations:

- power a given machine on or off

Omitted from the above list are features planned for future versions of AMP but not currently implemented. These include sensor data for the set of supported “sleep” states, for the current sleep state if the machine is sleeping, for power-cap data, and for DVFS (dynamic voltage/frequency scaling) values; and operations for putting the machine into a given sleep state, for setting power caps, and for performing DVFS operations.

The power-efficiency curve of a server, denoted $P_{on}$, gives the server’s power use as a function of CPU utilization level $x$. Power-efficiency depends on the nature of the workload being executed and on the server itself. Differences due to workload are on the order of 10%. [9, 10] Differences due to server hardware, even for nominally identical machines, are on the order of a few per cent. Therefore it is necessary for the power-efficiency curve to be determined separately for each server, and desirable for it to be measured dynamically and to be remeasured when the nature of the workload changes. If dynamical measurement is not possible, a default curve, empirically measured for each server under a benchmark workload, should be provided instead.

In the absence of DVFS, the power-efficiency curve for a given server under a given type of workload is close to linear [10] except for a discontinuity above zero due to idle power usage. It may therefore be modeled by the following functional form:

$$P_{on}(x) = \begin{cases} P_{off} & \text{if } \Theta_{on} = 0 \\ P_{idle} + \eta x & \text{if } \Theta_{on} = 1 \end{cases}$$

where $\Theta_{on}$ is 1 if the machine is on and 0 if it is off, $P_{off}$ is the power consumed by the machine when it is powered off (typically a few watts), $P_{idle}$ is the power consumed by the machine when it is powered on but not under load (frequently 50% or more of its peak power usage), and $\eta$ is the efficiency of the server.

2.2 Exchange of control

Central to the AMP’s design is the dynamic classification of machines into those that are in use (‘assigned’) and those that are idle (‘unassigned’). The reason for this is that by far the greatest savings of power are achieved by powering off machines; which in turn can only be done when the machines are not performing useful work.

AMP can be seen as a clearing-house to which other management components (hereafter referred to as ‘AMP clients’ or just ‘clients’) can come to request machines for use, and to relinquish machines that they no longer need. AMP keeps track of the pool, and the current assignment of each machine. At any given time, each machine is either assigned to a specific AMP client, or unassigned. Machines that are unassigned are under AMP’s exclusive
control. At any time, one of the client programs may contact AMP and request that a given machine be assigned to it. AMP is not bound to honor all such requests (if, for example, the machine is already assigned or is unavailable for another reason) but does so whenever possible.

All machines that are assigned to a client are always powered on, as the machine is not useful to the client unless it is on. However, machines that are unassigned could be in a wide variety of states, including powered off, in deep sleep, powered on, and transient states between the above. Future incarnations of AMP may also perform power-management on machines that are assigned, e.g., by feedback-controlled DVFS. It would be desirable if application managers did not have to provide target usage levels to AMP, though perhaps that will be necessary. In any case, it would not be necessary to change the request/release behavior.

**Requesting control:** When an AMP client decides that it requires more machine resources, it has two ways to do this. It can either request a specific machine, or it can request any machine that matches a set of criteria that AMP understands. In the latter case, AMP has flexibility to choose the machine that it regards as best. The former case is appropriate when the client is not in a position to offer a choice, either because only one machine will meet its needs or because it cannot express its requirements in a form that AMP can understand. Once it has selected a machine to assign to the client, AMP is responsible for getting the machine ready to do useful work, e.g. by putting it in the powered-on state. The client is not aware of this process.

**Releasing control:** When a client no longer requires a given machine, it may release that machine to AMP. Note that AMP cannot take a machine from one of its clients—the client must voluntarily release control. Before releasing a machine to AMP, the client is expected to stop using the machine for useful work, e.g., by vacating its VMs or redirecting request traffic away from it. When the client had released the machine, AMP is immediately free to power the machine down, put it into a sleep state, keep it as a hot spare, etc., at its sole discretion.

### 2.3 Cost functions

Upon request, AMP supplies information about the relative desirability of using the different machines known to it. This information takes the form of a “cost” function for each machine, representing a cost value (in arbitrary units) as a function of resource usage. The cost function conveys AMP’s own assessment about driving the machine at that usage level. In the present implementation, it comprises the power-efficiency curve $Power(x)$ and additional penalty terms for machines that need to be powered on or that have an atypically high power-on count. The first of these penalty terms actually plays two roles: it provides a disincentive for unnecessary power-on operations, and improves system responsiveness by guiding the client toward machines that are available without any time-delay.

The reason for conveying desirability using a cost function and letting the requestor make the final decision—rather than, say, always having AMP select the machine to be assigned—is that AMP is ignorant of the uses to which machines are put once they are assigned. As has been discussed elsewhere [6], enterprise-quality application management requires deep knowledge of the applications being managed, of administrator-defined policies, etc., all of which is beyond the purview of a hardware manager. AMP can provide only information to be factored into the application manager’s decision. AMP handles extreme cases, in which a machine must not be used, by making the machine unavailable for assignment.

For the present version of AMP, each server’s cost function has the following form:

$$Cost(x) = Power(x) + p_1 (1 - \Theta_{on}) + p_2 (c - c_{min})$$

where $p_1$ and $p_2$ are adjustable parameters, $Power(x)$ is the power-efficiency curve for the machine in question, $c$ is the power-on count of the machine, and $c_{min}$ is the minimal value of $c$ found in the population of machines AMP is managing. The first of the three terms captures the cost of electrical power, and guides the application manager toward requesting machines that are more efficient. The second term captures the cost of powering on a machine, and favors machines that are already on. The third term favors machines that are “younger” as measured by power-on count. Because this is a synthetic function (i.e., not required to convey real monetary cost), the values \{p_1, p_2\} may be chosen freely in order to guide the system toward the desired overall behavior. A plausible initial approach was to choose values based on the power-consumption of the machines when they were idle, $P_{idle}$, which for the experimental hardware used below was approximately 80 watts. Accordingly, after a little experimentation we selected values that led to appropriate behavior: $p_1 = P_{idle}$, and $p_2 = P_{idle}/4$, for all servers. It bears emphasis that these are not firmly established recommendations, but rather a starting point to show viability of our approach.¹

### 2.4 Adjustment of power states

AMP is solely in charge of whether and when to change the power states of the machines under its control. Its power state calculator needs to balance competing desiderata such as saving energy by powering machines off vs. reducing wear-and-tear by limiting the rate of power cycling.

As described above, the executor periodically (or upon receipt of a relevant event) triggers a calculation of how to adjust the power states of the machines managed by AMP but presently unassigned. For the present version of AMP we implemented three different power state adjusters:

**NeverPowerOff**: powers machines on as necessary; but immediately powers off every unassigned machine and every newly-released machine. This is expected to reduce power use as much as AMP is able to do so, but also to stress the machines by frequent power cycles.

**AlwaysPowerOn**: powers machines on as necessary, but immediately powers off every unassigned machine and every newly-released machine. This is expected to reduce power use as much as AMP is able to do so, but also to stress the machines by frequent power cycles.

**Basic**: attempts to maintain a given fraction of the available machines as hot spares (machines that are unassigned but ready for use) and, at the same time, imposes a minimum threshold on the frequency of power-cycles. In order to avoid situations in which machines are needed but cannot be powered on due to the threshold, it is expressed as $min\ time\ to\ run$: the minimum time the machine has to run after power-on before it may be powered off again. In Section 5, we will investigate the way in which the system’s behavior changes for different values of hot spare fraction $f_{Hot}$ and $min\ time\ to\ run\ mttr$.

### 3. IMPLEMENTATION

AMP is implemented in Java as “plain-old Java objects”, which can optionally be wrapped as OSGi bundles. The simplicity of the

¹In the experiments below, the application manager used the cost functions effectively as a tie-breaker to differentiate between placements that appeared to be otherwise equally good. In such cases, the system behavior is relatively insensitive to changes in the cost function’s parameters.
runtime required for AMP has proven to be beneficial during the development process, and has resulted in more rapid development than would have been possible otherwise.

3.1 Programmatic interfaces to AMP

Here we describe specifics of the current interfaces through which clients interact with AMP. The semantics of these interfaces was described in Section 2; the intent of this section is to provide a more developer-centric view of interactions with AMP.

The primary AMP interface with which AMP clients interact is named AmpService. This interface exposes the following primary methods:

- `getMachineIDs()` Provides the caller with the IDs of the machines that AMP manages.
- `getMachineData(id)` Given a machine ID, return information about that machine in a MachineData object.
- `request(id)` The client is requesting that a specific machine be assigned to it.
- `requestAny(filter)` The client is requesting that any machine that matches the provided filter be assigned to it. The filter is an object that AMP can call to determine if the client is prepared to accept a given machine.
- `release(id)` The client is releasing control of a machine. The machine will become unassigned.

In addition to AmpService, the other interface of interest is MachineData. This provides AMP clients with data about about a single machine—primarily two pieces of information: firstly, whether it is currently available for assignment; and secondly, the Cost(x) curve.

The AmpService interface also provides methods by which an AMP client can subscribe to changes in the set of machines that AMP manages, and changes to the MachineData for those machines. This is in order to permit AMP clients to keep up-to-date with AMP’s changing view of the world. For example, when it is possible that AMP is able to dynamically update the Cost(x) curves to take into account recent observations, AMP clients will be able to obtain updates to those curves.

3.2 Internals of AMP’s sensors and effectors

The sensor-effectuer used in the prototype connects to our testbed blade servers’ management module using the same protocol used by the IBM Director product, gathering sensor data and making changes using a combination of proprietary blade-specific messages and IPMI [11] commands. It was configured to gather power state and power usage data for every blade in the chassis every 60 seconds. The implementation used within AMP was a standalone Java library developed purely for the purposes of testing blade hardware. No attempt was made to harden it for enterprise use or to optimize its performance.

The power curves for one of blades was measured prior to the experiment using a standalone application developed specifically for that purpose. A tunable CPU-intensive workload was started within a virtual machine running on the server, and power usage and server CPU usage were measured for a range of CPU usage levels. These were then fit to a linear curve, which was then given to AMP’s sensor-effectuer as the default power curve to use for all blades. This was deemed sufficient to demonstrate the combined AMP-APC system, though it will obviously not be adequate for production environments.

3.3 Modifications to APC

As mentioned in the introduction, AMP was designed to work in concert with application managers such as APC. But because the latter predated AMP by several years, some modifications had to be made to it in order to permit it to work with AMP. Some of these modifications were discussed in [6], while others are new in the present design.

For the work in [6], APC had to be fitted with inputs whereby it received cost information about physical machines from outside entities, and with additional logic to incorporate those costs into its placement decisions. It also needed additional bookkeeping to track physical machines that were known to it but that it could not immediately use.

For the present design, it had to be fitted with further bookkeeping and logic to manage the request/release protocol described above. Additionally, for the experiments below the power-performance tradeoff described in [6] was temporarily disabled in favor of a straightforward consolidation scheme whereby VMs are migrated as needed until the total CPU utilization level of each physical machine drops below a given threshold. This scheme permits consolidation where possible without degrading performance. The reason for the change was that, for the purposes of studying AMP’s behavior, it was useful to simplify the way in which APC processed the information it received from AMP. It also illustrates how the separation of concerns between AMP and APC permits either one to be significantly modified without affecting the other.

4. EXPERIMENT

In order to validate the architecture, we implemented AMP, deployed it with APC on a small cluster, and conducted a simple experiment designed to illustrate the exchange of control between AMP and APC and the resultant CPU utilization and power consumption.

The system under test consists of four IBM LS20 blades (“physical machines”) venturep005, venturep006, venturep007, and venturep008, housed in a single IBM BladeCenter chassis, each with 4GB RAM and a single AMD processor with two cores, running VMware ESX 3.5. These blades are connected to a shared underlying fibrechannel storage system, and the VMware installations are managed by a VMware Virtual Center server. This configuration permits live migration of virtual machines.

In this system there are eight virtual machines (VMs), each of which can execute on any of the physical machines. Each VM runs Linux and an Apache Tomcat application server with a custom CPU-consuming web application installed. An external load driver directs HTTP traffic to the eight VMs. By varying the number of simulated clients that the driver sends to each virtual machine, we can vary the CPU demand exerted by those virtual machines.

For simplicity, in this case APC and AMP run in a single process on a single remote management machine. They communicate only through the mechanisms discussed in Section 2.

APC is configured to treat the eight virtual machines as ‘black boxes’. It does not know that the CPU demand exerted by the virtual machines is coming from HTTP workload, and it has no control over that HTTP workload. APC is given only information read from the hypervisors—including the current state of each virtual machine; upon which physical machine each virtual machine is running; and the amount of CPU and memory that each is using; the capacities of the physical machines; and the current CPU and memory usage of each physical machine.

AMP and APC have very different views of the system. While APC is concerned mainly with the scheduling and manipulating entities that run on the physical machines, AMP is concerned with sensing and manipulating properties of the physical machines them-
selves, and of the BladeCenter chassis. AMP does not know or care what the physical machines are used for, but it does know in which BladeCenter they reside, and how to turn them on and off.

In the experiment, which is designed to show basic interactions between APC and AMP, we use two of the currently implemented adjusters (see Section 2.4), NeverPowerOff and AlwaysPowerOff. Consequently, in this experiment, all machines that are powered off are unassigned. Physical machines that are powered on can be either be assigned or unassigned. AMP here has a single client: APC.

5. SIMULATION

To investigate the behavior of the APC-AMP collaboration on larger systems and longer timescales than were practical in the test environment used for the experimental prototype, it was necessary to perform a series of numerical experiments performed on a simulated data center. Though the system is still much smaller than real data centers, it was sufficient to give an indication of system behavior and permitted us perform hundreds of experiments, each of one of which would have taken weeks to perform in the lab, in a few tens of hours.

In the experiments below, the simulated system consisted of 20 physical machines, each of which had a CPU capacity of 4000 MHz and was running a hypervisor. Twenty virtual machines were distributed among the hypervisors. Each hypervisor was capable of hosting any virtual machine. Each simulation lasted a total of 15 days of simulator time. To account for transients, data from the first day was discarded.

The CPU demand timeseries for the VMs were selected round-robin fashion from a set of 10 predefined timeseries derived from measured CPU usage values taken in November 2008 on individual virtual machines running workload for a large (anonymous) customer in an enterprise data center. The raw data, consisting of CPU usage levels in per cent taken every 15 minutes over a period of a week, was normalized so that the highest value in each timeseries equalled the CPU capacity of one physical machine. Figure 3 shows the resulting timeseries.

5.1 Timeseries

One set of runs, described in this section, gathered timeseries data for each of the three power state calculation methods noted in Section 2.4: NeverPowerOff, AlwaysPowerOff, and Basic configured with fraction of hot-spare \( f_{Hot} = 0.1 \) and min time to run \( \mu_{min} = 15 \) min. Three observables were measured every hour: (a) total energy consumed by all machines; (b) maximum value of power-on count among the machines; (c) total time-delay in granting requests. The timeseries of values for these observables are shown in Fig. 4.

The power usage of the Basic case lay between that of the NeverPowerOff and AlwaysPowerOff runs, as is shown by the slopes of the three cases the energy-use plot, in which Basic lies between
During periods of increasing demand, as APC requested additional machines, it showed a marked tendency to request machines that had it had not been assigned before, or that been assigned to it fewer times than the others. This is attributable in part to APC’s own internal preference for avoiding unnecessary changes, and in part to the additional cost of machines with higher power-on counts (recall that for AlwaysPowerOff, to assign a machine is to power it on). Similarly, during periods of decreasing demand, APC tended to prefer releasing machines with higher power-on counts. This led to an emergent “wave effect” in which the set of assigned servers slowly worked its way around the total population, picking up new members as it went, holding them for a while, and then releasing them until the next time the wave came around. This in turn led to an overall slow increase in the max power-on count of any one server even though no actual limit on the power-on count was imposed. The wave was present just as strongly in the Basic case, with the differences that servers on the leading edge of the wave were often powered on some time before they were requested, and servers on the trailing edge were sometimes left powered on for while after they had been released.

Figure 4(c) shows the amount of simulated time that passed between when APC requested a machine and when AMP granted that request, summed over all requests (all requests were granted, because APC was the only requestor). The time delay for any single request is exactly the time required to make the requested machine ready for use—i.e., to power it on if it needed to be powered on. NeverPowerOff is of course always 0, while AlwaysPowerOff climbs steadily. The Basic case has a much smaller growth rate, even though the fraction of hot spares was small and the minimum time to run was a small fraction of the demands’ natural period. This is again attributable to the wave effect.

5.2 Parameter dependence

Next we investigate of the way in which the system’s behavior depends on the tunable parameters of the Basic case.

For each value of $f_{Hot}$ and $mtr$ in sequence that spanned the parameter space, an independent simulation run was done as above. At the end of each run, for the 14 days during which data was taken, the following quantities were measured: (a) time-averaged total power use, (b) the average daily increase in the maximum power-on count, and (c) average daily increase in total time-delay in granting requests. The results are shown in Fig. 5.

As expected, the power use for $f_{Hot} = 0$ and $mtr = 0$ was equal to that of the AlwaysPowerOff case, and grew with increasing $f_{Hot} = 0$ and $mtr = 0$ to an asymptotic value equal to that of the NeverPowerOff case. What may be more surprising was the relative insensitivity of the power use to the value of $mtr$ up to $mtr = 12$. The rate of power-on operations was never more than 1.5 starts per day for any machine, presumably due to the wave effect discussed above, and was less than or equal to 1.0 for $mtr = 9$, and also for or $f_{Hot} >= 0.25$. Time delays decreased quickly with increasing $f_{Hot}$, dropping effectively to zero for $f_{Hot} >= 0.2$ regardless of $mtr$.

Together, these results indicate that AMP can achieve a reasonable compromise among the multiple competing concerns in play: providing machines to requestors quickly, reducing power waste, and protecting server lifespan. For example, for $f_{Hot}$ somewhere in the range of 0.1 to 0.2 and $mtr$ somewhere between 6 and 9 wastes only a little power, accrues little time-delay, and keeps the power-cycle rate at or below the nominally acceptable rate of once per day.\footnote{The maximum rate regarded as “acceptable” is of course a matter of opinion. Many system administrators regard every single power}

The maximum power-on count for the NeverPowerOff case was of course zero. The difference between the Basic and the AlwaysPowerOff cases turned out to be relatively small. Interestingly, a detailed investigation of APC’s behavior in the AlwaysPowerOff case revealed an emergent phenomenon that accounts for this.

The cyclic variations in the slopes of the Basic and AlwaysPowerOff curves follow the daily peaks and troughs of the demand. The NeverPowerOff case shows a cyclic variation as well, possibly too small to be seen at the figure’s resolution, due to the additional energy used by a machine under load; its smallness is due to the dominance of $P_{idle}$.

The maximum power-on count for the NeverPowerOff case was of course zero. The difference between the Basic and the AlwaysPowerOff cases turned out to be relatively small. Interestingly, a detailed investigation of APC’s behavior in the AlwaysPowerOff case revealed an emergent phenomenon that accounts for this.
6. RELATED WORK

Among the dozens of papers that treat dynamic server consolidation, we have space to discuss only those that most closely relate to the two main issues considered in this paper: namely, hardware management and cooperative multi-manager systems.

Hardware management: Most of the literature studies a strictly limited set of hardware management concerns in conjunction with highly simplified application environment. One thread of research considers (of hardware) only server energy efficiency: e.g., refs [5, 12]. Another treats peak power use at the server and/or cluster level, sometimes in conjunction with energy efficiency [9]. A third considers datacenter cooling [15, 16], and a fourth focuses on exploiting server heterogeneity to save power. [17]

Chase et al. [2] use market-based control of power and performance in which wear-and-tear is mentioned but is not incorporated into the market model. The suggestion is to externally “damp” the frequency of changes, e.g., to once per day. Chen et al. [4] formulate an optimization problem in which the total cost of server use includes terms for both energy use and wear-and-tear, the latter in the form of an averaged cost per server power cycle. However, the problem formulation assumes that each reboot of each server is equally costly—i.e., it admits of no differentiation between servers by age or reliability.

Cooperative managers: Ref. [7] presents a two-manager scheme similar to ours, but which works only for web applications, and in which the power manager receives application response times from the application manager, thereby achieving only limited separation of application and hardware concerns. In ref. [18], a hardware manager and an application manager interact via a centralized arbiter that incorporates knowledge of both sides of the problem. They also include a third manager that explicitly considers migration cost, something that, at present, APC does only implicitly. More recently, Kumar et al. [19] present an architecture in which a VM placement manager and a power manager collaborate on dynamic server consolidation and server frequency scaling. It differs from our approach in several respects: It is specific to virtualized environments; the separation of concerns is incomplete because both managers have access to information specific to the other’s management domain and possess the expertise to use it effectively; and a third component is required to prevent the two managers from interfering with each other.

Ref. [20] develops a scheme (unfortunately not implemented) for coordinated feedback control among multiple managers that operate on different scales and address different aspects of hardware management. Much of this work is complementary to ours, and indicates a path whereby AMP can evolve beyond just doing power on/off operations. However, in their system, a single manager is responsible both for managing application placements and for managing server power states.

7. ASSESSMENT

The particular problem decomposition embodied by AMP and APC was originally motivated by the purely pragmatic concern of implementability: it was simply not practical to write a single controller that could handle all the aspects of both application and hardware management we are dealing with. Nevertheless, having implemented it and (through simulation) indicated that it can give good system behavior, we note a number of other attractive features it possesses.

Our division of the problem allows each manager to be enhanced separately to incorporate more, and more complex, real-life factors in the problem formulation and solution. The application manager performs its work without knowledge of hardware concerns; the hardware manager acts without knowledge of the applications. The separation of concerns allows us to address hardware-management issues, such as power-cycle frequency of a machine, independently of the application manager. Similarly, application management concerns, such as optimizing responsiveness while preserving constraints for or against collocation of applications on machines, may be addressed independently of the hardware manager.

It also permits reuse of either component in situations in which the other must be replaced. We can plug an alternative power manager into the system without modifying the optimization problem that AMP is solving. Similarly, we can reuse the same AMP component with different application managers, be they alternative implementations or specialists in, e.g., clustered web applications or black-box VMs.

As with in any such division, there is a risk that encapsulating a part of the overall system functionality in an autonomous agent will introduce suboptimalities into the system’s behavior. In the present case, the most significant such suboptimality was that machines were sometimes kept in a powered-on but idle state. In principle, a single component that combined the functions of AMP and APC could have made use of all powered-on machines, and could therefore have achieved a better tradeoff among application perfor-
mance, power savings, and reduced wear-and-tear. This suboptimality must however be traded off against the greater complexity and rigidity of the monolithic solution.

Within AMP itself, the separation of the “autonomics” from the sensor-effector permits the latter to encapsulate the means whereby sensor data is obtained and commands are put into effect, permitting it to be used with different hardware configurations (such as rack servers) by means of different sensor-effector implementations. Further, because the cost-function calculator and the power state adjustment calculator are separate subcomponents with their own APIs, AMP’s “autonomics” may be upgraded simply by plugging an improved implementation of either subcomponent. Thus, for example, three separate power state calculators were implemented for the three methods described in Section 2.4, and multiple versions of the cost-function calculator were made during the course of our experiments.

8. CONCLUSION

In this paper we have presented AMP, an autonomic manager in charge of the power states of physical server machines in an enterprise data center. AMP is designed to work in conjunction with other autonomic managers, and is responsible for cutting excess power use and preserving the lifespan of the servers it manages.

The present versions of the AMP subcomponents are clearly only starting points, and present a number of opportunities for future enhancements. The form of the cost function, and the values chosen for its parameters, deserve further attention; as does the addition of terms to deal with power budgeting and thermal management.

The power state adjustment calculator deserves similar attention. It should incorporate profiles of expected usage levels for different times of day and days of the week. This would enable it to start up additional hot spares shortly before an expected surge in demand and thereby incur less delay in assigning machines without greatly increasing power use. A means of dynamically building up this profile by observing actual usage patterns over time would be of significant value. In addition, the power state calculator’s mechanism for limiting reboot frequency, in which machines are made to run for some fixed minimum length of time, should be replaced with something that permits machines to be powered down sooner. Finally, it should be augmented to ensure that no machine is left powered down for an unduly long time, because that too can contribute to hardware failure.

9. ACKNOWLEDGEMENTS

The authors wish to express sincere thanks to Scott Piper and Anthony Low for letting us use their blade-management utility library in the AMP prototype’s sensor-effector, and to Nick Gruendler and Jerry Ackaret for sharing their expertise on server wear-and-tear.

10. REFERENCES