Automatic Run-Time Selection of Power Policies for Operating Systems

Nathaniel Pettis *, Jason Ridenour, and Yung-Hsiang Lu †
Electrical and Computer Engineering
Purdue University, West Lafayette, Indiana, USA
{npettis,ridenoja,yunglu}@purdue.edu

Abstract

A significant volume of research has concentrated on operating-system directed power management (OSPM). The primary focus of previous research has been the development of OSPM policies. Under different conditions, one policy may outperform another and vice versa. In this paper, we explain how to select the best policies at run-time without user or administrator intervention. We present a hardware-neutral architecture portable across different platforms running Linux. Our experiments reveal that changing policies at run-time can adapt to workloads more quickly than using any of the policies individually.

1. Introduction

Operating systems (OSs) manage resources, including processor time, memory space, and disk accesses. Recently, energy has become a crucial resource for OSs to manage as well [4, 7, 18]. This is predominantly due to the growing popularity of portable systems that require long battery life. Power management is also important in high-performance servers because performance improvements are limited by the excessive heat in the system [11]. Finding better policies has been the main focus of OSPM research in recent years [3]. A policy is an algorithm that chooses when to change a component's power states and which power states to use. Existing studies on power management make an implicit assumption: only one policy can be used to save power. Hence, those studies focus on finding the best policies for unique request patterns. Some policies allow their parameters to be adjusted at run-time [5, 6], but the algorithms remain the same. As demonstrated in previous studies, significantly different policies may be needed to achieve better power savings in different scenarios. In this paper, we present a new concept for power management: automatic policy selection. Instead of choosing a policy in advance, a group of policies can be eligible at run-time, and one is selected in response to the changing request patterns. This is especially beneficial for a general-purpose system, such as a laptop computer, where usage patterns can vary dramatically when the user executes different programs.

Three challenges arise for automatic policy selection. First, a group of policies must be eligible to be selected. We propose an architecture as the framework upon which new policies can be easily added. Second, eligible policies must be compared to determine which policy can save more power for the current request pattern. Third, the best eligible policy must be selected to manage a hardware component and the previous policy must stop managing the same component. Usually, power management is conducted by OSs, and changing a policy requires rebooting the system [13]. One unique advantage of our approach is that new policies can be added and selected without rebooting the system.

This paper has the following contributions. (a) We present the concept of automatic policy selection. Using our architecture, policies can be added, compared, and selected while a system is running. (b) Our architecture allows easier implementation and comparison of policies. Moreover, all policies are compared simultaneously so repeatable workloads are unnecessary. (c) We present a reference implementation in Linux and experimentally demonstrate that automatic policy selection can adapt to workload variations more quickly than individual policies.

* Nathaniel Pettis is sponsored by the U.S. Department of Education GAANN Fellowship.
† This work is supported in part by NSF CAREER CNS-0347466 and by the Purdue Research Foundation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

3-9810801-0-6/DATE06 © 2006 EDAA

2. Related Work

Most users are familiar with power management for block access devices, such as hard disks. Users can set the timeout values in Windows' Control Panel or using Linux's **hdparm** command. This is the most widely-used "timeout policy." Karlin et al. [12] propose a 2-competitive timeout algorithm, where the timeout value is the break-even time of the hardware device. The *break-even time* is defined as the amount of time a device must be shut down to save energy. Douglis et al. [6] suggest an adaptive timeout scheme to reduce the performance penalties for state transitions while providing energy savings. Hwang et al. [10] use exponential averages to predict idleness and shut down the device when the predicted idleness exceeds the break-even time. Several studies focus on stochastic optimization using Markov models [2,16] and generalized stochastic Petri Nets [15]. While many of these methods adapt to system behavior, some policies may adapt more quickly than others for different workloads. Our method extends beyond adaptive policies by providing another level of adaptiveness, namely, policy selection.

Several projects [4, 18] have focused on creating power-aware OSs. Both of these systems are essentially schedulers, where the energy consumption of each process is limited by reducing the allocated time quanta. Furthermore, each system uses a single policy, whereas our method selects the best policy at run-time. Microsoft Windows' OnNow API [14] provides application developers the ability to designate I/O activities as low priority to allow queuing until the device is awakened. Individual devices' power states are controlled by the device driver, which presumably implements a single policy such as those mentioned previously. OnNow provides a mechanism to set the timeout values and the device state after timeout, but policies cannot be changed without administrator intervention. Our method removes the administrator from the policy selection process and allows complex policies.

This paper does not present any new policies. Instead, we explain how to select the proper policy automatically at run-time. We demonstrate that policy selection achieves superior results to a single policy with adaptive parameters because policy selection adapts to workloads at least as quickly as adaptive policies (and often much more quickly). Our method allows a single policy to be dynamically selected from a set of policies for each device without rebooting the system, allowing experiments without disrupting system availability. We believe that our method is the first proposing run-time policy selection and a reference implementation in Linux.

3. HAPPI Overview

We begin by describing the OS support necessary to facilitate power policy selection. This design specifies homogeneous requirements for all policies so they can be easily integrated into the OS and selected at run-time. Homogeneous requirements are necessary to allow significantly different policies to be compared by the OS. We refer to this architecture as the Homogeneous Architecture for Power Policy Integration (HAPPI). HAPPI is currently capable of supporting power policies for disk, DVD-ROM, and network devices but can easily be extended to support other I/O devices. To implement a policy in HAPPI, the policy designer must provide:

1. A function that predicts idleness and controls a device's power state.
2. A function that accepts a trace of device accesses, determines the actions the control function would take, and returns the energy consumption and access delay from the actions.

3.1. Policy Set

Each device has a set of policies that are capable of managing the device. A policy is said to be *eligible* to manage a device if it is in the device's policy set. A policy becomes eligible when it is loaded into the OS and is no longer eligible when it is removed from the OS. The policy is considered *active* if it is selected to manage the power states of a specific device by HAPPI. Each device is assigned only one active policy at any time. However, a policy may be active on multiple devices at the same time by creating an instance of the policy for each device. When a policy is activated, it obtains exclusive control of the device's power state. The policy is responsible for determining when the device should be shut down and requesting state changes. An active policy may update its predictions and request device state changes on each device access or a periodic timer interrupt. The set always includes a "null policy" that keeps the device in the highest power state.

3.2. Measuring Device Accesses

Policies monitor device accesses to predict idleness and determine when to change power states. We refer to the data required by policies to make decisions as *measurements*. For the policies included in this paper, the measurements include traces of recent accesses for each device controlled by the policy. Whenever the device is accessed, HAPPI captures the size and time of the access. HAPPI also records the energy and delay for each device. Energy is accumulated after each access and after every second of idleness. We define delay

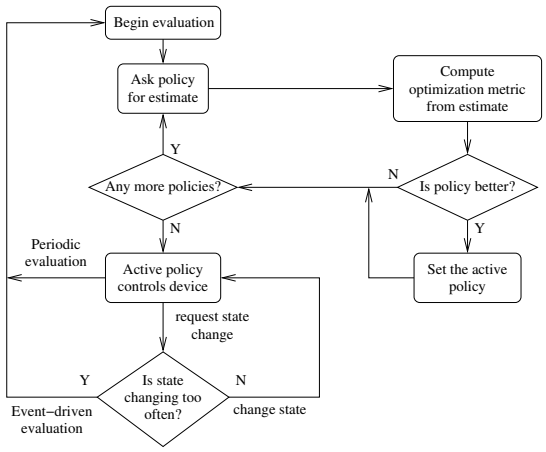


Figure 1. Policy selection under steady-state operation.

```
FIXED-TIMEOUT-ESTIMATOR(accesses)
     /* t_o = timeout length */
     /* P_on, P_off, P_active = power states */
 1   energy ← 0
 2   delay ← 0
 3   last ← accesses[I].time
 4   for each q ∈ accesses[2 : n]
 5       do now ← q.time
 6          t_idle ← now − last
 7          if t_idle ≥ t_o
 8            then /* Device will change power states */
 9                 energy ← energy + P_on t_o
10                 energy ← energy + P_off (t_idle − t_o)
11                 energy ← energy + E_wake
12                 delay ← delay + t_wake
13            else /* Device will not change power states */
14                 energy ← energy + P_on t_idle
15          /* Energy for the request */
16          energy ← energy + P_active q.size / bandwidth
17          last ← now
18   return (energy, delay)
```

Figure 2. Sample estimator function for a fixed-timeout policy.

as the amount of time that execution blocks waiting for a device to awaken. We only accumulate delay for the device's first access while sleeping or awakening because Linux prefetches adjacent blocks on each access.

3.3. Policy Selection

Policy selection is performed by the *evaluator* and is illustrated in Figure 1. When the evaluator is triggered, it asks all eligible policies to provide an estimate of potential behavior for the current measurements. An *estimate* consists of energy consumption and total delay for the measurement data and provides a quantitative description of a policy's ability to manage the device. To accomplish this, each policy must provide an estimation function that uses HAPPI's measurement data to analyze what decisions the policy would have made if it were active when the measurements were taken. The energy and delay for these decisions are computed by the estimation function and returned to the evaluator. An active policy for each device is selected by the evaluator after it receives estimates from all policies. The evaluator selects each active policy by choosing the best estimate for an optimization metric, such as total energy consumption or energy-delay product. If another policy's estimate is better than the currently active policy, the inferior policy is deactivated and returned to the set of eligible policies. The superior policy is activated and assumes control of the device's energy management. Otherwise, the current policy remains active. If the null policy produces the best estimate, none of the eligible power management policies can save power for the current workload. Under this condition, power management is disabled until the evaluator is triggered again.

To illustrate how an estimator works, we provide an example. Figure 2 shows an estimator function for a fixed-timeout policy. The estimator iterates over all device accesses, determining the time between each access. If the idleness t_{idle} exceeds the timeout length t_o (lines 8–12), the device shuts down and consumes $P_{on}t_o$ energy in the idle state. The remaining $(t_{idle} - t_o)$ time is spent in the off state. The device consumes E_{wake} energy to awaken from the off state before servicing the request. Since the device must change power states, a delay of t_{wake} is incurred before the access begins. However, if $t_{idle} < t_o$ (lines 13–14), the device does not shutdown and consumes energy $P_{on}t_{idle}$ before serving the next request. No delay occurs before the request begins. After all accesses have been considered, the total energy consumption and delay is returned to HAPPI.

The evaluator determines when re-evaluation should take place and performs the evaluation of eligible policies. In this paper, we use average power as our optimization metric. To minimize average power, the evaluator requests an estimate from each policy and selects the policy with the lowest energy estimate for the device access trace. Since average power is energy consumption over time and the traces record the same amount of time, the two metrics are equivalent.

Policy mispredictions may be classified into two types: missed opportunities to save energy and incorrect shutdowns for overestimated idleness. The first type oc-

curs with policies that make decisions on each access. Since no accesses have occurred, the policy cannot recognize the long period of idleness. To reduce the missed opportunities to save energy, we evaluate all policies once every 20 seconds to determine if a better policy exists. We select this interval because it exhibits quick response to workload changes without thrashing between policies. Shorter intervals detect changes in workload more quickly, but policies thrash when changing workloads, whereas a longer interval reduces thrashing at the cost of slower response to workload changes. The second type of misprediction occurs when workloads change from periods of long idleness to short idleness. To avoid several incorrect shutdowns, HAPPI notes the time between state transition events and re-evaluates the policies for the device if three transitions are recorded within two break-even times. This type of misprediction is infrequent but is easily detectable by this heuristic.

4. Implementation

To validate HAPPI's ability to select policies at run-time, we implement the architecture in the Linux 2.6.5 kernel. Policies and evaluators are implemented as kernel modules, but measurements must be compiled into the kernel. Function calls are inserted to the device drivers for the disk, DVD-ROM, and network card to record the size and time of each access for HAPPI. These measurements are taken continuously during system operation. We also add functions to maintain policy sets and issue state change requests. Since our experimental hardware is not fully ACPI compliant, we implement a function that returns the power, transition energy, and transition delay for each state of each device. These values are included within the the ACPI specification [1] but are not provided by our experimental hardware's BIOS. This is not a limitation of HAPPI. If the hardware is ACPI-compliant, the information is available to HAPPI automatically. If the hardware is not ACPI-compliant, the power parameters can be loaded into HAPPI by the administrator as constants in a kernel module during the installation process. Policies need these values to compute the power consumed in each state and determine when to change power states. The Linux kernel is optimized for performance and exploits disk idleness to perform maintenance operations such as dirty page writeback and swapping. To facilitate power management, we use the 2.6 kernel's **laptop_mode** option, which delays dirty page writeback until the disk services a demand access or the number of dirty pages becomes too large. Without **laptop_mode**, the disk is never idle long enough to save energy.

The evaluator and policies are implemented as loadable kernel modules. A computer may be used for many different applications with very different power management goals. The system administrator selects an evaluator that optimizes for the computer's specific application and inserts the module into the kernel using the **insmod** command. This is the only interaction required by an administrator. From this point onward, the evaluator selects power policies automatically. When a policy is inserted into the kernel using **insmod**, the policy calls a function to register its estimation function and control function with HAPPI. After insertion, the policy is eligible for selection. The evaluator is notified that a new policy is present and re-evaluates all policies. After the best policy is selected for each device, HAPPI enters the steady-state operation described in Section 3.3.

Since the policy is a kernel module, it may be added or removed from the OS at run-time, allowing policy designers to experiment without rebooting the system. This feature is particularly important because existing OSPM approaches require rebooting the system to add a new policy, significantly inconveniencing the user. By using HAPPI, a "plug-and-play policy" is possible. That is, a policy may be used on any hardware platform with Linux and the HAPPI architecture without further modification. Using kernel modules and ACPI allows our approach to be portable across different platforms. Policies may be shared between designers for comparison or included with Linux distributions to provide state-of-the-art OSPM to the general user. A policy may acquire a device's hardware power parameters at run-time through ACPI. Thus, a single HAPPI policy can be written to allow management of multiple devices, while existing solutions, such as [13], are device-specific by implementing a policy in a device driver. Our approach allows code to be reused, rather than implementing a policy for every device in the system. We also believe that our methodology applies to other OSs with loadable components.

5. Experiments

5.1. Experimental Setup

We manage three devices in our experiments: a Fujitsu laptop hard disk (HDD), a Samsung DVD drive (DVD), and a NetXtreme integrated wired network card (NIC). We determine the parameters for the devices from datasheets [8, 17] and experiments. Table 1 shows the information defined by the ACPI specification for each device. The active state is the state where the device can serve requests. The sleep state is a reduced power state in which requests cannot be served. Changing between states incurs energy and delay shown in Table 1.

		HDD	DVD	NIC
Active	Power	0.85 W	2.64 W	0.500 W
Sleep	Power	0.25 W	0.3 W	0.002 W
	Delay	4.5 s	2.68 s	1.996 s
	Energy	17.1 J	67 J	0.333 J

Table 1. Power states for devices.

#	HDD	DVD	NIC
1	Idle 45–75 s	Bursty	Idle 45–75 s
2	Bursty	Idle	Bursty
3	Busy	Idle	Busy
4	Periodic 60 s	Busy	Periodic 60 s
5	Busy	Idle	Idle

Table 2. Access patterns for workloads.

5.2. Sample Policies

We consider four power management policies: the null policy, 2-competitive timeout [12], exponential prediction [10], and adaptive timeout [9]. For our experiments, we assign an exponential weight, α, of 0.5, meaning that the current idleness contributes 50% of the prediction and the previous prediction contributes 50%. In adaptive timeout the policy has initial value of $0.5t_{be}$ and changes by $0.1t_{be}$ on each access. Our experiments will demonstrate that, at a particular time, one policy may be superior to the others. When the request pattern changes, another policy may become better. We also demonstrate that different policies may be active simultaneously to manage different devices.

5.3. Workload Characteristics

To illustrate HAPPI's ability to track changes in workloads and select policies, we execute applications that provide a wide range of activities for HAPPI to manage. The activity level of each device for each workload is indicated in Table 2. The workloads include:

Workload 1: Web browsing + buffered media playback from DVD.

Workload 2: Download video and buffered media playback from disk.

Workload 3: CVS checkout from remote repository.

Workload 4: E-mail synchronization + unbuffered media playback from DVD.

Workload 5: Kernel compile.

5.4. Experimental Results

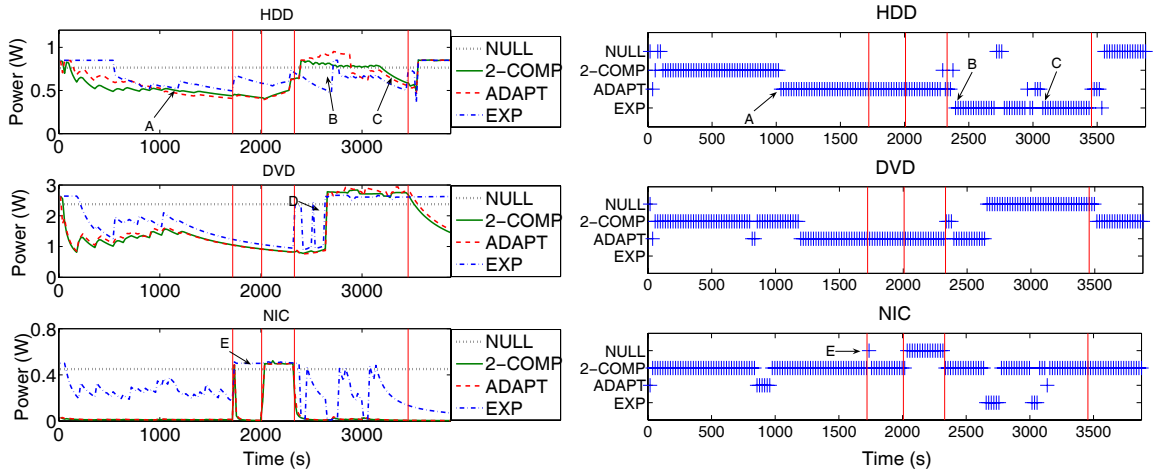
Figure 3(a) shows the estimated average power of the four policies (NULL, 2-COMP, ADAPT, and EXP) at each evaluation for the three devices during the benchmark. The vertical lines indicate the divisions between workloads. Figure 3(b) is a Gantt chart of the selected policies over time. For Workload 1 on the HDD, we observe from Figure 3(b) that 2-COMP is selected for the first half of the workload and ADAPT is selected for the second half of the workload. Figure 3(a) shows that ADAPT requires time to learn the proper timeout value and produces superior energy savings to 2-COMP at point A. Since HAPPI can choose policies at run-time, we are able to save more energy by using 2-COMP until adaptation is completed. We notice a similar occurrence for the HDD in Workload 4. At point B, EXP adapts to the periodic nature of the e-mail synchronization. Shortly after B, a dirty page writeback causes EXP to mispredict and remain active after the access. During this time, HAPPI switches to ADAPT until EXP adapts to the workload at C. Again, HAPPI is capable of dynamically selecting more effective policies more quickly than adaptive policies can adapt.

Estimates sometimes increase sharply, such as at points D and E. They correspond to accesses where the selected policy would mispredict (point D) or where the workload changes such that no idleness exists for the policy to save energy (point E). For example, point D marks the first time that the DVD is accessed after completing Workload 1. Hence, significant idleness exists. EXP wants to immediately shutdown the DVD after the access, but another access immediately occurs. ADAPT and 2-COMP must wait for a timeout and do not mispredict, so their estimates do not increase. Point E references the time where the video is downloaded from the Internet in Workload 2. No policy can save energy because insufficient idleness exists. Therefore, all the policies' estimates increase and NULL is selected.

5.5. Discussion

The results from Section 5.4 indicate HAPPI can adapt more quickly than individual policies. Adaptive policies are limited by the speed at which they adapt and maintain stability. If a policy is capable of adapting quickly to a workload and outperforms the other policies in the system, HAPPI selects the policy when it is evaluated. Hence, HAPPI adapts at least as quickly as any adaptive policy. However, we observe that HAPPI often adapts more quickly than individual policies.

Fundamental differences in access patterns between devices can be revealed by HAPPI's policy selection process. For example, at point B, three different policies are active on three different devices. This information can be used at run-time to select appropriate policies and provide insight for policy designers that may seek to exploit these properties in future policies. Even though this



(a) (b)

Figure 3. Policy selection for workloads. (a) Estimated energy consumption for each policy on devices for experimental workload. (b) Selected policies for devices at each evaluation.

paper compares only four policies, many new policies can be added. Our approach allows other researchers to perform experiments with their innovative policies on real machines easily. The actual energy savings depend on the set of policies. HAPPI provides an environment so that policies can be easily implemented and automatically selected, encouraging the development of sophisticated policies that can save more energy.

6. Conclusion

Despite a considerable amount of research and open standards, sophisticated dynamic power management has not been widely adopted. The difficulty to implement and compare policies with current technology is a significant barrier to the acceptance of OSPM. We propose an architecture called HAPPI to simplify the implementation of policies and select the proper policy at run-time. We validate HAPPI's requirements by implementing the architecture in Linux. To our knowledge, this is the only architecture that allows policies to be selected automatically at run-time. Our experiments indicate that policy selection is highly adaptive to workload and hardware types, supporting our claim that automatic policy selection is necessary to achieve better energy savings.

References

[1] Advanced Configuration and Power Interface. http://www.acpi.info.

[2] L. Benini, A. Bogliolo, G. A. Paleologo, and G. D. Micheli. Policy Optimization for Dynamic Power Management. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 18(6):813–833, June 1999.

[3] L. Benini and G. D. Micheli. System-Level Power Optimization: Techniques and Tools. *ACM ToDAES*, 5(2):115–192, April 2000.

[4] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle. Managing Energy and Server Resources in Hosting Centers. In *ACM Symposium on Operating Systems Principles*, p. 103–116, 2001.

[5] E.-Y. Chung, L. Benini, A. Bogliolo, Y.-H. Lu, and G. D. Micheli. Dynamic Power Management for Nonstationary Service Requests. *IEEE Transactions on Computers*, 51(11):1345–1361, November 2002.

[6] F. Douglis, P. Krishnan, and B. Bershad. Adaptive Disk Spin-down Policies for Mobile Computers. In *USENIX Symposium on Mobile and Location-Independent Computing*, p. 121–137, 1995.

[7] C. S. Ellis. The Case for Higher-level Power Management. In *Workshop on Hot Topics in Operating Systems*, p. 162–167, 1999.

[8] Fujitsu Computer Products, Inc. Fujitsu MGT-AH Series 5400 RPM Ultra ATA/100, 2.5-inch Mobile Drive data sheet, 2003.

[9] R. Golding, P. Bosch, and J. Wilkes. Idleness Is Not Sloth. In *USENIX Winter Conference*, p. 201–212, 1995.

[10] C.-H. Hwang and A. C.-H. Wu. A Predictive System Shutdown Method for Energy Saving of Event-driven Computation. *ACM ToDAES*, 5(2):226–241, April 2000.

[11] R. Joseph and M. Martonosi. Run-time Power Estimation in High Performance Microprocessors. In *ISLPED*, p. 135–140, 2001.

[12] A. Karlin, M. Manasse, L. McGeoch, and S. Owicki. Competitive Randomized Algorithms for Nonuniform Problems. *Algorithmica*, 11(6):542–571, June 1994.

[13] Y.-H. Lu and G. D. Micheli. Comparing System-Level Power Management Policies. *IEEE Design and Test of Computers*, 18(2):10–19, March 2001.

[14] Microsoft Corporation. OnNow Pow. Mgmt. Architecture for Applications, December 2001.

[15] Q. Qiu, Q. Wu, and M. Pedram. Dynamic Power Management of Complex Systems Using Generalized Stochastic Petri Nets. In *DAC*, p. 352–356, 2000.

[16] T. Simunic, L. Benini, P. Glynn, and G. D. Micheli. Dynamic Power Management for Portable Systems. In *International Conference on Mobile Computing and Networking*, p. 11–19, 2000.

[17] R. Winterton. DVD/CD Rendering: Optimizing for Power on Mobile Platforms, July 2004.

[18] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. ECOSystem: Managing Energy As A First Class Operating System Resource. In *International Conference on ASPLOS*, p. 123–132, 2002.