Purdue University
Purdue e-Pubs

ECE Technical Reports | Electrical and Computer Engineering

3-15-2007

Implementation Guides for a Homogeneous Architecture for Power Policy Integration in Operating Systems

Nathaniel Pettis
Purdue University, pettis.eddie@gmail.com

Yung-Hsiang Lu
Purdue University, yunglu@purdue.edu

Follow this and additional works at: http://docs.lib.purdue.edu/ecetr

Pettis, Nathaniel and Lu, Yung-Hsiang, "Implementation Guides for a Homogeneous Architecture for Power Policy Integration in Operating Systems" (2007). *ECE Technical Reports.* Paper 351.
http://docs.lib.purdue.edu/ecetr/351

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

Implementation Guides for a Homogeneous Architecture for Power Policy Integration in Operating Systems

Nathaniel Pettis and Yung-Hsiang Lu
School of Electrical and Computer Engineering
Purdue University, West Lafayette, Indiana, USA
{npettis,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. Hence, it is difficult, or even impossible, to design the "best" policy for all computers. We present a software framework called the Homogeneous Architecture for Power Policy Integration (HAPPI) that selects the best policy for a given workload at run-time without user or administrator intervention. This framework is portable across different platforms running Linux. HAPPI specifies common requirements for policies and provides an interface to simplify the implementation of policies in a commodity OS. HAPPI can select the best policy among a set of distinct policies at run-time. This technical report describes HAPPI's implementation and provides a sample policy.

Contents

1 Introduction **2**

2 HAPPI Overview **2**
2.1 Policy Set . . . 2
2.2 Measurements . . . 3
2.3 Policy Selection . . . 3

3 Implementation **3**
3.1 Environment Setup . . . 3
3.2 Configuring Hardware Specific Parameters . . . 4
3.3 Inserting and Removing Policies . . . 4
3.4 Recording Device Accesses and State Transitions . . . 5
3.5 Maintaining Access History . . . 6
3.6 Advanced Measurements . . . 8
3.7 Evaluating and Changing Policies . . . 8

4 Installation and Configuration **10**
4.1 Configuring IDE Devices . . . 10
4.2 Configuring PCI Devices . . . 11
4.3 Configuring SCSI Devices . . . 12
4.4 Compiling HAPPI . . . 12
4.5 Loading Evaluators . . . 12
4.6 Loading Policies . . . 12

5 Sample Policy **13**
5.1 Inserting and Removing Policies . . . 13
5.2 Predicting Idleness and Changing Power States . . . 14
5.3 Evaluation . . . 14

6 Sample Measurement **15**
6.1 Inserting and Removing Measurements . . . 15
6.2 Updating Measurements . . . 15
6.3 Providing Measurements to Policies . . . 15

7 Running Experiments Using HAPPI **15**
7.1 Testing a New Policy . . . 16
7.2 Comparing Policies . . . 16

8 Acknowledgments **16**

1 Introduction

Power management has become an important design criterion for a variety of modern computing systems, from embedded systems to high-performance computers. Power management research has focused on the creation of better algorithms to reduce energy consumption. These algorithms are called *policies*. Existing studies assume that only a single policy may be used to control a device's power states. However, significantly different policies may be necessary to save energy for different workloads.

In our previous work [9], we demonstrate that different policies should be selected at run-time based upon the current workload. We call this mechanism automatic policy selection. This technical report describes the Homogeneous Architecture for Power Policy Integration (HAPPI), a framework that selects the best power policy from a library of policies at run-time. In HAPPI, policies are implemented as kernel modules and manage power states at the operating system (OS) level. HAPPI defines homogeneous requirements for all policies, allowing a single policy to control multiple devices simultaneously. This interface allows policies to be reused between devices, in contrast to existing methods in Linux [3] and Windows [8] that require policies to be implemented in each device driver.

2 HAPPI Overview

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.

2.1 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. 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.

2.2 Measurements

We refer to the data required by policies to make decisions as *measurements*. HAPPI always provides 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. These accesses are used by HAPPI to determine how well each policy controls the device. Measurements may also be very complex. For example, some policies require more complex measurements, such as probability matrices for access rates [5].

HAPPI also provides software energy and delay models that may be used to monitor the energy consumption and performance of the computer. Energy is accumulated after each access and after every ten seconds of idleness. We use simple state-based models to measure energy. This is consistent with other OS-based power management schemes [12]. We define delay 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.

2.3 Policy Selection

Policy selection is performed by the *evaluator*. 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 a function, called an *estimator*, 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 estimator 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.

3 Implementation

This section focuses on HAPPI's implementation. We implement the architecture in the Linux 2.6.17 kernel to demonstrate HAPPI's ability to select policies at run-time and provide a reference for future OSPM. HAPPI's implementation is split into two halves: statically compiled code and loadable kernel modules. The statically compiled code includes the recording of device accesses, wrapper functions for state transitions, and glue logic to maintain lists of policies and devices. Policies, evaluators, and most measurements are implemented as loadable kernel modules that may be inserted and removed at run-time. These modules are described in Section 3.3, Section 3.6, and Section 3.7, respectively. The only measurement that is statically compiled into the kernel is a device's access history. This measurement will be discussed in Section 3.5.

3.1 Environment Setup

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 accessed at least once every five seconds and is never idle long enough to save energy. We must also adjust the commit interval of journaling file systems, such as `ext3` and `ReiserFS`, because `laptop_mode` does not delay commits. Increasing the commit interval increases the amount of data loss during power failure, similar to using `laptop_mode`. For our experiments, we use a five minute commit interval. Using `laptop_mode` and longer commit intervals pose an increased risk of data loss in the event of a system crash

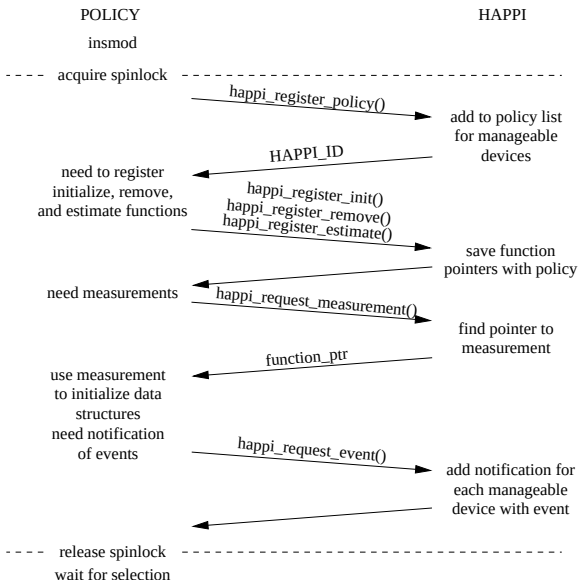

Figure 1: Timeline for policy insertion. Arrows indicate function calls and returns. Dotted lines indicate synchronization points. Notifications include accesses, state transitions, and file operations.

or power outage. However, we note that this risk of data loss is a fundamental property of dirty page writeback and unavoidable even if we disable `laptop_mode`. The aforementioned environment setup merely increases the window of vulnerability.

3.2 Configuring Hardware Specific Parameters

The Advanced Configuration and Power Interface (ACPI) specification [1] describes hardware power parameters that modern hardware should provide to ensure power management support. Although processors are widely supported, I/O devices and peripherals often lack ACPI support. Hence, the OS cannot automatically obtain the power consumption of these devices. HAPPI provides hardware parameters to policies through two methods: a statically compiled header file and run-time modification. HAPPI includes a header file (`include/linux/happi.h`) that contains locations to specify each device's power consumption. HAPPI also provides an interface through the `proc` virtual file system where power parameters may be changed at run-time to simulate different hardware.

3.3 Inserting and Removing Policies

Figure 1 illustrates a timeline of actions for policy insertion. The left column indicates actions taken by the policy. The right column shows actions taken by HAPPI. Arrows indicate function calls and return values. A policy must register with HAPPI before it may be selected to control devices. Registration begins when a policy is inserted into the kernel using `insmod`. A spin lock protects the policy list and must be acquired before the policy can begin registering with HAPPI. We use a spin lock rather than a semaphore because HAPPI's data structures may be called from both process and interrupt context [2]. The policy calls the `happi_register_policy` function to inform HAPPI that it is being inserted into the kernel and indicates the types of devices the policy can manage. HAPPI responds by returning a unique `HAPPI_ID` to identify the policy on all future requests. The policy registers callback functions to begin the policy's control of a device (`initialize`), stop the policy's control of a device (`remove`), and provide an estimate to the evaluator for a device (`estimate`). Then, the policy initializes local data structures for each eligible device.

After initializing local data structures, the policy requests notification of specific system events by calling the `happi_request_event` function. These events, listed in Table 1, include notification after each device access,

Name	Description
HAPPI_NOTIFY_BLOCK	Filtered block device requests
HAPPI_NOTIFY_BLOCK_SHORT	Unfiltered block device requests
HAPPI_NOTIFY_BLOCK_STATE	Block device state change
HAPPI_ALL_BLOCK_EVENTS	All block requests and state changes
HAPPI_NOTIFY_NET	Filtered network device requests
HAPPI_NOTIFY_NET_SHORT	Unfiltered network device requests
HAPPI_NOTIFY_NET_STATE	Network device state change
HAPPI_ALL_NET_EVENTS	All network requests and state changes
HAPPI_ALL_ACCESSES	Filtered requests from all device types
HAPPI_ALL_STATES	State changes from all device types
HAPPI_FILE_OPEN	File opened by application
HAPPI_FILE_CLOSE	File closed by application
HAPPI_FILE_READ	File read by application
HAPPI_FILE_READ_DISK	File read from disk by application
HAPPI_FILE_WRITE	File written by application
HAPPI_FILE_WRITE_DISK	File written to disk by dirty page writeback
HAPPI_FILE_OPS	Any file access by application

Table 1: All notifications provided by HAPPI. Filtered block device requests occur 1000 ms after the previous access. Filtered network device requests occur 250 ms after the previous access.

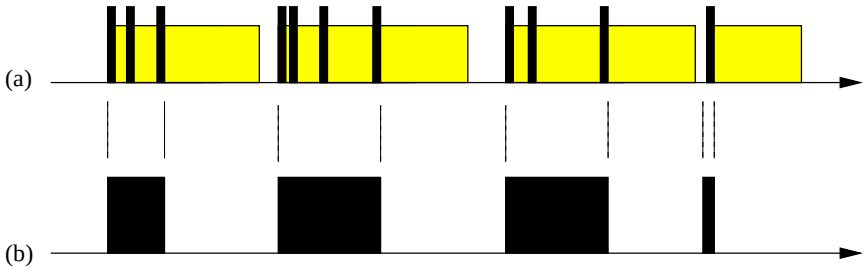

Figure 2: Accesses pass through a short filter before being seen by policies. (a) Unfiltered accesses with filter shown. (b) Filtered accesses.

state transition, and file access. However, these events are received by only the active policy to reduce the overhead of multiple policies running simultaneously. All registered measurements receive all requested notifications because measurements are common to all policies. After the notifications have been created, the policy releases the spin lock and is eligible for selection. Since policy registration uses the `insmod` command, administrator privilege is required to add new policies. Hence, policies do not cause any security breaches.

3.4 Recording Device Accesses and State Transitions

Simunic et al. [11] observe that policies predict more effectively if a 1000 ms filter is used for disk accesses and 250 ms filter is used for network accesses. These filters allow bursts of accesses to be merged into a single access. This process is demonstrated in Figure 2. Figure 2(a) shows a sequence of accesses, represented by solid bars. The shaded area represents a filter. The filter window restarts after each access. Each access that occurs after the filter expires is considered a new access. Figure 2(b) illustrates how policies view the accesses after the filter. An access is defined by a time span of activity extending from the first access of the burst to the completion of the last access of the burst. This representation preserves the amount of idleness between the end of an access and the beginning of the next access.

The filtering mechanism also prevents policies' predictions from being skewed by rapid bursts of accesses. Without the filtering mechanism, we observe that policies significantly mispredict the amount of idleness between accesses.

These bursts of accesses occur because an I/O access consists of two parts: a top-half and a bottom-half [7]. When the application performs a read or write operation, it uses system calls to the OS to generate requests. The OS passes these requests on to the device driver on behalf of the application. This process is called the top-half. The application may continue executing after issuing write requests but must wait for read requests to complete. The device driver constitutes the bottom-half. The bottom-half interfaces with the device, returns data to the application's memory space, and marks the application as ready to resume execution. The mechanism allows top-half actions to perform quickly by returning to execution as soon as possible. Bottom-half tasks are deferred until a convenient time. This mechanism allows the OS to merge adjacent blocks into a single request or enforce priority among accesses. Since the bottom-half waits until a convenient time to execute, the mechanism is referred to as *deferred work*. Since accesses may be deferred, multiple accesses may be issued to a device consecutively.

We insert functions to record device accesses after the access completes because a significant delay may occur between an access's issue and completion during state changes. By recording accesses after completion, policies may assume that the device is in the active power state and make decisions immediately, rather than wait until the device is fully awake. We record block requests in the `__end_that_request_first` function of `block/ll_rw_blk.c`. Network requests are recorded at several places throughout the kernel. We record network sends after each call to the `hard_start_xmit` function pointer and network receives in the `netif_rx` function of `net/core/dev.c`.

Deferred work plays an important role in managing state transitions in Linux. When a state transition is requested, a command is passed to a bottom-half to update the device's power state. The actual state transition may require several seconds to complete and does not notify Linux upon completion. The exact power state of a device during a transition is unknown to Linux because the commands are handled at the device driver-level. Device accesses are managed in device drivers, as well, implying that the status of outstanding requests are also unknown and cannot be used to infer power states. HAPPI could obtain the exact power state of a device by modifying the bottom-half in the device driver. However, drivers constitute 70 percent of Linux's source code [4]. Any solution that requires modifying all device drivers is not scalable. Modifying the subset of drivers for the target machine is not portable. Hence, we estimate state transition time using ACPI information and update the state after the time expires.

3.5 Maintaining Access History

All policies require knowledge of device accesses to predict idleness and provide estimates for policy selection. The method for measuring device accesses directly affects HAPPI's ability to select the proper policy for different workloads. In Section 3.4, we describe how a filter merges deferred accesses into a single access. When a request passes through the filter, HAPPI records the access in a circular buffer. We use a circular buffer rather than a dynamically-allocated list to reduce the time spent in memory allocation and release and limit the amount of memory consumed by HAPPI. After HAPPI records the access, the active policy and all measurements are notified of the event. Since all policies require information about device accesses, these functions are statically compiled into the kernel. Access histories are the only components of HAPPI that are not loaded or removed at run-time.

We determine the circular buffer's length experimentally because the proper buffer length depends on workloads. We choose five different workloads to induce different levels of idleness on three different devices (a IDE disk, a CD-ROM, and a PCI network card). These workloads are described below:

Workload 1: Web browsing + buffered media playback from CD-ROM.

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

Workload 3: CVS checkout from remote repository.

Workload 4: E-mail synchronization + sequential access from CD-ROM.

Workload 5: Kernel compile.

Figure 3(a) illustrates an access trace consisting of five unique workloads for the IDE disk. Each workload is separated by a vertical line and labeled above the figure. Figure 3(b) illustrates the amount of history (in seconds)

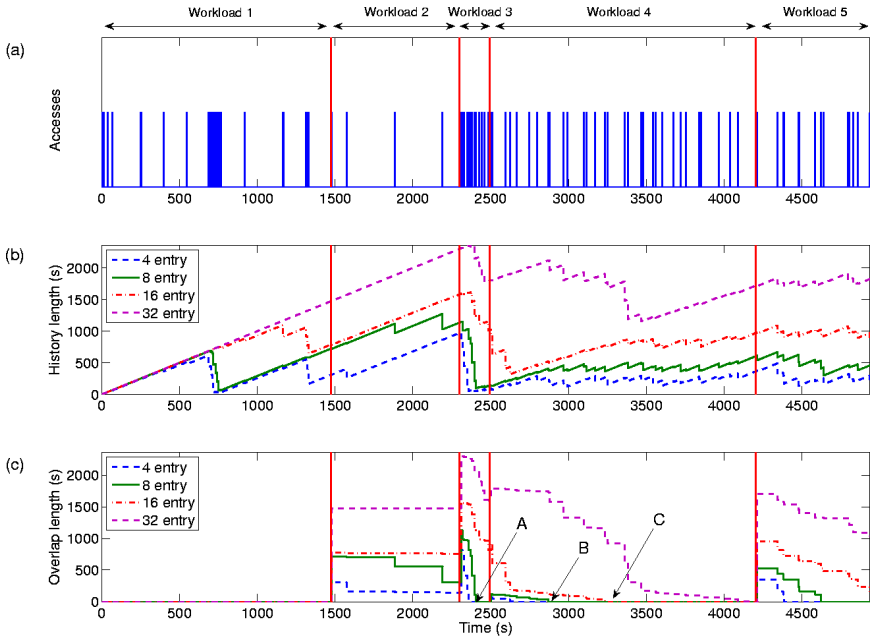

Figure 3: Accesses, history buffer length, and overlap between workload histories for different window sizes for desktop workloads.

retained by HAPPI for circular buffer sizes of 4, 8, 16, and 32 entries. When no accesses occur, the history length increases linearly. If a new access overwrites another access in the circular buffer, the history length decreases sharply. An ideal history provides full knowledge of the current workload and zero knowledge of previous workloads. The ideal history would appear as a linear slope beginning at zero for each workload. A circular buffer naturally discards history as new accesses occur. Figure 3(c) shows how much history overlaps with previous workloads. This plot appears as a staircase function because history is discarded in discrete quantities as accesses are overwritten in the circular buffer.

Our implementation targets interactive workloads, common to desktop environments. We use an 8-entry buffer because this buffer quickly discards history when workloads change but maintains sufficient history to select policies accurately. We observe from Figure 3(c) that the 8-entry buffer requires 107 seconds to discard Workload 2 (indicated at point A) and 380 seconds to discard Workload 3 (point B). In contrast, the 16-entry buffer requires 760 seconds to discard Workload 3 (point C) and cannot completely discard Workload 2 before Workload 3 completes. The 4-entry buffer discards history more quickly than the 8-entry buffer but does not exhibit a sufficiently long history to estimate policies' energy consumption accurately. Systems with less variant workloads, such as servers, may use a larger buffer, such as a 32-entry buffer. A larger buffer requires longer to discard past workloads but allows for a better prediction of the current workload in steady-state operation. The buffer length is set by the administrator.

Figure 4 depicts the second disk of a server containing three disks running the SPECWeb99 benchmark with 20–460 simultaneous connections, increasing in increments of 40. This figure illustrates 32, 64, and 128-entry buffers. We notice in Figure 4(a) that accesses begin at 60 connections, decrease at 260 connections, and increase at 340 connections. This effect occurs because each of the three disks in the system contain different data sets. At 20 connections, all accesses are serviced from the first disk. At 260 connections, most of the files on the disk are cached in memory. At 340 connections, the working set exceeds the size of physical memory, so the disk becomes active again. The best opportunities to save energy occur between 260 and 340 connections. We observe at points A and B that only the 32-entry buffer discards the previous history before the workload changes. The 32-entry buffer also maintains a linear increase in history length in Figure 4(b). Hence, we choose a 32-entry buffer for server workloads.

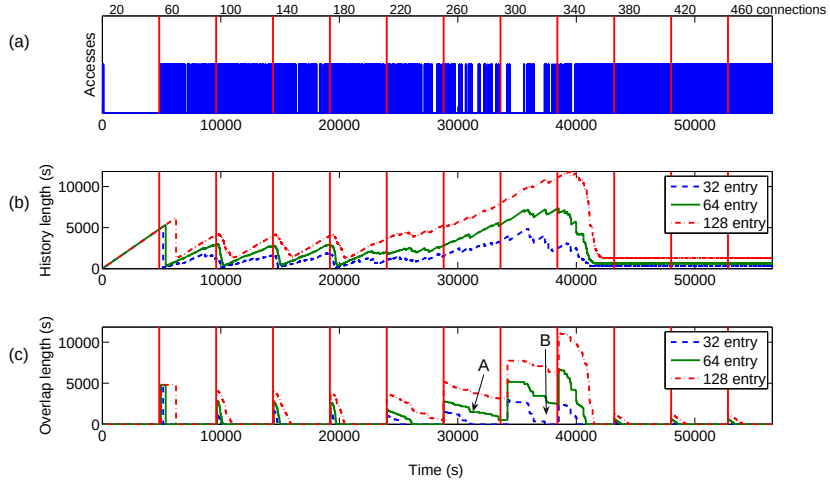

Figure 4: Accesses, history buffer length, and overlap between workload histories for different window sizes for server running SPECWeb99.

3.6 Advanced Measurements

HAPPI always provides an access history for each device to facilitate policy selection. However, some policies require more complex data than access history, such as probability matrices for access rates [5]. Advanced measurements can be directly computed from the history of recent accesses. Since such information is not required by all policies, HAPPI does not provide the information directly. HAPPI provides the minimum common requirements for policies. This design is based upon the end-to-end argument of system design [10] by providing the minimum common requirements to avoid unnecessary overhead. Although it does not directly provide these complex measurements, HAPPI provides an interface for measurements to be added as loadable kernel modules. A new measurement registers a callback function pointer with HAPPI that returns the measurement and requests events similar to the other policies. If a policy requires additional measurements, the policy calls the `happi_request_measurement` function with an identifier for the measurement. HAPPI returns a function pointer to the policy for retrieving the measurement data.

We implement measurements as separate kernel modules because several policies may require the same measurement. By separating the measurement from the policies, the measurement is computed once for all the policies in the system. Since measurements are always needed, they receive all requested events, whereas inactive policies do not respond to events. If policies were individually responsible for generating measurements, their measurements would only consider the time when the policy has been active. Thus, policies would consider different time spans in their estimator functions. Implementing measurement as separate modules also allows measurements to be improved independently of policies.

3.7 Evaluating and Changing Policies

HAPPI automatically chooses the best policy for each device for the current workload and allows power policies to change at run-time, whereas existing power management implementations require a system reboot. HAPPI's evaluator is responsible for selecting the active policy. The evaluator is a loadable kernel module, allowing the system administrator to select an evaluator that optimizes for specific power management goals, for example, to minimize energy consumption under performance constraints. Since the evaluator is a loadable module, the administrator may change evaluators without rebooting if power management goals change. The administrator inserts the module using the `insmod` command. From this point onward, the evaluator selects power policies automatically. When a policy is inserted into the kernel using `insmod`, the evaluator is notified that a new policy is present and re-evaluates all

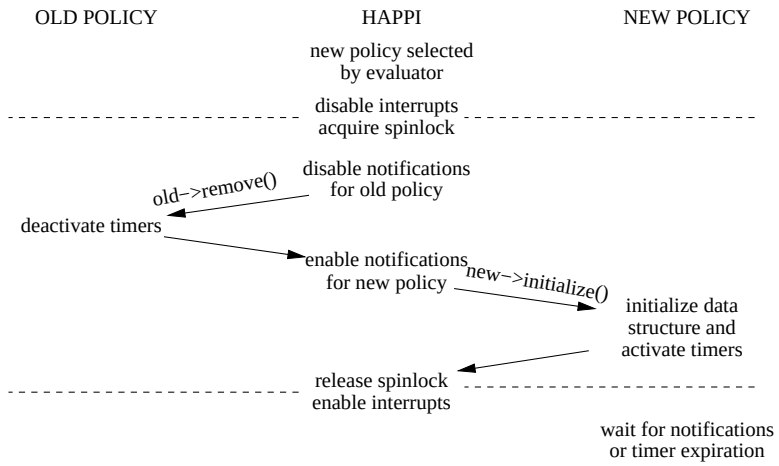

Figure 5: Timeline for policy selection. Arrows indicate function calls and returns. Dotted lines indicate synchronization points. Notifications include accesses, state transitions, and file operations.

policies. After the best policy is selected for each device, the policy controls the device until the next evaluation.

If the evaluator changes the active policy, the old policy must relinquish control of the device's power states and the new policy must acquire control. Figure 5 illustrates how HAPPI changes policies at run-time without rebooting. The left column indicates actions taken by the old policy. The middle column describes HAPPI's actions. The right column indicates the new policy's actions. When notified by the evaluator to change the active policy, HAPPI disables interrupts and acquires a spin lock protecting the device. Disabling interrupts prevents any of the old policy's pending timers from expiring and blocks accesses from being issued or received from the device. Acquiring the spin lock prevents HAPPI from interrupting the old policy if it is currently issuing a command to the device. Once the spin lock is acquired, we are guaranteed that the old policy is not currently controlling the device. HAPPI deactivates all notifications for the old policy and calls the old policy's `remove` function to delete any pending timers and force the policy to stop controlling the device. After the old policy has successfully stopped controlling the device, HAPPI enables the notifications for the new policy and calls the new policy's `initialize` function. The new policy uses this function to update any stale data structures and activate its timers. At this point, HAPPI enables interrupts and releases the device's spin lock, allowing the new policy to become active. The performance loss for disabling interrupts and acquiring locks is negligible.

Replaced policies may elect to save or discard their current predictions. If history is saved, the information may be used when selected in the future. In our policies, we elect to discard all previous history when a policy is replaced in favor of a different policy. A policy is replaced because its estimate indicates that it is incapable of saving as much energy as another eligible policy for the current workload. Replacement implies that a policy's idleness prediction is poor. Hence, discarding previous history resets the policy's predictions to an initial value when providing another estimate and often allows the policy to revise its prediction much more quickly than by saving history.

Changing policies indicates that (a) the old policy is mispredicting or (b) the new policy can exploit additional idleness. HAPPI must evaluate policies frequently enough to detect these conditions. Figure 6 illustrates two workloads where these conditions occur. Vertical bars indicate device accesses. Figure 6(a) depicts a workload changing from long to short idleness. This transition is likely to induce mispredictions because policies over-predict the amount of idleness. Policies quickly correct their predictions to avoid frequent shutdowns. When the previous workload is discarded from HAPPI's access trace, a new policy is selected that predicts idleness more effectively and consumes less energy for the new workload. Figure 6(b) depicts a workload changing from short to long idleness. Policies that make decisions on each access cannot recognize and exploit long periods of idleness because no accesses occur to update the policies' predictions.

To reduce missed opportunities to save energy, we evaluate policies frequently. HAPPI evaluates all policies once every 20 seconds to determine if a better policy is eligible among the available policies. We select this interval because

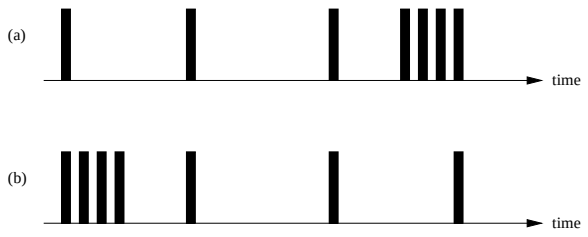

Figure 6: Vertical bars indicate device accesses. (a) Workload changing from long to short idleness. (b) Workload changing from short to long idleness.

it exhibits quick response to workload changes without thrashing between policies. Here, thrashing means changing policies too often, in particular, changing policies every time policies are evaluated. Policy thrashing is a problem because deselected policies discard their previous predictions. Hence, a momentarily deselected policy may require several minutes to refine its prediction, reducing opportunities to save energy. 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. We observe that a 20 second period evaluates quickly but does not frequently cause thrashing. To further reduce the likelihood of thrashing, we assign a small bonus to the currently selected policy. The bonus reduces the current policy's energy estimate by 5%. Our experiments demonstrate that this bonus significantly reduces thrashing and improves overall energy savings.

4 Installation and Configuration

HAPPI does not automatically detect system hardware. Hence, some manual steps must be performed to make HAPPI work with new hardware. These instructions assume a basic understanding of C and the tools necessary to compile the Linux 2.6 kernel.

4.1 Configuring IDE Devices

The power parameters for IDE devices are described in terms of buses and master-slave configuration. The configuration of a system's devices may be determined from the /sys filesystem as shown below. Using `bash` syntax, the "#" symbol represents comments and "$" represents commands.

```
$ cd /sys/bus/ide/devices

# List the different IDE buses on the system. This machine has two buses.
# The .0 indicates that device is master. (.1 indicates slave)
$ ls
0.0  1.0

$ ls 0.0
block:hda    # Device hda lies on bus IDE0 as master

$ ls 1.0
block:hdc    # Device hdc lies on bus IDE1 as master
```

This information indicates that the machine has two buses, `IDE0` and `IDE1`. The disk `hda` is the master device on `IDE0`. The CD-ROM `hdc` is the master device on `IDE1`. If the disk and CD-ROM are located on the same bus as master and slave, the information will appear similar to the following:

```
# List the different IDE buses on the system.  This machine has one bus
# with two devices.  0.0 is IDE0 master.  0.1 is IDE0 slave.
$ ls
0.0  0.1

$ ls 0.0
block:hda    # Device hda lies on bus IDE0 as master

$ ls 0.1
block:hdb    # Device hdb lies on bus IDE0 as slave
```

After determining the master-slave configurations for each bus, the devices may be configured. The header file `include/linux/happi.h` provides macros that describe each device's power parameters. As described in Section 3.2, this information is defined in the ACPI device specification but often unavailable for I/O devices. A subset of these macros are shown below. The IDE macros use the first number to indicate the bus number and the second number to indicate master (0) or slave (1). The power parameters include active power (D0), standby power (D1), transition time, and transition energy. Here, we use C comment syntax (/* */).

```
/* See include/linux/happi.h -- ide0 master */
#define IDE00_POWER_D0     850                 /* active power (mW) */
#define IDE00_POWER_D1     250                 /* standby power (mW) */
#define IDE00_TIME_D1_D0   MS_TO_JIFFIES(4500) /* transition time (jiffies) */
#define IDE00_ENERGY_D1_D0 17100000            /* transition energy (uJ) */

/* ide0 slave */
#define IDE01_POWER_D0     -1                  /* -1 indicates no device */
#define IDE01_POWER_D1     -1                  /* -1 indicates no device */
#define IDE01_TIME_D1_D0   -1                  /* -1 indicates no device */
#define IDE01_ENERGY_D1_D0 -1                  /* -1 indicates no device */
```

4.2 Configuring PCI Devices

HAPPI supports fine-grained power management of PCI devices, such as network cards. To enable network power management, HAPPI needs to know the vendor and device number for the device. To obtain this information, perform the following steps.

```
# Some output not shown for brevity.
$ lspci
02:00.0 Ethernet controller: Broadcom Corporation NetXtreme BCM5705M Gigabit
Ethernet (rev 01)
```

Locate the PCI address of the device in the output of `lspci`. This number is in the first column of information. In this example, the network card is located at PCI address 02:00.0. We can use this information to locate the vendor and device number from the /sys filesystem.

```
$ cd /sys/bus/pci/devices

# All PCI devices are located under this directory, listed by PCI address
$ ls
0000:02:00.0

# Choose the PCI device
$ cd 0000\:02\:00.0
```

```
# Display the vendor and device numbers
$ cat vendor
0x14e4
$ cat device
0x165d
```

These numbers must be provided to HAPPI in `include/linux/happi.h` to allow power management. In this file, change the macros for `PCI0_VENDOR` and `PCI0_DEVICE` to the numbers determined previously.

```
/* In include/linux/happi.h */
#define PCI0_VENDOR              0x14e4
#define PCI0_DEVICE              0x165d
```

The PCI device number has no significance in these macros. The numbers exist to allow many PCI devices to be managed by HAPPI. Note that the PCI device driver must support power management for HAPPI to successfully control its power states.

4.3 Configuring SCSI Devices

The procedure for configuring SCSI devices is identical to PCI devices, except that the vendor and device number are written to the `SCSI0_DEVICE` and `SCSI0_VENDOR` macros.

4.4 Compiling HAPPI

The compilation process for HAPPI is the same as for any Linux kernel, except that the `CONFIG_HAPPI` and `CONFIG_HAPPI_ACPI` configuration options must be set. These options may be set manually in the `.config` file or through the `menuconfig` option. The configuration options for HAPPI may be found under the Power Management options. To compile the kernel in three commands:

```
$ make menuconfig
$ make all
$ sudo make modules_install install && reboot
```

4.5 Loading Evaluators

The evaluator is responsible for selecting the best policy for each device in the system. Evaluators are implemented as loadable kernel modules. To compile an evaluator, type the following command, where `HAPPI_SRC` is the location of the HAPPI kernel code, or use the supplied Makefile:

```
$ make -C ${HAPPI_SRC} SUBDIRS=${PWD} modules
```

After compiling the evaluator, insert the evaluator into HAPPI using `insmod evaluator.ko`. An evaluator must be loaded before any policy may become active.

4.6 Loading Policies

Policies are implemented as loadable kernel modules. To compile policies, type the following command, where `HAPPI_SRC` is the location of the HAPPI kernel code, or use the supplied Makefile:

```
$ make -C ${HAPPI_SRC} SUBDIRS=${PWD} modules
```

After compiling the policy, insert the policy into HAPPI using `insmod policy.ko`. An evaluator must be loaded before the policy may become active.

```
 1 static int __init exp_average_init(void)
 2 {
 3    unsigned long flags;
 4
 5    spin_lock_irqsave(&happi_mutex, flags);
 7
 8    /* Policy may control block and network devices */
 9    HAPPI_ID = happi_register_policy(HAPPI_BLOCK_POLICY |
10                                     HAPPI_NETWORK_POLICY);
11
12    /* Register all required functions */
13    happi_register_init(HAPPI_ID, &initialize_variables);
14    happi_register_eval(HAPPI_ID, &evaluate);
15    happi_register_exit(HAPPI_ID, &restore_core_structures);
16
17    /* Initialize all local data structures */
18    initialize_all_pdevice();
19
20    /* Request block and network device accesses */
21    happi_request_device_event(HAPPI_ID, HAPPI_NOTIFY_BLOCK,
22                               &handle_device_access);
23    happi_request_device_event(HAPPI_ID, HAPPI_NOTIFY_NETWORK,
24                               &handle_device_access);
25
26    spin_unlock_irqrestore(&happi_mutex, flags);
27    return 0;   /* success */
28 }
29 module_init(exp_average_init);
```

Figure 7: Initialization code for EXP. Called when a policy is inserted into the kernel.

5 Sample Policy

This section provides a full sample policy with description of its various parts. The policy described here is the exponential average policy [6]. For brevity, we will abbreviate the exponential average policy as EXP. This policy uses exponential averages to predict the length of the next idleness interval based upon previous intervals. The length of the next idleness interval is computed by the equation $I[k+1] = \alpha i_k + (1-\alpha)I[k]$, where i_k is the actual length of the last idle interval, $I[k]$ is the previous idleness prediction, and α is a weight factor in the range $(0,1)$. If $I[k+1]$ is longer than the breakeven time of the device, EXP shuts down the device. The breakeven time is defined as the amount of time that the device must remain asleep to save energy. If $I[k+1]$ is shorter than the breakeven time, the device remains in the active state until the next access occurs.

5.1 Inserting and Removing Policies

Figure 7 contains the initialization code for HAPPI. The basic structure described in Section 3.3 is evident. Line 5 acquires the spin lock. Lines 8–10 register the policy with HAPPI and indicate that the policy can control both block and network devices. Lines 12–15 provide HAPPI with the policy's callback functions. Lines 17–18 initialize local data structures. Lines 20–24 request notification of device accesses on block and network devices. Line 26 releases the spin lock, allowing the kernel to resume execution. Line 29 is a macro provided by Linux to indicate the function that should be called on `insmod`.

Figure 8 contains the code required to remove the policy from the kernel. Line 5 acquires the spin lock. Line 6

```
 1 static void __exit exp_average_exit(void)
 2 {
 3    unsigned long flags;
 4
 5    spin_lock_irqsave(&happi_mutex, flags);
 6    delete_all_pdevice();
 7    happi_unregister_policy(HAPPI_ID);
 8    spin_unlock_irqrestore(&happi_mutex, flags);
 9 }
10 module_exit(exp_average_exit);
```

Figure 8: Removal code for EXP. Called when a policy is removed from the kernel.

frees all the local data structures that have been created during initialization. Line 7 removes the policy from HAPPI. Line 8 releases the spin lock. Line 10 is a macro provided by Linux to indicate the function that should be called on `rmmod`.

5.2 Predicting Idleness and Changing Power States

Figure 9 provides the code for idleness prediction and state selection. This function is called after every unfiltered access, as described in Section 3.4. Line 8 locates the local data structure associated with the device. This data structure contains the device's idleness prediction (pd→predict), breakeven time (pd→tbe), and prediction history (pd→old_predict). Line 9 locates the device's circular buffer of accesses. Lines 11–15 compute the idleness between the current and the previous access. Line 14 determines the time between the first access of each burst. Line 15 subtracts the duration of the first burst. Recall that Figure 2 illustrates how an access is interpreted by policies in HAPPI. Lines 17–18 save the previous prediction. This is used by the evaluator in Section 5.3 to improve the estimator's accuracy. Lines 20–21 compute the predicted idleness for the next period. The code for this function is shown on Lines 32–40.

At this point, pd→predict contains the prediction for the next idle period. On Line 24, the policy checks if the idleness exceeds the device's breakeven time. If the idleness is shorter than the breakeven time, the device should not be shutdown. In this case, shown on Line 25, the policy returns to HAPPI. If the predicted idleness exceeds the breakeven time, the policy sets a timer to shutdown the device. This is shown on Lines 27–28. We use a timer to delay the shutdown because the filter has not yet expired. By waiting until the filter expires, we reduce the likelihood that another access will immediately awaken the device. This behavior is described previously in Section 3.4.

5.3 Evaluation

Figure 10 provides the code for EXP's evaluation function. This function is called by the evaluator periodically to determine EXP's ability to control the device's power states. Lines 5–9 locate the data structures for the device and initialize the estimate's energy and delay to zero. Lines 11–13 compute the initial idleness prediction. If the policy is not currently active for the device, the initial prediction is equal to the breakeven time (pd→tbe). However, if the policy is currently active, the policy must determine its actual prediction for the access. Recall that this value is recorded on Line 21 in Figure 9. Neglecting this initial value causes a large estimate error.

Lines 15–37 compute the energy consumption and delay for each access. Lines 20–22 compute the actual idleness (`t`) between the current and the previous accesses and the time required for any state change to occur (`awake_time`). Recall that a short timer is used to wait until the access filter expires before shutting down the device. Line 25 checks if the policy would have shut down the device before the current access. If not, the energy for remaining in the active state is accumulated in Line 26, and execution jumps to the end of the loop. If the device would have shut down, Lines 30–34 are executed to accumulate the energy for shutting down after `awake_time` and awakening the device after `t - awake_time`. Line 35 saves the current prediction in case the policy is selected. Line 36 updates the predicted idleness before the next loop iteration.

We omit the energy computation since the final access in the buffer. This code is very similar to the loop in Lines 15–37, except for two omissions. First, the device is not awakened after being shut down because no access has occurred. Second, the prediction is not updated. After computing the energy consumption since the final access, the estimate is returned to the evaluator on Line 40.

6 Sample Measurement

This section provides a sample measurement with description of its various parts. The measurement described here is a probability matrix for access rates [5]. This particular measurement considers four types of accesses: an access occurring after a previous access, no access after an access, an access occurring without a previous access, and no access after no access. These probabilities are necessary to perform stochastic optimizations for power management [5].

6.1 Inserting and Removing Measurements

Figure 11 contains the initialization code for the state transition probability model measurement. The basic structure is similar to the description for policies in Section 3.3. Line 8 acquires the spin lock. Line 10 registers the measurement with HAPPI provides a callback function to return the measurement to a policy. Lines 13–23 configure a periodic timer for each device. The measurement wakes up every `EVAL_PERIOD` and determines if an access has occurred during the previous `EVAL_PERIOD`. Since the measurement is not called on every access, the measurement does not need to request notification of events. Line 25 releases the spin lock, allowing the kernel to resume execution. Line 28 is a macro provided by Linux to indicate the function that should be called on `insmod`.

6.2 Updating Measurements

Figure 12 contains the code that is called every `EVAL_PERIOD` to compute the state probability matrix. Lines 3–4 determine the individual device being measured. Line 7 checks if an access occurred the last time the measurement was called. Line 10 checks if an access occurred in the previous `EVAL_PERIOD` by observing the circular buffer. Lines 12–15 subtract the oldest access from the measurement's history. Lines 17–22 add the current access to the measurement's history. Lines 24–25 records if an access occurred to be used the next time the measurement is called.

6.3 Providing Measurements to Policies

A measurement's purpose is to provide information to a policy. To accomplish this, the measurement provides a callback function to HAPPI that policies may use to obtain the measurement. Figure 13 details this process. Lines 1–5 are provided by the measurement. The function returns the measurement associated with the device as void pointer to facilitate a generic interface with HAPPI. Recall from Line 10 of Figure 11 that the measurement registers this function with HAPPI. Lines 7–11 are provided by the policy. Line 8 is a header file containing the description of the measurement's data structure. Line 9 is a function pointer that stores the callback for multiple uses. Line 10 obtains the measurement callback from HAPPI using the `happi_request_measurement` function. This function accepts a string containing the name of the measurement and returns the measurement. Line 11 demonstrates how the measurement is called from the policy. The policy may use the data as necessary to compute idleness.

7 Running Experiments Using HAPPI

One of HAPPI's greatest advantages is the ability to perform experiments easily on actual hardware. This section describes how to setup common experiments using HAPPI. The evaluator must have `printk` statements included to print each policy's estimate to the Linux kernel log. These print statements may be extracted by shell scripts to compute the policy's effectiveness.

7.1 Testing a New Policy

HAPPI includes a null policy to compute the energy consumption of a computer system without power management. To determine a policy's effectiveness, it should be compared to the NULL policy. To insert the policies into HAPPI, type the following commands.

```
$ sudo insmod evaluator.ko
$ sudo insmod null.ko
$ sudo insmod new-policy.ko

# Run experiment
$ dmesg > kernel-log.txt
```

The `kernel-log.txt` file may be parsed to determine each policy's estimates and the active policy. Estimates are indicated by the `EVAL` keyword. Policy assignments are indicated by the `ASSIGN` keyword.

7.2 Comparing Policies

When comparing multiple policies, both policies should be inserted simultaneously. The following commands should be executed.

```
$ sudo insmod evaluator.ko
$ sudo insmod null.ko
$ sudo insmod new-policy-1.ko
$ sudo insmod new-policy-2.ko

# Run experiment
$ dmesg > kernel-log.txt
```

After the experiment is completed, the `kernel-log.txt` file may be parsed to determine what workloads favor each policy.

8 Acknowledgments

This work is supported by NSF CAREER CNS-0347466. 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.

References

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

[2] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly Media, 2006.

[3] D. Brownell. Linux Kernel 2.6.17 Source: Documentation/power/devices.txt. http://www.kernel.org, July 2006.

[4] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating Systems Errors. In *ACM Symposium on Operating Systems Principles*, pages 73–88, 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] C.-H. Hwang and A. C.-H. Wu. A Predictive System Shutdown Method for Energy Saving of Event-driven Computation. *ACM Transactions on Design Automation of Electronic Systems*, 5(2):226–241, April 2000.

[7] R. Love. *Linux Kernel Development*. Sams Publishing, 2004.

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

[9] N. Pettis, J. Ridenour, and Y.-H. Lu. Automatic Run-Time Selection of Power Policies for Operating Systems. In *Design Automation and Test in Europe*, pages 508–513, 2006.

[10] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-To-End Arguments in System Design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

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

[12] 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 Architectural Support for Programming Languages and Operating Systems*, pages 123–132, 2002.

```
 1 static int handle_device_access(struct happi_device *hd)
 2 {
 3   struct pdevice *pd;
 4   struct blk_q *q;
 5   int recent, previous;
 6   unsigned long diff, idleness;
 7
 8   pd = get_pdevice(hd);   /* Local data structure*/
 9   q = hd->accesses;       /* Recent accesses for device */
10
11   /* Compute time since last access */
12   recent = q->last;
13   previous = (recent + REQ_QUEUE_SIZE - 1) % REQ_QUEUE_SIZE;
14   diff = q->jiffies[recent] - q->jiffies[previous];
15   idleness = diff - q->elapsed_time[previous];
16
17   /* Save the previous prediction for estimator */
18   pd->old_predict[recent] = pd->predict;
19
20   /* Predict length of next idle period */
21   pd->predict = compute_prediction(pd, pd->predict, idleness);
22
23   /* If idleness shorter than breakeven time, stay awake */
24   if (pd->predict < pd->tbe)
25     return 0;
26
27   /* If longer, set shutdown timer to expire after filter */
28   reset_timer(hd);
29   return 0;
30 }
31
32 /* Computes idleness prediction */
33 static inline unsigned long
34 compute_prediction(struct pdevice *pd, unsigned long predict,
35                    unsigned long diff)
36 {
37   unsigned long new = pd->alpha * (diff / 1000) +
38                       (1000 - pd->alpha) * (predict / 1000);
39   return new;
40 }
```

Figure 9: Idleness prediction and state selection code. Called after every filtered access is completed.

```
 1 static struct estimate *evaluate(struct happi_device *hd)
 2 {
 3   /* Variable declarations omitted for space */
 4
 5   pd = get_pdevice(hd); est = &pd->est;
 6   s = hd->dstates; q = hd->accesses;
 7
 8   /* Initialize estimate */
 9   est->energy = 0; est->delay = 0;
10
11   /* Compute initial prediction */
12   predict = initialize_estimate_prediction(pd);
13   i = q->first; pd->old_predict[i] = predict;
14
15   /* Compute prediction, energy, and delay for each access */
16   for (i = q->first, then = q->jiffies[q->head];
17        (i != q->first) || firsttime;
18        i = (i + 1) % REQ_QUEUE_SIZE, then = now, firsttime = 0) {
19
20     /* Compute time since previous access */
21     now = q->jiffies[i]; t = now - then;
22     awake_time = q->elapsed_time[i] + pd->wait;
23
24     /* Was predicted idleness longer than breakeven time? */
25     if ((predict < pd->tbe + q->elapsed_time[i])||(t < awake_time)){
26       estimate_for_state(est, hd, ACPI_STATE_D0, t);
27       goto next;
28     }
29
30     estimate_for_state_transition(est, hd, ACPI_STATE_D0,
31                                   ACPI_STATE_D1, awake_time);
32     estimate_for_state_transition(est, hd, ACPI_STATE_D1,
33                                   ACPI_STATE_D0, t - awake_time);
34 next:
35     pd->old_predict[i] = predict;  /* Prediction before access */
36     predict = compute_prediction(pd, predict, t);  /* New predict */
37   }
38
39   /* Energy since final access omitted for space */
40   return est;
41 }
```

Figure 10: Estimator code for EXP. Called by evaluator to compute the energy and delay for a given device.

```
 1 static int __init model_markov_init(void)
 2 {
 3   struct list_head *hds;
 4   struct happi_device *d;
 5   struct list_head *d_scan;
 6   unsigned long flags;
 7
 8   spin_lock_irqsave(&happi_mutex, flags);
 9
10   HAPPI_ID = happi_register_measurement("MARKOV", &return_measurement);
11   initialize_all_device_markov()
12
13   /*
14    * Markovian transition probability matrix is periodic, so it doesn't
15    * need to know about any device accesses or state transitions.  It
16    * will read these later when it wakes up.
17    */
18   hds = get_all_happi_devices();
19   list_for_each(d_scan, hds) {
20     d = list_entry(d_scan, struct happi_device, list);
21     happi_set_timeout(HAPPI_ID, RESTART_NOW, d, EVAL_PERIOD,
22                       &handle_periodic);
23   }
24
25   spin_unlock_irqrestore(&happi_mutex, flags);
26   return 0;               /* Loaded successfully */
27 }
28 module_init(model_markov_init);
```

Figure 11: Initialization code for state transition matrix measurement. Called when the measurement is inserted into the kernel.

```
static void handle_periodic(unsigned long hd_ul)
{
  struct happi_device * hd = (struct happi_device *) hd_ul;
  struct device_markov * d = get_device_markov(hd);

  int i;
  enum sr_states cstate = d->cstate;  /* Current state */
  enum sr_states nstate, oldstate;    /* Next/old state */

  nstate = did_access_occur(hd, jiffies - EVAL_PERIOD, jiffies);

  /* Subtract off the current window value from the transition matrix */
  i = d->ptr[cstate];
  oldstate = d->window[cstate][i];
  d->matrix[cstate][oldstate]--;

  /* Insert the new value into the window and increment pointer */
  d->window[cstate][i] = nstate;
  d->ptr[cstate] = (i + 1) % WINDOW_SIZE;

  /* Add the access to the transition matrix */
  d->matrix[cstate][nstate]++;

  /* Update the current state */
  d->cstate = nstate;
}
```

Figure 12: Measurement update code. Called periodically to update the measurement.

```
/* Callback provided by the measurement */
static void *return_measurement(struct happi_device * hd)
{
  return (void *) get_device_markov(hd);
}

/* Obtaining the measurement for the policy */
#include <markov.h>
static void * (*get_markov_model)(struct happi_device *);
get_markov_model = happi_request_measurement("MARKOV");
struct device_markov *md = (struct device_markov *) get_markov_model(hd);
```

Figure 13: Measurement return code. Called by a policy to obtain the measurement's data.