Monitoring system activity for OS-directed dynamic power management

Luca Benini* Alessandro Bogliolo Stefano Cavallucci Bruno Riccó

DEIS - Università di Bologna
Bologna, ITALY 40136

Abstract

[1] In this paper we describe a workload monitoring system that has been specifically designed for supporting dynamic power management in personal computers with tight power constraints (such as laptop or notebook computers). Our monitoring system is minimally intrusive, and has negligible impact on system activity. Moreover, it can be used both for on-line system monitoring and off-line data collection.

We used our monitoring tool to collect data on the usage of system resources (disks, CPU, keyboard and mouse) for a laptop computer, under several workload conditions. Our analysis shows that resource usage is strongly resource and workload dependent, and that on-line usage monitoring capability is a critical issue of the implementation of effective power management policies.

1 Introduction

Reducing the power dissipation of hardware components in portable computers is a primary design target. Numerous computer-aided design techniques for low power have been proposed [1], focusing on chip-level power optimization. Unfortunately, a portable computer is far more complex than a single chip and the complexity of such a system is well beyond the scope of any power-optimization tool. Nevertheless, the power consumption of portable computers has been kept under control, and battery lifetime for a given performance level is steadily improving.

This successful effort is rooted in technological innovations and designer's ingenuity. Low power system components (CPUs, hard disk drives, displays, etc.) are available thanks to improved integration and careful optimization. The first task of the system designer is to select components and organize the system architectures trying to achieve the best balance between cost, performance and power.

Component selection is just the first step toward a low-power design. Computers are flexible and are designed to effectively support widely varying workloads. Different system components may be stressed by different users (or by the same user at different times), and it is almost never the case that all resources in a laptop computer are operated at their maximum performance level at the same time. In other words, there is a large amount of idleness during operation. *Dynamic power management* techniques exploit idleness to reduce power dissipation [2].

The key idea in dynamic power management is that component can be switched into a low-power *sleep* state when they are not required for system operation (*i.e.*, they are idle). Several sleep states can be defined spanning the trade-off between power consumption and time required to return to a fully operational state. Similarly, several operational states can be defined spanning the trade-off between power and performance. A successful implementation of dynamic power management relies on two key assumptions: *i*) the availability of *power manageable* components (also called *resources*) that support multiple states of operation; *ii*) the existence of a *power manager* that drives component transitions with the correct protocol and implements a power management *policy* (an algorithm that decides when to shut down idle components).

The importance of power management in the design of successful computer architectures is well understood by industry leaders in the software and hardware areas. An *advanced configuration and power management interface* specification (ACPI), has been jointly developed by Intel, Microsoft and Toshiba [9]. The ACPI specifies the conventions and protocols for the communication between resources and power manager. Its main purpose is to ease the implementation of power-managed computers, both in terms of hardware and software development. One key assumption behind ACPI is that the power manager is a module of the operating systems (OS). Hence, OS-directed dynamic power management appears to be the frontier in system design for laptop computers.

ACPI provides an interface for the communication between hardware components and software power manager (in the OS), but it does not specify neither the power management policy nor the information that should be used to drive the policy's decisions. In this paper we present a tool that has been specifically developed for collecting such information and making it available to the system designer for analysis and to the power manager for use. Our tool is based on the same system model that led to the development of ACPI, but it is completely independent from it.

More specifically we implemented a system monitoring and instrumentation package based on the Linux [7] operat-

*Luca Benini is also with Hewlett Packard Laboratories and Stanford University

[1]We would like to thank Prof. Giovanni De Micheli, at Stanford, and Daniele Gordini, at DEIS, for many useful suggestions.

ing system for monitoring the usage of system components with high accuracy and low overhead. The package collects the information required to: *i*) design the power manager; *ii*) drive power management policies and *iii*) estimate the activity level of system components. While previously developed OS-based monitoring tools focused mainly on performance estimation, our package is tailored toward collecting and analyzing data that is relevant for dynamic power management.

The data collection tool is designed for on-line as well as off-line data analysis. In the first mode, it provides the data used by the power management policy to take decisions during system operation. In the second mode it can be used to study usage patterns and design effective power management policies. Particular care has been taken to minimize the perturbation of normal system activity caused by monitoring.

The paper is organized as follows. In the next section we review the basic concepts of OS-directed power management, and we propose a general model for formulating power management policies in a rigorous fashion. In Section 3 we describe the architecture of our instrumentation and analysis tool, while in Section 4 we discuss implementation details and provide experimental evidence and insights gained thanks to the data collected by our tool. Section 5 concludes the work.

2 OS-directed power management

The high-level architecture of the software interface between user applications and hardware can be summarized as follows. Applications communicate with *hardware abstractions* provided by the operating system. The operating system implements explicit communication to hardware devices through *device drivers.* Notice that some hardware components are not controlled through device drivers. For instance, the CPU and main memory are not visible as devices, but are implicitly controlled by the OS (both are needed for running the OS itself). Hardware components that perform very-high bandwidth communication with the CPU tend to be controlled bypassing device drivers for performance reasons.

The OS is most effective in power-managing components that are explicitly controlled through device drivers. Hardware components controlled through device drivers will be called *explicit.* Some of the components which are largest contributors to the power budget (CPU, main memory) are not abstracted as devices. We call such components *implicit.* Individually monitoring and managing implicit components is difficult and usually requires some form of hardware support. We take a conservative (albeit sub-optimal from the power viewpoint) approach: the only implicit device that is monitored is the CPU itself, while all explicit devices can be monitored.

2.1 Power management

As shown in Figure 1, a complete power manager consists of three conceptual blocks: a *policy*, a *controller* and an *observer.* The policy takes decisions on power management commands, that are actuated through the controller. The observer monitors the system activity and informs the policy of the activity level of the system. The focus of this work is on the design of the observer block.

Without observation, it is almost impossible to devise effective power management policies. The simplest policy is

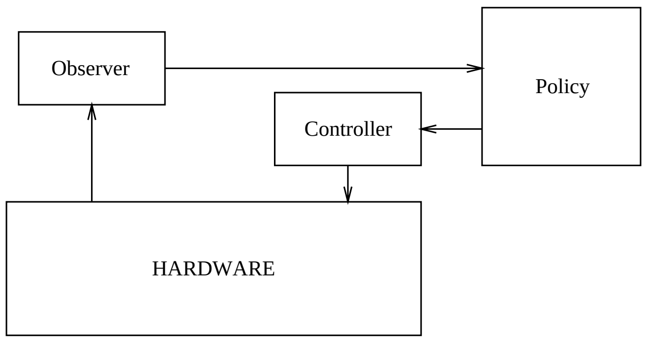


Figure 1: Components of a power manager: policy, observer and controller

greedy: devices are switched to a low-power state as soon as they are inactive. The greedy policy rises two major issues. First, if there are multiple sleep states, how to choose the most appropriate one. Second, and most important, since transition to sleep states and back to active have a power and performance cost, the power manager should guarantee that, in average, the state transitions do reduce power without compromising performance beyond an acceptable level. It is important to realize that the decision on when to transition to sleep state, and what state to choose critically depend on the operating conditions. For instance, if the usage pattern is bursty, it is highly convenient to never turn off a component during an usage burst, and to de-activate as soon as a period of inactivity is detected.

The task of choosing an optimal policy that minimizes power consumption under performance constraints (or vice-versa) based on usage patterns is called *policy optimization.* Recent studies [3, 4, 10, 11] have shown that policy optimization relies on the availability of observation data on the usage patterns of the system components. Policies such as the greedy one, that ignore observation data, are bound to be much less effective than policies that are customized on usage.

Furthermore, personal computers are highly flexible, general-purpose machines that can be used in a wide variety of ways. The load of system components changes over time, depending on which applications are running and what the user is doing with them. Hence, the most effective power management policies should be *adaptive.* Dynamic adaptation of a policy can be performed only if we assume the availability of an observation mechanism that collects data on usage patterns on-line, during normal system operation.

In the next section we will describe the basic requirement and the architecture of a system activity observer, that has been specifically designed for interaction with a policy within the operating system of a personal computer. The data can be just collected for off-line processing or analyzed and processed on-line.

3 System monitoring

The power management policy requires input information regarding the usage of each hardware resource. To be more specific, power management policies [3, 4, 10, 11] need information regarding: *i*) the distribution of inter-arrival times of request to the resources *ii*) the distribution of service times for the requests. Such data can be easily extracted from a stream of time-stamped events E_i that can be represented as a 3-tuple $E_i = (t_i, r_i, s_i)$, where t_i is the time-stamp, r_i is the resource identifier and s_i is the resource-dependent identifier of the type of event. Before describing the architecture of the observer, we analyze the basic requirements and constraints for its implementation. *Low perturbation of normal system activity.* This is the primary constraint. Monitoring should be transparent to the end user and should modify the usage patterns of hardware resources as little as possible. Notice that this requirement is subtly different from the one enforced for performance monitoring. When monitoring performance, it is important not to perturb the system when it is actively servicing requests, while periods of inactivity are of no interest and they can be reduced or changed by the monitoring tool. In contrast, we are interested in monitoring both the *busy* and the *idle* times.

Flexibility. It should be easy to monitor multiple types of resources, to select which resources to monitor and when, and to arbitrarily filter the stream of events. Moreover, the number and types of observed resources should be dynamically controllable. This feature is particularly useful for laptop computers where new devices can be installed during system operation (*i.e.*, plug-and-play capability).

Accuracy. Well-known system utilities give access to cumulative counts of accesses to system resources. This functionality is not sufficient to obtain accurate statistics of inter-arrival times and service times. One important feature of the observer is the capability of time-stamping the events with high resolution.

These requirements have been taken into account in our implementation, with particular emphasis on non-intrusiveness. Since for power management purposes both idle and active periods are of interest, the best time for executing monitoring activity is immediately before and after active periods. In this way, the duration of active periods is only slightly increased (if the monitoring activities have short execution time compared to the active periods) at the expense of a small decrease of the duration of idle periods.

3.1 Observer architecture

The observer is implemented as an extension of the Linux operating system [7]. The core data structure for storing the time-stamped events is stored in kernel memory space, that is linked to run in physical-address space. Hence, storing events in kernel space prevents the usage of memory paging, thus avoiding the severe performance penalty possibly caused by TLB misses.

On the other hand, storing the event list in kernel space imposes a tight limitation on its maximum size. In our implementation, the list cannot grow larger than 64KB, which corresponds to $L_{max} = 4096$ events. The event list is implemented as a circular buffer and it is allocated once for all (for performance reasons). The circular structure protects against memory violations. If the number of unprocessed events stored in the list grows larger than the number of slots, older events are overwritten. Event loss causes a decrease in accuracy in monitoring but does not damage normal system operation.

The size limitation of the event list in kernel memory is not a concern if events are processed and discarded as soon as they are registered (on-line monitoring). However, event loss should be avoided if the observer is collecting long event traces for off-line processing. The observer supports off-line monitoring through a simple dumping mechanism that can be summarized as follows. Whenever the number of unprocessed events reaches a value $L_{low} < L_{max}$, a wake-up signal is sent to a dedicated process. The process is

```
void NotifyEvent (event_list, resource_id,
              event_type, online_proc)
{
   time = GetTimeStamp();
   UpdateLast(event_list, time, resource_id,
              event_type);
   if (online_proc) {
      ProcessLastEvent(event_list);
   }
   else {
      if (NEvents(event_list) == $L_{low}$)
         SendSignal(dump_proc);
   }
}
```

Figure 2: Procedure that updates the event list

normally inactive, waiting for the wake-up signal, thus it does not alter normal system activity. Whenever the wake-up signal is asserted, the process becomes active and can be scheduled.

We decided not to modify the default scheduling algorithm of the Linux operating system, therefore, we cannot guarantee that the waiting process will be scheduled as soon as the signal is asserted. If the list-processing process is not scheduled right after signal assertion, new events can be generated before the event list is examined. L_{low} is set to a smaller value than L_{max} to minimize the probability of event loss in the interval between the assertion of the wake-up signal and the scheduling of the process that empties the list. Clearly, the execution of this process does alter normal system activity. However, the perturbation is limited by the fact that the list is processed only when it is almost full.

The high-level pseudo-code of the list-update procedure, `NotifyEvent`, is shown in Figure 2. The parameters are the event list (implemented as a circular queue), the unique resource identifier, the event type and a Boolean flag, `online_proc`, that has value `true` when on-line processing of the events is performed. Upon entering `NotifyEvent`, the current time is recorded by procedure `GetTimeStamp` (to be analyzed later) and the new event is stored in the circular queue. Then, if on-line processing is active, the newly stored event is processed and "consumed" by function `ProcessLastEvent`. If off-line processing is selected, the number of events in the list is checked against L_{low}. If there are L_{low} events in the list, the wakeup signal is sent to the process `dump_proc`. The process becomes active and will be scheduled. Notice that the signal is issued only when the number of events in the list is *equal* to L_{low}. This avoids the issue of multiple signals if `dump_proc` is not scheduled right away.

Function `GetTimeStamp` exploits a special feature of the X86 architecture (which, needless to say, is de-facto standard for personal computers). The function accesses a 64 bit register containing the number of clock cycles from the activation of the CPU. The time resolution of the time-stamp obtained by `GetTimeStamp` is equal to the clock cycle (e.g., 10ns for a 100MHz Pentium processor), which is more than sufficient for the typical time granularity of power-management transitions (microseconds in the fastest case). Counter count saturation is not an issue because the counter would take more than 500 years to saturate the count for a processor with 1GHz clock frequency.

The on-line event processing function `ProcessLastEvent` and the process for examining and freeing the event list off-line can be freely customized by the system designer. In the next section we describe an example of usage of the monitor for off-line analysis. In this case, the only function of dump_proc is to append the time-stamped events at the end of a file.

3.2 Monitoring explicit resources

Explicit resources are managed by the OS through device drivers. Our monitoring approach requires a simple modification of the device drivers of the resources under observation. The modification is simply the insertion of a call to NotifyEvent immediately before the driver sends execution commands to the hardware and immediately after the driver becomes ready to communicate to the kernel that the hardware has completed the required service. The insertion points of the calls to NotifyEvent strongly depend on the resource class being monitored and on the interpretation given to service initiation and completion. In particular, since we are interested in monitoring the actual usage of resources, we need to distinguish between *char devices* and *block devices*. Char devices are blocking: when the driver of a char device is serving a request, it cannot accept any other request. In contrast, requests to block devices are enqueued and possibly reordered before being served. As a consequence, char devices can be monitored at a higher abstraction level (thanks to the direct mapping between a request and the corresponding service) while block devices need to be monitored at physical level.

In the serial-port driver, calls to NotifyEvent are located at the beginning and at the end of the receive_char and transmit_char routines, that are directly called by the kernel. In the *IDE*-disk driver, event notification is performed by the handlers that physically manage data transfers. Before being translated into a call to the handler, a high-level read/write command is processed by a hierarchy of routines implementing the disk-management policy. Monitoring high-level read/write commands wouldn't provide direct information about the actual workload of the disk.

In any case, monitoring does not change the flow of execution of the device driver, and it has minimal impact on the execution time, because the execution of NotifyEvent just requires the update of a few memory locations and the access to the cycle counter for obtaining the time-stamp. Notice also that NotifyEvent is called only when necessary and only for resources that are actively monitored. At boot time, the observer is initialized by specifying which resources should be monitored.

3.3 Monitoring the CPU

The CPU and all hardware components required for its operation (chipset, RAM, bus controllers, etc.) are not controlled through device drivers. Fortunately, it is possible to monitor the CPU and its ancillary components by observing that OS kernel itself is nothing else than executable code running on the CPU. Hence, whenever the kernel is running, the CPU is active.

There is a simple way to detect when the kernel is not running and no active process needs the CPU. In the Linux scheduler, a process is defined, called *idle process* that has lowest execution priority, thus it is scheduled when neither the kernel nor the user processes need to execute. Detecting the scheduling of the idle process is a simple way to monitor activity and idleness of the CPU.

A call to NotifyEvent is inserted immediately after the priority computation of the scheduler returns the idle process as the highest priority process, and immediately before the exit from the idle process. The CPU idle is the time spent in the idle process.

The CPU idle state is nothing but a loop of hlt instructions (*i.e.*, the idle task), periodically interrupted by a timer interrupt that activates the scheduler to recompute priorities. We remark that priority computation does not require exiting from the idle state, since the scheduler always executes in the context of the last active process. As long as the idle task is re-scheduled as the highest priority process, no context switch is performed and no CPU activity is detected.

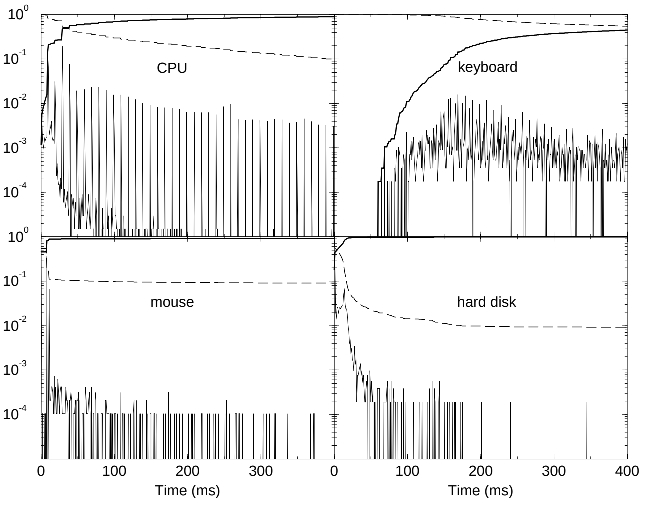


Figure 3: Statistical analysis of the inter-arrival time. For each device, three curves are plotted in lin-log scale: the probability density (solid line), the probability distribution (bold line) and its complement to 1 (dashed line). Data refer to software development.

4 Implementation and results

The monitoring system described in the paper has been implemented as an add-on package (called ipm) for the RedHat-4.1 distribution of Linux 2.0.29. In the current ipm release, monitoring support is available for CPU, keyboard, serial and parallel ports, PS2 mouse, IDE hard disk and CD-ROM. We tested our package on a HP Omnibook 5500CT with 133MHz Pentium processor and 48MB of RAM. To evaluate the impact of system monitoring on execution time we used the Linux kernel re-compilation as a CPU and disk intensive benchmark. Re-compilation was first performed without monitoring and then repeated while collecting event traces for CPU only, Hard Disk only, CPU and Hard Disk, entire system. Each experiment was repeated 10 times and the average CPU usage was taken as a significant measure of the actual performance (the maximum deviation being less than 0.01%). The system was re-booted after each run to empty the cache. The measured monitoring overheads were of 0.13% for the CPU, 0.14% for the hard disk, 0.30% for CPU and disk, and 0.38% for the entire system. Notice that the overhead of concurrent CPU and disk monitoring is slightly larger than the sum of the overhead of the separate monitoring of the two resources. This is because of the larger number of disk events generated by the dump_proc routine. On-line system monitoring wouldn't cause this additional overhead.

A second set of experiments were performed to show how real-world data provided by `ipm` can be used to steer power management. Full-system monitoring was enabled and event traces were collected under typical operating conditions corresponding to different workloads: editing (*Editing*), file-system browsing (*FSBrowsing*), software development (*SoftDev*) and graphical interactive games (*Games*). The maximum size of a one-hour full-system event stream was of about 3MB.

Traces were studied off-line to extract significant statistical properties. Four random variables were characterized for each resource: inter-arrival time (IT), service time (ST), number of service requests per time slice (SR) and percentage busy time per time slice (BT). Probability densities and distributions of ITs are plotted in Fig. 3 for CPU, keyboard, mouse and hard-disk. Though all data refer to the same session of system development, the four devices have completely different statistics. The probability density of the CPU presents regular peaks at multiples of 10ms, that is the period of the timer that interrupts the idle task to re-compute priorities. Since decisions are usually taken upon priority computation, most of the service requests are processed at multiples of 10ms. The time-continuous behavior of the first 200ms is due to the effect of interrupt signals that may force the CPU to awake at any time. Notice that the probability of the CPU staying idle for more that 1s is almost negligible.

The statistical distribution of keyboard events differs from that of other devices for two main reasons: *i*) it doesn't start from zero (the minimum IT time depends on the user typing speed, that is much slower than 1ms) and *ii*) it doesn't decay significantly in the first second. In contrast, the time between most mouse's events is around 1ms. This is due to the rate at which events are generated when the mouse is moved. This rate does not depend on the user. Finally, the IT of hard disk services is mainly concentrated below 100ms, and only once every 100 requests it is above 0.5s.

In the following we refer to a simple example to demonstrate how off-line processing of event traces can be used to find optimal policies for power management. We refer to an ideal power manageable hard disk with a unique *active* state and a unique *sleep* state. In the sleep state, not only the electronic components of the drive are turned off, but the disk is also spun down, thus making power consumption almost zero. On the other hand, transitions between sleep and active states are slow (mainly because of mechanical inertia) and cause additional power consumption (the additional current absorbed by the motor to accelerate the disk). Though in principle the disk can be put in sleep mode whenever it goes idle, in practice this is a good choice only when the idle period is *long enough* to compensate the additional spin-up power consumption and the overall performance penalty. Once power consumptions and transition times are known, it is possible to evaluate the *minimum idle time* (MIT) that makes the sleep state convenient. We assume this time to be MIT=100*sec.*

Since service times (ST) are always negligible if compared with the idle times of interest, IT provides a good measure of the idle time. From Fig. 3 we read that the probability of having idle periods longer than MIT is well below 1%. This percentage, however, doesn't tell us how much power we can save by switching the disk to the sleep state whenever IT is greater than MIT. This is shown in Fig. 4.a. For each value of MIT, the solid curve reports the overall time (per hour) spent by the disk in idle periods longer than MIT when the user is doing software development (*i.e.*, alternating editing, compilation and execution tasks). The starting point of the curve (corresponding to MIT=0) represents the total idle time of the disk, that is more than 99% of the reference hour. The point corresponding to MIT=100*sec.* represents the total amount of time in which we could save power by switching the disk to the sleep state. Fig. 4.b shows that this figure is strongly dependent on the operating conditions.

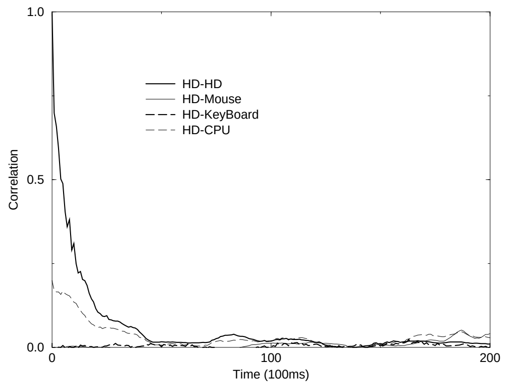
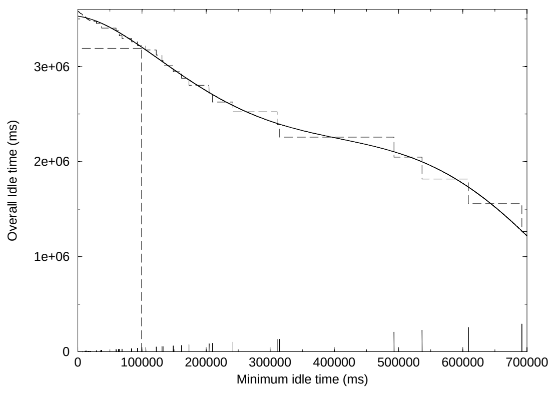


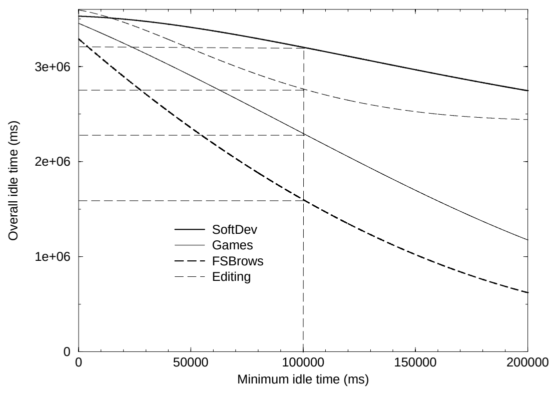
Figure 5: Auto and cross correlation of random variables SR, representing the number of service requests received by a resource in a time slice of 100ms.

The graphs of Fig. 4 refer to the ideal situation of a power manager that knows in advance the length of each idle period and decides consequently whether to switch to the sleep state or stay active. Unfortunately, inter-arrival times are random variables. When a resource goes idle we don't know when the next service request will be issued. To actually implement a power manager that takes advantage of idle periods longer than MIT, we need a statistical criterion to recognize those periods as soon as possible by predicting their length. Since the only information available to make a prediction is the history of the system, we study the correlation among the event of interest (*i.e.*, the time of the next request) and all previously collected data. Discrete-time random variables SR and BT (with discretization time step of 100ms) were used for correlation analysis. Auto and cross correlations of random variables SR are plotted in Fig. 5 as a function of time. We remark that all correlations are greater than zero. This means that the activity of any part of the system usually increases the probability of a request to be sent to the disk. However, cross correlations are much lower than the autocorrelation of subsequent disk events. As a consequence, the history of the disk is the best candidate to steer the prediction of the arrival time of the next disk event.

Policy optimization for power management is a complex task that is far beyond the scope of this paper. Here we refer to a time-out mechanism that is the simplest implementation of a power manager based on the history of the resource to be managed. A time-out counter is re-started each time the disk goes idle. The disk is switched to the sleep state if it is still idle when the time-out is reached. The assumption behind this policy is that there is a good correlation between the event A="the disk has been idle for time-out seconds" and the event B="the disk will be idle for at least MIT more seconds". The conditional probability of event B given event A is plotted in Fig. 6 as a function of the time-out period (assuming MIT = 100s). The time-out



a)



b)

Figure 4: Ideal sleep time (per hour) of the example hard disk as a function of the minimum idle time required to save power by switching the resource off and on. Data refer to a) software development and b) four different workloads.

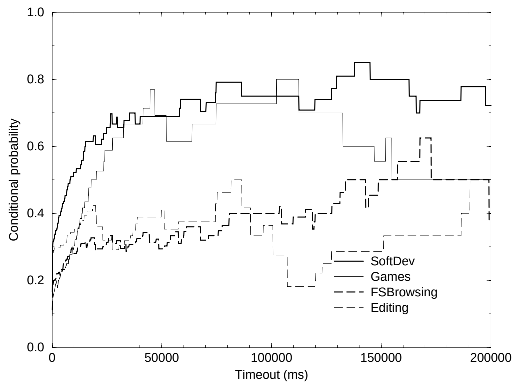


Figure 6: Conditional probability of the disk staying idle for at least 100 more seconds as a function of the elapsed idle time.

has to be chosen in order to maximize this probability. Interestingly, the conditional probability strongly depends on the operating Conditions: a time-out of 100s leads to a 80% of good predictions if the user is developing software, while it gives almost no information (the conditional probability being less than 20%) if user is editing a text document.

The results of our analyses can be summarized as follows. First, under the usage models we experimented, all system resources are idle for a significant fraction of the operation time. Thus, power management techniques that minimize the power dissipation of idle components show good promise. Second, the duration of idle periods, although nondeterministic, can be estimated with good accuracy, given the availability of tools for collecting a significant amount of past history. Third, the distribution of usage and idle periods strongly depends on usage models. In other words, the workload for all system resources is strongly non-stationary. Hence, on-line monitoring is key for implementing effective power-management policies. Such policies should be able to adapt to non-stationary operating conditions. Finally, although some cross correlation between usage of different resource does exist, it is not a dominant factor with respect to auto correlation. Consequently, the single-resource, adaptive policies based on past system history, seem to be a good choice for the practical implementation of dynamic power management.

5 Conclusion

In this paper we presented a workload monitoring tool specifically designed for support of aggressive power management strategies for laptop computers, and more generally for any power-sensitive system whose resources are managed by an operating system. Our implementation is based on the Linux operating system and has been tested on machines based on the Intel X86 architecture. A low-overhead system activity monitoring tool is a key component of any OS-based dynamic power management architecture, because it provides the input data required by the power-management policy to take optimal decision on when and how to switch off idle resources. We performed extensive experimentation and monitoring of system activity with our tool and we presented a detailed statistical analysis of resource usage. In particular we addressed the key issue of predicting the duration of idle periods.

References

[1] W. Nebel and J. Mermet (Eds.), *Low power design in deep submicron electronics.* Kluwer (1997).

[2] L. Benini and G. De Micheli. *Dynamic Power Management: design techniques and CAD tools.* Kluwer (1997).

[3] M. Srivastava, A. Chandrakasan and R. Brodersen, "Predictive system shutdown and other architectural techniques or energy efficient programmable computation," *IEEE Transactions on Very Large Scale Integration Systems* vol. 4, no. 1, pp. 42-55, March 1996.

[4] R. Golding, P. Bosh et al, "Idleness is not sloth," in *Proceedings of Winter USENIX Technical Conference*, pp.201–212 (1995).

[5] Derman, C. and Veinott A. Jr. Constrained Dynamic Programming. Management Science (1970).

[6] L. Kleinrock. *Queuing Systems* Voll. I,II. Wiley, 1976.

[7] L. Torvalds, "Linux kernel implementation," *Proceedings of Open Systems. Looking into the Future. AUUG '94*, pp. 9–14, Sept. 1994.

[8] Microsoft, "OnNow: the evolution of the PC platform," *http://www.microsoft.com/hwdev/ONNOW.HTM* 1997.

[9] Intel, Microsoft and Toshiba, "Advanced Configuration and Power Interface", *http://www.teleport.com/ acpi/* 1996.

[10] C. Hwang et al., "A predictive system shutdown method for energy saving of event-driven computation," in *Proceedings of IEEE International Conference on Computer Aided Design*, pp. 28-32, Nov. 1997.

[11] G. Paleologo, L. Benini, A. Bogliolo and G. De Micheli, "Policy optimization for dynamic power management" to appear in *Proceedings of Design Automation Conference*, June 1998.