A Dynamic Disk Spin-down Technique for Mobile Computing

David P. Helmbold, Darrell D. E. Long and Bruce Sherrod[†]
Department of Computer Science
University of California, Santa Cruz

Abstract

We address the problem of deciding when to spin down the disk of a mobile computer in order to extend battery life. Since one of the most critical resources in mobile computing environments is battery life, good energy conservation methods can dramatically increase the utility of mobile systems. We use a simple and efficient algorithm based on machine learning techniques that has excellent performance in practice. Our experimental results are based on traces collected from HP C2474s disks. Using this data, the algorithm outperforms several algorithms that are theoretically optimal in under various worst-case assumptions, as well as the best fixed time-out strategy. In particular, the algorithm reduces the power consumption of the disk to about half (depending on the disk's properties) of the energy consumed by a one minute fixed time-out. Since the algorithm adapts to usage patterns, it uses as little as 88% of the energy consumed by the best fixed time-out computed in retrospect.

1 Introduction

Since one of the main limitations on mobile computing is battery life, minimizing energy consumption is essential for maximizing the utility of wireless computing systems. Adaptive energy conservation algorithms can extend the battery life of portable computers by powering down devices when they are not needed. The disk drive, the wireless network interface, the display, and other elements of the mobile computer can be turned off or placed in low power modes to conserve energy. Several researchers have even considered dynamically changing the speed of the CPU in order to conserve energy [7, 22]. We show that a simple algorithm for deciding when to power down the disk drive is even more effective in reducing the energy consumed by the disk than the best fixed time-out value computed in retrospect.

Douglis *et al.* [5] show that the disk sub-system on portable computers consumes a major portion of the available energy (Greenawalt [8] states 30% or more). It is well-known that spinning the disk down when it is not in use can save energy [5, 6, 15, 23]. Since spinning the disk back up consumes a significant amount of energy, spinning the disk down immediately after each access is likely to use more energy than is saved. An intelligent strategy for deciding when to spin down the disk is needed to maximize the energy savings.

Current mobile computer systems use a fixed time-out policy. A timer is set when the disk becomes idle and if the disk remains idle until the timer expires then the disk is spun down. This time-out can be set by the user, and typical values range from 30 seconds up to 15 minutes. Douglis *et al.* [5], Golding *et al.* [6], and other researchers [8, 14, 15] have proposed algorithms which spin the disk down more aggressively, conserving much more power than these relatively long time-outs.

We use a simple algorithm called the *share algorithm*, a machine learning technique developed by Herbster and Warmuth [10], to determine when to spin the disk down. Our implementation of the share algorithm dynamically chooses a time-out value as a function of the (recent) disk activity. Since the algorithm adapts to the recent disk access patterns, it is able to exploit the bursty nature of disk activity.

We show that the share algorithm reduces the power consumption of the disk to about *one half* (from 16% to 82% depending on the disk's physical characteristics, 54% on average) of the energy consumed by a one

[†]This research was supported by the Office of Naval Research under Grant N00014-92-J-1807 and by research funds from the University of California, Santa Cruz.

minute fixed time-out. In other words, the simulations indicate that battery life is extended by more than 17% when the share algorithm is used instead of a one minute fixed time-out.[1] A more dramatic comparison can be made by examining the energy wasted (*i.e.* the energy consumed minus the minimum energy required to service the disk accesses) by different spin-down policies. The share algorithm wastes only about 26% of the energy wasted by a one minute fixed time-out.

As noted above, large fixed time-outs are poor spin-down strategies. One can compute (in retrospect) the best fixed time-out value for the actual sequence of accesses that occurred and then calculate how much energy is consumed when this best fixed time-out is used. On the trace data we analyzed, the share algorithm performs better than even this best fixed time-out; on average it wastes only about three-quarters (77.3%) of the energy wasted by the best fixed time-out.

The share algorithm is efficient and simple to implement. It takes constant space and constant time per trial. For the results presented in this article, the total space required by the algorithm is never more than about 2400 bytes. Our implementation in **C** takes only about 100 lines of code. This algorithm could be implemented on a disk controller, in the BIOS, or as part of the operating system.

Although we have concentrated primarily on traces of disk activity, the principles behind the share algorithm can be applied to the other devices in a mobile computing environment. In particular, the algorithm could be applied to the wireless interface, the display, or other peripherals. Since the algorithm exploits the bursty nature of device usage, it can be used to conserve energy on any device with irregular use patterns.

The rest of this article proceeds as follows. Section 2 contains a brief survey of related work. We formalize the problem and define our performance metrics in §3. Section 4 describes the share algorithm. We present our empirical results in §5, future work in §6, and conclusions in §7.

2 Related Research

One of the simplest algorithms for disk spin-down is to pick one fixed time-out value and spin-down after the disk has remained idle for that period. Most, if not all, current mobile computers use this method with a large fixed time-out, such as several minutes [5].[2] It has been shown that energy consumption can be improved dramatically by picking a shorter fixed time-out, such as just a few seconds [5, 6]. For any particular sequence of idle times, the ***best fixed time-out***[3] is the fixed time-out that causes the least amount of energy to be consumed over the sequence of idle times. Note that the best fixed time-out depends on the particular sequence of idle times, and this information about the future is unavailable to the spin-down algorithm *a priori*.

We can compare the energy used by algorithms to the minimum amount of energy required to service a sequence of disk accesses. This minimum amount of energy is used by the *optimal algorithm* which "peeks" into the future before deciding what to do after each disk access. If the disk will remain idle for only a short time, then the optimal algorithm keeps the disk spinning. If the disk will be idle for a long time, so that if the energy used to spin the disk back up is less than the energy needed to keep the disk spinning, then the optimal algorithm immediately spins down the disk. In a sense the optimal algorithm uses a long time-out when the idle time will be short, and uses a time-out of zero when the idle time will be long.

Although both the optimal algorithm and the best fixed time-out use information about the future, they use it in different ways. The optimal algorithm adapts its strategy to each individual idle time, and uses the minimum possible energy to service the sequence of disk requests. The best fixed time-out is non-adaptive, using the same strategy for every idle time. In particular, the best fixed time-out always waits some amount of time before spinning down the disk, and so uses more energy than the optimal algorithm. Since both the best fixed time-out and the optimal algorithm require knowledge about the future, it is impossible to implement them in any real system. Even so, they provide very useful measures with which we can compare the performance of other algorithms.

A critical value of the idle period is when the energy cost of keeping the disk spinning equals the energy needed to spin the disk down and then spin it back up. If the idle time is exactly this critical value then the optimal algorithm can either immediately spin-down the disk or keep it spinning since both actions incur the same energy cost. Thus the energy cost to spin-down and then spin-up the disk can be measured in terms of the number of seconds that that amount of energy

[1]We assume that the disk with a one minute time-out uses 30% of the energy consumed by the entire system. Thus, if t is the time the system can operate on a single battery charge with 1 minute time-outs, and t' is the time the system can operate using the share algorithm, we have $t = 0.7t' + 0.15t'$. So $t' > 1.176t$ and the battery life is extended by more than 17%.

[2]For example, on a contemporary Macintosh 520C laptop computer, the user can set the time-out to 30 seconds or any whole number of minutes from 1 to 15.

[3]There could be several equally good time-outs, in that case we define the best fixed time-out to be the smallest of these equally good time-out times.

would keep the disk spinning. We call this number of the seconds the *spin-down cost* for the disk drive. Obviously, the spin-down cost can be different for different makes and models of disk drives.

One natural algorithm uses a fixed time-out equal to the spin-down cost of the disk drive. If the actual idle time is shorter than the spin-down cost, then this algorithm keeps the disk spinning and uses the same amount of energy as the optimal algorithm on that idle period. If the length of the idle time is longer than the time-out, then the algorithm will wait until the time-out expires and then spin down the disk. This uses exactly twice the spin-down cost in total energy (to keep it spinning before the spin-down, and then to spin it back up again). This is also twice the energy used by the optimal algorithm on that idle period, since the optimal algorithm would have immediately spun down the disk. For larger idle periods, this algorithm never consumes more than twice the energy used by the optimal algorithm.

An algorithm is called *c-competitive* or has a *competitive ratio* of c if it never uses more then c times the energy used by the optimal algorithm [20, 11]. So this natural algorithm is 2-competitive, and we will refer to it as the *2-competitive algorithm.* It is easy to see that the 2-competitive algorithm has the best competitive ratio of all constant time-out algorithms.[4] As we will see in §5, even this very simple algorithm uses much less energy than the large fixed time-outs currently in use.

Since the 2-competitive algorithm uses a fixed time-out, its performance never surpasses the best fixed time-out. In fact, the best fixed time-out for a sequence of idle times usually uses much less energy than the 2-competitive algorithm. Since the 2-competitive algorithm uses *one* predetermined time-out, it is guaranteed to be reasonably good for *all* sequences of idle times, while the best fixed time-out depends on the particular sequence of idle times.

Randomized algorithms can be viewed as selecting time-out values from some distribution, and can have smaller (expected) competitive ratios. Although we still compute the competitive ratio based on a worst-case idle time between accesses, we average the energy used over the algorithm's random choice of time-out. Karlin *et al.* [12] give an (expected) $(\frac{e}{e-1})$-competitive randomized algorithm. If the spin-down cost is s, their algorithm chooses a time-out at random from $[0, s]$ according to the density function

$$p(\text{time-out} = x) = \frac{e^{x/s}}{e-1} .$$

[4]Any algorithm that uses a larger time-out, say $(1 + \Delta)s$, for a spin-down cost s, is only $2 + \Delta$-competitive when the idle times are large; an algorithm that uses time-out smaller than s is less than 2-competitive when the idle time is between its time-out and s.

They show that this $(\frac{e}{e-1})$-competitive randomized algorithm is optimal in the following sense: every other distribution of time-outs has an idle time for which the distribution's (expected) competitive ratio is larger than $\frac{e}{e-1}$.

Both the 2-competitive algorithm and the $(\frac{e}{e-1})$-competitive randomized algorithm are worst case algorithms; that is, these algorithms perform competitively even if the idle times are drawn adversarially. Without making some assumptions about the nature of the data, and thus abandoning this worst case setting, it is difficult to improve on these results.

One can get better bounds by assuming that the idle times are drawn independently from some fixed (but unknown) probability distribution instead of chosen adversarially. With this assumption, the best fixed time-out on the past idle times should closely approximate the best fixed time-out for the future idle times. Krishnan *et al.* [14] introduce an algorithm designed to operate under this assumption. Their basic algorithm operates in two phases. In the first phase it predicts arbitrarily while building a set of candidate time-outs from the idle times. After obtaining enough candidate time-outs, the algorithm then tracks the energy used by the candidates and chooses the best candidate as its time-out. Their full algorithm repeatedly restarts this basic algorithm with each restart using more candidates and running for exponentially increasing amounts of time. When processing the t^{th} idle period the full algorithm tracks order $\sqrt{t}$ candidates and takes order $\sqrt{t}$ time to update its data structure. They essentially prove that the energy used by this full algorithm *per disk access* quickly approaches that of the best fixed time-out when the idle times are independent and identically distributed. However, their algorithms may perform very poorly while collecting the candidate time-outs and are only estimating the best fixed time-out. This makes it very unlikely that these algorithms will use less energy than the best fixed time-out.

If the fixed distribution on idle times is known in advance, then one can analyze the distribution in order to choose a good fixed time-out time. Greenawalt [8] takes this approach using a Poisson distribution to model the disk drive idle times. He uses this rather strong assumption to solve for the single time-out value giving the best expected performance (over the random choice of idle times). Note that even if a sequence of idle times is drawn from the assumed distribution, the best fixed time-out for that particular sequence will be at least as good as the time-out value computed from the distribution.

Upon examining disk traces, we observe that idle times are not drawn according to some simple fixed distribution. So, in contrast to the above algorithms, we assume that the data is time dependent, having both extended busy and idle periods. We use a simple adaptive algorithm that exploits these different periods by shifting its time-out after each trial. The share algorithm uses constant time and space, and is simple to implement. In trace-driven simulations, it performs better than all of the algorithms described above, and even conserves more energy than the best fixed time-out.

Douglis *et al.* [4] have recently studied some incrementally adaptive disk spin-down policies. The policies they consider maintains a changing time-out value. Whenever the disk access pattern indicates that the current time-out value may be either too long or too short, the current time-out is modified by an additive or multiplicative factor. Whereas we concentrate on the energy used by the spin-down algorithm, Douglis *et al.* pay particular attention to those spin-downs likely to inconvenience the user and analyze the tradeoff between energy consumed and these undesirable spin-downs. In Golding *et al.* [6], similar incrementally adaptive policies are evaluated. We compare the performances of the share algorithm and these incrementally adaptive policies in §5.4.

3 Problem Description

We define the disk spin-down problem as follows. A disk spin-down algorithm continually decides whether or not to spin down the disk drive based on the patterns of previous usage. Alternately, we can view the disk spin-down algorithm as suggesting, after each disk access, a delay or *time-out* indicating how long an idle disk is kept powered up before spinning it down. We can then treat the problem as a sequence of trials, where each trial represents the idle time between two consecutive accesses to the disk. The disk is spun down if and only if it remains idle for at least the number of seconds equal the time-out. These two views of the disk spin-down problem are equivalent when the algorithm is allowed to choose a different time-out for each trial.

We measure the performance of the algorithms in terms of "seconds of energy" used, a measure introduced by Douglis *et al.* [5]. One "second of energy" is the difference in energy consumed between a spinning disk and a spun down disk over one second. One second of energy corresponds to some number of joules, depending on the model of disk drive used. Using seconds of energy allows us to discuss disk drives in general while avoiding a joules/second conversion factor.

We use the term "spin-down cost" to refer to the total cost of choosing to spin down the disk. This cost equals the energy required to spin the disk down (if any), plus the energy needed to spin the disk back up. We measure the spin-down cost in terms of the amount of time the disk would have to spin to consume the same amount of energy; in other words, a spin-down cost of s means that spinning the disk down and starting it up again consumes as much energy as keeping the disk spinning for s seconds. If we assume that a mobile computer user's disk usage is independent of the type of disk that they have, then this single parameter, the *spin-down cost* s, is the only statistic about the physical disk that we need for our simulations. Douglis *et al.* [5] compute this value for two disks, giving spin-down costs of 5 seconds and 14.9 seconds. Golding *et al.* [6] gives disk statistics that correspond to a spin-down cost of 9 or 10 seconds. We analyze the share algorithm's performance for spin-down costs varying from 1 to 20 seconds.

We define the following metrics, summarized in Table 1, to measure and compare the performance of algorithms. For each trial, the *energy* used by an algorithm is the total amount of energy, measured in seconds, that the algorithm uses. The energy use of an algorithm on a given trial depends on whether or not the algorithm spins down the disk: if the time-out is less than the idle time then the algorithm spins down the disk and uses a fixed amount of energy (that required to keep the disk spinning until the time-out plus that required to spin the disk down and then back up) regardless of the length of time the disk is idle; if the time-out is larger than the idle time then the algorithm keeps the disk spinning and uses energy proportional to the length of the idle time. The *excess energy* used by the algorithm is the amount of additional energy used by the algorithm over the optimal algorithm. We find it convenient to scale the excess energy, and denote this scaled excess energy for a time-out x as *Loss*(x).

4 Algorithm Description

The share algorithm is a member of the multiplicative-weight algorithmic family that has been developed by the computational learning theory community. This family has a long history and excellent performance for a wide variety of on-line problems [3, 13, 18, 9, 16, 17, 21]. Algorithms in this family receive as input a set of "experts," other algorithms which make predictions. On each trial, each expert makes a prediction. The goal of the algorithm is to combine the predictions of the experts in a way that minimizes the total error, or loss, over the sequence. Algorithms typically keep one weight per expert, representing the quality of that expert's predictions, and predict with a weighted average of the experts' predictions.

After each trial the weights of the experts are up-

$$\text{Energy used by time-out} = \begin{cases} \text{idle time} & \text{if idle time} < \text{time-out (don't spin down)} \\ \text{time-out} + \text{spin-down cost} & \text{if idle time} > \text{time-out (spin down after time-out)} \end{cases}$$

$$\text{Energy used by optimal} = \begin{cases} \text{idle time} & \text{if idle time} < \text{spin-down cost (don't spin down)} \\ \text{spin-down cost} & \text{if idle time} > \text{spin-down cost (spin down immediately)} \end{cases}$$

$$\text{Excess energy} = \text{Energy used by time-out} - \text{Energy used by optimal}$$

$$\mathit{Loss} = \frac{\text{Excess energy}}{\text{idle time}}$$

Table 1: Energy and loss statistics during each trial.

dated: the weights of misleading experts are reduced (multiplied by some small factor), while the weights of good experts are usually not changed. The more misleading the expert the more drastically the expert's weight is slashed. This method causes the predictions of the algorithm to quickly converge to the those of the best expert.

Herbster and Warmuth have recently developed the method of a "sharing update" [10]. Briefly stated, this update takes some of the weight of each misleading expert and "shares" it among the other experts. Thus an expert whose weight was severely slashed, but is now predicting well, can regain its influence on the algorithm's predictions. In essence, the algorithm pays the most attention to those experts performing well during the recent past. This adaptability allows us to exploit the bursty nature of disk accesses, and perform better than the best fixed time-out.

For the disk spin-down problem, we interpret each expert as a different fixed time-out, although we could substitute an arbitrary algorithm for each expert. In our experiments we used 100 evenly spaced time-outs between zero and the disk's spin-down cost. Although it is easy to construct traces where the best fixed time-out is larger than the spin-down cost, this does not seem to happen in practice. Note that the 2-competitive algorithm and the randomized $(\frac{e}{e-1})$-competitive algorithms also choose time-outs between zero and the spin-down cost. Reducing the space between experts tends to improve the algorithm's performance. On the other hand, the running time and memory requirements of the algorithm increase as additional experts are added.

Let us denote the predictions of the experts as x_1 to x_n (since each expert predicts with a fixed time-out, these predictions do not change with time). The current weights of the experts are denoted by w_1 to w_n, and these weights are initially set to $\frac{1}{n}$. We use $Loss(x_i)$ to denote the loss of expert i on a given trial (the loss is the normalized excess energy described in §3).

The share algorithm uses two additional parameters. The learning rate, η, is a real number greater than one and controls how rapidly the weights of misleading experts are slashed. The share parameter, α, is a real number between zero and one, and governs how rapidly a poorly predicting expert's weight recovers when that expert begins predicting well. Although these parameters must be chosen carefully to prove good worst-case bounds on the learning algorithm, the real-world performance of multiplicative weight algorithms appears less sensitive to the choice of parameters (for another example see Blum [2] on predicting calendar events). In our experiments, akin to a train and test regimen, we used a small portion of the data (the first day of one trace) to find a good setting for η and α, and then use those settings on the rest of the data (the remaining 62 days) to collect our results. We chose $\eta = 4.0$ and $\alpha = 0.08$. Small perturbations in these parameters have little effect on our results. The performance of the algorithm changes by less than a factor of 0.0013 as α varies in the range 0.05 to 0.1. Similarly, different values of η between 3.5 to 4.5 cause the algorithm's performance to change by at most a factor of 0.0018. We intend to explore methods for self-tuning these parameters in the future.

We can now precisely state the algorithm we use: Herbster and Warmuth's [10] variable-share algorithm. On each trial the algorithm:

1. Uses a time-out equal to the weighted average of the experts:

$$\text{time-out} = \frac{\sum_{i=0}^{n} w_i x_i}{\sum_{i=0}^{n} w_i},$$

2. Slashes the weights of poorly performing experts

$$w'_i = w_i e^{-\eta Loss(x_i)},$$

3. Shares some of the remaining weights

$$\mathit{pool} = \sum_{i=1}^{n} w'_i (1 - (1-\alpha)^{Loss(x_i)})$$

$$w_i'' = (1-\alpha)^{Loss(x_i)} w_i' + \frac{1}{n} pool .$$

The new w_i'' weights are used in the next trial.

Herbster and Warmuth [10] show that if the loss function meets certain properties then this algorithm has very good performance, even in the worst case. Although the loss function we use does not have the properties required for their proof, we show in the next section that its empirical performance is excellent.

The algorithm runs in constant time and space where the constants depend linearly on n, the number of experts chosen by the implementor. However, the algorithm as stated above has the drawback that the weights continually shrink towards zero. Our implementation avoids underflow problems by bounding the ratio between weights and periodically rescaling the weights.

5 Experimental Results

In this section we present trace-driven simulation results showing that our implementation of the share algorithm outperforms the best strategies currently available. We use the simulation results to compare the energy used by our implementation with the energy used by various other proposed algorithms, as well as the (impractical) best fixed time-out and optimal algorithms.

5.1 Methodology

We used traces of HP C2474s disks collected from April 18, 1992 through June 19, 1992 (63 days) [19].

We compare the share algorithm with several algorithms, including the 2-competitive algorithm, the randomized ($\frac{e}{e-1}$)-competitive algorithm, (an approximation to) the best fixed time-out, and the optimal algorithm. These other algorithms are described in §2. Although no practical algorithm can do as well as the optimal algorithm, the optimal algorithm's performance provides an indication of how far we have come and how much room is left for improvement.

There are two minor differences between the best fixed time-out we plot and the actual best fixed time-out for the 63 day trace. First, since it is computationally expensive to compute the exact best fixed time-out (quadratic in the number of disk accesses), we used a close approximation. We evaluated 10,000 different time-out points evenly spaced from 0 to 100 seconds on each day's trace data, and plot the best of these values as the best fixed time-out. Note that the best fixed time-out is guaranteed to be within $\frac{1}{100}$th of a second of the value we use and Figure 1 shows how little the energy used varies near the best fixed time-out. Furthermore, we can use the number of spin-downs to bound the difference in energy used by the best fixed time-out and our approximation. In all cases we found that the energy used by the best fixed time-out is within 0.4% of the numbers we report. Second, we estimate the best fixed time-out on each individual day's trace data. Thus instead of using the best fixed time-out for the entire 63 day trace, we use a series of daily best fixed time-outs computed from just that day's disk activity. This will tend to *underestimate* the energy use of the best fixed time-out since consecutive days are no longer constrained to use the same time-out.

As in all trace-driven simulations, it is difficult to model the interaction between the spin-down decisions of the algorithm and the sequences of disk accesses. If an algorithm spins down the disk then the following disk request must wait until the disk spins up again. It is impossible to know what effect this would have on the future disk accesses. Furthermore, these delays may adversely impact the computer's performance as perceived by the user. Although we are unable to explicitly account for these effects, we do provide the number of spin-downs done by the various algorithms in §5.5 as an indication of the trace reliability and the intrusiveness of the spin-down policy on the user.

Since the implementation of the share algorithm has several parameters (number and predictions of the experts, learning rate, and share rate) we used a train-and-test regimen. One disk was selected and its trace data for the first day was used to find reasonable settings for the parameters. The values determined from this partial trace are 100 experts, $\eta = 4$, and $\alpha = 0.08$, and these are the defaults used in our experiments. The rest of that trace as well as traces from other disks were used to validate that these settings do in fact tend to perform well. Although the η and α parameters must be carefully chosen for the worst-case bound proofs, for practical purposes the choice of these parameters tends to have little impact on the algorithm's performance. As mentioned in §3, the performance varies by less than one-fifth of one percent as the share rate varies from 0.5 to 1 or the learning rate varies from 3.5 to 4.5. This observation is in agreement with other empirical work on multiplicative weight algorithms [2]. The number of experts and distribution of the experts appears to have more impact on the algorithm's performance.

Before presenting our main results, we discuss the differences between various fixed time-outs on a typical day's trace data. Figure 1 illustrates how the fixed time-outs perform on the trace data for a Monday using a spin-down cost of 10. The figure shows fixed time-outs in the range 0 to 10 seconds. This figure shows that very small fixed time-outs are very expensive, the best fixed time-out is around 2.52 seconds, and the energy cost slowly rises for larger time-outs.

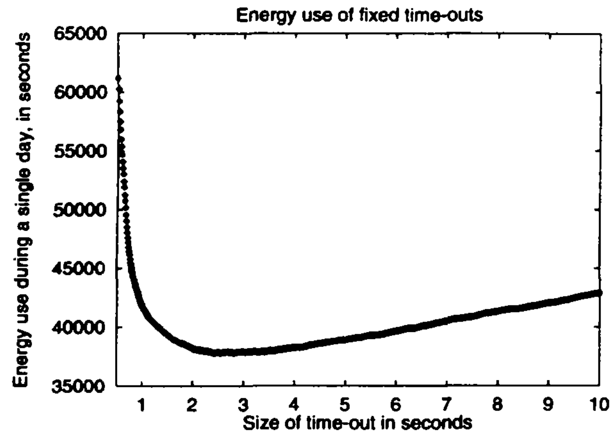

Figure 1: Energy cost of fixed time-outs, Monday, April 20 using a spin-down cost of 10.

Table 2 shows that almost all of the idle times are very small, so very small time-outs spin down the disk too aggressively and pay too many spin-up costs. As the best fixed time-out becomes larger, fewer spin-downs occur so fewer spin-down costs are paid. On the other hand, the disk remains spinning longer before it spins down, and more energy is spent keeping the disk spinning. The best fixed time-out tends to be on the border between a clump of slightly smaller idle times, and a region with few idle times. Increasing the spin-down cost makes the spin-ups relatively more expensive, and thus the best fixed time-out shifts to the right. Figure 2 shows how the best fixed time-out grows as a function of the spin-down cost. At a spin-down cost of 19, we see a large increase in the best fixed time-out. This is caused by a cluster of idle times around 30 seconds.

5.2 Main Results

Table 2: Frequencies of idle time ranges in a typical day of the trace. There are 142,694 idle times in this day.

Idle time (seconds)	Frequency count
0	37,252
0 - 1	102,146
1 - 10	1,747
10 - 30	917
31 - 100	498
100 - 600	131
> 600	3

Figure 3 summarizes the main experimental results of this article. For each value of the spin-down cost, we show the daily energy use (averaged over the 62 day test period) for all the algorithms described in this section: the 2-competitive algorithm, the $(\frac{e}{e-1})$-competitive randomized algorithm, the approximate best fixed time-out, and the share algorithm. We also include the energy use of the one minute and 30-second fixed time-outs for comparison (we believe that these are indicative of what is commonly used in portable computers today). We also show the energy used by the optimal algorithm to give some idea of scale and the limits on possible improvements (although no practical algorithm is likely to approach this theoretical best performance). The figure shows that the share algorithm is better than the other practical algorithms, and even outperforms the best fixed time-out. Our implementation uses be-

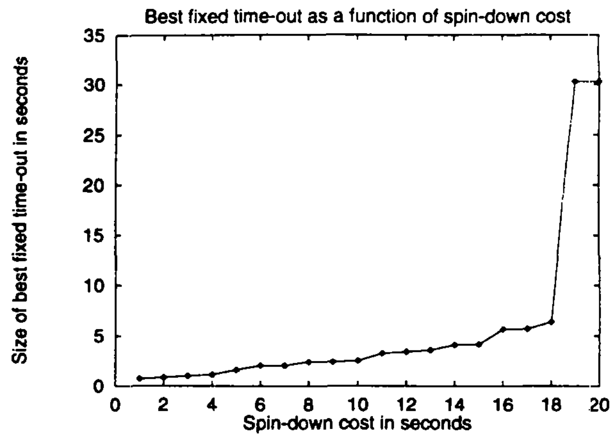

Figure 2: Best fixed time-out, as a function of the spin-down cost.

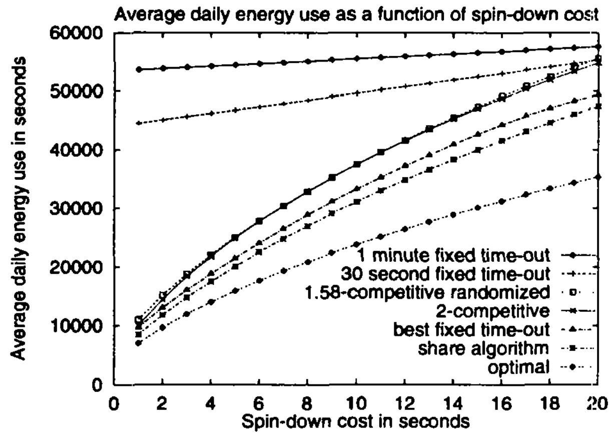

Figure 3: Average energy use per day as a function of the spin-down cost.

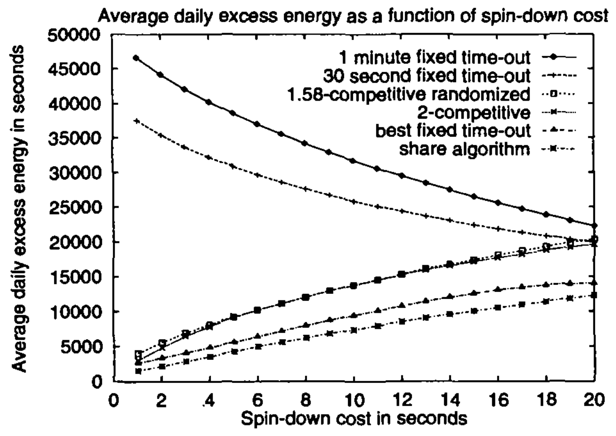

Figure 4: Average excess energy per day as a function of the spin-down cost.

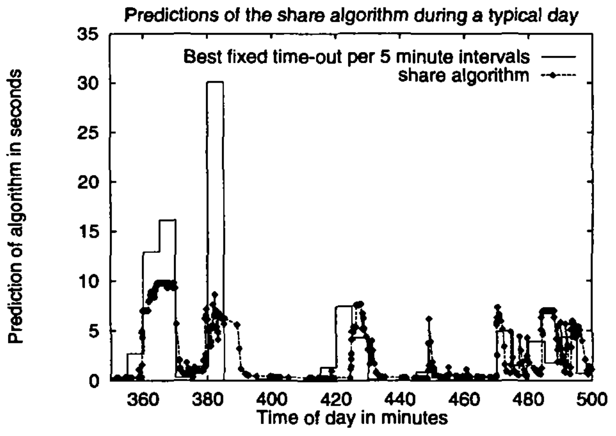

Figure 5: Time-outs suggested by the algorithm, on each successive trial during a 24 hour trace, with a spin-down cost of 10.

tween 88% (at spin-down cost 1) and 96% (at spin-down cost 20) of the energy used by the best fixed time-out. When averaged over the 20 time-outs, our implementation uses only 93% of the energy consumed by the best fixed time-out. Figure 4 plots the *excess energy* used by each algorithm, essentially subtracting off the energy used by the optimal algorithm from Figure 3. This indicates how much more energy was used by the algorithm than the theoretical minimum energy needed to complete the requested sequence of disk accesses.

Compared to the performance of a one minute fixed time-out, we see a dramatic performance improvement. Over the twenty spin-down costs, the share algorithm averages an excess energy of merely 26.3% the excess energy of the one minute time-out. In terms of total energy, the new algorithm uses 54.6% of the total energy used by the one minute fixed time-out (averaged over the different spin-down costs). Stated another way, if the battery is expected to last 4 hours with a one minute time-out (where 1/3 of the energy is used by the disk) then almost 45 minutes of extra life can be expected when the share algorithm is used.

Notice that the ($\frac{e}{e-1}$)-competitive randomized algorithm performs slightly worse than the 2-competitive algorithm (recall that ($\frac{e}{e-1}$) ≈ 1.58, so we would expect it to do better). Although this result seems surprising at first, it is easily explained when we consider the distribution of idle times in our traces. Most idle times are short – much shorter than the spin-down cost, and the 2-competitive algorithm will perform optimally on the short idle times. Only when the disk stays idle for longer than the spin-down cost will the 2-competitive algorithm be suboptimal. The ($\frac{e}{e-1}$)-competitive randomized algorithm performs ($\frac{e}{e-1}$)-competitively for all idle times, including those shorter than the spin-down cost. While the 2-competitive algorithm does worse on the longer idle times, it does much better on the more frequent short idle times.

This comparison illustrates a disadvantage of worst case analysis. Good worst-case algorithms spread their performance evenly across all possible inputs. While the ($\frac{e}{e-1}$)-competitive algorithm has a better worst case bound, it does not perform as well under realistic workloads as algorithms with weaker worst-case bounds but better performance on the likely cases. In practice, it is desirable for algorithms to perform best on the likely input values.

One class of algorithms adapts to the input patterns, deducing from the past which values are likely to occur in the future. This approach was taken by Krishnan *et al.* [14] (see §2). Their algorithm keeps information which allows it to approximate the best fixed time-out for the entire sequence, and thus its performance is about that of the best fixed time-out. We use an algorithm that takes this approach a step further. Rather than looking for a time-out that has done well on the entire past, the share algorithm attempts to find a time-out that has done well on the recent past.

As Figure 4 shows, the share algorithm consistently outperforms the best fixed time-out, consuming an average of 79% of the excess energy consumed by the best fixed time-out. If the data is truly drawn according to a fixed probability distribution, then the best on-line policy will be some fixed time-out whose value depends on the distribution. Since the share algorithm outperforms the best fixed time-out, it is exploiting time dependencies in the input values. Of course, it is not surprising that there are time dependencies as it is well-known that user access patterns exhibit bursty behavior [1].

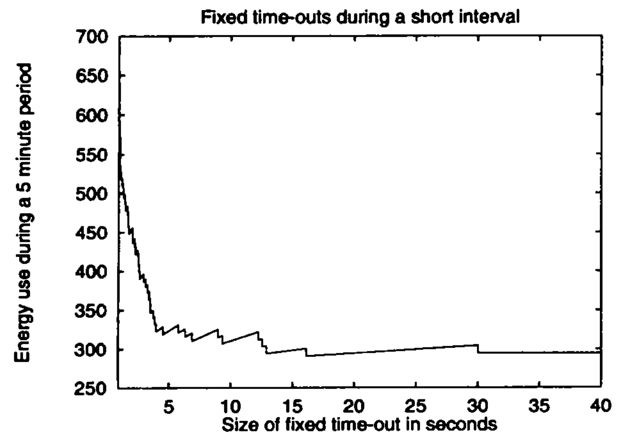

Figure 6: Performance of fixed time-outs in the interval from 360 to 370 minutes of Figure 5.

5.3 Predictions of the Share Algorithm

Figure 5 shows the predictions of the algorithm during a portion of a typical day (Monday, April 20), using a spin-down cost of 10. The figure also shows the fixed time-out that would have minimized the energy use on each five minute interval of that period. Note that these minimizing time-outs require knowledge of the future, and as the interval size shrinks, the time-outs minimizing the energy on the interval approach the time-outs used by the optimal algorithm. Therefore it is unreasonable to expect any practical algorithm to perform as well as the best fixed time-outs on small intervals. However, this figure does illustrate some interesting aspects of the share algorithm. We can see that the time-outs used by the share algorithm vary dramatically throughout the day, by at least an order of magnitude. While the disk is idle, the predictions slowly become smaller (smaller than the best fixed time-out), and the algorithm spins the disk down more aggressively. When the disk becomes busy, the algorithm quickly jumps to a much longer time-out. These kinds of shifts occur often throughout the trace, sometimes every few minutes. The time-outs issued by the share algorithm tend to follow the best fixed time-outs over five minute intervals, which in turn reflects the bursty nature of disk accesses.

Figure 6 shows the performance of many fixed time-outs over the ten minute interval from 360 to 370 minutes shown in Figure 5. Due to our choice of experts, the share algorithm can only make predictions less than or equal to the spin-down cost (10 in this example). However, as Figure 5 shows, sometimes the best fixed time-out is not in this range. Yet, our performance remains good overall. The explanation for this is that when the best fixed time-out is large there is a large region which is nearly flat, with all time-outs in the region using similar amounts of energy. For example, Figure 6 shows that time-out values between 9 and 10 (within the range considered by the share algorithm) perform almost as well as the best fixed time-out. In particular, although the best fixed time-out is 16.115, the energy used by the 9.34 time-out is within 5% of the energy used by the 16.115 time-out over this 10 minute interval.

5.4 Adaptive Results

Douglis *et al.* [4] consider a family of incrementally adaptive spin-down schemes. This family of algorithms has been shown to save significant amounts of energy. These schemes change the time-out after each idle time by either an additive or multiplicative factor. When an idle time is "long" the time-out is decreased so that the algorithm spins down the disk more aggressively, wasting less energy waiting for the time-out. When an idle time is "short" the time-out is increased to reduce the chance of an inappropriate spin-down. We compare the share algorithm with several of these schemes for a spin-down cost of 10 seconds.

Since the traces are composed mostly of short idle times, there is the danger that an adaptive algorithm's time-out value will quickly become too large to be effective (see Figure 1). To prevent this problem we prevent the time-out from exceeding 10 seconds, the same value as the largest expert used by the share algorithm. Without this bound on the time-out, the incrementally adaptive algorithms perform poorly. We also used 10 seconds as the definition of "long" idle times: treating idle times less than 10 seconds as "short" and idle times longer than 10 seconds as "long."

We considered three ways of increasing the time-out: doubling it, by adding 1 second, and adding 0.1 seconds. Similarly, we considered three ways of decreasing the time-out: halving it, decreasing it by 1 second, and decreasing it by 0.1 seconds. This gives us nine variations in the incrementally adaptive family. These values were also used in Douglis *et al.* [4] and the long version of Golding *et al.*[6]. The time-out should never become negative, we constrained the time-out to be at least one decrement amount above zero.

We compared each of these nine algorithms with the share algorithm and the daily best time-outs on three different traces. The results are in Table 5.4.

Since each day contains 86400 seconds, all of these algorithms are saving significantly over a no-spin-down policy. Only the share algorithm can beat the daily best fixed time-outs, which it does on two of the traces. In addition the share algorithm is less intrusive than the daily best fixed time-out, with fewer spin-up delays.

Table 3: Spin-downs and energy costs in seconds of the adaptive algorithms, averaged over the two-month traces.

Algorithm		Trace 1 SDs	Trace 1 cost	Trace 2 SDs	Trace 2 cost	Trace 3 SDs	Trace 3 cost
Daily best fixed		2065	33007	710	11646	409	4886
Share algorithm		1889	31017	689	11707	373	4808
10 second time-out		1378	37426	493	13774	294	6613
+ 0.1	− 0.1	1378	37411	495	13792	294	6376
+ 0.1	− 1.0	1471	34924	649	12963	312	5630
+ 0.1	÷ 2.0	1775	31424	788	12550	356	4923
+ 1.0	− 0.1	1378	37417	494	13812	294	6576
+ 1.0	− 1.0	1378	37260	504	13478	294	6444
+ 1.0	÷ 2.0	1440	35589	563	12666	503	5742
× 2.0	− 0.1	1381	37395	499	13782	294	6594
× 2.0	− 1.0	1378	37431	549	13430	294	6604
× 2.0	÷ 2.0	1564	36047	494	13832	320	6157

Of the nine algorithms in the incrementally adaptive family, the "add 0.1 second and divide by 2.0" algorithm saved the most energy. However, even this algorithm uses up to 7.2% more energy than the share algorithm and does worse than the the daily fixed time-out on all three traces.

It appears that the better incrementally adaptive algorithms decrease the time-out rapidly but increase it only slowly. Decreasing the time-out rapidly allows greater savings if the next idle time is long. The disadvantage of a rapid decrease is that an inappropriate spin-down may occur if the next idle time had an intermediate duration. However, the preponderance of small idle times (see Table 2) makes this relatively unlikely. A slow increase in the threshold allows the algorithm to perform well when two nearby long idle times are separated by one or two short idle times.

It appears from Table 5.4 that some of the incrementally adaptive algorithms are primarily exploiting the 10 second bound on their time-outs. Since any spin-down done by the 10 second time-out is also done by all of the other algorithms, we can infer that some of the add/subtract algorithms do exactly the same spin-downs as the 10 second time-out, although sometimes they may do these spin-downs with a slightly smaller time-out.

An interesting property of Table 5.4 is that the daily best fixed time-out uses slightly less energy than the share algorithm on trace 2. This is due in part to the fact that the best fixed time-out is recalculated for each day's data. On trace 2, it varies from about 1 second to about 6 seconds depending on the day. If the same time-out was used every day then energy consumed would certainly be larger than that used by the share algorithm.

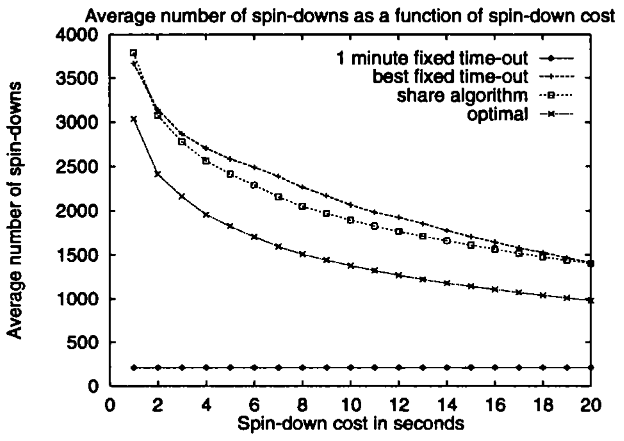

Figure 7: Average spin-downs for each algorithm as a function of spin-down cost.

5.5 Spin-downs

Figure 7 shows the average number of spin-downs used per day as a function of the spin-down cost for the share algorithm, the best fixed time-out, the optimal algorithm, and the one minute fixed time-out. We can see from this figure that the share algorithm recommends spinning the disk down less often than the (daily) best fixed time-outs. Our simulations show that the share algorithm tends to predict when the disk will be idle more accurately than the best fixed time-out, allowing it to spin the disk down more quickly when both algorithms spin down the disk. In addition, the share algorithm more accurately predicts when the disk will be needed, enabling it to avoid disk spin-downs ordered by the best fixed time-out. Thus, in addition to using less energy than the best fixed time-out, the share algorithm also spins the disk down less often, minimizing wear and tear on the disk and inconvenience to the user.

6 Future work

Many avenues of further exploration exist. Several variations on the share algorithm have the potential to dramatically improve its performance. A "self tuning" version of the share algorithm that requires no (or fewer) parameters would be useful. It would be comforting to have a good worst case analysis of the share algorithm in this setting. One issue that cries out for further study is how the additional latency imposed by disk spin-down effects the user and their applications. Another issue of critical importance is studying how algorithms like the share algorithm perform on other power management and systems utilization problems.

6.1 Extensions to the Share Algorithm

There are a number of improvements to the share algorithm that we are exploring. One promising approach is to modify the set of experts used by the algorithm. There is no reason that the experts have to be fixed time-outs. They could be previous idle times, a second-order estimate of the next idle time, or any function of the past disk accesses. In principle, any other spin-down algorithm could be used as an expert – when the algorithm performs well it will have high weight and the share algorithm would mimic its predictions. Similarly, when several prediction methods are used as experts, the share algorithm can exploit the strengths of each of method.

We are interested in better methods for selecting the algorithms learning rate and share rate. It may be possible to derive simple heuristics that provide reasonable values for these parameters. A more ambitious goal is to have the algorithm self-tune these parameters based on its past performance. Although cross validation and structural risk minimization techniques can be used for some parameter optimization problems, the on-line and time-critical nature of this problem makes it difficult to apply these techniques.

Although the scaled excess energy used in this article is an attractive loss function, other loss functions may prove even more beneficial. There is a version of the share update which shares a fixed amount from each expert per trial, instead of an amount proportional to the loss of that expert [10]. While the worst case bounds proven for the fixed share version of the algorithm are not as good as the version we use, we have not yet fully explored this variant's empirical behavior.

6.2 Related Problems

The power management problem can be viewed as a type of *rent-to-buy* problem [14]. A single rent-to-buy decision can be described as follows: we need a resource for an unknown amount of time, and we have the option to rent it for \$1 per unit time, or to buy it once and for all for \$$c$. For how long do we rent the resource before buying it?

Many interesting problems can be modeled as a sequence of rent-to-buy decisions. This is called the *sequential rent-to-buy* problem, or just the rent-to-buy problem [14]. For example, the disk spin-down scenario can be modeled as a rent-to-buy problem as follows. A *round* is the time between any two requests for data on the disk. For each round, we need to solve the disk spin-down problem. Keeping the disk spinning is viewed as renting, since energy is continuously expended to keep the disk spinning. Spinning down the disk is viewed as a buy, since the energy to spin-down the disk and spin it back up upon the next request is independent of the remaining amount of idle time until the next disk access.

Many other systems problems can be viewed as rent-to-buy problems. The power conservation problem for any mobile computer component which can be powered down for a fixed cost, such as disk drives, wireless interfaces, or displays, can be viewed as a rent-to-buy problems. A thread trying to acquire a lock on a shared device can either busy-wait or block (and context switch). This *spin/block* problem can also be viewed as a rent-to-buy problem. Deciding virtual circuit holding times in an IP-over-ATM network is yet another example of a rent-to-buy problem.

Since the share algorithm is relatively simple and effective for the disk spin-down problem, it is natural to consider its application to other rent-to-buy situations.

7 Conclusions

We have shown that a simple machine learning algorithm is an effective solution to the disk spin-down problem. This algorithm performs better than all other algorithms that we are aware of, often using less than half the energy consumed by a standard one minute time-out. The algorithm even outperforms the impractical best fixed time-out. The algorithm's excellent performance is due to the way it adapts to the pattern of recent disk activity, exploiting the bursty nature of user activity.

The existing algorithms for the disk spin-down problem that we are aware of make either worst-case assumptions or attempt to approximate the best fixed time-out over an entire sequence. We discuss why algorithms with good worst case bounds do not necessarily perform well in practice. Our simulations show that the new algorithm outperforms worst-case algorithms by a significant amount.

We believe that the disk spin-down problem is just one example of a wide class of rent-to-buy problems for which the new algorithm is well suited. In addition to disk spin-down, other problems in this class are of importance to mobile computing, such as: power management of a wireless interface, admission control on shared channels, and a variety of other power management problems. Other rent-to-buy problems where the algorithm can be applied include applications such as deciding when a thread that is trying to acquire a lock should busy-wait or context switch or computing virtual circuit holding times in IP-over-ATM networks [14].

Our implementation of the share algorithm is efficient, taking taking constant space and constant time per trial. This constant is adjustable, and adjusts the accuracy of the algorithm. For the results presented in this article, the total space required by the algorithm is never more than about 2400 bytes, and our implementation in C requires only about 100 lines of code. This algorithm could be implemented on a disk controller, in the BIOS, or in the operating system.

Acknowledgments

We are deeply indebted to J. Wilkes, R. Golding, and the Hewlett-Packard Company for making their file system traces available to us. We are also grateful to M. Herbster and M. Warmuth for several valuable conversations, and to the anonymous referees for their comments.

References

[1] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout, "Measurements of a distributed file system," in *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, (Asilomar, Pacific Grove, CA), pp. 198–212, ACM, Oct. 1991.

[2] A. Blum, "Empirical support for Winnow and weighted-majority based algorithms: results on a calendar scheduling domain," in *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 64–72, Morgan Kaufmann, 1995.

[3] N. Cesa-Bianchi, Y. Freund, D. Haussler, D. P. Helmbold, R. E. Schapire, and M. K. Warmuth, "How to use expert advice," Tech. Rep. UCSC-CRL-94-33, University of California, Santa Cruz, 1994.

[4] F. Douglis, P. Krishnan, and B. Bershad, "Adaptive disk spin-down policies for mobile computers," in *Proceedings of the Second Usenix Symposium on Mobile and Location-Independent Computing*, (Ann Arbor, MI), Usenix Association, Apr. 1995.

[5] F. Douglis, P. Krishnan, and B. Marsh, "Thwarting the power-hungry disk," in *Proceedings of the Usenix Technical Conference*, (San Francisco, CA), pp. 292–306, Usenix Association, Winter 1994.

[6] R. Golding, P. Bosch, C. Staelin, T. Sullivan, and J. Wilkes, "Idleness is not sloth," in *Proceedings of the Usenix Technical Conference*, (New Orleans), pp. 201–212, Usenix Association, Jan. 1995.

[7] K. Govil, E. Chan, and H. Wasserman, "Comparing algorithms for dynamic speed-setting of a low-power cpu," in *The First Annual International Conference on Mobile Computing and Networking (MobiCom)*, (Berkeley, CA), pp. 13–25, ACM, 1995.

[8] P. Greenawalt, "Modeling power management for hard disks," in *Proceedings of the Conference on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pp. 62–66, IEEE, Jan. 1994.

[9] D. Haussler, J. Kivinen, and M. K. Warmuth, "Tight worst-case loss bounds for predicting with expert advice," Tech. Rep. UCSC-CRL-94-36, University of California, Santa Cruz, Nov. 1994.

[10] M. Herbster and M. K. Warmuth, "Tracking the best expert," in *Proceedings of the Twelfth International Conference on Machine Learning*, (Tahoe City, CA), pp. 286–294, Morgan Kaufmann, 1995.

[11] A. Karlin, M. Manasse, L. Rudolph, and D. Sleator, "Competitive snoopy caching," in *Proceedings of the Twenty-seventh Annual IEEE Symposium on the Foundations of Computer Science*, (Toronto), pp. 224–254, ACM, Oct. 1986.

[12] A. Karlin, M. S. Manasse, L. A. McGeoch, and S. Owicki, "Competitive randomized algorithms for non-uniform problems," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pp. 301–309, 1990.

[13] J. Kivinen and M. Warmuth, "Using experts for predicting continuous outcomes," in *Computational Learning Theory: Eurocolt '93*, vol. New Series Number 53 of *The Institute of Mathematics*

and its Applications Conference Series, pp. 109–120, Oxford University Press, Dec. 1993.

[14] P. Krishnan, P. Long, and J. S. Vitter, "Adaptive disk spin-down via optimal rent-to-buy in probabilistic environments," in *Proceedings of the Twelfth International Conference on Machine Learning (ML95)*, (Tahoe City, CA), pp. 322–330, Morgan Kaufman, July 1995.

[15] K. Li, R. Kumpf, P. Horton, and T. Anderson, "A quantitative analysis of disk drive power management in portable computers," in *Proceeding of the Usenix Technical Conference*, (San Francisco), pp. 279–291, Usenix Association, Winter 1994.

[16] N. Littlestone, "Learning when irrelevant attributes abound: A new linear-threshold algorithm," *Machine Learning*, vol. 2, pp. 285–318, 1988.

[17] N. Littlestone, *Mistake Bounds and Logarithmic Linear-threshold Learning Algorithms.* Ph.D. dissertation, University of California Santa Cruz, 1989.

[18] N. Littlestone and M. K. Warmuth, "The weighted majority algorithm," *Information and Computation*, vol. 108, no. 2, pp. 212–261, 1994.

[19] C. Ruemmler and J. Wilkes, "UNIX disk access patterns," in *Proceedings of the Usenix Technical Conference*, (San Diego, CA), pp. 405–420, Usenix Association, Winter 1993.

[20] D. D. Sleator and R. E. Tarjan, "Amortized efficiency of list update and paging rules," *Communications of the ACM*, vol. 28, pp. 202–228, Feb. 1985.

[21] V. Vovk, "Aggregating strategies," in *Proceedings of the Third Annual Workshop on Computational Learning Theory*, (Rochester, NY), pp. 371–383, Morgan Kaufmann, 1990.

[22] M. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for reduced CPU energy," in *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI)*, (Monterey, CA), pp. 13–23, Usenix Association, Nov. 1994.

[23] J. Wilkes, "Predictive power conservation," Tech. Rep. HPL-CSP-92-5, Hewlett-Packard Laboratories, Feb. 1992.