Controlling Head Parking in Laptop Hard Drives

Daniel Gnoutcheff '11
Union College
2008-2009 CT Scholars Independent Research
Adviser: Prof. Chris Fernandes

December 8, 2009

In modern laptops, hard drives are significant power sinks; therefore, drive manufacturers are quite interested in developing ways to save power without reducing performance. However, hard drives are also one of the most fragile components in modern laptops, and they are the component that is responsible for the most important thing in a laptop - the user's data. Many hard drive power management features cause wear and tear, so care must be taken to manage these features properly. Unfortunately, modern drives tend to use inadequate control mechanisms, and there is one common firmware bug that can cause very rapid wear and premature failure. This is a very dangerous bug that can - and should - be addressed by hard drive manufacturers.

The Problem

The bug is related to the hard drive feature of HEAD PARKING. A typical drive consists of a set of magnetic platters and a corresponding set of read-write heads that actually access and modify data on the platters (see Figure 1 on the following page). Normally, the heads are positioned over the platters, floating a very short distance above the platter surfaces. The heads create air drag for the spinning platters, increasing the power needed to maintain their speed. Also, the close proximity of the heads and the platters means that physically jarring the hard drive could cause a head to come into contact with a fast-moving platter - a catastrophic HEAD CRASH.

So, to save power and to reduce the risk of a head crash, it is sometimes desirable to perform a head park. When the disk is not in use, the heads may be taken completely off the platters. When the disk is next accessed (for either a read or a write), the heads are pulled back onto the platters ("unparked") and normal operation resumes. Heads can be parked and unparked fairly quickly, and the platters remain spinning while the heads are unparked, so head parking is usually transparent to the user and the operating system.

Nonetheless, head parking must be used with care. A head park puts a significant amount of stress on the heads, and in many implementations, the
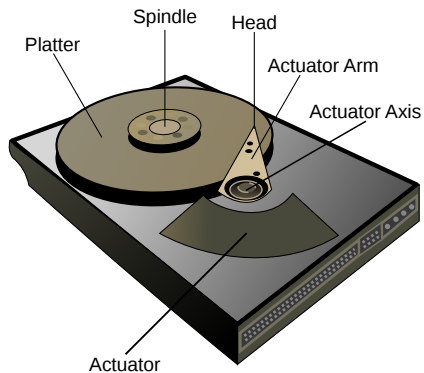
Figure 1: A simplified diagram of a hard drive's platters and associated hardware, in normal operation. Usually, the heads are very close to the platters, and an air bearing created by the fast-moving platters is used to keep the heads from hitting them. (Figure based on the graphic from `http://commons.wikimedia.org/w/index.php?title=File:Hard_drive-en.svg&oldid=13777056`)

heads may come into physical contact with other components when parking. Thus, head parks cause wear and tear, and practically all drives have a limit on the number of parks that they can perform before there is a risk of failure. With the most recent advancements, modern laptop hard drives typically can tolerate up to 600,000 park/unpark cycles.

To decide when to park, many hard drives use a very simple FIXED TIMEIN algorithm. This algorithm assumes that if the drive has not been accessed for n seconds (where n is a predefined constant), then it will probably remain idle for a reasonably long time. Thus, the hard drive will always perform a head park after n seconds of idle time. For laptop hard drives, n typically defaults to a value on the order of 6 seconds.

Sadly, it is all too easy to imagine a scenario where this simple algorithm fails. Consider a setup used to measure daily temperature variations on a large number of days; say we have a high-precision thermometer set up to give a reading every 10 seconds. Assume that a laptop has been set up to record these readings; at each reading, a program writes the temperature to a log file and instructs the OS to ensure that the data is actually saved to disk (e.g. with the POSIX fsync call). Assume that this is the only running program on the laptop that performs disk IO. Thus, we get a disk write every 10 seconds. In *every one* of these 10 seconds intervals, the heads will be parked 6 seconds in and then unparked 4 seconds later; thus, we get 6 park/unpark cycles per minute. At this rate, we will reach 600,000 cycles after about 1,667 hours. Assuming we run this laptop for 6 hours a day, this means that a brand new hard drive will reach its cycle limit after 278 days - much less than a year. That is unacceptable.

Alarmingly, this problem is not limited to contrived scenarios. Nearly every

Linux distribution - including the popular Ubuntu[1] - has a disk access pattern that triggers this bug, even when the system is idle. Many laptops, including those considered to have relatively strong Linux support, ship with hard drives that have this problem, and there is empirical evidence that this bug can cause drives to fail within a year of purchase[2]. This has led to a wave of bad publicity - including an alarming Slashdot article[3] - and a lot of very concerned users (myself included). Clearly, something must be done.

How to Fix It

It might initially appear that this bug can be fixed by having the operating system guarantee a disk access pattern that avoids the problem. However, for most OSs, this guarantee is impossible. Consider the temperature logger again, where all disk IO is created by a single process. That process simply reads the temperature, saves the value, syncs to disk, and repeats after 10 seconds. Between each temperature reading, the OS has no IO to do, so it has no excuse to access the disk between readings. If we assume a fixed timein of 6 seconds, this means that the OS is forced to allow a head park before the next reading. So every time we do a reading, the OS will be asked to write data to a disk that's parked - and this will happen every 10 seconds. The only way the OS can avoid a park/unpark every 10 seconds is to delay the write, allowing the disk to stay parked for a reasonably long time. But the logger program tries to do each write synchronously, so it will get blocked for long periods of time. This will cause unacceptably poor performance and, in our case, ruin the experiment. This is unacceptable behavior, so the the OS is forced to skip the delays and access the disk immediately - thus leading to exactly the kind of excessive head parking that we're trying to avoid.

Thus, thanks to the need for synchronous IO and the demand for speed, the OS has very little control over its disk access pattern. Any process, including an unprivileged one belonging to an untrusted user, can cause a disk access at any time, allowing just about any kind of access pattern to arise under just about any OS. While tweaking the IO subsystem can *reduce* the risk of a dangerous access pattern, it cannot eliminate it, and when the danger involves damaged hardware and lost data, any risk is too great.

An alternative solution is disabling head parking completely. This is a popular option among Linux users trying to workaround this bug, and it is a straightforward (and dramatic) way to guarantee that excessive head parking will not occur. However, this method misses out on the potential benefits of head parking, and it is surprisingly difficult to implement. There is no standard interface for controlling automatic head parking, and some drives will continue parking even when all power management has ostensibly been turned off. Worst of all, many laptop hard drives are mounted with limited ventilation, apparently un-

---

[1] https://launchpad.net/ubuntu/+source/acpi-support/+bug/59695
[2] http://paul.luon.net/journal/hacking/BrokenHDDs.html
[3] http://hardware.slashdot.org/article.pl?sid=07/10/30/1742258

der the assumption that the drive's power management features will reduce the amount of heat released. When power management is disabled, these drives heat up - and it is well understood that heat accelerates drive failure.

Thus, we are forced to face the challenge of handling head parking correctly. We must develop a new mechanism, a new PARK DECISION ALGORITHM, that determines when it is appropriate to park. It must be simple enough to be easily implemented in hard drive firmware. It should keep the heads parked for as long as possible so as to maximize power savings. Finally, and most importantly, it should be robust enough so that no matter what the disk access pattern is, the PARK RATE (the number of head parks per unit time) is low enough that the drive will have a reasonably long lifetime.

The Budgeter and the Suggester

One way to control the park rate is to add a mechanism I call the BUDGETER. The Budgeter is best thought of as a "relief valve" that can be added to an existing park decision algorithm (the "SUGGESTER") and prevents hardware damage in the case that the suggester behaves badly.

The mechanism is simple. Before operation, we define the maximum number of parks allowed during a defined amount of time. For example, we may say that we will allow up to 5 parks in a 10 minute period. During operation, we use the suggester algorithm to decide when to park, and we stop once 10 minutes is up or 5 parks have been performed. If the parks are "used up" first, we stop parking and wait until the block of time ends. Then, we repeat for the next block. In other words, this mechanism works by dividing time into successive blocks and, within each block, stopping all parking once the maximum number of parks is reached.

While this mechanism effectively fixes the original bug, it has efficiency problems. If too many parks are suggested, we may end up "wasting" parks early in a time block and thus losing some good opportunities later. Thus, in addition to using this relief valve, it is also desirable to use a suggester that can adapt to the given disk access pattern and at least try to control the number of parks it suggests. To this end, we present and evaluate a few algorithms that improve upon the fixed timein by attempting to automatically calculate an appropriate timein.

New Suggesters

Sliding Window - Period Frequency

The sliding window period frequency mechanism (codenamed SWPERIODFRE- QUENCY) assumes that every IDLE PERIOD (the time between two successive disk accesses) has a length given by an independent random function that has a probability distribution approximated by the last n idle periods (where n is a predefined constant). This assumption is believed to be reasonable, as programs

Time: 0   2   4   6   8   10   12   14   16   18   20   ...
Freq: | 1000 | 10 | 0 | 100 | 0 | 0 | 0 | 0 | 0 | 0 | 50 | ...
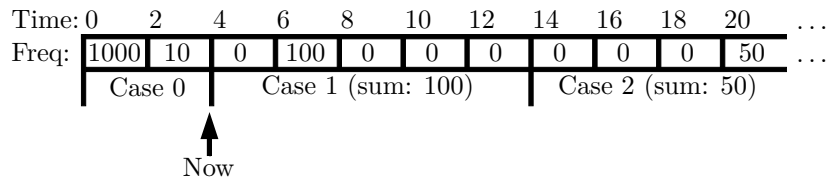   Case 0   Case 1 (sum: 100)   Case 2 (sum: 50)

↑
Now

Figure 2: An example of a frequency table maintained and analyzed by SWPeriodFrequency. In this situation, 4 seconds after the last disk access, there are more entries under case 1 (don't park) than under case 2 (do park), so we don't park.

Time: 0   2   4   6   8   10   12   14   16   18   20   ...
Freq: | 1000 | 10 | 0 | 100 | 0 | 0 | 0 | 0 | 0 | 0 | 50 | ...
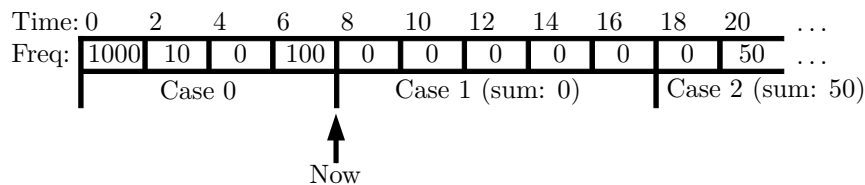   Case 0   Case 1 (sum: 0)   Case 2 (sum: 50)

↑
Now

Figure 3: The same frequency table as in Figure 2, analyzed again 4 seconds later (assuming no intervening disk access. Now, there are more entries in case 2 than in case 1, so we do park.

that access the disk on a regular basis often do so on fixed intervals, and these programs tend to operate in the background for long periods of time. To store the necessary history data, this algorithm maintains a frequency table of the lengths of the last n idle periods.

To determine when it is appropriate to park, let s denote the amount of time since the last disk access and define m to be the MINIMUM PARK TIME, the minimum amount of time for which we want to stay parked once we decide to park. At any given time, we park if there is at least a 50% probability that we would stay parked for at least m seconds. To estimate this probability, we consider three cases. Case zero is where the idle period is shorter than s; we already know that we are in an idle period of at least s, so we can ignore this case. Case one is where the idle period length is at least s but less than $s + m$; this is the case where we shouldn't park, because we would stay parked for less than m seconds. Finally, case two is where the idle period length is at least $m + s$; this is the case where we should park. We look at the frequency table to determine which of case 1 and 2 has been more common in the recent past, and we park if and only if case 2 has been more common. Figure 2 has an example of a situation where we don't park; Figure 3 shows a situation where we do.

This is probably the most "sophisticated" algorithm presented here. It maintains complete information about the disk usage history, so it is quite "accurate". However, it is rather complex, perhaps too much so to be implemented in hard drive firmware. We must have enough storage space not only for the frequency
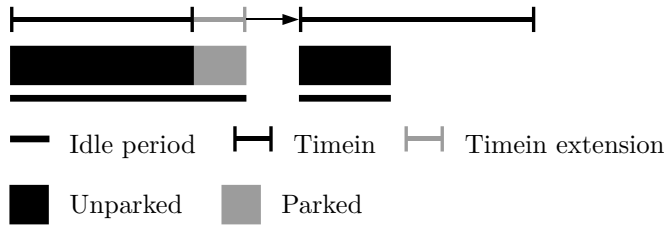
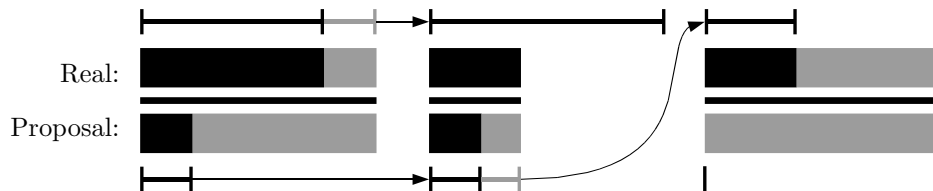Figure 4: Example of extending the real timein under the Proposer algorithm.

Figure 5: An extension of Figure 4, adding the proposed timein. We simulate how the disk would behave under the proposal and make any necessary extensions. After a period of time, the real timein is replaced by the proposal and we reset the proposal to zero.

table but also for a rather large queue to facilitate the discarding of old entries. Even with several optimizations, this mechanism requires many kilobytes of memory and, in certain scenarios, many processor cycles. Worst of all, the history is maintained in terms of the number of idle periods, not the amount of time. The time represented can vary widely, and a large burst of disk IO - in other words, a very large number of very small idle periods - can end up dominating the frequency table, flushing out useful data.

Proposer

The Proposer algorithm essentially increases the park timein whenever it appears appropriate to do so. We initialize the timein to some predefined constant (e.g. 6 seconds), and then, every time there is a head unpark, we determine if we have been parked for at least the minimum park time (m). If we have not (i.e. we just had a BAD PARK), then we increase the timein to be equal to the length of the idle period that caused the bad park. Then, future idle periods of that length won't lead to more bad parks. Figure 4 shows an example.

Of course, it is sometimes safe (and desirable) to make the timein shorter, and so far we have no way to do that. We thus introduce the PROPOSED TIMEIN, separate from the real timein. The proposed timein is initialized to zero. Then, for each idle period, we determine how the disk would have behaved if the proposed timein were real. If the proposal would have led to a bad park, we

6

extend the proposal to be equal to the length of the idle period that caused the simulated bad park, as is shown in Figure 5 on the preceding page. In this way, we "develop" a proposed timein for a predefined amount of time (the COOK TIME, on the order of 5 minutes). Then, we set the real timein equal to the value we developed, reset the proposal to zero, and continue.

The Proposer has several interesting properties. First, it is very simple and easy to implement; it needs only 2 variables, and it only requires that we run a small $O(1)$ algorithm on each disk access. Also, its window size (i.e. the amount of disk usage history it considers) is based on time, making it insusceptible to floods of short idle periods. However, the history is "stored" in a very limited form and a lot of potentially useful information is discarded. For example, if there is a process that, on *rare occasions*, accesses the disk after 10 seconds of inactivity, a timein of, say, 4 seconds might still be appropriate. However, this algorithm cannot distinguish between an idle period length that occurs frequently or one that occurs rarely, so it will insist on a timein of at least 10 seconds.

Evaluation

Both of these algorithms use numerous assumptions and simplifications; to determine if they behave well in practice, we need to test them against a real world situation. To do this, we collect data on the disk access pattern of a real system, implement each of these algorithms, and then run simulations based on the collected data to see how these algorithms behave.

Data collection

The Linux kernel contains a switch called BLOCK_DUMP that is designed to assist investigations of disk IO and power management. When block_dump mode is enabled, the disk IO subsystem becomes very verbose, printing into the kernel log buffer information about every disk and file access. Furthermore, if the kernel is configured appropriately during compilation, it will print timestamps in each log message[4]. This data can provide complete information about the disk access pattern, and it is a perfect source of data for our simulations.

For about a month, I used the block_dump mechanism to record the disk access pattern of my laptop while using it in the middle of a college term, forming what I call the DISK LOG. Care had to be taken while recording this data. It was not safe to use the existing logging system; when it saves a message about a disk access, it causes another disk access, causing another disk access message to be printed, leading to another disk access, etc. The result is an infinite loop of logged disk accesses, consuming CPU time, disk bandwidth, and disk space. And, of course, the loop dramatically distorts the disk access pattern, making the data nearly useless. Since this data collection was performed over a long period of time on a production system, simply disabling the logging system

---

[4]This is the CONFIG_PRINTK_TIME option, accessed from the Kernel Hacking menu.

was not an option. Instead, a small program was inserted between the kernel log buffer and the main logging system, redirecting block_dump output to a file on a ramdisk. Even then, care had to be taken, as block_dump also prints information about ramdisk accesses. It was thus necessary to filter out messages not related to real disk IO.

The collected data is representative of how I use my Linux system during a typical college term. It covers a variety of activities (web surfing, word processing, programming, note taking, software installation, etc.), and it also covers a few different tweaks and modifications I have made to the disk IO subsystem. However, it *is* limited to a single user and a single operating system.

Finally, it should be noted that the log data does not include information about when the system is shut down or suspended. Such periods sometimes appear as very long idle periods, which is inaccurate since this time does not count toward the hard drive's uptime. To address this problem, all idle periods longer than 5 minutes are ignored. On the laptop in question, real idle periods longer than 5 minutes are rare, and since the algorithms being tested are supposed to handle any disk access pattern, the modified data is still suitable for testing.

Frequency of parks

Simulations of the proposed decision algorithms and of the existing fixed timein algorithm were implemented in Java. Each algorithm implementation supports the "recording" of the length of the most recent idle period and the querying of the algorithm's effective timein value after that period. Then, a combination of Unix awk/sed scripts and a Java driver program were used to read the disk log and use it as a script for a disk IO simulation. The decision algorithm simulators were given idle period lengths from the disk log, and after each period, the appropriate timein was generated. The driver program keeps track of how many parks the algorithm ends up performing as well as the total amount time that the heads are parked. This continues until the disk log data is exhausted.

This simulation was run for each of the suggester algorithms presented. The results are in Table 1 on the next page. It appears that both SWPeriodFrequency and Proposer successfully reduce the number of parks performed. Hard drive lifetime has increased by at least a year in all cases, and in some cases much more. Thus, it appears that these new algorithms have much less of a dependence on the Budgeter. So, in this respect, the two algorithms are tied. To break the tie, we consider the second criterion - efficiency.

Efficiency

Since head parks have a cost, we want to chose a decision algorithm that is most likely to find those times where it is most beneficial to "spend" a park. One way to do this is to look at the total park time of each algorithm - the higher the time, the more benefit gained from parking, and therefore the more efficient the decision algorithm. However, each test resulted in a different number of parks, which complicates comparison. In general, reducing the number

Description	Total park time	Number parks	Projected HD lifetime (years)
Original (fixed timein of 6 seconds)	481,111	16,509	2.4
Proposer (min parktime = 10 seconds)	354,247	9,822	3.6
SWPeriodFrequency	299,083	8,987	4.2
Proposer (min parktime = 15 seconds)	297,934	7,272	5.1
Proposer (min parktime = 30 seconds)	210,081	4,127	9.0

Table 1: Number of parks and hard drive lifetimes for the proposed suggesters, as compared to the 6 second fixed timein. The table is sorted by the number of parks in descending order. SWPeriodFrequency was run with a minimum parktime value of 30 seconds, and all of the Proposer implementations were configured with a cook time of 5 minutes. All of these runs covered 654,664 seconds (about 182 hours) of recorded hard drive runtime. Hard drive lifetime was estimated by assuming that the frequency of head parking would be about constant in the drive's lifetime, that the drive can perform 600,000 park/unpark cycles before there is a risk of failure, and that the drive is in operation during about a third of a day.

of parks reduces the total park time, and this correlation must be accounted for in any comparison. To deal with this, we need to establish a common scale that considers the effects of the disk access pattern and the allowed number of parks.

One approach is to consider the FIXED TIMEIN PARK TIME. By appropriate processing of the disk log, it is possible to determine the VIRTUAL TIMEIN of each algorithm, the fixed timein value that would have caused the corresponding number of parks. We can then determine the total parktime that the virtual timein would have led to. This will allow us to do some rough comparisons of the efficiencies our new algorithms to that of the fixed timein.

Another metric worth considering is IDEAL PARK TIME. Imagine that we had a perfect, magical park decision algorithm that could look into the future, and assume that we allowed this algorithm to perform n parks during a certain interval of time. What decisions would it make? The answer is straightforward; it would park right at the beginning of each of the n longest idle periods.

It's easy to see why this is the case. First, it's clear that we should try to park right at the beginning of the periods; if we are going to park during a certain period, we maximize the time parked by staying parked during the entire period.

We can also show, with great certainty, that we should park during the longest idle periods. Let I be the set of all idle periods in the given time interval, M the set of the n largest idle periods, and P the set of idle periods in

which we should be parked. Note that the total park time is $\sum P$, the sum of all of the lengths of the idle periods in P. Also note that $|P| \leq n$; that is, the number of idle periods we park in is limited to the number of parks that we are allowed to do.

Suppose (for contradiction) that there is an idle period $x \in P$ such that $x \notin M$. Then, there must be some $y \in M$ where $y \notin P$ (otherwise, P would contain more than $|M| = n$ elements, which is not allowed). Since $x \notin M$ and $y \in M$, y must be longer than x; therefore, we can increase $\sum P$ by replacing $x \in P$ with y; so x really shouldn't be in P. Therefore, we have $P \subseteq M$.

Finally, assume (for contradiction) that there is an element $z \in M$ where $z \notin P$. Since $P \subseteq M$, we can say that $|P| < n$ - that is, we have "room" for more parks. We can increase $\sum P$ by putting z into P; thus, we really should ensure that $M \subseteq P$. Combined with $P \subseteq M$, this means we have $P = M$ - that is, we should park during and only during the n longest idle periods.

No algorithm we develop is going to be made of magic. It will not have information about the future, and so it most likely will make mistakes. Thus, we cannot expect any real algorithm to achieve ideal park time. However, the concept remains useful for much the same reason that the traditional engineering concept of efficiency is useful; it allows us to gain a sense of how much room there is for improvement, and perhaps shed light on the sources of inefficiency.

Table 2 on the following page compares the performances of our new algorithms. They fared about as well as or a little better than the fixed timein mechanism. The Proposer algorithm tended to perform better; the most likely explanation is that the large bursts of very short idle periods flooded the history stored by SWPeriodFrequency, thus harming its performance.

The more surprising results come from comparisons with ideal parktime. In terms of efficiency, all of the new algorithms perform *much* worse than does the original 6 second timein. It seems that, in general, efficiency decreases dramatically as the number of parks increases. This is even true of the fixed timein, as Figure 6 on page 12 demonstrates.

Since we want to reduce the number of parks without losing the benefits of head parking, we need to address this inefficiency. Since all three algorithms show very similar efficiencies, and since SWPeriodFrequency and Proposer algorithms are essentially designed to calculate a reasonable timein value, it's reasonable to assume that they all have the same source of inefficiency. Thus, we will consider only the fixed timein and assume that the timein has been set by some external (and possibly automated) mechanism. Assume a fixed timein of n seconds. We know that this will cause a park in every idle period that is longer than n seconds; let m denote the number of such periods. So far, this matches the ideal behavior in that we always park in the m longest periods. However, we *don't* park right at the beginning of these periods; rather, we wait n seconds before parking. Since we have this delay with every park, we end up wasting $n \times m$ seconds of potential park time. As long as we use timeins, we cannot avoid this waste - and with my system at least, trying to reduce m results in a lot of waste.

Description	Total park time	Number parks	Virtual fixed timein	Fixed timein park time		Ideal park time	
				Total	% achieved	Total	% achieved
Original (fixed timein of 6 seconds)	481,110	16,509	6.00	481,110	100.00%	580,165	82.00%
Proposer (min parktime = 10 seconds)	354,247	9,822	18.54	323,041	110.00%	505,144	70.00%
SWPeriodFrequency	299,083	8,987	20.33	305,977	98.00%	488,664	61.00%
Proposer (min parktime = 15 seconds)	297,934	7,272	25.69	261,568	114.00%	448,408	66.00%
Proposer (min parktime = 30 seconds)	210,081	4,127	33.94	212,219	99.00%	352,294	60.00%

Table 2: The park-time performances of the original 6 second fixed timein mechanism, compared with the proposals. The new algorithms seem to be at least on par with fixed timeins, but the ideal park time numbers reveal gaping inefficiencies in all three algorithms.
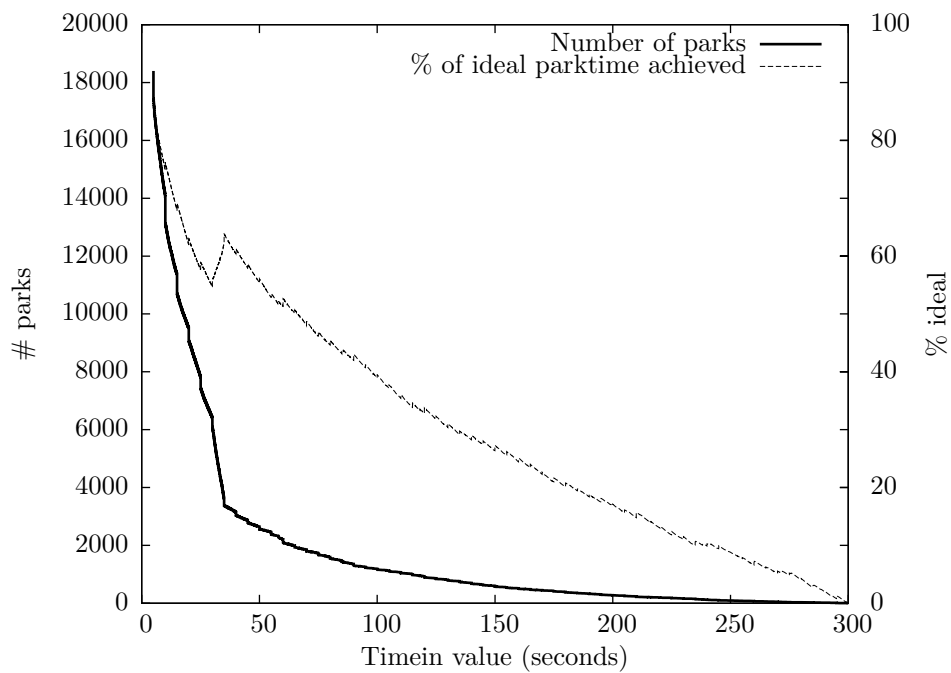
Figure 6: The behavior of the fixed timein algorithm for different timein values, using the collected disk log. As the timein increases, the number of parks decreases (unsurprisingly). However, with this disk log, we also see a dramatic fall in efficiency (as measured by the fraction of ideal parktime achieved).

Conclusions

Out of the three decision algorithms considered - the original fixed timein, SW-PeriodFrequency, and Proposer - it seems that the Proposer (combined with a Budgeter as a relief valve) is the best option. It controls the frequency of parks, and at least with the disk log collected, it is the most effective (by a small margin) at maximizing the total time spent parked. Further, it is very simple to implement, placing low demands on whatever system component it is implemented in.

However, there is much that should be done before we make a push to get this implemented. All testing was done on only one data set from only one user from only one operating system; it is not at all clear that the Proposer will perform as well under different conditions; more data collection and testing is imperative.

Furthermore, power usage implications need to be considered more carefully. While some power is saved when the heads are parked, this is somewhat offset by the power used by the actual park/unpark operation. More data must be collected on the actual power usage involved, as this may have significant implications on how a park decision algorithm should be designed and/or configured.

Finally, there is the problem of efficiency. In terms of maximizing total parktime, the Proposer is quite far from ideal, and it seems that this is due to the use of a timein-based design. It is likely that we have already taken timeins as far as they can go, and alternatives should be investigated. The algorithms presented so far assume that each idle period is independent, but future algorithms might try to find patterns in disk usage, perhaps even monitoring individual processes. Locality of reference has not been considered, but this principle gives us a great deal of information about disk accesses, and a decision algorithm could take advantage of this information.

But as crude as the Proposer-Budgeter combination is, it is still an unquestionable improvement over the fixed timein. Fixed timeins are blind and sometimes dangerous, and their replacement is possible and very necessary. There is no excuse for their continued use.