Saving Energy by Coordinating Hard Disk Accesses

"Studienarbeit" in Computer Science

written by
Björn Beutel
born 26th July, 1969 in Mainz

Department of Computer Science
(Distributed Systems and Operating Systems)
University of Erlangen-Nürnberg

Advisors: **Dr.-Ing. Frank Bellosa**
Prof. Dr. rer. nat. Fridolin Hofmann

Begin: 16th December, 2001
Submission: 17th April, 2002

Contents

Chapter 1

Introduction

The last few years have seen a significant rise in the number of mobile computers like laptops or PDAs. Advantages in chip technology have widened the range of their applications: mobile WWW browsers, audio players or mobile phones with lots of additional features were not available until recently. And this trend will probably continue for the next years. Mobile video players or video phones may be future applications.

A mobile computer that deserves its name must be independent of external power supply for a reasonable time period. To that aim, battery packs are used, which are, of course, very limited in their capacity. To extend a mobile computer's uptime, all its components should be investigated for ways to reduce their power consumptions. There are two main types such measures:

1. In the design process of a mobile computer, components should be selected that closely match the requirements. This is rather easy if the mobile device is designed for one single purpose. Oversized general-purpose processors, for example, are not only expensive, but their consumption might be more than an order of magnitude higher than the consumption of a special-purpose processor.

2. A mobile computer should be able to adapt its power requirements to the current workload.

   Newer mobile processors, for example, may reduce their clock rate and voltage to adapt for changing performance demands, reducing their power consumption.

   If a laptop is unused for some time, it may switch off some of its components like the display or the hard disk.

   Part of the electric circuitry, like RAM or the processor may be switched off after saving its state on the hard disk.

Together with the processor and the display, the hard disk drive is a mobile computer's main energy drain. In 1998, Jacob Lorch and Alan Smith [11] stated:

> Later models of portable computers seem to spend a greater percentage of their power consumption on the hard disk than earlier models. Presumably, this is because later models have substantial relative savings in other components' power

> but not as much savings in hard disk power. These forecasts suggest that as time progresses, power-saving techniques might become more important for the display and hard disk [...]

The present paper will focus on power-saving strategies for hard disks which may be implemented in an operating system.

Chapter 2 presents the basics of hard disk consumption reduction. A drive's power consumption depends on the pattern of the read/write accesses it has to serve. Additionally, modern hard disks offer *power saving modes* which may be activated by the drive itself or by software.

The operating system can control disk accesses and power modes, considering information from different sources, particularly from the *device driver*, the *file system* and *applications*. In chapter 3, I will present an integrated concept, called Coop-I/O, that uses information from all these sources. It consists of the following components:

1. A power mode control that uses an adaptive algorithm to shut down the disk when it is not needed.
2. An update policy that is trimmed to reduce the number of write access bundles.
3. Cooperative file operations (*open*, *read* and *write*) that may be delayed or cancelled when activating the hard disk immediately is too energy-expensive. These operations can be used by applications.

Chapter 4 describes an exemplary implementation of that concept which is based on the Linux kernel, version 2.4.10.

Chapter 5 reports the energy-saving results when testing that implementation, using real-world applications as well as parameterised tests.

Chapter 6 gives hints how applications may use the cooperative file operations. Possible future work is also suggested.

The Coop-I/O concept may be embedded in a larger energy-saving concept that encompasses all components of a computer system. For example, Heng Zeng, Xiaobo Fan, Carla Ellis, Alvin Lebeck and Amin Vahdat [10] have proposed to introduce an energy account system that "[...] unifies energy accounting over diverse hardware components and enables fair allocation of available energy among applications." A similar system has been suggested by Gaurav Banga, Peter Druschel and Jeffrey Mogul [9]. The present work is complementary to theirs and should fit in nicely.

Chapter 2

Background

Operating systems may reduce a hard disk's power consumption by switching between the disk's power-saving modes.

2.1 Power-Saving Modes

Modern hard disks make use of several modes that are associated with different levels of power consumption. The ATA standard, also known as IDE, defines three power saving modes [5]:

Idle: The hard disk is rotating and the hard disk interface is active. Typical power consumption for new-generation mobile hard disks is 0.75–2 W.

Standby: The hard disk spindle motor is off, but the hard disk interface is active. Typical power consumption is 0.25 W.

Sleep: The hard disk spindle motor is on, and the hard disk interface is inactive. It can only be activated by a reset command. Typical power consumption is 0.1 W.

The ATA standard furthermore defines the *active* mode, in which the hard disk resides when reading, writing or seeking. I will subsume standby mode and sleep mode as *resting* modes; active mode and idle mode will be subsumed as *running* modes.

Some of the mobile hard disks on market split up the idle mode into 2–3 modes that have different power consumption and need different time intervals to go back to active mode. They switch autonomously between these power saving modes, using fixed time-outs or adaptive algorithms.

A non-negligible amount of time and energy is needed to enter and leave resting modes. Therefore, resting modes should be activated as soon as possible, and the resting period should persist as long as possible. Of course, entering a resting mode is only worthwile if the interval up to the next disk access will be long enough. The minimum interval between two disk accesses for which switching pays off is called the *break-even* time. It only depends on the characteristics of the hard disk device.

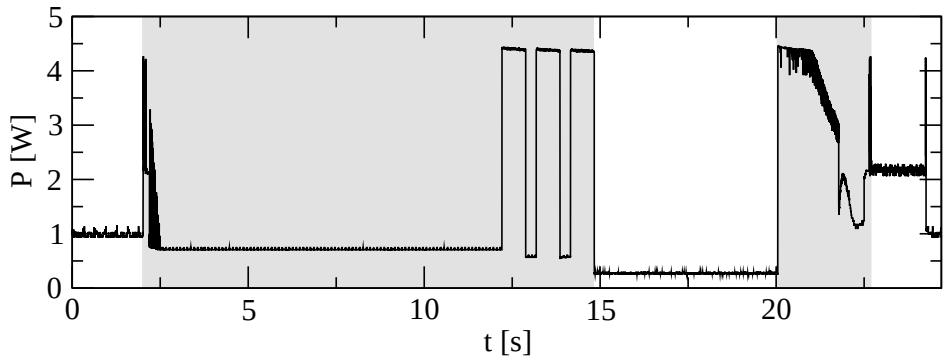

Figure 2.1: Power consumption of the IBM DCRA-22160 during an idle–standby–idle turnaround.

2.2 The IBM DCRA-22160 Drive

For all power measurements, I have used a 2.5-inch IBM DCRA-22160 drive, sold as Travelstar 2XP, with the following characteristics, as reported by its manufacturer [4]:

Storage capacity	2160 MB
Number of disks	3
Rotational Speed	4900 rpm
Interface	ATA-3
Dissipation in Idle mode	1.1 W
Dissipation in Standby mode	0.3 W
Dissipation in Sleep mode	0.1 W
Time from Standby to Idle	2.3 s typical, 9.5 s maximal
Time from Idle to Standby	1.7 s typical, 5.0 s maximal

As the measurements have shown, the time from idle to standby, as reported by IBM, only comprehends the time it takes to start or stop the spindle motor. The time to lock the heads is not included.

The DCRA-22160 uses adaptive power management. It conforms to the ATA standard, but it splits up the idle mode into two submodes: *performance idle* and *low power idle* [4] [8].

> In performance Idle mode, all electronic components remain powered and full frequency servo remains operational. [...]
> In low power idle mode, additional electronics are powered off, and the head is parked near the mid-diameter of the disk without servoing. [...]
> The transition time is dynamically managed by users recent access pattern, instead of fixed times. [...] [4]

Figure 2.1 shows the power consumption of the IBM DCRA-22160 during an idle–standby–idle turnaround. The dissipation changes in reaction to the changing operating modes:

$t = 0$ s: The hard disk is in low power idle mode and dissipates about 1 W.

$t = 2$ s: The disk receives a shut-down command. The big shaded region in figure 2.1 is the shut-down interval. First, the disk consumes up to 4 W to stop the spindle motor, but only for a very short time period. Then, the disk stays in a *pre-standby* state, which is not documented in the drive's technical documentation. The disk draws about 0.75 W. This state lasts for about 9 s. Finally, the power consumption reaches three peaks, each one continuing for 0.5–1 s. This is caused by the head lock mechanism.

$t = 14.8$ s: After locking the heads, the disk has reached standby mode, and power consumption falls to about 0.275 W.

$t = 20$ s: The drive receives a write command and starts to spin up. The smaller shaded region shows the hard disk switching from standby mode to active mode. Starting the spindle motor is quite expensive, energetically. After 2.5 s, the disk has run up and may execute read or write accesses.

$t = 22.5$ s: In the test scenario, only a single block gets written. Then, the disk stays in performance idle mode for about 2 s, dissipating about 2 W. Finally, after a short power peak, it switches to low power idle mode.

The duration of performance idle mode is controlled by the disk itself and is variable. The time algorithm is not documented. I ran this "turnaround test" several times and got results where the disk switched to low power idle immediately, and others where the disk stayed in performance idle mode for 4 s.

Ideally, the disk should go to standby mode immediately at the beginning of a standby period. Of course, when a disk operation has been processed, we do not know whether we are at the beginning of a standby period or not. So what we need is an approximative algorithm that estimates whether it pays off to shut down the hard disk. The easiest approximation is to shut down when a fixed time period after the last disk access has elapsed. The ATA-5 standard supports such a fixed time-out. The standby timer is controlled by the drive without intervention from the operating system. The user can set the time-out according to her needs. Linux supports this via the *hdparm* command.

2.3 The Break-Even Period

The hard disk can save an optimum of energy if it goes to sleep whenever the hard disk will be not accessed for the so-called *break-even period* at least. This is the minimum time period between two disk accesses when it pays off to change to standby mode (and back). A time interval without any disk access that is longer than the break-even period is called a *standby period*.

Let us suppose we have the following values for a specific hard disk drive:

P_i: The power consumption in idle mode.
P_s: The power consumption in standby mode.
E_{is}: The energy needed to change from idle mode to standby mode.
t_{is}: The time needed to change from idle mode to standby mode.
E_{si}: The energy needed to change from standby mode to idle mode.
t_{si}: The time needed to change from standby mode to idle mode.

For an interval $t \geq t_{is} + t_{si}$ in which no hard disk access takes place, we are interested in the following values:

$E_i(t)$: The energy needed to stay in idle mode.
Since the power consumption is (nearly) constant in idle mode, we have $E_i(t) = tP_i$.

$E_s(t)$: The energy needed to switch to standby mode at the beginning of the interval, to stay in standby mode, and to switch back to idle mode at the end of the interval.
Since the standby interval is $t - t_{is} - t_{si}$ and power consumption is constant in this interval, we have $E_s(t) = E_{is} + (t - t_{is} - t_{si})P_s + E_{si}$.

t_{be}: The break-even time, which is the time for which $E_s(t_{be}) = E_i(t_{be})$.

Inserting into the last equation and resolving for the break-even time yields:

$$t_{be} = \frac{E_{is} - (t_{is} + t_{si})P_s + E_{si}}{P_i - P_s}$$

For the IBM DCRA-22160 hard disk, I measured the following values, using the test environment described in section 5.1:

$$\begin{aligned} P_i &= 0.945 \text{ W}, & P_s &= 0.275 \text{ W} \\ t_{is} &= 12.8 \text{ s}, & t_{si} &= 2.5 \text{ s} \\ E_{is} &= 16.8 \text{ J}, & E_{si} &= 8.0 \text{ J} \end{aligned}$$

So the break-even time for that hard disk is 30.7 s. Note that this value is device-specific. Yung-Hsiang Lu and Giovanni De Micheli [2] report break-even times of 6.39 s for a Fuijitsu MHF 2043AT disk and 35.0 s for an Hitachi DK23AA-60 drive.

2.4 Existing Energy-Saving Strategies

Up to now, the following strategies have been used or researched to use hard disk mode switching to reduce energy consumption:

- The hard disk device can switch to standby mode autonomously. For this aim, the ATA standard provides a command to set a shut-down time-out. If the disk is idle for longer than the time-out interval, it will go to standby mode automatically.

- The operating system can switch a hard disk to a resting mode using an ATA command. Normally, this is done either on a fixed time-out basis or because a user action has happened, such as closing the laptop or explicitly activating power saving mode.

- Applications can access the disk as rarely as possible by buffering the data they have read or will write. This strategy is used, for example, by most MiniDisc players that need about 2 s to read an audio data block. The playing time for such a block is 10 s. In the meantime, the MiniDisc spindle motor can rest.

2.5 A Cooperative Approach

In his pre-master's thesis [3], Steffen Meyer has implemented a scheme that combines energy-accounting with the concept of a *market place*. He assumes that energy is a limited resource, as is true for battery-driven laptops or PDAs. For every process, the operating system keeps an energy account, which holds the amount of energy the process is allowed to spend for hard disk accesses. If a process creates a child, the parent can determine the amount of energy it will transfer to the child. A file will also get its own account when being opened; its initial amount will be transferred from the account of the creating process.

The system calls for file operations are substituted by energy-aware variants that have two additional parameters, namely the amount of energy they may spend at most, and a time-out. Every process that requests a hard disk operation has to go to the market place before the operation is actually executed. Here, the process' energy bid is compared with the estimated amount of energy it takes to execute the operation. If the energy bid does not suffice, the operation will be delayed until the time-out is reached. Meanwhile, bids for disk accesses of other processes can arrive at the market place. If the sum of the bids reaches the amount of energy needed, all operations are executed and the energy for the disk access is taken from the files' accounts in the order the bids have reached the market place.

Steffen Mayer implemented this concept on a Minix operating system. Since "real world applications" are rare for Minix, the test programs were simulating the behaviour of such applications. The concept of cooperative processes is appealing, and it was proven to save energy. But the current implementation imposes many decisions on the user's programs. An application has to decide:

- How much energy it should bid for a certain disk access.
- How long it should wait until raising the bid or resigning.
- How much it should raise its bid.
- How much energy it will transfer to a child process when creating one.

Since energy consumption for hard disk accesses depends on the actual drive, the application has to deal with aspects of specific devices. Furthermore, if an application's energy account runs out, the application's file operations will almost certainly be blocked.

Chapter 3

An Integrated Power-Saving Concept

In this chapter, I will present an integrated concept to coordinate hard disk accesses which I call *Coop-I/O*. It consists of three major parts:

- The operating system monitors and controls the hard disk modes; it will switch a hard disk to standby mode when it assumes that the drive is in a standby period. This is done by a simple adaptive algorithm called *device-dependent timeout with early shut-down* (DDT/ES).
- The operating system uses an *update* policy that is trimmed to save energy by the following means:
  1. Each drive is updated separately. If a drive is updated, all its dirty buffers are written back.
  2. Dirty buffers are written back preferably when another disk access happens.
  3. When the operating system decides to shut down a hard disk drive, it will previously write back all dirty buffers of that drive.
- New cooperative file operations *open_coop()*, *read_coop()* and *write_coop()* can wait until another processes access the hard disk. They all get a *delay* and a *cancel flag* as additional parameters. If a file operation needs to access a hard disk, and that hard disk is in standby mode, the operation will be suspended until either the disk drive has run up by another I/O request or the delay has elapsed. When the delay has elapsed and the file operation *cancel flag* is set, the operation will be cancelled. In all other cases, it will finally be executed.

While working out this concept, I had to make certain conceptual decisions that I will present and reason on in the following sections.

3.1 Use Sleep mode or Standby mode?

Sleep mode needs less power (0.1 W on a IBM DCRA-22160, compared to 0.275 W in standby mode), but it deactivates the hard disk interface. So every hard disk access in the operating system has to check whether the hard disk is currently in sleep mode and wake it

DDT Policy

Shut down the hard disk if $t_{la} + t_{be} \leq t$.

The variables have the following meanings:
t: The current time.
t_{la}: The last hard disk access.
t_{be}: The break-even period.

Figure 3.1: Definition of the DDT Policy

up if necessary. Sleep mode can only be aborted by a software reset. After a software reset, the disk is either in standby mode or in idle mode, depending on the hard disk model. If the time for a software reset takes longer than just running up from standby mode, it is preferable to use standby mode instead.

Since the reset time depends heavily on the drive type, and since the device is not signalling via interrupt when the drive is ready after a reset, I decided to use standby mode as the only resting mode.

3.2 When should the Hard Disk Shut Down?

Several shut-down policies have been suggested [1] [7]. Yung-Hsiang Lu and Giovanni De Micheli [2] have compared some of them to the ideal "oracle" policy that shuts down the hard disk at the beginning of every standby period and runs it up again so that it is just ready at the end of that standby period. Their criteria were power consumption, the number of shutdowns (the less the better), percentage of incorrect shutdowns, interactive performance, and the algorithm's memory and computation requirements.

The DDT Policy

As Lu and De Micheli have found out, the device-dependent time-out policy (*DDT*), which uses the break-even time of a drive as its time-out parameter, has good power-saving facilities, and its algorithm is fast, simple and storage-efficient. DDT uses the break-even period as spin-down timeout. This policy is defined in figure 3.1.

The DDT/ES Policy

For scenarios with very short or regular disk accesses, this algorithm can be improved: Assume that disk accesses are bundled in time intervals of nearly the same length, separated by standby periods. This regularity may be exploited to shut down the disk *before* the break-even interval that started with the last disk access has elapsed.

DDT/ES Policy

Shut down the hard disk if

$$t_{la} + t_{be} \leq t$$

or

$$t_{fa} + t_{lb} \leq t \leq t_{fa} + t_{lb} + t_1 \text{ and } t_{la} + t_2 \leq t \ .$$

The variables have the following meanings:

- t: The current time.
- t_{lb}: The length of the last busy interval.
- t_{fa}: The first access in the current busy interval.
- t_{la}: The last hard disk access.
- t_{be}: The break-even period.
- t_1: Tolerance when comparing last and current busy interval.
- t_2: Small timeout to detect the end of a busy interval.

Figure 3.2: Definition of the DDT/ES Policy

In chapter 2, we have seen that, ideally, the disk should shut down at the beginning of every standby period. An intervall between two such periods will be called a *busy period*. It starts and ends with a disk access.

The idea of the modified DDT policy, which I call *device-dependent time-out with early shut-down* (DDT/ES), is the following: We use the original DDT policy. Additionally, when the length of the current busy interval *so far* is only somewhat longer than the length of the last busy interval and we guess that the current busy interval has ended, then we shut down the hard disk immediately. We can guess that the current busy interval has ended if the disk is idle for a small timeout of 1–2 s.

This modification is based on the observation that consecutive busy periods often have similar lengths, for example if disk accesses are rare or regular. Imagine that a disk is active for 1–2 s in intervals of 1–5 minutes. The DDT policy would spend the break-even interval after disk activity in idle mode until switching to standby mode. The DDT/ES policy will switch to standby mode already after 3–4 s. The algorithm adapts very fast to changing busy intervals. It is defined in figure 3.2.

The hard disk driver has to keep track of t_{lb}, t_{fa}, and t_{la}. Furthermore, it should know t_{be} for the drives under its control.

The effects of DDT/ES on the disk mode switching for some test applications are shown in chapter 5.

3.3 Bundling Write Accesses

After a disk access, the hard disk stays in active or performance idle mode for some time. In these modes, energy consumption is higher than in low power idle mode or in standby mode.

So write accesses should be bundled to maximise the time the hard disk spends in energy saving modes.

For efficiency reasons, the Linux operating system, like most modern operating systems, does not execute write accesses immediately. Instead, it writes the data into the *block buffer* in main memory and marks the buffer as dirty. The buffer will be written later to disk, when one of the following conditions applies:

- An explicit update command like *sync()* or *fsync()* forces the system to write back the buffers of a filesystem or a file, respectively.
- The buffer is dirty for a certain period (30 s for Linux) and is written back to prevent data loss in case of a crash. That period is called *dirty buffer lifespan*. This is the most frequent cause for writing back when there is few I/O traffic. In Linux, a dirty buffer whose lifespan has elapsed is not written back immediately, but when it is found by the update kernel task, which wakes up every 5 s. This means a buffer may be dirty for 30–35 s.
- A certain percentage (30% in Linux) of block buffers is dirty. To avoid I/O jams, some of them are written back. This is the most frequent cause for dirty buffer updates when there is heavy I/O traffic.
- The system needs main memory and writes back some dirty buffers that it will reclaim as free memory later.

This policy is not optimal to save energy. Imagine a Linux user process that constantly writes to a file with a rate of one disk block per second, for example a voice recorder. In every second, a new buffer gets dirty. After some time, the update kernel task finds old dirty blocks that it writes back. This will happen each time the update kernel task wakes up again, so the disk has to write in intervals of 5 s although the maximum write-back time for a buffer is 30 s. This may prevent standby periods.

File-Dependent Update Intervals

To make updates less frequent, one could introduce different file categories with different update timings:

- Temporary files do not need to be written to hard disk, since they are of no use after a system crash. But since they may share some block buffers with other files (i-node buffers, bitmap buffers, i-node bitmap buffers, directory entries) that must be written back to disk, they will be not as efficient as a filesystem solely dedicated for disk based temporary files. In Linux 2.5, such a file system will presumably be introduced. That would be useful for data base servers that need huge intermediate files.
- A file that is only of use if it is complete could be handled like this: First, treat the file like a temporary file. If the file has been written completely, the user process either calls *fsync()* to update synchronously, or it calls a derivation of *fsync()* that does not wait for the update to complete, but that guarantees that the update will complete in a time period given as argument.

- A “regular” file is updated in regular periods. Otherwise too much information could be lost when the system crashes. In current file systems, all dirty buffers get the same update interval. Setting the update time on a per-file basis could be an interesting idea.

I will not implement any of these file categories since it would be difficult to mix files with different update timings on the same file system. A mixture of different update policies on the same drive could perhaps even worsen the energetic behaviour.

Drive-Specific Cooperative Update

I will use another policy that is called *Drive-Specific Cooperative Update*, because it updates each drive independently of all others and is preferably executed when another disk access takes place. It is composed of four strategies:

Write back all buffers. We write back all dirty buffers instead of only the oldest ones, so we have to update at most once per *dirty buffer lifespan*, which is 60 s in our implementation. Original Linux may update some buffers each time when its update task wakes up, so it may update every 5 s.

Of course, there might be buffers which just got dirty and which are currently busy, i.e. under frequent modification by, for example, numerous small sequential write system calls. On the one hand, these buffers could be written back later, avoiding a redundant write access. This is what Linux currently does. On the other hand, being young is only a very vague symptom of a busy buffer: Many buffers are only modified once over a long period, and older dirty buffers may be busy as well. Besides, an additional write access has only marginal costs if it is bundled with other accesses.

Update each drive separately. Updates are treated separately for each drive. This will not compromise file system consistency and it may increase the update interval for a single drive even more. It may also balance system I/O load since different drives will probably updated at different times. Besides, this is a prerequisite for cooperative updates.

For each drive, we have to watch the age of the oldest dirty buffer. If it has reached the dirty buffer lifespan, we write back all buffers for that drive.

Update cooperatively. The operating system can choose, within the *dirty buffer lifespan*, when it will write back the buffers for a drive. We exploit this to update cooperatively: When half of the lifespan has passed, we wait for a disk access on that drive. This or the expiration of the full lifespan will trigger the update process.

By attaching to a disk access that has to happen anyway, we can update at very little costs.

Update on shut-down. If the operating system has decided to shut down a drive, it first writes back all dirty buffers that contain blocks of that drive. This minimises the risk that the disk has to spin up again soon solely because there are some old dirty buffers that must be updated.

3.4 Bundling Read Accesses

What is good for write accesses is not necessarily good for read accesses. When an application reads some data from disk, it probably needs that data for further processing. If an interactive program had to wait because the operating system delays a read operation to save energy, the user would irritated. Furthermore, the operating system can only delay one file operation per process (disregarding asynchronous file operations), so the number of accesses to be delayed simultaneously is quite limited.

Hence, read accesses should only be delayed if the application permits. For that aim, I will introduce cooperative system calls.

3.5 Cooperative Read Operations

The essential file operations in most operating systems are the *open()*, the *read()* and the *write()* system call. The system calls *close()* and *lseek()* usually do not access the disk directly, but operate on data in main memory. So we have to introduce three cooperative variants: *open_coop()*, *read_coop()* and *write_coop()*.

We assume that a file is situated on a single disk. This may not be true for a RAID (Redundant Array if Inexpensive Disks) or if a file system is spread across multiple partitions via *logical volume management* (LVM).

A *read_coop()* operation is quite straightforward. It behaves like the ordinary read operation most of the time, working on data blocks and indirect blocks. The i-node data of an open file is already buffered, so we need not care about it. If the operation needs the data of a hard disk block, it will probably find it in main memory already, residing in a block buffer. If it needs a block that is not buffered in memory, the operation must check if the relative hard disk is in running mode. If it is, the read access can take place immediately. If not, the operation has to block itself until either the hard disk is spinning up or until the time-out has elapsed. If the time-out has elapsed, the operation is to be cancelled if the *cancel flag* was set. In all other cases, the operation will proceed.

The same strategy applies for *open_coop()*, provided that the operation will not create a new file or truncate an existing one. Note that the operation may access several disk drives via mount points and symbolic links when walking along the file name's path. This is handled properly by our approach.

3.6 Cooperative Write Operations

When a cooperative write operation might be cancelled to save energy, it is essential to assure the consistency of the file system. Imagine a write operation that will enlarge the file so that it needs a new block for its data. In a traditional Unix file system, this will cause the following write accesses to take place (not necessarily in this order):

- In the block bitmap, a free block is found and marked as occupied.

- The new data block is registered in the block list of the i-node or in an indirect block that is owned by that i-node. If a new indirect block is needed, the same steps are taken as for a new data block.
- The new data block is written.
- The block containing the i-node is written to reflect the new file length as well as the new *m-time*.

If, for example, the operating system would only modify the block bitmap and cancel afterwards, a data block would be marked as occupied although it is not used. Of course, a file system check would reclaim that block, but if this scenario recurs many times without any file system checks in between, the file system could run out of blocks. Other combinations of committed and aborted block writes could also corrupt the file system.

To solve this problem, I have considered three strategies. All these strategies may be explained in terms of *transactional operations*. A transactional write operation can be in one of three states:

Preliminary: The effects of the write operation are not yet visible to other processes.

Committed: The write operation cannot be cancelled any more and its effects are visible to other processes. In contrast to database semantics, I will not necessarily assume that the operation is completed. This deviation reflects the semantics of Unix file operations.

Aborted: The effects of the write operation will never be visible to other processes and are going to be reversed.

Private Buffer State

The *private buffer state* strategy uses a new state for block buffers, the *private state*. A private buffer is owned by the process that has invoked the write operation. It must not be written to the hard disk as long as it is private. A buffer can only be private during the execution of a preliminary write operation.

- If a process, while executing a preliminary write, creates or modifies a buffer, the buffer will be marked as private and will be owned by that process. If the buffer is already dirty, it has to be written back to disk before marking it as private. Alternatively, the process can decide to delay the modification or to commit or abort the transactional operation.
- If another process is going to read or write a private buffer, the owner process of that buffer must either commit or abort the transactional write immediately, or the other process must be suspended until the owner process' write has been committed or aborted.
- If a process *commits* a transactional write, all its private buffers are changed to the non-private state.
- If a process *aborts* a transactional write, all its private buffers must be deleted.

Shadow Buffers

The *shadow buffer* strategy is similar to the first, but instead of an additional buffer state, it uses a *shadow buffer*, which is a private block buffer that may coexist with the normal, public buffer of that block. No block can have more than one shadow buffer. Shadow buffers only exist during a preliminary write. A shadow buffer is owned by the process that has created it.

- If a preliminary write operation is going to modify a block without a shadow buffer, a shadow-buffer copy of that block will be created.
- If another process is going to modify a block that has a shadow buffer, then either the owner process must commit or abort its write operation immediately or the other process must be suspended until the write operation of the owner process has been committed or aborted.
- If another process is going to read a block that has a shadow buffer, it may simply ignore the shadow buffer.
- If a process *commits* a write operation, all its shadow buffers become public buffers, replacing the old public buffers for these blocks.
- If a process *aborts* a write operation, all its shadow buffers have to be deleted.

Note that "modifying a block" starts when the block content is examined for the purpose of changing it. This is called an *update*. For example, if a file needs a new data block, a bitmap block must be searched for a free block and the bit representing the new data block must be set. The search and the modification must be delayed if another process is already modifying that block.

Early Commit/Abort

A cooperative write operation is preliminary in the beginning. When the first modification of a block buffer is going to take place, the operation has to decide if it will commit or abort. Using this strategy, there is no need for private buffers or private buffer states. This strategy is called *early commit/abort*.

When reading a block, the *write_coop()* operation will wait for the drive the same way as the *read_coop()* operation does. But if it has to modify a buffer, it can exploit the fact that a modified buffer is not written back to disk immediately, but somewhen before its dirty buffer lifespan has expired:

Assume we are going to modify a buffer and the drive is in standby mode. If there is another dirty buffer for the same drive (or the buffer to be modified is already dirty), the drive must run up in the near future anyway to write back that buffer. So we can immediately modify our buffer at almost no cost: When the other buffer is updated to disk, our buffer will be updated in the same sweep, as described in section 3.3.

Therefore, a write to a block buffer should be only delayed as long as the drive is in standby mode *and* there are no dirty buffers for the relative drive. Since a write operation's first

buffer modification involves committing the operation, a write can be committed even when the hard disk is not running.

Unfortunately, this strategy may not be optimal in a scenario when a cancellable write operation is already committed although the drive is in standby mode. If the write operation subsequently has to read a block from disk, it may need to spin up the disk, although its cancel flag is set.

Fortunately, the energy waste in this case is not that big, since the hard disk would run up anyway in the near future to save the dirty buffers. Besides, if the file is written sequentially and has been modified previously, the blocks that are needed by the write operation are most probably already buffered.

Use Simplest Approach

For Coop-I/O, I have chosen the early commit/abort strategy, although non-optimal, because it allows write operations that have not yet been committed to be cancelled without much effort. The two other strategies need source code modifications in many more places – for example for file access via memory mapping – to prevent a mix of preliminary and effective data in block buffers. Changing the Ext2 file system file semantics to a transactional concept would be too time-consuming for a thesis. Some transactional file systems for Linux already exist, like Ext3, JBD and ReiserFS. It might be easier to adapt one of them.

An *open_coop()* operation that has to create or truncate a file uses the same strategy as *write_coop()*.

Chapter 4

Implementation

The concept presented in chapter 3 has been implemented in the SuSE derivative of the Linux Kernel, version 2.4.10. Originally, I modified the official kernel sources provided on `http://www.kernel.org`, but when I used the SuSE 7.3 distribution for testing, I had to realise that they do not cooperate properly.

The Linux kernel supports a variety of file systems via its internal *virtual file system* layer (VFS). Most of them are disk-based, but there are also network-based file systems like NFS, and special-purpose file systems like *proc*. Most Linux systems use the *Ext2* file system as their primary filesystem. It has been developed especially for Linux and has been optimised for speed. I chose to implement cooperative file operations in this file system only. Since the VFS and a particular file system closely interact, the VFS has also been changed in many places. As a consequence, adding cooperative file operations to another file system type should not be too hard.

Furthermore, the kernel supports several hard disk interfaces. The most popular one is the ATA interface. It is mostly called IDE in the Linux society, so I will use that term from now on. The SCSI interface is also supported, and there are several other hard disk drivers. I decided to implement power mode control only in the IDE hard disk driver.

The kernel changes can be divided into three parts:

- The IDE driver has been enhanced by a power mode control for hard disk drives, which includes the DDT/ES algorithm of section 3.2.
- The VFS and the Ext2 file system have been modified to support the drive-specific cooperative update policy of section 3.3. I have also introduced cooperative system calls using the concept of sections 3.5 and 3.6.
- The block device code, which is the glue between a particular block device driver and the file system, has been augmented to enable cooperation of the disk drivers' power mode control with the file system's update mechanism and the cooperative file operations.

```
#ifdef COOP_IO
static int write_some_buffers_coop (kdev_t first_dev, kdev_t last_dev)
#else
static int write_some_buffers (kdev_t dev)
#endif
{
  ...
#ifdef COOP_IO
  if (first_dev && bh->b_dev < first_dev || last_dev && bh->b_dev > last_dev)
#else
  if (dev && bh->b_dev != dev)
#endif
  ...
}

#ifdef COOP_IO
static inline int write_some_buffers (kdev_t dev)
{
  return write_some_buffers_coop (dev, dev);
}
#endif
```

Figure 4.1: A Cooperative Version of a Linux Function.

Practices

Since the changes are spread in source files that sum up to about 30 000 lines, I have marked all source code changes by enclosing them with "#ifdef COOP_IO" and "#endif" preprocessor directives, so they can be found more easily. In the file system code, I often had to write a cooperative version of a function. The cooperative version gets the name of the original function with the suffix "_coop". If the function is rather short, I have written a new cooperative version. If the function is lengthy, I have changed the name of the original function, added any necessary new parameters and changed the function body, and I have written a dummy version of the original function that calls the new cooperative version. Figure 4.1 shows an example.

The call graphs for some file system operations are quite extensive and the functions involved are spread across multiple files in multiple directories in the Linux source tree. So some of these call graphs are visualised in this paper for a better understanding. Only the calls that are relevant for our purposes are included.

When the VFS operates on an object (like an i-node or a super block) of an actual file system (like Ext2), it sometimes has to call a function that depends on the actual file system. To this end, the VFS uses a function pointer provided by the object it is working on. This is what is called a *method call* or *virtual function call* in object oriented programming. In the call graphs, a virtual function call is shown as a dotted line.

4.1 Power Mode Control for ATA/IDE Drives

Drive-Specific Information

The Linux IDE driver may control multiple hardware interfaces. Each hardware interface may be connected to a *master* device and a *slave* device via the same line. For each hard disk, the driver has a description that reflects the properties and state of that device. I have augmented that description by the following entries:

break_even_period: This is the device-specific break-even period needed for the DDT/ES algorithm. Since there is only one drive for which I could compute its break-even period, I have hard-coded it into the initialisation code. A full grown-driver would need a table with known hard disk drives and their break-even periods.

last_access, first_access, busy_period: These values are also needed for the DDT/ES algorithm.

power_mode: Since the power mode is checked very frequently, it will be remembered here.

new_power_mode: If a power mode change of the disk is requested, the new power mode is put here.

The IDE Power Task

A power mode switch might take rather a long time, since it may write all dirty buffers back to that drive, or it may execute an IDE command that actually changes the drive's mode, and wait for its completion. For that purpose, I have introduced a kernel thread called *idepower*, which serves all IDE drives.

The *idepower* thread normally sleeps and waits for a semaphore that signals that a power mode change has been requested. If some function wants to change the power mode, either explicitly or implicitly (for example, because it starts a disk access), it sets *new_power_mode* to the new power mode and increases the semaphore value. So the power task will wake up and search all IDE drives for a new value in *new_power_mode*. Then it emits a power mode command to the hard disk, changes its internal power state variables and waits for the semaphore again. Note that this scheme also works if several drives have to change their power modes simultaneously, since the semaphore's value always reflects the number of power mode changes that the power task still has to handle. When changing the power mode, the power task also informs the file system when dirty buffers must be written back or cooperative file operations that are blocked must be awoken.

But things are yet a little bit more difficult. The value of *new_power_mode* might be changed several times until the power task takes notice of it. The semaphore must only be increased for the first change. Besides, some functions, like disk accesses, change the power mode implicitly by emitting other IDE commands, so they must inform the power task. Finally, since *new_power_mode* might be changed in interrupt code, race conditions must be avoided. These difficulties are all handled in the power task and in the function that requests a new power mode. This function may set *new_power_mode* to one of the following values:

ide_power_standby: The drive's mode has changed to *standby*, and the power task should know that (not used).

ide_power_running: The drive's mode has changed to *running*, and the power task should know that.

ide_power_changed: The drive's mode may have changed, but the caller is not sure about it, so the power task has to find out. This must be used, for example, after sending an IDE reset command or an IDE raw command via *ioctl()*.

ide_power_go_to_standby: The power task shall emit an IDE command to switch to standby mode.

ide_power_go_to_running: The power task shall emit an IDE command to switch to idle mode.

There are two main reasons why a new power mode might be requested:

1. A hard disk access is sent to the device driver. This implicitly changes the disk's power mode to *running*.
2. The drive must be explicitly shut down as decided by the DDT/ES standby algorithm.

Besides, some special IDE commands leave the disk in an undefined power mode, so they request the power task to check. Finally, the new *ioctl()* command *HDIO_SET_POWER_MODE* may explicitly change the power mode.

Going to Standby

The DDT/ES algorithm, which depends on the time of the last disk access (see section 3.2), is implemented in *ide_check_for_standby()*, which is a timer-based function that is called once per second. Since disk accesses may be very frequent, this is more efficient than to use a dedicated timer for each drive that has to be restarted when a disk access has taken place. The standby function has to check every IDE drive for the standby trigger conditions of section 3.2 and shuts down the drive if the conditions apply.

The time information that is needed by the DDT/ES algorithm is updated by the function *ide_notify_access()*. This function is called for each disk access and it updates the time of the last access as well as the time of the first access in the current busy period and the length of the drive's last busy period. It also implicitly changes the power mode to *running*.

When the IDE power control, in response to a shut down request, causes the file system to write back dirty buffers, *ide_notify_access()* should not be executed, since this would restart the disk immediately. Furthermore, updating the time information would degrade the performance of the DDT/ES algorithm.

It is a little bit tricky to make *ide_notify_access()* not be executed in that context. If it were called by the IDE driver itself, we had no hint for the origin of the write request, since the request is processed in an interrupt handler and not in the context of the calling task. The file system could pass a shut-down indicator in the buffer head, but this is dangerous, since

other tasks might access (and even write to disk) the buffer at the same time. Alternatively, the block device interface could pass a shut-down indicator in the I/O request struct, but then the file system and the block device had to pass it across a long function call chain.

I decided to call *ide_notify_access()* from the block device interface where all the file system's disk accesses have to go through. We even do not need to pass a shut-down indicator from the file system: *ide_notify_access()* can check whether it is called in the power task's context. If it is, it must not execute, since the power task only writes blocks when shutting down the drive.

4.2 Disk Update

In section 3.3, I have introduced the drive-specific cooperative update policy. The file system does not know about drives, it only deals with device numbers. To implement the update policy, I had to introduce a mapping of device numbers to drives as part of the file system. For each drive, the file system must also keep track of the number of dirty buffers and of the time when the oldest dirty buffer got dirty. I have implemented the mapping as a list of drives. Since the list must be searched for every hard disk access, the run-time costs must be modest. In fact they are, because the list is very short.

The device numbers of the partitions on a drive are stored as a range from *first_dev* to *last_dev*. This works, since the device numbers that belong to one drive are contiguous. It would not be sufficient to store the first device number only, since the range length depends on the device driver. The SCSI driver, for example, uses 4 bits of the device number for the partition number, while the IDE driver uses 6 bits.

The following functions deal with the file system's drive list:

fs_add_drive (kdev_t first_dev, kdev_t last_dev): Registers a drive that maps the device numbers from *first_dev* to *last_dev*. This is called from every device driver that supports Coop-I/O.

fs_remove_drive (kdev_t first_dev): Unregisters a drive that maps the device number range that starts at *first_dev*.

get_dirty_buffer_count (kdev_t dev): Get the number of dirty buffers for the drive that contains partition *dev*. This may be called by a cooperative file operation to check if it should block or cancel. See section 4.3 for more information.

change_dirty_buffer_count (kdev_t dev, int increment): Adds *increment* to the number of dirty buffers for the drive that contains partition *dev*. The *increment* may be negative. This function is called by the buffer manager. If the number of dirty buffers changes from 0 to a positive value, all cooperative file operations that are blocked are woken up. See section 4.3 for more information.

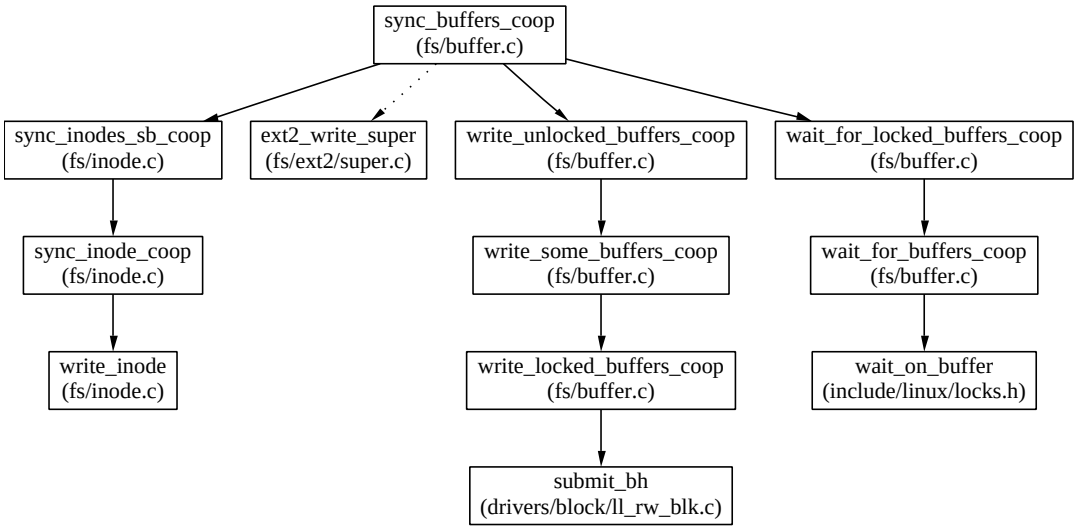

Figure 4.2: Call Graph for *sync_buffers_coop()*.

Updating a Single Drive

Data that must be rewritten to disk may reside in one of three buffer types: *block buffers*, *i-node buffers* and the *super blocks*. Each of them has a mechanism to be marked as dirty or clean. For each drive, we keep track of the number of its dirty buffers in that drive's *dirty_buffer_count*.

Actually, the early commit/abort algorithm of section 3.6 only needs to know if the file system has some data to be written back to a certain drive. But we cannot simply use the number of dirty block buffers, since an i-node is only written to its block buffer (which it shares with other i-nodes) when an update function is called. So the i-node's block buffer may be clean before updating, although the i-node itself is already dirty. Thus we have to consider the i-node buffers for the *dirty_buffer_count*.

A super block is only written to its block buffer when updating, too. But this block buffer is marked as dirty as soon as the internal super block has changed, so we do not need to take the internal super block into account.

The function *sync_buffer_coop()* writes back all dirty buffers for a drive that is specified by a range of device numbers. When this function is called by the update task, it returns immediately after the writes have been requested, so other drives that concurrently wait to be updated are served earlier. When this function is called by the drive's power mode control, because it intends to shut down a drive, it waits until all dirty buffers are written back, so the power mode control will not shut down the drive too early. Figure 4.2 shows the call graph of *sync_buffer_coop()*.

The Update Task

I also had to modify the update task, which in the original Linux wakes up every 5 s. The Coop-I/O version of the update task also wakes up when a drive is accessed and the file

system finds out that it is opportune to update that drive, as explained in section 3.3. In either case, the update task checks each drive: When the file system has requested an update for it or when its dirty buffer lifespan has elapsed, the drive is updated using *sync_buffer_coop()*.

The need of a cooperative update is checked for every time a drive is read from or written to. If there are any dirty buffers for the drive and the drive's oldest dirty buffer is older than half of the dirty buffer lifespan, the update task is woken up and induced to update that drive.

4.3 Cooperative File Operations

The concept of cooperative file operations has been introduced in sections 3.5 and 3.6. As stated there, a file operation may block whenever it is going to access a disk or to make a clean block buffer dirty by modifying it. I wrote a function *wait_for_drive()* that handles the blocking mechanism. It is used throughout the cooperative file code.

The function *wait_for_drive()* usually waits until a certain drive is running. Additionally, it may wait until there exists at least one dirty buffer for that drive. This facility is used when blocking a write access, as explained in section 3.6. Besides, *wait_for_drive()* can be told whether it should cancel on timeout.

When blocked in *wait_for_drive()*, a task may be awoken by one of four events:

The timer has elapsed. If *wait_for_drive()* should cancel on timeout, it returns with *-ETIME*. Else it returns without error.

The drive has run up. The file operation can go on, so *wait_for_drive()* returns without error.

The number of dirty buffers for the drive has become non-zero. If *wait_for_drive()* is also waiting for that event, it simply returns without error. If not, it is ignored.

A signal has arrived. The blocked file operation should be aborted with *-EINTR*, so *wait_for_drive()* returns with that error code. The cooperative operation should not use Linux' implicit restart mechanism, since the signal could be sent to abort it.

The implementation of the cooperative file operations is straightforward: The functions that implement the standard file operations have to be enhanced by the timeout parameter and the cancel flag. When a block is going to be read from disk, the function *wait_for_drive()* has to be called. For a write operation, or an open operation that truncates an old file or creates a new one, a point has to be found where the operation decides to commit or to abort. But the changes are extensive nevertheless, because the whole call hierarchy of the file functions must be examined, taking into account the interactions between VFS and Ext2, to find the code lines and functions to be changed.

4.4 *read_coop()*

The call syntax for a cooperative read operation is as follows:

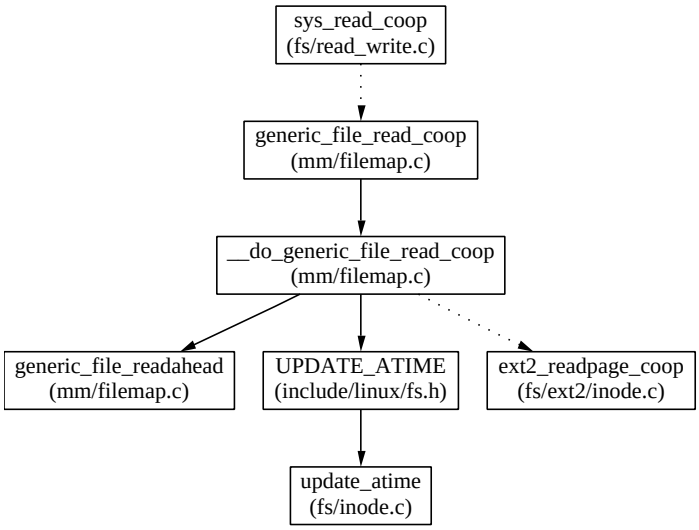

Figure 4.3: Call Graph for *sys_read_coop()*.

*int read_coop (int fd, void *buf, size_t count, int delay, int cancel_flag);*

The first three parameters have the same meaning as for *read()*. The parameter *delay* is the maximum delay (in seconds) that the read may be blocked for energy-saving issues. The parameter *cancel_flag* can be non-zero to indicate that the operation should be cancelled with *errno = ETIME* if it could not be executed without spinning up the drive until *delay* has passed.

When a *read_coop()* system call has reached the kernel via a software interrupt, its entry point within the kernel (the function *sys_read_coop()*) is looked up in a table defined in "arch/i386/kernel/entry.S". In the entry point function, the delay time, which is relative to the invokation time, is translated into the equivalent number of ticks (or *jiffies*) since system startup, because it is easier to have a time point relative to a global reference point.

Then *sys_read_coop()* does its job as sketched by its call graph in figure 4.3. It may try to read ahead some blocks that are not yet needed, which improves system performance. When the drive is in standby mode, we do not read ahead, since this could unnecessarily run up the drive. It may also update the time of the i-node's last access (*a-time*). This could be inefficient, since the i-node will be made dirty and has to be written to disk later. I recommend to mount the file systems with the mount option *noatime*, because the *a-time* is normally not needed.

In Linux 2.4, the read/write operations operate on memory pages in order to use the same data structures as are used for memory mapping. The function *ext2_readpage_coop()* assures that a full page, which consists of a device-dependent number of blocks, is resident in memory. If some blocks of that page are not yet buffered, it waits for the drive and reads them in, using the interface function *submit_bh()* to the block device, as shown in figure 4.4. The function *ext2_get_block_coop()* maps the file's block index to a block number on the drive, using the i-node and indirect blocks. You can read more about it in section 4.5.

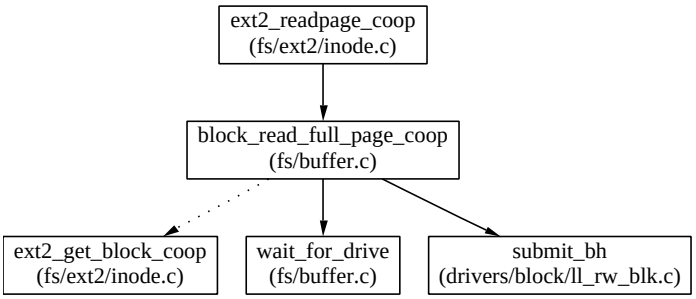

Figure 4.4: Call Graph for *ext2_readpage_coop()*.

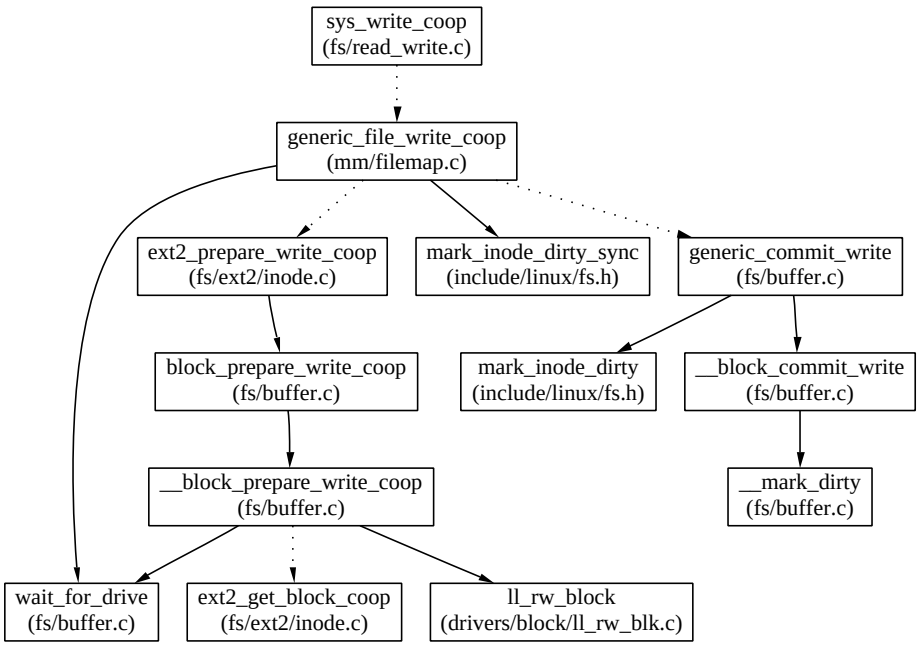

Figure 4.5: Call Graph for *sys_write_coop()*.

4.5 *write_coop()*

The call syntax for a cooperative write operation is as follows:

*int write_coop (int fd, void *buf, size_t count, int delay, int cancel_flag);*

The first three parameters and the return value have the same meaning as for *write()*. The parameters *delay* and *cancel_flag* are the same as for *read_coop()* in section 4.4.

The entry point for a cooperative write operation is *sys_write_coop()*, whose call graph is shown in figure 4.5. The Ext2 file system uses the page structure for writing (starting with Linux 2.4). Before committing the write operation, we have to test that the drive is ready or that there are at least any dirty buffers. If the write operation is committed, we have

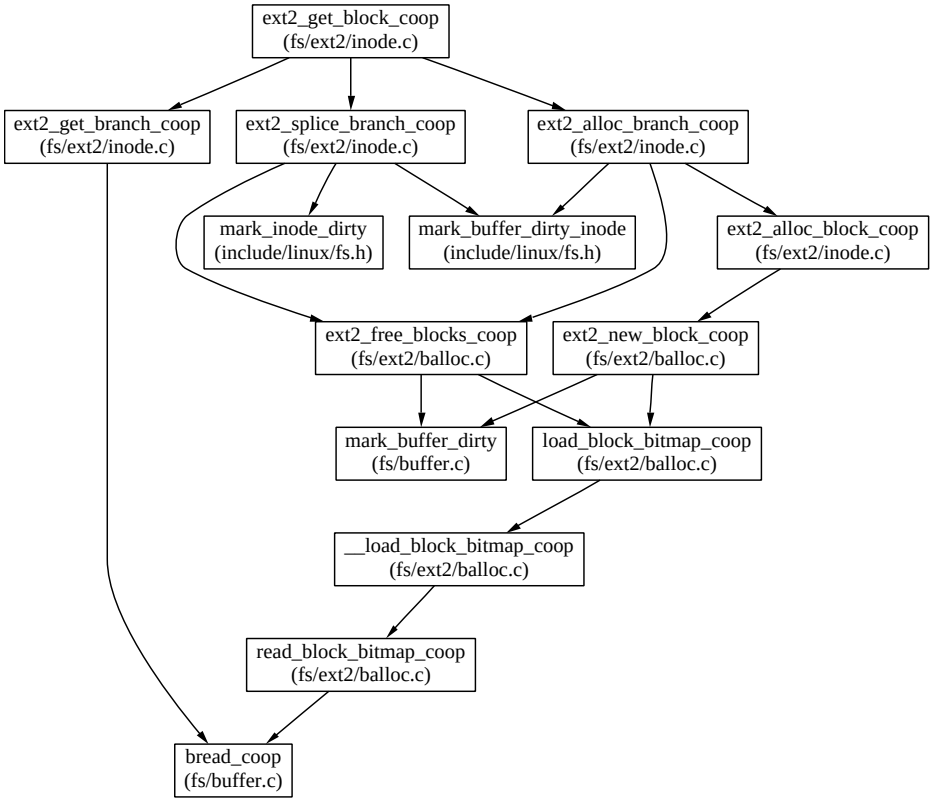

Figure 4.6: Call Graph for *ext2_get_block_coop()*.

to update the i-node's time of last modification (*m-time*) and mark the i-node as dirty using *mark_inode_dirty_sync()*. Then the page is read in by *ext2_prepare_write_coop()*, it is changed in main memory, and the block buffers that have been modified are marked as dirty by *ext2_commit_write()*. This might change the i-node's *size* field.

The function that maps a file's block index (which is the byte index in that file divided by the block size) to the sector number of an Ext2 file system is called *ext2_get_block_coop()*. It can also look up a yet non-existing block, adding a new block to the file. Its call graph is diplayed in figure 4.6. For simplicity, I have omitted the calls that may block when the file or the file system is in synchronous mode.

Normally, an existing buffer is found by *ext2_get_branch_coop()*. This function starts with the i-node (which is always in memory for an open file) and finds the block number on the drive. If it needs to follow single, double, or triple indirect block, it asks the buffer cache for it, which in turn may read the block if it is not cached.

A write operation may also ask *ext2_get_block_coop()* for a yet non-existing block. (Since Ext2 files may have holes in them, this is not necessarily a block beyond the current file size.) Depending on the file's current configuration, that block may also need a new single, double, and even triple indirect block. These new blocks are allocated by *ext2_alloc_branch_coop()*. This function looks for free blocks in the partition. They are found and allocated in the block bitmap. If something went wrong during allocation, all blocks that are already allocated must be freed again. The function *ext2_splice_branch_coop()* inserts the newly allocated blocks in

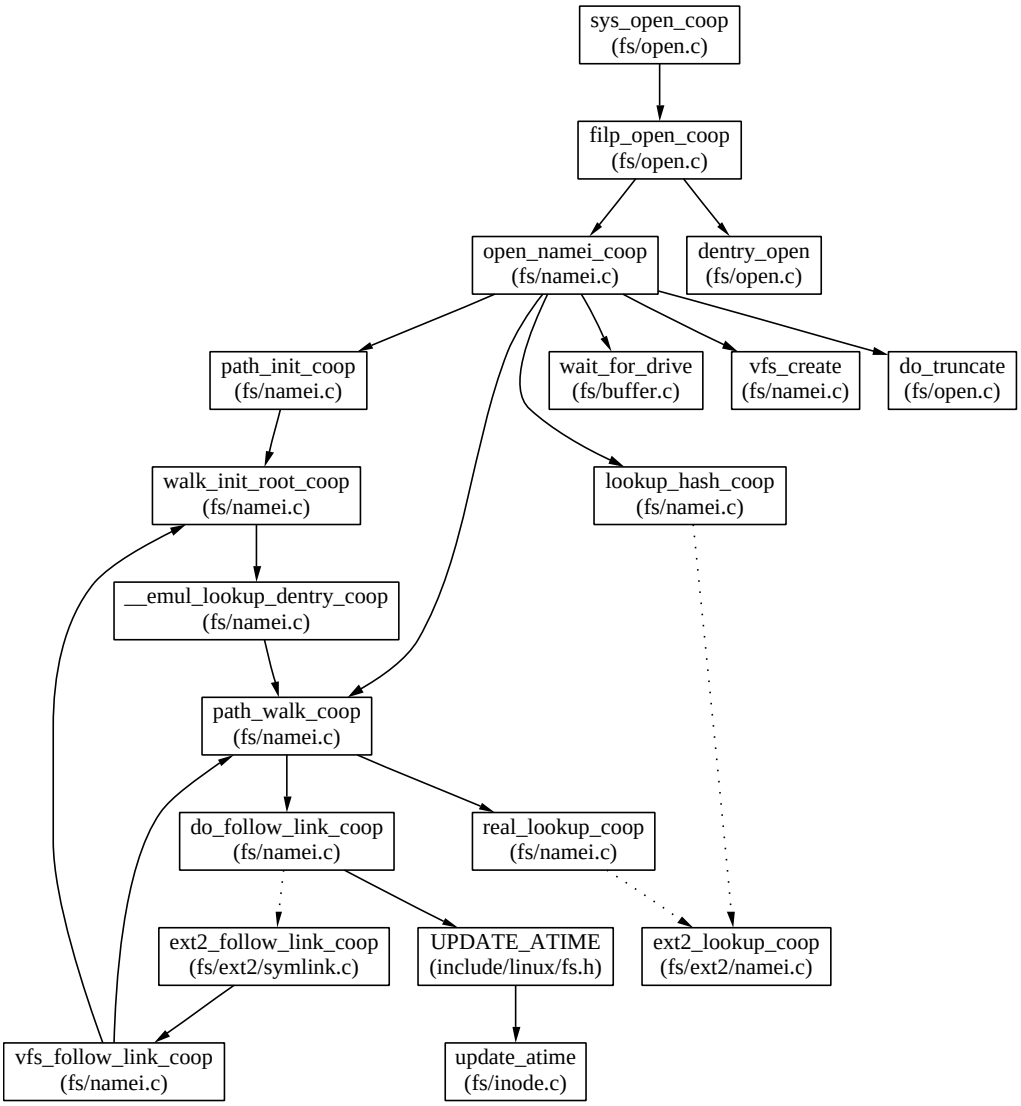

Figure 4.7: Call Graph for *sys_open_coop()*.

the i-node structure, obeying the structure of indirect blocks.

4.6 *open_coop()*

The call syntax for a cooperative open operation is as follows:

*int open_coop (char *name, int flags, mode_t mode, int delay, int cancel_flag);*

The first three parameters and the return value have the same meaning as for *open()*. The parameters *delay* and *cancel_flag* are the same as for *read_coop()* in section 4.4.

The entry point for a cooperative open operation is *sys_open_coop()*. Figure 4.7 shows its call graph. The main work is done in the function *open_namei_coop()* which traverses the path

name and loads the i-node of that name into the i-node buffer. If the file has to be created or truncated, this will also be done here. The function *dentry_open()* will create the process' file structure for the i-node.

The start directory for the file search is set by *path_init_coop()*. If the path is relative, the current directory will be used. If the path is absolute, the start directory will be set to the root directory in *walk_init_root_coop()*.

Then the path is walked along in *path_walk_coop()*. The directory entries are buffered in the *dentry cache*. If a directory entry has yet not been read, it is looked up by *real_lookup_coop()*. If a path element to be followed is a symbolic link, this is done by recursively calling *walk_init_root_coop()* for an absolute symbolic link or *path_walk_coop()* for a relative symbolic link. When a symbolic link is followed, the *a-time* of that link is updated. This can be prevented by mounting the file system with the option *noatime*.

Normally, calling *walk_init_root_coop()* simply has the effect that the directory from which the path walk is started is the root directory. But Linux is able to emulate other operating system environments for non-Linux processes, like Solaris on SPARC-Linux. Such an emulated environment is called a *flavour*. Since a non-Linux application may expect certain non-Linux files at standard places, for example shared libraries, Linux has to use a flavor specific root when executing such an application. This flavor specific root is stored as a path name relative to Linux' native root. It is made the start directory for the path walk by *__emul_lookup_dentry_coop()*.

If the i-node to be opened has been found and the *open_coop()* has to truncate the file, we must commit the operation before actually truncating. So we call *wait_for_drive()* to check (and maybe wait for) the conditions of the *early commit/abort* concept as established in section 3.6. We only truncate the file if we have committed the operation.

If *open_coop()* has been called with the *create* flag set, and the file does not yet exist, we also have to decide whether to commit or abort by calling *wait_for_drive()*. If we commit, we must create the file.

In the current implementation, I have decided *not* to write cooperative versions of *do_truncate()* and *vfs_create()*. This does not strictly follow the early commit/abort concept, since these functions may have to read in blocks from the disk and might delay in that case. But if the concept were implemented thoroughly, many additional functions had to be changed for a small expected energy-saving effect: When truncating a file, we must modify the i-node, the block bitmap, and the *m-time* of the parent directory. If the file is currently memory mapped, special care is needed. File creation is even more complex.

The function *ext2_lookup_coop()*, whose call graph is shown in figure 4.8, looks up a file name in a directory. It is called when walking along a path name. First, it loads the directory's i-node by calling *iget_coop()*. Then, it traverses the directory entries in *ext2_find_entry_coop()*. Since directories are very similar to ordinary files and since ordinary files are accessed using the memory mapping data structures, the Ext2 file system uses these data structures as well when accessing a directory file.

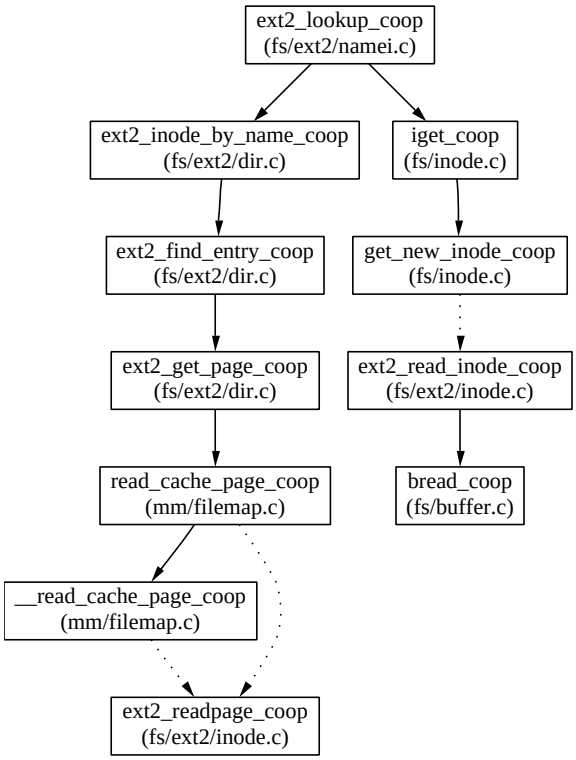

Figure 4.8: Call Graph for *ext2_lookup_coop()*.

4.7 Changed Source Files

These are the files that I changed in the Linux Kernel 2.4.10:

arch/i386/kernel/entry.S
drivers/block/ll_rw_blk.c
drivers/ide/ide-disk.c
drivers/ide/ide-probe.c
drivers/ide/ide.c
fs/buffer.c
fs/inode.c
fs/namei.c
fs/open.c
fs/read_write.c
fs/ext2/balloc.c
fs/ext2/dir.c
fs/ext2/file.c
fs/ext2/inode.c
fs/ext2/namei.c
fs/ext2/symlink.c
mm/filemap.c
include/config.h
include/asm-i386/unistd.h
include/linux/blkdev.h
include/linux/coop.h
include/linux/ext2_fs.h
include/linux/fs.h
include/linux/hdreg.h
include/linux/ide.h
include/linux/pagemap.h

Chapter 5

Validation

5.1 Test Environment

To validate the Coop-I/O concept, a power measuring configuration was needed. I have used a monitoring computer that was equipped with a four-channel analog-to-digital converter (ADC) developed at the University of Erlangen-Nürnberg. It measures the voltage drop at defined resistors in the power supply lines with a resolution of 256 steps and at a rate of up to 20 000 samples per second. The maximum voltage drop that is correctly converted is 50 mV. The samples are read in via the standard parallel port. The software driver to read the data from the interface has been written by Christian Winter [6].

I used only one ADC channel with a rate of 10 000 samples per second to measure the power dissipation of the IBM DCRA-22160 hard disk. Since its maximum power consumption lies at about 4 W, I used a resistor of 0.05Ω, so the maximum voltage drop was equivalent to an amperage of 1 A or to a dissipation of 5 W.

The target computer was a Siemens SCENIC Edition Mi7 desktop with a Pentium II processor clocked at 350 MHz and 128 MBytes of main memory. It was running SuSE Linux 7.3, either the original kernel, version 2.4.10, or the modified Coop-I/O kernel. The system was equipped with a Fujitsu MPG3204AT hard disk and an IBM DCRA-22160 hard disk, which was used as the test drive.

5.2 Testing a Cooperative Audio Player

The first test step was to examine whether Cooperative I/O is able to save energy in a real-world application at all. Since hard disk power management works best for regular file accesses, I have tested the system with a modified version of AMP, an MPEG audio layer 3 player. In the modified version, AMP creates a thread that reads from hard disk and puts the data into a 1 MB buffer. When the player thread needs some data, it reads them from the buffer. The read thread and the player thread synchronise by the use of semaphores. The buffer is divided into two semi-buffers. When a semi-buffer is empty, the reader refills it by the use of a system read call which might be cooperative, while the player get data from the other semi-buffer. I had to add about 150 source lines to incorporate the changes.

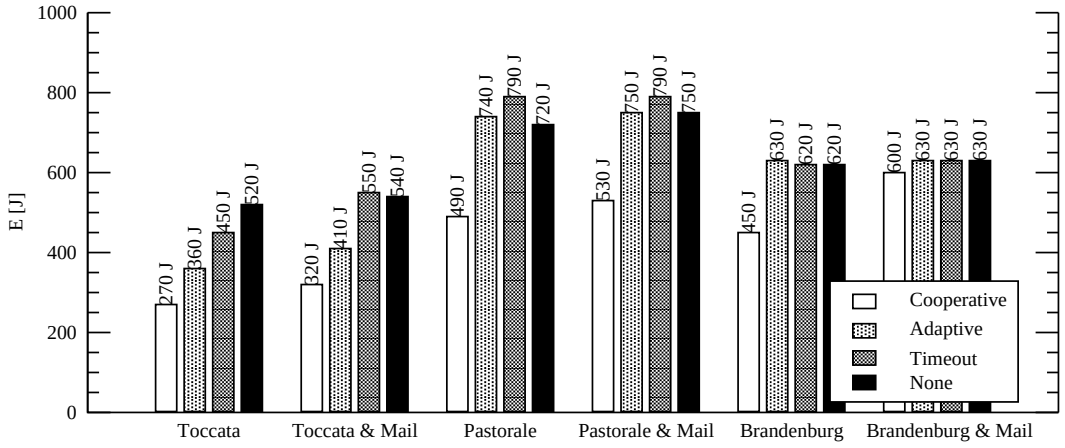

Figure 5.1: Energy consumption for three sound files played with four different policies.

AMP has been tested under the following four strategies:

Cooperative: Use the DDT/ES standby algorithm. To read in new data, use the *read_coop()* system call with a delay that is equivalent to the playing time for one semi-buffer.

Adaptive: Use the DDT/ES standby algorithm. Use *read()* to read in new data.

Timeout: Use the fixed timeout standby of the ATA standard with a timeout interval of 30 s.

None: Do not use any power-saving measures at all.

Every strategy has been tested by playing the following three audio files. *Delay* is the time interval in which one semi-buffer is played.

Audio File	Bitrate	Delay	Duration
Toccata	64 kb/s	64 s	534 s
Pastorale	128 kb/s	32 s	719 s
Brandenburg	160 kb/s	25 s	598 s

I have also examined how well the power-saving strategies work when an asynchronous second application runs while playing an audio file. For that aim, the test computer has concurrently executed a mail reader that examined the input mailbox of a remote computer via POP3 every 2 minutes. If there was any mail in it, the mail was stored in the local mailbox on the test hard disk. Mail was sent in intervals of 60–265 seconds, controlled by a pseudo-random generator. For every test pass, the random generator was initialised to the same value, so the timely sequence of read/write operations was the same for every test, with a tolerance of about one second. Figure 5.1 shows the results.

The cooperative strategy is surprisingly power-efficient in these tests. This is not only caused by the cooperation of multiple processes, because some tests have only one process doing I/O. Instead, it can be explained by the following behaviour: When the drive is in standby mode, a cooperative read is delayed until the data is really needed, i.e., the semi-buffer to be read will soon be played. When the delayed read operation is eventually performed, the other semi-buffer gets empty very soon and is read in immediately, since the hard disk drive is still running. This effectively bundles two subsequent read operations. You can see this

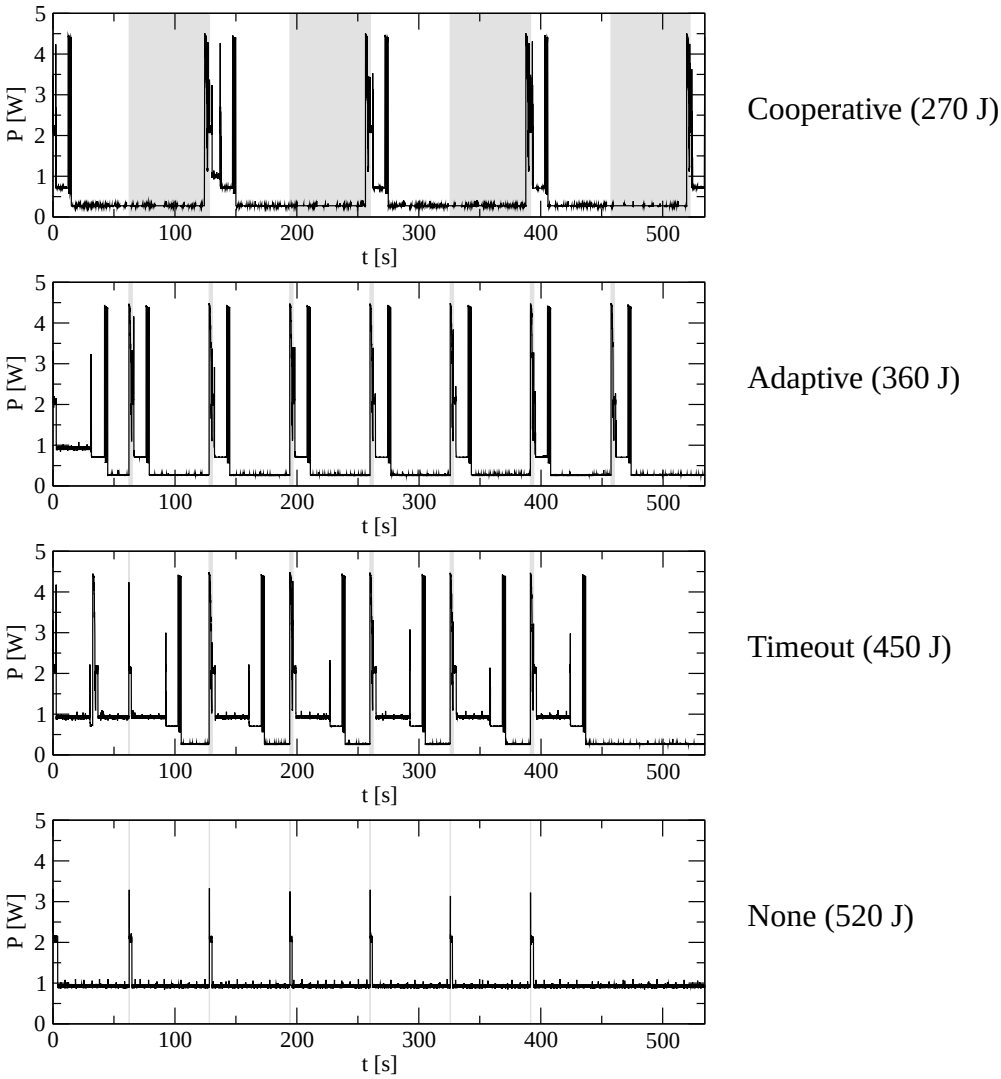

Figure 5.2: The hard disk's power inputs when playing "Toccata" using the four energy saving strategies. In the shaded regions, the program has called *read()* or *read_coop()* and waits for its completion.

behaviour in figure 5.2.

In that figure, you may also note that AMP initiates a read operation at $t = 455$ s when using the Cooperative or Adaptive strategy, but not when applying the strategies Timeout and None. This is caused by Linux' *read-ahead* policy. The original Linux kernel reads ahead the remainder of the file as part of its file access at $t = 395$ s. The modified kernel omits the read ahead at that point since the disk is in standby mode when the file access is started.

Playing "Brandenburg" needs nearly the same energy for the hard disk, regardless which strategy is used. Because the delay for this audio file is only 25 s, no strategy will normally try to shut down the hard disk in this test scenario. But there is an exception: When "Brandenburg" is played using the cooperative strategy, sometimes the power mode control shuts down the drive soon after the test has started. This depends on the pattern of the previous disk accesses. If the disk has shut down, the next read will be delayed, so the interval be-

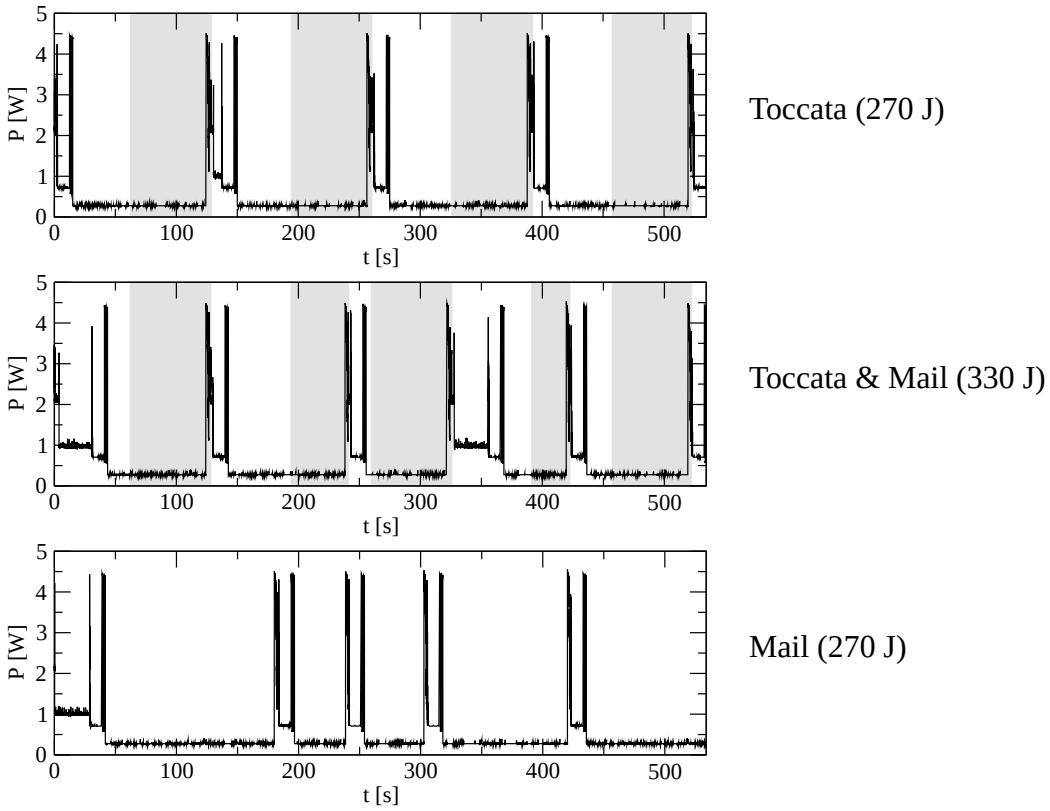

Figure 5.3: The hard disk's power inputs when playing "Toccata", receiving Mail or doing both using Coop-I/O. In the shaded regions, the MP3 player has called *read_coop()* and waits for its completion.

tween two reads may be more than the break-even time. But if the power mode control does not shut down the drive soon after test start, the reads will take place in 25 s intervals, so the drive will never be idle for a standby period. So the power mode control's behaviour depends on the cooperative disk access delays which again depend on the power mode. This positive feedback is hard to predict and control.

The Timeout strategy needs more energy even than the strategy "None" when playing "Toccata & Mail", "Pastorale" or "Pastorale & Mail". This behaviour is caused by the unlucky relation of the standby timeout and the hard disk access pattern, which makes the disk go to standby shortly before the disk is used; in that case, switching to standby mode is more expensive than staying in idle mode.

Figure 5.3 shows how disk accesses of two independent tasks may interact. The "Toccata" task cooperatively reads one semi-buffer in every period of 64 s; the "Mail" task writes in intervals of 2 minutes, provided that mail has arrived. The write accesses are delayed by the cooperative update scheme. If the disk accesses were not coordinated, the hard disk's energy consumption would be about 400 J. This means that Coop-I/O is a working energy-saving concept.

Process	Period	Idle min.	Idle max.	Seed
1	100 s	20 s	100 s	10 001
2	160 s	40 s	160 s	10 002
3	200 s	50 s	180 s	10 003
4	260 s	60 s	160 s	10 004
5	300 s	100 s	220 s	10 005

Figure 5.4: The parameters used for the five processes in the parameterised tests.

5.3 Parameterised Tests

To get an idea under which circumstances Coop-I/O saves energy, I have written two little test programs, *test_read* and *test_write*, which simulate single tasks that periodically read or write data, respectively. In a test set, five such test programs with different parameters are simultaneously started and the overall disk power consumption is observed.

The C programs *test_read* and *test_write* are very similar. They take the following parameters:

mode: Cooperative mode or non-cooperative mode.

file_name: The file to read from or to write to, respectively.

period: The length of a period in which a single read/write operation takes place, in seconds.

block_size: The size of the data block to read or write during a period, in KBytes.

idle: The time to wait at the beginning of a period until the read/write operation is started, in seconds.

idle_min, idle_max, seed: To get some non-regular behaviour, the fixed *idle* parameter may be replaced by a range from *idle_min* to *idle_max* seconds which limits the idle time. The idle time for each period is determined by the C library function *rand()*, which returns a pseudo-random number. To get reproducable results, the pseudo-random number generator is initialised by *srand(seed)*.

The programs run for an integer number of periods, but at least for 1000 s. In each period, they sleep for *idle* seconds, execute the read/write operation, and finally wait until the period is over. When they run in cooperative mode, they call *read_coop()* or *write_coop()* with a delay that ends when the current period also ends. In non-cooperative mode, they call *read()* or *write()*.

The running time of these test programs and the number of disk accesses only depend on *period*, while the amount of data read or written depends on *period* and *block_size*. Both values are independent of the power-saving strategy that is being used, so the work done by these test programs is the same for all power-saving strategies. This helps in comparing the energy consumptions. For all test series, I have used a group of five read/write processes with the parameters shown in figure 5.4. Normally, all these processes read or write a chunk of 256 KBytes in each period.

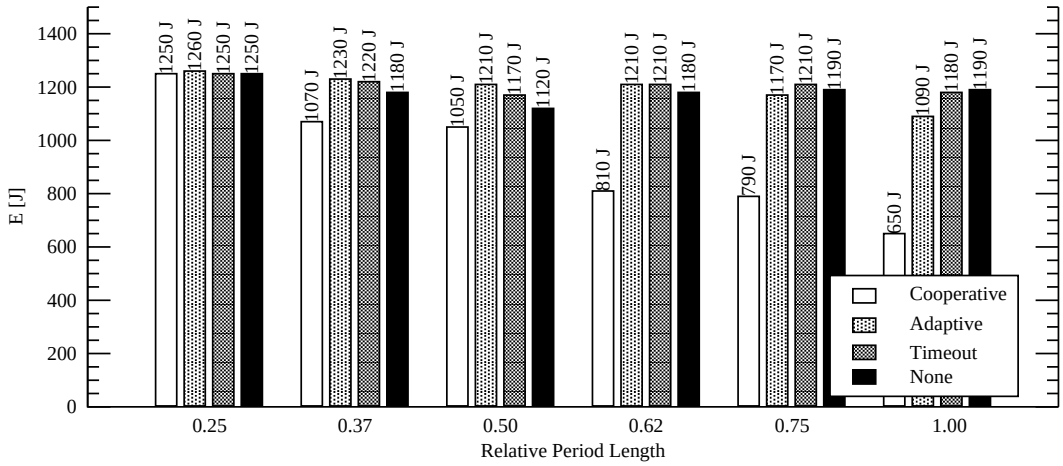

Figure 5.5: Energy consumption for five read processes, varying the period length.

Reads with Varying Period Length

In the first test series, I have varied the period length for read operations. For each test, five *test_read* processes have been started, using the parameters of figure 5.4, but *period*, *idle_min* and *idle_max* were multiplied by the *relative period length* of that test. I used relative period lengths of 0.25, 0.37, 0.50, 0.62, 0.75, and 1.00. Each test has been executed in combination with the four power-saving strategies. Figure 5.5 shows the measured consumptions.

Running with a relative period length of 0.25, the test's consumption is nearly equal for all four strategies. Here, the read requests are so frequent that the drive has no chance to shut down. The longer the period length, the more often is the power mode control able to go to standby mode for all strategies but "None", as could be seen from the time dependent energy consumption (not shown). The Cooperative strategy is able to exploit these shutdowns since following cooperative reads are delayed. This is the reason why energy consumption drops steadily with increasing period length for "Cooperative". Since the non-delayed disk accesses are still too frequent, "Adaptive" and "Timeout" are unable to save power for relative period lengths from 0.37 to 0.75. They may even consume more power than "None" by initiating disadvantageous shut-downs. For the test with relative period length 1.00, the Adaptive strategy seems to outplay "Timeout" and "None", although this is only a trend.

We may conclude that cooperative reads only save energy if the read accesses are not too frequent.

Writes with Varying Period Length

Ordinary write operations are usually buffered and written back by the update daemon which is trimmed to be cooperative. So one may wonder whether explicitly cooperative write operations have noteworthy energy-saving effects.

I have executed the same test series as in the previous test, only using write operations instead of read operations. Again, the tests have been run under all four strategies. The results are displayed in figure 5.6.

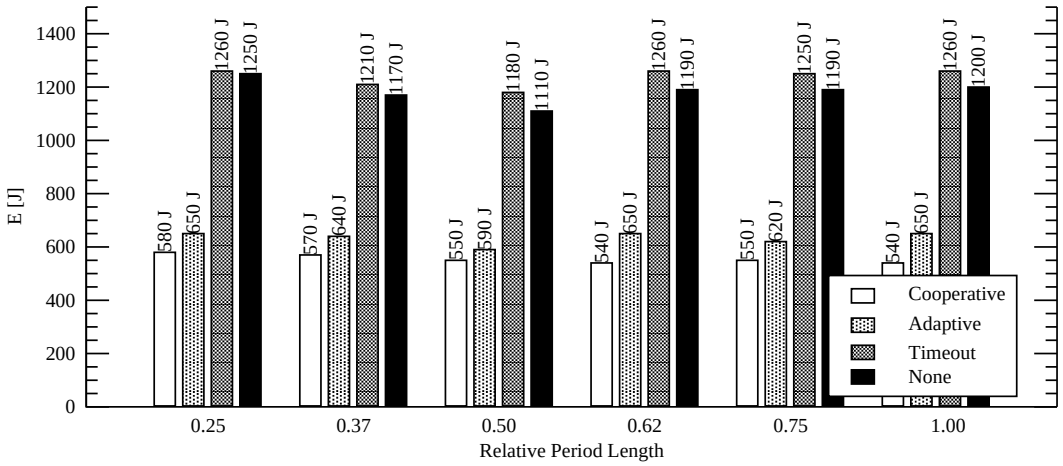

Figure 5.6: Energy consumption for five write processes, varying the period length.

The frequency of write operations seems to have little effect on the energy consumption. For the Cooperative and Adaptive strategies, the dissipation stays constantly on a low level for all period lengths. This is caused by the cooperative update policy that writes back quite regularly in intervals of 60 s, if no other disk access happens. The amount of data written has almost no influence on the energy consumption. The small difference between the Cooperative and the Adaptive strategy is an evidence that cooperative write operations are not as effective as cooperative read operations. The strategies “Timeout” and “None” have a persistently high dissipation. This shows that the original Linux 2.4 update strategy does not match power-saving requirements.

Varying Number of Cooperative Processes

In the previous tests, the test programs were all cooperative. But in a real-world scenario, the majority of running processes will be non-cooperative. Thus, I have also examined the behaviour of a mixture of cooperative and non-cooperative processes. To this end, I have run test sets, each with five reading processes using the parameters in figure 5.4, but only the first n processes of them were running the Cooperative strategy, while the remaining $5-n$ processes used the Adaptive strategy. For the results, see figure 5.7.

The energy consumption steadily declines with increasing proportion of cooperative processes. The decline is larger when all processes get cooperative (compare $n = 4$ to $n = 5$), but having a single cooperative process suffices to save energy.

Varying Block Size

Linux may read ahead up to 128 KBytes of a file when it assumes that it is accessed sequentially. In section 4.4, we have seen that the implementation of the cooperative read operation refrains from reading ahead when the disk is not running, in contrast to standard Linux. This behaviour may degrade power-efficiency when reading small blocks. So I have tested the energy consumption for the five reading processes of figure 5.4 with smaller block sizes, as

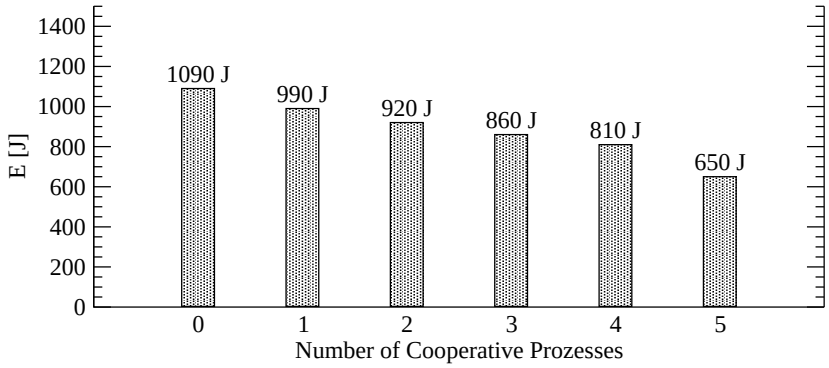

Figure 5.7: Energy consumption for five read processes, varying the number of cooperative processes.

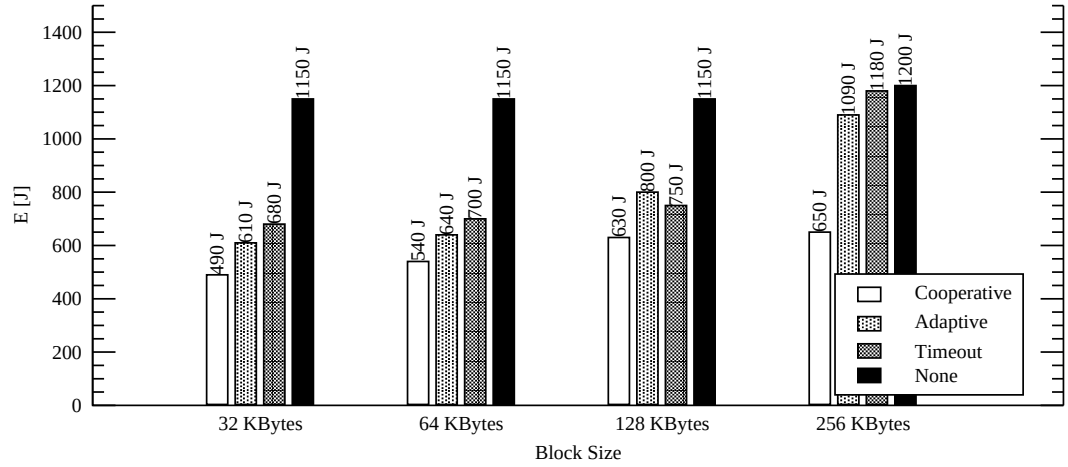

Figure 5.8: Energy consumption for five read processes, varying the block size.

shown in figure 5.8.

The Cooperative strategy is still better than the Timeout strategy, but the advantage for small block sizes is smaller than for 256 KBytes blocks. This is caused by the fact that Linux' read-ahead function is called *before* the requested data is read in. If the drive is in standby mode, no data is read ahead although the drive will run up in a moment to read in the requested data. Unfortunately, Linux' current read-ahead function must be modified to be called later. This remains work to be done.

Chapter 6

Conclusion

6.1 Summary

In the present work, I have examined how an operating system may support energy-saving measures for hard disks, with or without hints from the applications.

In chapter 2, I have outlined the strategies that are commonly applied to reduce the energy consumption of a hard disk drive. The central point is the use of the drive's power-saving modes that have lesser power requirements and should be activated when the disk is not in use. Since the transition between power modes needs time and energy, a power saving mode (e.g. standby mode) should only be activated if no hard disk accesses are to be expected for the *break-even period*.

In chapter 3, I have introduced Coop-I/O, an integrated cooperative power-saving strategy that is composed of the following mechanisms:

1. A simple adaptive shut-down algorithm called DDT/ES that tries to guess the length of the current busy interval.
2. An update policy that writes back all dirty buffers of a drive at once, tries to attach to other disk accesses and starts an update when a drive is shutting down.
3. Explicitly cooperative file operations that may wait for other hard disk accesses for a specified interval in order to save energy.

In chapter 4, I have described the changes in the Linux kernel that were necessary to implement that concept. The ATA/IDE driver was augmented by a power mode control, while the virtual file system and the Ext2 file system were modified to support the cooperative drive-specific update policy and the cooperative file operations.

In chapter 5, I have tested Coop-I/O concept by running a modified, cooperative MP3 audio player together with an email reader. There was evidence that Coop-I/O may save energy. Parameterised tests were used to examine the conditions under which Coop-I/O really saves energy and when it is most effective. The Coop-I/O implementation has shown a robust energy-saving behaviour.

6.2 Application Areas

Write operations are coordinated by the buffer/update mechanism even without explicitly calling *write_coop()*, so all applications that write to disk may benefit from Coop-I/O without any need of modifications. A user program can improve its cooperability by using *write_coop()*.

Applications that may profit from cooperative read operations must be able to defer or to cancel them. Streaming applications like audio or video players are a primary field for cooperative operations.

If explicit cooperative operations are to be used in an application program, it has to be modified. Depending on the application's needs, several usage strategies are conceivable, among them the following ones:

- If the file operation is of minor importance and should only be executed if it does not consume much energy:
  Call a cooperative file operation with parameters *delay = 0* and *cancel_flag = 1*. This will execute the file operation only if the hard disk does not have to run up.

- If the file operation may be deferred for a period that is known in advance:
  Call a cooperative file operation with the known *delay* and *cancel_flag = 0*. Since the process may be blocked for a longer time, it is often advisable to create a second thread that is dedicated for I/O. This has been exemplified by the AMP audio player in section 5.2.

- If the file operation may be deferred, but the period is not known in advance:
  The application may use a second thread that starts a cooperative operation with a long *delay*. If the operation must be executed at a certain point of time and is still blocked, the thread is interrupted by a signal and aborts the operation. The file operation may then be done by the appropriate standard system call.

6.3 Future Work

Since a thesis is limited in time, there is still room for improvements and extensions. The following areas that seem worth investigating are listed with increasing estimated complexity.

- In section 5.2, we have seen that power mode control and cooperative I/O operations may influence each other in a positive feedback loop, leading to an energy-saving behaviour where small variations in the access pattern may have a big effect. It would be desirable to break this loop, for example by passing the possible time range of the file operation to the IDE driver when requesting a disk access. The shut-down algorithm then has to be modified to consider these time ranges instead of time points.

- Adding a power mode control to the SCSI driver that has the same functionality as the one of the ATA/IDE driver should be easy, since SCSI's definition of power modes is similar to ATA's.

- Other Linux file systems may also be modified to implement cooperative file operations. This could be interesting for non-Unix file systems like the FAT file system or for transactional file systems.

- Using the early commit/abort policy, a blocked cooperative write operation might block other file operations that access the same file. In such cases, implementing the concept of shadow buffers, as presented in section 3.6, might improve system performance. Besides, write accesses might be aborted later, thus improving the system's power saving behaviour.

Glossary

Active Mode: Hard disk mode. The disk is either reading, or writing, or positioning.

ATA: *AT Attachment.* A standard interface and protocol for hard disk drives, also known as IDE. Version 4 and later have a packet interface extension (ATAPI) included that is used to control other device types like CD-ROM drives.

Break-Even Period: The shortest time interval for which switching to standby in the beginning and switching back to running mode at the end is more energy-efficient than staying in idle mode. The break-even period is device-specific.

Busy Period: A time interval between two standby periods that does not contain a standby period itself.

DDT: *Device Dependent Time-out policy.* Switches the disk drive to a resting mode after it has been idle for the break-even period.

DDT/ES: *DDT with Early Shut-down.* Like DDT, but switches to resting mode earlier if the current busy period is roughly as long as the previous one.

Dirty Buffer Lifespan: The time interval that a buffer may be dirty until it is written back to disk.

Ext2: The 2nd version of the extended file system for Linux. The most popular file system type in the Linux world.

IDE: *Intelligent Drive Electronics.* Another name for the ATA standard.

Idle Mode: Hard disk mode. The hard disk is not active, but the spindle motor is on and the interface is active.

Resting Mode: Hard disk mode. Subsumes standby mode and sleep mode.

Running Mode: Hard disk mode. Subsumes active mode and idle mode.

Sleep Mode: Hard disk mode. The spindle motor is off and the hard disk interface is inactive and may only be reactivated by a reset.

Standby Mode: Hard disk mode. The spindle motor is off, but the hard disk interface is active.

Standby Period: A time interval of hard disk inactivity that is longer than the break-even period, so changing to standby mode would be energetically profitable.

VFS: *Virtual File System.* The part of the Linux kernel that is common to all file system types. It communicates with the individual file systems via a well-defined (but badly documented) interface.

Bibliography

[1] Yung-Hsiang Lu, Giovanni De Micheli: *Adaptive Hard Disk Power Management on Personal Computers.* In: *Ninth Great Lakes Symposium on VLSI 1999.* pp. 50–53. http://akebono.stanford.edu/users/luyung/papers/glsvlsi-99.pdf

[2] Yung-Hsiang Lu, Giovanni De Micheli: *Comparing System-Level Power Management Policies.* In: *IEEE Design & Test of Computers.* Special Issue on Dynamic Power Management of Electronic Systems. March–April 2001. pp. 10–18. http://akebono.stanford.edu/users/luyung/papers/dnt01.pdf

[3] Steffen Meyer: *Energieeinsparung durch kooperative Gerätenutzung.* (Energy Savings by Cooperative I/O) Pre-Master's Thesis SA-I4-2001-03. Institut für Informatik der Universität Erlangen-Nürnberg. http://www4.informatik.uni-erlangen.de/SA/pdf/SA-I4-2001-03-Meyer.pdf

[4] *OEM Hard Disk Drive Specification for DCRA-22160 (2160 MB) 2.5-Inch Hard Disk Drive with ATA Interface, Revision (2.0).* IBM Corporation 1996. http://www.storage.ibm.com/hdd/support/dcra/dcra_sp.pdf

[5] *Information Technology – AT Attachment with Packet Interface – 5 (ATA/ATAPI-5).* Published as ANSI NCITS 340-2000. Draft version: http://www.t13.org/project/d1321r3.pdf

[6] Christian Winter: *Measuring Power Consumption with Linux.* Pre-Master's Thesis SA-I4-2002-07. Institut für Informatik der Universität Erlangen-Nürnberg.

[7] Fred Douglis, P. Krishnan and Brian Bershad: *Adaptive Disk Spin-down Policies for Mobile Computers.* In: *Computing Systems,* 8(4), pp. 381–413, Fall 1995. http://www.douglis.org/fred/work/papers/adapt.ps.gz

[8] *Adaptive Power Management for Mobile Hard Drives*. IBM Corporation 1999 (White Paper). http://www.almaden.ibm.com/almaden/mobile_hard_drives.html

[9] Gaurav Banga, Peter Druschel and Jeffrey Mogul: *Resource containers: A new facility for resource management in server systems.* In: *Proceedings of the Third Symposium on Operating System Design and Implementation (OSDI) 1999.* http://www.cs.rice.edu/~druschel/osdi99rc.ps.gz

[10] Heng Zeng, Xiaobo Fan, Carla Ellis, Alvin Lebeck and Amin Vahdat: *ECOSystem: Managing Energy as a First Class Operating System Resource.* Technical Report CS-2001-01. Duke University. http://www.cs.duke.edu/ari/millywatt/ecosystem.pdf

[11] Jacob R. Lorch, Alan Jay Smith: *Software Strategies for Portable Computer Energy Management.* In: *IEEE Personal Communications Magazine,* 5(3). pp. 60–73, June 1998. http://research.microsoft.com/~lorch/papers/survey.pdf