Cooperative I/O—A Novel I/O Semantics for Energy-Aware Applications

Andreas Weissel, Björn Beutel, Frank Bellosa

University of Erlangen, Department of Computer Science
{weissel,bnbeutel,bellosa}@cs.fau.de

Abstract

In this paper we demonstrate the benefits of application involvement in operating system power management. We present Coop-I/O, an approach to reduce the power consumption of devices while encompassing all levels of the system—from the hardware and OS to a new interface for cooperative I/O that can be used by energy-aware applications. We assume devices which can be set to low-power operation modes if they are not accessed and where switching between modes consumes additional energy, e.g. devices with rotating components or network devices consuming energy for the establishment and shutdown of network connections. In these cases frequent mode switches should be avoided.
With Coop-I/O, applications can declare open, read and write operations as *deferrable* and even *abortable* by specifying a time-out and a cancel flag. This information enables the operating system to delay and batch requests so that the number of power mode switches is reduced and the device can be kept longer in a low-power mode. We have deployed our concept to the IDE hard disk driver and Ext2 file system of Linux and to typical real-life programs so that they make use of the new cooperative I/O functions. With energy savings of up to 50%, the experimental results demonstrate the benefits of the concept. We will show that Coop-I/O even outperforms the "oracle" shutdown policy which defines the lower bound in power consumption if the timing of requests can not be influenced.

1 Introduction

Mobile devices and embedded systems characterize one major trend in computer system design. One common aspect of new developments in this area is a limited power supply and the need to economically handle the energy resource. Energy-aware system design has therefore been widely recognized to be an important and major challenge [19].

To enable *dynamic power management*, the control and reduction of power consumption at run time, many techniques and algorithms have been suggested. Research in this area has focused mainly on the operating system level. The OS tries to save a maximum amount of energy by setting devices and components to low-power modes if they are not accessed. Because a mode switch itself consumes energy, this action will only pay off if the time to the next request is long enough. An introduction to low-power modes of devices is given in section 2.

Our contribution to this research area is *Coop-I/O*—an approach that integrates the application layer into dynamic power management of devices. We introduce a new operating system interface for cooperative I/O which can be exploited by energy-aware applications. File operations are equipped with two additional parameters—a time-out and a cancel flag. The operating system tries to batch deferrable requests in order to create long idle periods during which switching to a low-power mode pays off.

An example of a deferrable and abortable write operation is the periodic auto-save function of a text editor. If an auto-save has to be aborted because the disk is shut down, the next auto-save can be performed non-cooperatively with up-to-date data. Deferrable, but not abortable read operations could fill the read buffer of an audio- or video player. The time-out can be set to the play time of this buffer. Other examples are background processes like cron jobs, daemons or logging mechanisms. A web browser could use a memory cache and abortable reads and writes to access its disk cache. If the disk is not running, data will be cached only in memory and not on disk.

Our concept consists of three major parts, which will be presented in detail in the next sections:

- We introduce new cooperative file operations that have a time-out and a cancel flag as additional parameters (see section 3). If a file operation needs to access a disk drive and that drive is shut down, the operation will be suspended until either the disk drive has spun up due to another I/O request or the time-out has elapsed. When the time-out is reached and the file operation cancel flag is set, the operation will be aborted. In all other cases, it will be finally executed. The new functions are compatible with the legacy interface.
- The operating system caches disk blocks in *block buffers* in main memory. Modified block buffers are periodically written to disk by an update mechanism. We present an update policy that is redesigned to save energy (see section 4).
- The operating system will control the hard disk modes; it will switch the disk drive to a low-power mode when it assumes that the drive will not be accessed for a certain time. This is controlled by a simple algorithm called *device-dependent time-out* (DDT) (see section 5).

Figure 1 presents the whole concept. Coop-I/O integrates all levels—hardware, operating system (driver, cache and file system) and the application layer—to reduce energy consumption

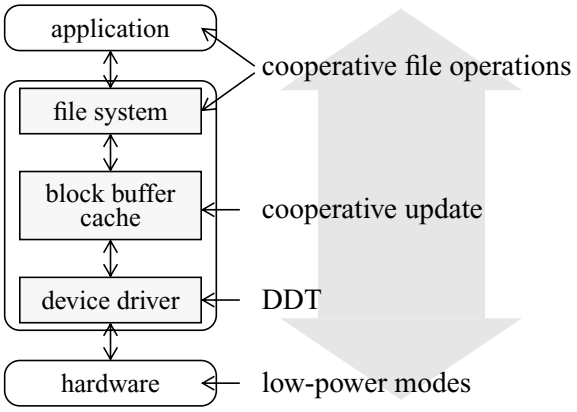


Figure 1: components of Coop-I/O

We have implemented Coop-I/O in Linux. The extensions to the IDE device driver, the buffer management and the file system are presented in section 6. As we will show, only little effort is needed to rewrite existing programs and energy-savings are possible even if many unmodified applications are running. The results of our experiments can be found in section 7. We will show that even the "oracle" device shutdown policy, which has complete knowledge about, but no influence on the timing of future requests, can be surpassed by Coop-I/O. Related work is presented in section 8.

2 Low-Power Modes of Devices

Modern peripheral devices make use of several operation modes that are associated with different levels of power consumption. Operating systems may reduce the power consumption of a device by switching between the device's operation modes.

The ATA standard, also known as IDE, defines four operation modes for hard disks [1]: *active, idle*, *standby* and *sleep*.

- *Active mode:* the hard disk is reading, writing, seeking or spinning up/down.
- *Idle mode:* the disk is rotating and the interface is active. The head is moved to a parking position.
- *Standby mode:* the hard disk spindle motor is off, but the hard disk interface is still active.
- *Sleep mode:* both the hard disk spindle motor and the interface are off. To reactivate, a reset command is needed.

A device is "shut down" if it is put into standby or sleep mode. A non-negligible amount of time and energy is needed to enter and leave these modes. Therefore, entering standby or sleep mode will only pay off if the interval to the next disk operation is long enough. The minimum interval between two disk operations for which switching pays off is called the *break-even time*. It only depends on the characteristics of the hard disk device. A time interval of idleness (an idle period) that is longer than the break-even time is called a *standby period*.

The break-even time of the IBM hard disk we used in our tests is 8.7 s (see section 7.2). Note that this value is device-specific. Lu et al. [15] report break-even times of 6.39 s for a Fuijitsu MHF 2043AT disk and 35.0 s for an Hitachi DK23AA-60 drive.

3 Cooperative File Operations

Contemporary operating systems serve read requests immediately and buffer write requests before storing the data on disk according to a time-out policy. This can cause a hard disk to spin up even if the disk was shut down only a few seconds before. The application developer has no possible way to exert influence on this

mechanism. With Coop-I/O, applications can soften this strict timing to enable a flexible schedule of device requests.

The classical device interface of operating systems has been designed to hide aspects regarding hardware management. The application programmer is not aware of power-saving techniques inside the operating system. With Coop-I/O we abandon this concept to some extent. We enable the application programmer to support the operating system's efforts to save energy. The details concerning the power management algorithm are still hidden from the application layer.

The essential file operations in most operating systems are provided by the system calls `open()`, `read()` and `write()`. The operations `close()` and `lseek()` usually do not access the disk directly, but operate on data in main memory. So we introduce three cooperative variants: `open_coop()`, `read_coop()` and `write_coop()`. The legacy interface, `open()`, `read()` and `write()`, is mapped to the cooperative functions with zero time-out and inactive cancel flag.

The user-specified time-out value indicates when the operation is initiated at the latest, not when the operation has finished. As with the classical Unix file I/O interface the user does not know when the operation will be completed.

3.1 Interactions between the disk cache and cooperative operations

For efficiency reasons, modern operating systems do not serve write requests to disk drives immediately. To reduce the number of slow disk operations, they keep copies of disk blocks (called *block buffers*) in a cache in main memory. A write operation copies data into block buffers which are marked as *dirty* to indicate that they differ from the blocks on disk. The *dirty buffer life span* determines the time when these buffers must be written back to disk to prevent data loss in case of a crash. A special *update* task periodically writes out dirty buffers with an elapsed life span.

Cooperative read operations. If a cooperative read references a disk block that is not cached in memory, the operation will have to check if the corresponding hard disk is in active or idle mode. If it is, the read request can be served immediately. If not, the operation will have to block itself until either the hard disk spins up due to another request or until the time-out has elapsed. If the time-out has elapsed and the *cancel flag* is set, the operation will have to be aborted. Otherwise the drive has to be activated.

Cooperative write operations. As we never modify disk blocks directly (data is always written to the block buffer cache) there seems to be no need for cooperative writes. The update task defers write operations until their dirty buffer life span has elapsed. We can easily modify this mechanism to be cooperative and wait for other device accesses (read operations); see section 4.2.

However, writing to a disk block can induce a read operation if the block is not yet cached and must be read before it can be modified. Thus a shut-down drive would have to run up immediately. In this case, a cooperative write operation will simply wait for the drive the same way as the `read_coop()` operation does.

But the situation is more complicated: if the write operation needs to read an uncached block *after* several modifications of cached blocks and the whole operation has to be cancelled (because the drive is in standby mode, the time-out has elapsed and the operation is declared abortable), we have to undo all previous modifications of cached blocks to assure file system consistency. All modifications issued by a write request can be understood as *one* transaction that must be performed completely (*commit*) or not at all (*abort*).

We decided to use the following approach which avoids the implementation of an undo mechanism: the *early commit/abort* strategy decides to commit or abort as soon as the first modification to a block buffer is going to take place.

Assume we want to modify a buffer and the drive is in standby mode. We can distinguish three situations:

1. The drive is activated due to another request before the time-out of our request is reached. We can commit the whole operation.
2. The drive is still shut down when the time-out of our request is reached. If there are no dirty buffers for the drive, it can be concluded that our request is the only one that wants to access the drive. Depending on the cancel flag, we have to activate the drive or abort the whole operation.
3. If there is another dirty buffer for the same drive (or the buffer to be modified is already dirty), the drive will 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 dirty block buffer is updated to disk, our buffer will be updated in the same sweep, as described in section 4.2.

Therefore, a write to a block buffer should be delayed only as long as the drive is in standby mode *and* there are no dirty buffers for that drive. Since a write operation's first buffer modification involves committing the operation, a write can be committed even if the hard disk is not running.

Due to the early commit/abort strategy it is conceivable that an abortable write operation will not be aborted after the time-out even if the disk is in standby mode. This is the case when a read follows a committed write to a cached block:

1. The disk is in standby mode and there exist dirty block buffers for that device.
2. The write operation has to modify several disk blocks.
3. According to the *early commit/abort* strategy the write operation is committed after the modification of the first block.
4. If a subsequent block is not cached, it has to be read from the disk. This action will be deferred until the time-out has elapsed.
5. As the complete operation is already committed, we have to spin up the disk to read the block.

Fortunately, the energy waste in this case is not that big because the hard disk would run up anyway in the near future to save the dirty buffers.

Cooperative file open operations. Opening a file results in reading its meta data (inode block etc.). If the file has to be created or to be truncated, open will cause write operations. Therefore we decided to provide a cooperative system call to open files. Read and write operations induced by `open_coop()` are mapped to their corresponding cooperative counterparts.

3.2 Using the new system calls

Many applications can be modified to be cooperative so that users will not notice changes in system behavior. Examples are low-priority tasks like cron jobs, logging mechanisms or applications with periodic I/O requests like audio and video players and voice recorders.

Only little effort is needed to make use of the new system calls. Because the cooperative reads and writes may block for the specified delay time, I/O should be decoupled from main processing. An additional thread reads or writes data cooperatively and caches it for the main thread. The main thread reads or writes data from/to this buffer in the same way as it formerly operated on the file system. The additional thread thus hides the cooperative operations from the original application. Our experiments show that the amount of changes is typically in the range of 100 to 150 lines of code.

Applications which write out temporary data and update it periodically can declare their requests as deferrable and *abortable*. An example is the autosave function of a text editor. If a write request is cancelled, the autosave thread can simply ignore it and issue another cooperative write with a time-out equal to the autosave period.

4 Cooperative update

In this section we will show how the caching mechanism can be optimized with respect to power consumption.

4.1 "Traditional" caching of disk blocks

The update process passes the modifications to the cached disk blocks on to the hard disk. In Unix systems an update takes place when one of the following conditions is met:

- An explicit update command like `sync()` forces the system to write back the buffers of a file system or a file.
- The *dirty buffer life span* has elapsed. This is the most frequent cause of writing back when there is little I/O traffic. A dirty buffer whose life span has elapsed is not written back immediately, but when it is found by the update task.
- A certain percentage of block buffers is dirty. To avoid I/O jams, some of them are written back. This is the most frequent cause of 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 common policy is not optimized to save energy. The update task has to access the disk each time it wakes up to write back dirty buffers. If the time between two updates is shorter than the break-even time, standby periods will never be reached. In this case the disk can not be set to standby mode to save energy.

4.2 Batching write requests

In active mode, energy consumption is higher than in idle or standby mode. Furthermore, switching between modes consumes a significant amount of energy. As a

consequence write requests should be batched to maximize the time the device can spend in low-power modes and to reduce mode switches.

To make updates and thus disk requests less frequent, we use a policy which updates each drive independently of all others and is preferably executed when another disk request is generated (*drive-specific cooperative update*). 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 life span (60 s in our implementation).
  An additional, possibly redundant write operation will have only marginal costs if it is batched with other requests.
- *Update cooperatively.* The operating system tries to join other hard disk accesses (read requests) and write back the dirty buffers possibly before the dirty buffer life span has elapsed. This or the expiration of the full life span will trigger the update process.
  By attaching to a request that has to happen anyway, we can update at very little cost.
- *Update each drive separately.* 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 be 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 life span, we will write back all buffers for that drive.
- *Update on shutdown.* If the operating system has decided to shut down a drive, it will first write back all dirty buffers that contain blocks of that drive. This minimizes the risk that the disk has to spin up again soon solely because there are some old dirty buffers that must be updated.

When an application reads some data from disk, it normally needs the data for further processing. Thus read operations are batched only if the application permits it by using `read_coop()`.

5 An adaptive shutdown policy

Many power management policies for hard disks have been suggested varying in complexity and usefulness (see the related work discussed in section 8). If the timing of hard disk requests cannot be influenced, the imaginary "*oracle*" policy is the policy that reaches maximum energy savings. It 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. To achieve this, the oracle policy needs information on future hard disk requests. This policy can only be simulated by analyzing request traces offline. The operating system of course is not able to perfectly predict the future and thus cannot achieve maximum reduction in energy consumption.

The spin-down policy is not central to our work because our focus lies on cooperative I/O and cache management. Thus we decided to choose a simple and easy to implement, but effective and proven, algorithm. Simple shutdown policies switch to a low-power mode after a fixed or adaptively changed time-out. As is stated in [15], 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. It is defined in the following way:

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 time of the last hard disk access

t_{be}:the break-even time

Figure 2: the DDT policy

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

6 Prototype Implementation in Linux

We have implemented the Coop-I/O concept in the Linux kernel, version 2.4.10. The kernel modifications can be divided into three parts:

- The VFS and the Ext2 file system have been modified to support the drive-specific cooperative update policy of section 4. We have also introduced cooperative system calls using the concept of section 3 (see sections 6.1 & 6.2).
- The block device code, which is the glue between a particular block device driver and the file system, has been augmented to enable cooperation between the disk drivers' power mode control, the file system's update mechanism and the cooperative file operations.
- The IDE driver has been enhanced by a power mode control for hard disk drives, which includes the DDT algorithm of section 5. See section 6.3 for a detailed description.

6.1 Cooperative file operations

A file operation may block whenever it is going to access a disk or to make a clean block buffer dirty by modifying it. The blocking mechanism is implemented in the new function `wait_for_drive()`.

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

- *The timer has elapsed.*
  If the request should be cancelled on time-out, `wait_for_drive()` will return `-ETIME`.
- *The drive is serving another request.*
  The file operation can go on.
- *The number of dirty buffers for the drive has become non-zero.*
  If `wait_for_drive()` is also waiting for that event, it will simply return without error. If not, it will be 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's implicit restart mechanism since the signal could be sent to abort it.

The implementation of the cooperative file operations (`open_coop()`, `read_coop()`, `write_coop()`) is straightforward: The functions that implement the standard file operations have to be enhanced by the `time-out` 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. We chose to implement the early commit/abort strategy as described in section 3.1.

6.2 Drive specific cooperative update

We have implemented the drive-specific cooperative update policy (see section 4.2). Since the file system does not know about drives, we 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.

The update task in the original Linux wakes up every 5 s. The cooperative 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 4.2. The need for a cooperative update is checked 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 life span, the update task will be woken up and induced to update that drive.

6.3 Power mode control for IDE drives

Drive-specific information. For each hard disk, the Linux IDE driver keeps a description that reflects the properties and state of that device. We have augmented the device structure with information needed by the DDT algorithm (break-even time and time of the last access) and a field indicating the current power mode.

The IDE power task. A power mode switch might take a rather 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. Instead of a mode switch function that could block for a long time, we 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. In this case it will wake up and emits a power mode command to the hard disk. 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. Some functions like disk operations change the power mode implicitly by emitting other IDE commands, so they must inform the power task.

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

- A hard disk request is sent to the device driver. This implicitly changes the drive's power mode to active.
- The DDT standby algorithm decides to shut down the drive.

Some special IDE commands leave the disk drive in an undefined power mode, so they request the power task to check.

Going to standby. The DDT algorithm is implemented as a timer-based function that is called once per second. Since disk requests may be very frequent, this is more efficient than using a dedicated timer for each drive that has to be restarted when a disk request has been served.

The information that is needed by the DDT algorithm (time of the last access) is updated with each disk request.

7 Experiments and Results

7.1 Data acquisition

To validate the Coop-I/O concept a power measuring environment was needed. A comparatively inexpensive four-channel analog-to-digital converter (ADC) has been designed and built to act as the data acquisition system. It measures the voltage drop at defined sense resistors in the 5 V lines leading from the power supply to the hard disk and interfaces to the standard parallel port on the data acquisition system to output the digital values. The voltage drop is acquired with a resolution of 256 steps and at a rate of up to 20000 samples per second. The maximum voltage drop that is correctly converted is 50 mV.

The target computer was a standard personal computer running Linux (kernel version 2.4.10). The system was equipped with a 2,5" IBM Travelstar 15GN (IC25N010ATDA04) hard disk [9] which was used as the test drive.

7.2 Determining hard disk parameters

We measured the power consumption of different mode switches of the IBM hard disk with our data acquisition system. The Travelstar splits up the idle mode into three submodes (*performance, active* and *low power*) that have different power consumption and timing characteristics. Observing recent user request patterns, an internal adaptive algorithm switches autonomously between these modes [9].

Figure 3 shows the power consumption of the IBM hard disk during an idle-standby-idle turnaround.

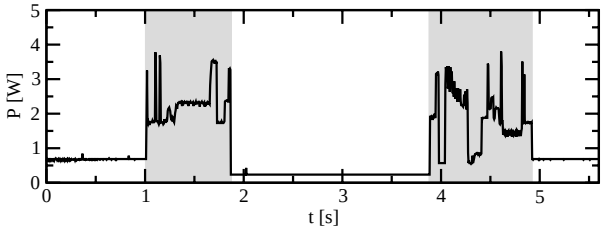


Figure 3: idle-standby-idle turnaround

t = 1s: The disk receives a shutdown command. The shaded region shows the hard disk switching from *low power idle* to *standby* mode.

t = 1.8 s: After stopping the spindle motor, the disk has reached *standby mode*, and power consumption drops to about 0.25 W.

t = 3.8s: The drive receives a write command and starts to spin up. The shaded region shows the hard disk switching from standby mode to *active mode*. Starting the spindle motor is quite expensive, energetically. After 1 s, the disk has spun up and may serve read or write requests.

t = 4.8 s: In this test scenario only a single disk block gets written. Then, the disk switches to *low power idle mode*.

We determined the following characteristics for the IBM Travelstar 15GN drive. Due to the undocumented internal adaptive algorithm of the firmware the time and energy values vary according to the recent access pattern. The following values present the average of several measurements:

time needed for a mode switch:
- low-power idle to standby: $t_{is} = 0.85$ s
- standby to low-power idle: $t_{si} = 1.03$ s

energy required to switch between modes:
- low-power idle to standby: $E_{is} = 1.91$ J
- standby to low-power idle: $E_{si} = 1.89$ J

power consumption of the low-power modes:
- low-power idle mode: $P_i = 0.657$ W
- standby mode: $P_s = 0.235$ W

break-even time: $t_{be} = 8.7$ s

7.3 Testing a cooperative audio player

We examined to what extent Cooperative I/O is able to save energy in a real-life situation. A typical application for hand-held or portable computers is a player for audio or video files. We have tested the system with a modified version of AMP, an MPEG audio layer 3 player for Linux, which makes use of the cooperative system calls. Thus we gained insight into the procedure and amount of modifications needed to make applications energy-aware.

The modified AMP creates a thread that reads from the hard disk and puts the data into a 512 kB buffer. When the player thread needs some data it reads from the buffer. The read thread and the player thread synchronize 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 cooperative system read call while the player reads from the other semi-buffer. We had to add only about 150 source lines to incorporate the changes.

AMP was tested under the following four strategies:

- *Cooperative:*
  Use the DDT standby algorithm together with the new buffer cache and update mechanism. 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.
- *ECU* (*E*nergy-aware *C*aching & *U*pdate):
  Use the DDT standby algorithm together with the new buffer cache and update mechanism. Use the standard `read()` system call instead of `read_coop()` to read in new data.
- *DDT only:*
  Use the DDT standby algorithm with the *original* buffer cache and update mechanism.
- *None:*
  Do not use any power-saving measures at all.

In addition to that we simulated the "*uncooperative oracle*" policy. We collected traces of hard disk requests issued by the original uncooperative AMP running on an unmodified Linux. We calculated the minimum in total energy consumption according to the following assumptions:

- The hard disk will be set to standby mode *immediately* after serving a request if the following idle period is a standby period, i.e. if it is longer than the break-even time.
- Otherwise the hard disk is not shut down. It switches immediately to low power idle mode.

The values for "oracle" resemble the theoretical lower bounds of power consumption that can be reached by shutdown policies *without* influencing the timing of requests (in contrast to Coop-I/O).

Each strategy has been tested by playing the following two audio files. Delay is the time interval in which one semi-buffer is played. The files have the same length (9 minutes), but different compression levels.

audio file	bit rate	delay
Toccata	64 kb/s	32 s
Pastorale	128 kb/s	16 s

We 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 minute. If there is any mail in it, the mail will be stored in the local mailbox on the test hard disk. Mail was sent in intervals of 15–60 seconds; the timing was controlled by a pseudo-random generator. For every test pass, the random generator was initialized to the same value, so the timely sequence of read/write operations was the same for each test with a tolerance of about one second. Figure 4 shows the results (all tests run for 534 seconds).

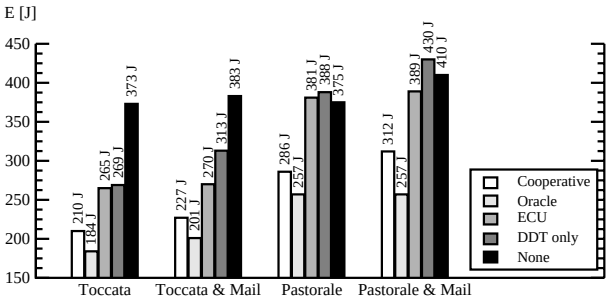
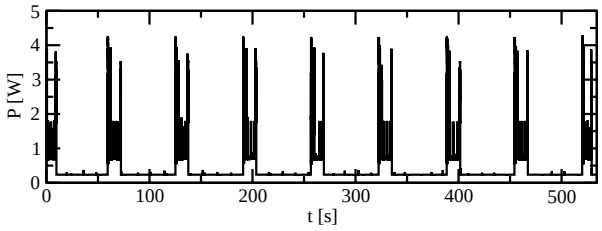


Figure 4: comparison of policies

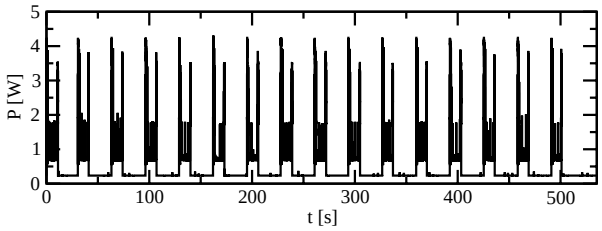
The cooperative strategy is surprisingly power-efficient in these tests. This is not only caused by the cooperation of multiple processes because some tests have only one process doing I/O. Instead, it can be explained by the following behavior: When the drive is in standby mode, a cooperative read is 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 because the hard disk drive is still in idle or active mode. This effectively batches two subsequent read operations. You can see this behavior in figure 5.

Playing "Pastorale" together with the mail reader consumes nearly the same energy, regardless which non-cooperative strategy is used. Because the delay for this audio file is only 16 s and there are several write requests, no strategy will normally try to shut down the hard disk in this test scenario. This is due to the short intervals between requests which seldom exceed the break-even time. Coop-I/O again groups requests together, so that longer idle periods and thus *standby periods* are achieved. As a consequence the drive can be set to standby mode more often and the energy consumption is reduced.

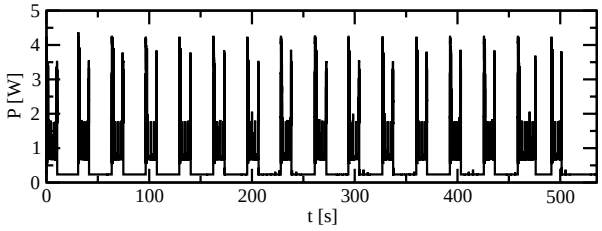
If we have a look at the times spend in active, idle and standby mode, it can be seen that "Oracle" saves more energy than "Cooperative" by keeping the drive in standby mode all the time it is not accessed (table 1). This is due to the DDT policy which always waits 8.7 seconds before setting the drive to standby mode. The oracle policy will shut down the drive immediately if the following idle period is longer than the break-even time.



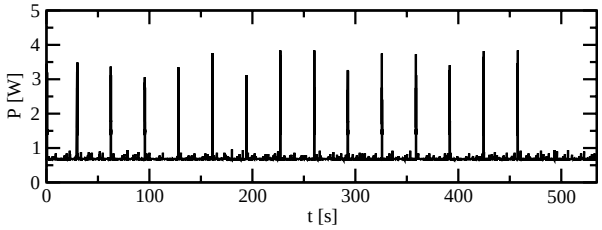
a) Cooperative (210 J)



b) Energy-aware Caching & Update (265 J)



c) DDT only (269 J)



d) None (373 J)

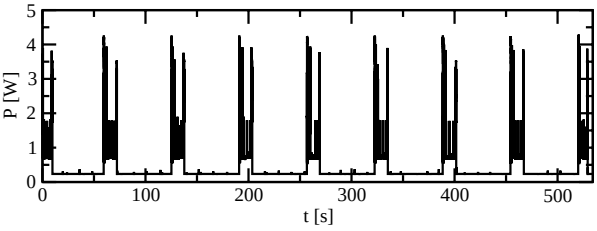
Figure 5: the four energy saving strategies (playing “Toccata”, without Mail)

policy	active	idle	standby
Cooperative	38 s	166 s	331 s
Oracle	89 s	0 s	456 s

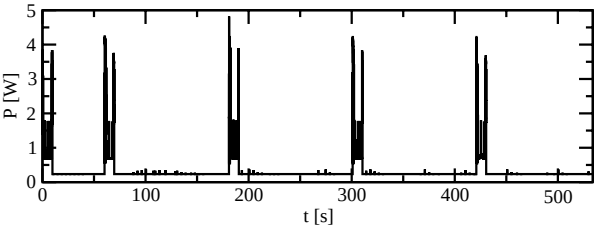
Table 1: times spent in the different power modes when playing “Pastorale”

Furthermore it can be seen that “Cooperative” reduces the time spent in active mode by almost 60%. There is almost no difference in energy consumption between the strategies “DDT only” and “None” when playing “Pastorale”. This behavior is caused by the unlucky relation of the shutdown time-out and the hard disk request pattern which sets the disk to standby shortly before the disk is accessed. In that case, switching to standby mode is more expensive than staying in idle mode.

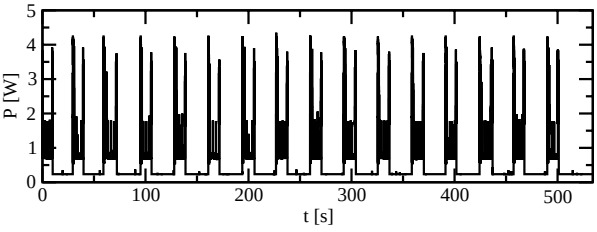
Figure 6 shows how disk requests of two independent tasks may interact. The “Toccata” task cooperatively reads one semi-buffer in every period of 32 s (figure 6a); the “Mail” task writes in intervals of 1 minute, provided that mail has arrived (figure 6b). The write requests are delayed by the cooperative update scheme (figure 6c). With Coop-I/O most of the requests of the two applications can be grouped together. As a consequence, the energy consumption of “Toccata & Mail” is only 17 J higher than without the mail application. If the requests were not coordinated, the hard disk’s energy consumption would be about 60 J higher (figure 6d). This means that Coop-I/O is a working energy-saving concept.



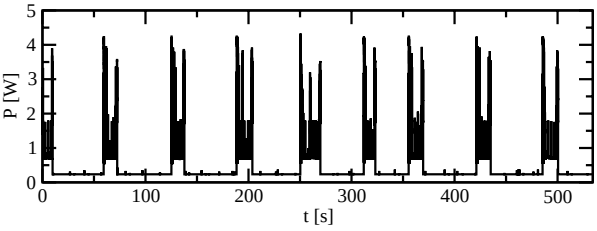
a) Toccata cooperative (210 J)



b) Mail cooperative (164 J)



c) Toccata & Mail non-cooperative (270 J)



d) Toccata & Mail cooperative (227 J)

Figure 6: interaction of two independent tasks

7.4 Parameterized tests

To simulate a workload where multiple tasks periodically read or write data, we implemented two simple test programs. The period and the idle time (the time to wait at the beginning of a period until the read/write operation is started) can be configured. To simulate non-regular behavior, minimum and maximum values for the idle time can be specified; the actual value is chosen by a pseudo-random generator. Groups of five read/write processes with varying period lengths and idle times were used to generate non-regular hard disk request patterns. We measured the energy consumption for a time window of 1000 seconds (figures 7 and 8).

Reads with varying period length. In the first test series, we have varied the period length and idle times for read operations. The average period lengths of the six tests range from 25 to 150 seconds, the average idle times lie between 7 and 120 seconds. Each test has been executed in combination with the four power-saving strategies. Figure 7 shows the measured consumptions.

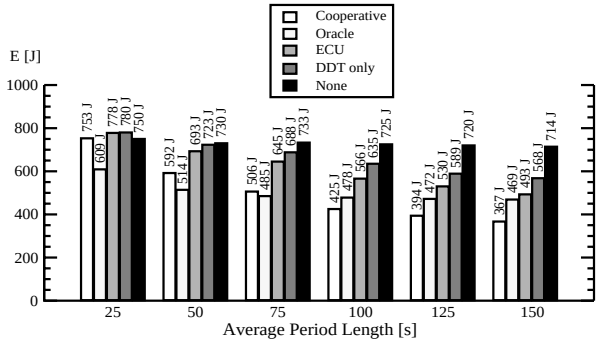


Figure 7: reads with varying average period length

Running with an average period length of 25 seconds, the energy consumption is nearly equal for all strategies (except "Oracle"). Here, the read requests are so frequent that the drive has no chance to shut down. "ECU" and "DDT only" consume even more power than "None" by initiating disadvantageous shutdowns. The longer the period length, the longer the power mode control can hold the disk in standby mode for all strategies but "None". The cooperative strategy generates the longest standby periods since following cooperative reads are delayed.

"Cooperative" even outperforms the oracle policy for average period lengths of 100 seconds and more. Table 2 shows the times spent in active, idle and standby mode for an average period length of 150 seconds. The cooperative policy batches hard disk requests. Consequently mode switches are less frequent and the time spent in active mode is reduced by more than 70%, while the time spent in low-power modes is increased. Thus "Cooperative" saves more energy than the oracle policy.

policy	active	idle	standby
Cooperative	29 s	153 s	868 s
Oracle	107 s	132 s	811 s

Table 2: times spent in the different power modes for an average period length of 150 seconds

Writes with varying period length. The energy-related characteristics of write operations are influenced by the sequence of requests and the update policy of the buffer management.

We have executed the same test series as in the previous section using write operations instead of read operations. Again, the tests have been run under all four strategies. The traces for the oracle policy where collected using the unmodified update mechanism. The results are presented in figure 8.

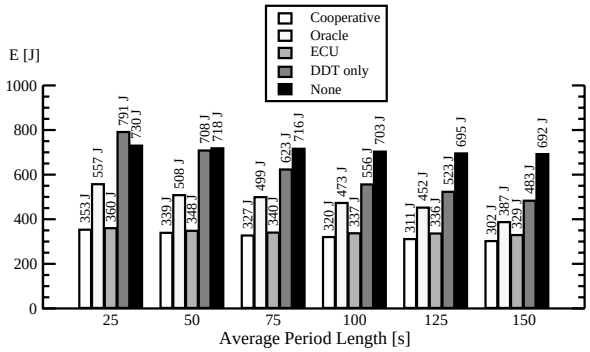


Figure 8: writes with varying average period length

The frequency of write operations seems to have little effect on the energy consumption. For the strategies "Cooperative" and "ECU" the consumption stays constantly on a low level for all period lengths. This is caused by the cooperative update policy that will write back quite regularly in intervals of 60 seconds, if no other disk request happens. The amount of data written has almost no influence on the energy consumption. The small difference between the cooperative and the "ECU" strategy indicates that cooperative write operations have a marginal but at least consistently positive benefit. The policies "DDT only" and "None" employ the original Linux 2.4 update strategy. "ECU" consumes significantly less energy than "DDT only" and "None" because of the improved update policies (see section 4.2). This shows that the original Linux update strategy does not match power-saving requirements.

7.5 Varying the number of cooperative processes

In the previous tests all programs were cooperative. But in a real-life scenario non-cooperative legacy applications will mix with cooperative programs. Thus, we have examined the behavior of a mixture of different numbers of cooperative and non-cooperative processes. Figure 9 presents the results of 6 read tests with a mixture of 5 processes including 0 to 5 cooperative applications.

The energy consumption steadily declines with increasing proportion of cooperative processes, but having a single cooperative process suffices to save energy

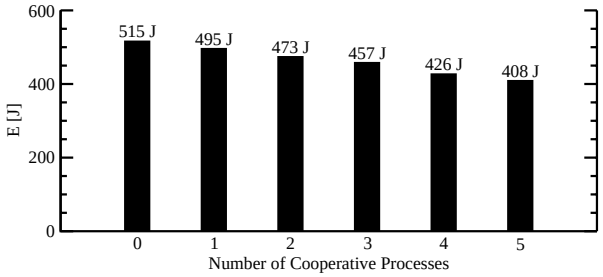


Figure 9: varying number of cooperative processes

8 Related Work

We distinguish three levels on which energy saving techniques can be applied: the hardware, the operating system and the application layer. We will show how power management is performed on each level and how Coop-I/O and existing techniques can be incorporated into an energy-aware system design.

Device-level power management. Attention has first been drawn to the lowest level leading to many research and industrial efforts at developing low-power hardware. Modern-day devices and their components can be shut down or put into low-power modes to save energy when they are not in use. These improvements have come along with OS-level support in the form of heuristic hardware-centric shutdown policies which are transparent to applications.

Three approaches to hard disk power management can be distinguished: algorithms with a fixed or adaptive time-out, policies that try to predict future requests by observing the utilization of the device and policies based on statistical models of requests. Several shutdown policies have been suggested [4],[7],[8],[10],[11],[14]. Lu et al. [15] have compared and evaluated several algorithms. Among commercial solutions to OS-level power management are the Advanced Configuration and Power Interface (ACPI) [3], Microsoft's OnNow [16] and ACPI4Linux [2].

Energy-aware operating system. As the next step more sophisticated energy-related methods have been introduced lifting energy to a first class operating system resource by unifying resource management, introducing energy accounting and enabling control of energy consumption and battery lifetime. These approaches have explicitly been designed to be not dependent on application software to be rewritten.

Zeng et al. present ECOSystem [19], a modified Linux, that unifies energy accounting over diverse hardware components and enables fair allocation of available energy among applications. ECOSystem provides system-wide energy management and can be configured to hit a specified target battery lifetime. Energy accounting is realized by implementing the powerful concept of resource containers which serve as the abstraction to which energy expenditures are charged. As Zeng et al. propose in their paper, future research should consider the interaction between ECOSystem and the applications. We find it interesting to investigate the potential of combining Coop-I/O and ECOSystem, being orthogonal to each other, to come closer to the vision of incorporating energy management in all levels of system design.

Cooperation between the OS and applications. While traditional power management schemes in operating systems do not distinguish different sources of requests, the power reduction technique introduced by Lu et al. [12] uses information about concurrently running tasks as an accurate system-level model of requesters. This *task-based power management* records the device and processor utilization of each process and shuts down a device when the overall utilization is low. Being somewhat orthogonal to each other it would be interesting to combine Coop-I/O and task-based power management.

Another approach by Lu et al. [13] is based on application involvement in energy management. New system calls are introduced which enable applications to inform the operating system about future hard disk requests. These system calls are similar to timers; they indicate in which time intervals, how often (once, periodically) and with which possible delay requests are issued. Thus applications have to inform the operating system about future requests *before* these requests are issued, normally at program start.

The cooperation policy of Coop-I/O enables processes to pass the *urgency* (delay time) of each individual request without the need for the programmer to

determine request patterns of the whole program run. Figure 10a shows a scenario with four processes; number one issues several hard disk requests. While Lu's approach (Fig 10b) arranges the schedule of processes to create a cooperative schedule of hard disk requests, Coop-I/O (Fig 10c) schedules the hard disk requests themselves without changing the process schedule.

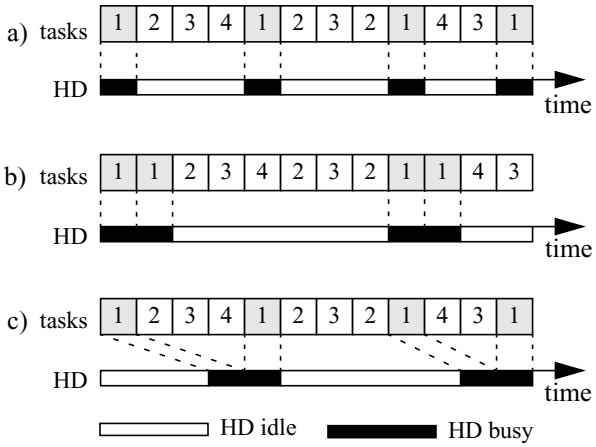


Figure 10: different request batching policies

Pouwelse et al. [18] demonstrate the benefits of power-aware applications. Applications can inform the OS about their processing demands, so the optimal processor speed can be selected that minimizes power consumption and still meets the application's deadlines.

Application adaptation. Orthogonal to our approach is *energy-aware application adaptation* presented by Flinn et al. [6],[17]. Applications are rewritten so that they can dynamically modify their behavior according to changing restrictions in energy consumption. An example of adaptation is a movie player with different quality of service modes (e.g. refresh rates or window sizes). To guide such adaptation, the operating system monitors energy supply and demand and informs applications about restrictions in energy consumption via upcalls.

Energy-aware system design. All four approaches—energy-aware hardware, operating system, application adaptation and Coop-I/O—are complementary to each other and can be used to form a comprehensive energy-aware system de-sign, as shown in figure 11.

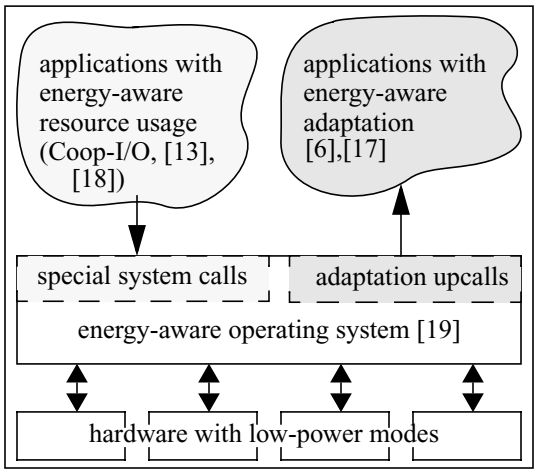


Figure 11: comprehensive energy-aware system design

9 Conclusion

In this paper we have presented an energy-saving concept for devices with low-power operation modes based on a cooperative relationship between the operating system and applications. While many efforts have been made to incorporate energy-related aspects in the design of operating systems and computer hardware, the interface between the OS and applications has been retained unchanged to make power management fully transparent to applications.

We have demonstrated that Coop-I/O, with its new interface functions, enables a higher reduction in power consumption than power management techniques which operate solely on the OS level. Coop-I/O outperforms the "oracle" policy which defines the theoretical lower bound in power consumption if the timing of requests is not influenced. These improvements come with the cost of modifying existing applications. We have shown that only little effort is needed to make use of the new functionality. Coop-I/O is the first approach to power management which controls not only the timing but also the execution of requests.

We plan to investigate the applicability of our concept to the control of other system components, e.g. a wireless network adapter. Our modifications to the operating system interface presented here concentrate on the basic file I/O operations. We plan to investigate to what extent other interface functions can be enhanced so that a wider range of energy-aware applications is able to contribute to OS power management. In our opinion the research area of application involvement in operating system power management facilities shows huge potential and should be further investigated.

Acknowledgments

The anonymous reviewers and our shepherd, Carla Ellis, have helped us to improve this paper with their useful feedback.

References

[1] American National Standards Institute. Information Technology – AT Attachment with Packet Interface 5 (ATA/ATAPI-5). Published as ANSI/INCITS 340-2000, Dec 2000

[2] M. Berger, S. Richter, ACPI4Linux. http://phobos.fs.tum.de/acpi/index.html

[3] Compaq, Intel, Microsoft, Phoenix, Toshiba. Advanced Configuration and Power Interface Specification 2.0a, Mar 2002

[4] F. Douglis, P. Krishnan, B. Bershad. Adaptive Disk Spin-down Policies for Mobile Computers. In *Proceedings of the Second USENIX Symposium on Mobile and Location Independent Computing*, Apr 1995

[5] F. Douglis, P. Krishnan, B Marsh. Thwarting the Power Hungry Disk. In *Proceedings of the 1994 Winter USENIX Conference*, Jan 1994

[6] J. Flinn, M. Satyanarayanan: Energy-aware Adaptation for Mobile Applications, In *Proceedings of the 17th Symposium on Operating Systems Principles SOSP'99*, pp. 48–63, Dec 1999

[7] P. Greenawalt. Modeling Power Management for Hard Disks. In *Proceedings of the Symposium on Modeling and Simulation of Computer and Telecommunication Systems*, Jan 1994

[8] D. Helmbold, D. Long, B. Sherrod. A Dynamic Disk Spin-Down Technique for Mobile Computing. In *Proceedings of the 2nd ACM International Conference on Mobile Computing (MOBICOM96)*, pp. 130–142, Nov 1996

[9] IBM Corporation. Hard Disk Drive Specifications for Travelstar 48GH, 30GN & 15GN, Rev. 2.0, Jan 2002

[10] P. Krishnan, P. Long, J. Vitter. Adaptive Disk Spin-Down via Optimal Rent-to-Buy in Probabilistic Environments. In *Proceedings of the 12th International Conference on Machine Learning*. pp. 332–330, July 1995

[11] Li-K; Kumpf-R; Horton-P; Anderson-T. A Quantitative Analysis of Disk Drive Power Management in Portable Computers. In *Proceedings of the USENIX Winter 1994 Conference*, pp. 279–292, Jan 1994

[12] Y.-H. Lu, L.Benini, G. De Micheli. Operating System Directed Power Reduction. In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design ISLPED'00*, pp. 37–42, July 2000

[13] Y.-H. Lu, L. Benini, G. De Micheli. Power-aware Operating Systems for Interactive Systems, In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 10(2):119–134, Apr 2002

[14] Y.-H. Lu, G. De Micheli. Adaptive Hard Disk Power Management on Personal Computers. In *Proceedings of the Ninth IEEE Great Lakes Symposium*, pp. 50–53, Mar 1999

[15] Y.-H. Lu, G. De Micheli: Comparing System-Level Power Management Policies. In *IEEE Design & Test of Computers. Special Issue on Dynamic Power Management of Electronic Systems*. pp. 10–18, Mar-Apr 2001

[16] Microsoft. OnNow Power Management http://www.microsoft.com/hwdev/onnow/

[17] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, K. R. Walker. Agile Application-aware Adaptation for Mobility. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles SOSP'97*, pp. 276–287, Oct 1997

[18] J. Pouwelse, K. Langendoen, H. Spis. Dynamic Voltage Scaling on a Low-power Microprocessor. In *Proceedings of the International Symposium on Mobile Multimedia Systems & Applications MMSA'2000*, Nov 2000

[19] H. Zeng, X. Fan, C. Ellis, A. Lebeck, A. Vahdat: ECOSystem: Managing Energy as a First Class Operating System Resource, In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct 2002