FlexFetch: A History-Aware Scheme for I/O Energy Saving in Mobile Computing

Feng Chen[1], Song Jiang[2], Weisong Shi[3], and Weikuan Yu[4]

[1]Dept. of Computer Science & Engineering
The Ohio State University
Columbus, OH 43210, USA
fchen@cse.ohio-state.edu

[2]Dept. of Electrical & Computer Engineering
Wayne State University
Detroit, MI 48202, USA
sjiang@eng.wayne.edu

[3]Department of Computer Science
Wayne State University
Detroit, MI 48202, USA
weisong@cs.wayne.edu

[4] Oak Ridge National Laboratory
Computer Science and Mathematics
Oak Ridge, TN 37831, USA
wyu@ornl.gov

Abstract

Extension of battery lifetime has always been a major issue for mobile computing. While more and more data are involved in mobile computing, energy consumption caused by I/O operations becomes increasingly large. In a pervasive computing environment, the requested data can be stored both on the local disk of a mobile computer by using the hoarding technique, and on the remote server, where data are accessible via wireless communication. Based on the current operational states of local disk (active or standby), the amount of data to be requested (small or large), and currently available wireless bandwidth (strong or weak reception), data access source can be adaptively selected to achieve maximum energy reduction. To this end, we propose a profile-based I/O management scheme, FlexFetch, that is aware of access history and adaptive to current access environment. Our simulation experiments driven by real-life traces demonstrate that the scheme can significantly reduce energy consumption in a mobile computer compared with existing representative schemes.

1 Introduction

An appealing feature of pervasive computing is its ability to access consistently maintained set of data and programs anywhere via network connections. Relying on this feature, people can keep working on their data without disruption when they move around. In the application of pervasive computing, a commonly used technique is data hoarding, where commonly used set of data are replicated on user's mobile computer, so that data can be accessed from the computer's local disk when the user is on travel or even disconnected, and the local data can be kept consistent by a replication system [11]. When data are available both on the local disk and on the server via wireless network, an intuitive choice would be to access the data from the local disk to save communication cost. However, this choice can be sub-optimal in terms of energy consumption, which can be a major concern because mobile devices are constrained by the limited lifetime of their batteries. The reason of the sub-optimality is that the hard disk is energy inefficient if the disk is in the standby state and only a small amount of data is accessed there.

1.1 Energy Saving Models in the Hard Disk and Wireless Card

Typically, a hard disk has four power-consumption states — *active*, *idle*, *standby*, and *sleep*. A disk carries out its reading or writing in its active state. In the idle state disk platters keep spinning with no data transfer. In the standby state the disk is spun down, disk electronics are partially un-powered, and disk heads are unloaded or parked. In the sleep state all the remaining electronics are powered off, and a hard reset is needed to reactivate the disk. To save energy, disk is spun down to the standby state if it has been idle for a threshold period of time, and to service a request it has to be spun up to the active state. We call the time period between the disk spin-up and its adjacent spin-down a *busy period*, and the time period between the disk spin-down and its adjacent spin-up a *quiet period*. Because of substantial energy consumption with disk state changes, the quiet period must

be sufficiently long to justify the energy cost for the disk spin-down/up. The minimum length of quiet period to pay off the cost is referred to as the *break-even* time. That is, if a disk stays in the standby state for a period of time that is less than the break-even time, it consumes more energy, rather than saves energy, by spinning down the disk. If data requests are always first attempted at the hard disk, quiet periods can be broken into pieces that are shorter than the break-even time and render the spin-down energy-saving efforts fruitless, or even harmful. In addition, it can take a delay of about one second or more for a spun-down laptop disk to service the first byte of a request [15].

The 802.11 wireless network interface card used in a mobile computer has a power consumption comparable to that for the hard disk. Compared with disks in terms of the energy saving model, a wireless card has its unique characteristics: (1) There are two energy modes: continuously-aware mode (CAM) and power-saving mode (PSM). In the CAM mode, the wireless card keeps active with a high power consumption. In the PSM mode, the wireless card turns radio off and periodically wakes up to check with access point. Data transmission can be carried out in both CAM and PSM, but with different latencies and bandwidths. While there is also time and energy cost for the mode changes, the cost is smaller than that for the hard disk. For example, spinning down a Hitachi DK23DA hard disk to the standby state needs 2.3 seconds and 2.94 joules energy. In contrast, a Cisco Aironet 350 wireless card takes only 0.41 seconds to switch from CAM mode to PSM mode with an energy cost of only 0.53 joules. This characteristic suggests that wireless connection can be a good choice for data access when the hard disk is not ready (in the standby mode) or is not worth being spun up to service only a small amount of data. (2) Wireless network bandwidth is usually lower than disk bandwidth. In addition, wireless network bandwidth may be changing with the variation of reception strength when user changes the location of his computer. This characteristic suggests that the network is preferred only when a small amount of data is requested and this amount is related to the current network bandwidth.

1.2 Tracking File-access History to Predict I/O Burst Size

Selecting an appropriate I/O source, either network or local disk, to service I/O requests based on the current disk state has been proposed previously. One representative example is BlueFS [15]. As a distributed file system, BlueFS duplicates data across multiple storage devices. For each request, it selects a target device currently of the lowest access cost. For example, if the hard disk is in the standby state and accessing the disk may raise high overhead, BlueFS would dispatch the request to the wireless network card in the CAM mode. There are several limitations in the scheme: (1) tracking recently received requests can only tell which device should *have* been used for the optimal energy consumption, but cannot confidently predict how much data *will* be accessed in the future. (2) This scheme cannot predict temporal distribution of files to be requested, including the distribution of think times between data requests, which is important because it directly affects energy and time of servicing the requests. As an example, a sequence of intermittent requests for files are preferred to be serviced through network because of its low utilization of disk bandwidth and high energy consumption for disk to stay idle waiting for the next request to arrive. (3) Without recording and analyzing file access history, BlueFS has to reactively make decisions by evaluating *opportunity cost*, or the time and energy wasted by using the current device (e.g., network) instead of activating the other device (e.g., disk). In our scheme, a proactive decision about device use is made at a very early stage to maximize its effectiveness by taking file access history into account. In fact, due to the stability of a program's behavior, the files accessed by a program at different execution stages as well as their access patterns are well predicable, and such techniques have been successfully applied in some areas such as prefetching in Web proxies [3].

1.3 Our Solution: Adaptive Selection of I/O Data Source

To address the aforementioned issues, we proposed a history-aware and environment-adaptive scheme called *FlexFetch*. The scheme tracks and records a program's file access history, which is independent of the I/O devices used or other environment settings such as current wireless bandwidth. The recorded file access profile is then used to adaptively determine the I/O source to service the requests in the next run of the program. FlexFetch is also adaptive to the change of environment, such as a forced disk spin-up or change of program's data request patterns by constantly re-evaluating its decision about I/O data source.

In the next section, we present the detailed scheme design. In Section 3 we describe the performance evaluation of FlexFetch with its comparison with the fixed data source schemes and the scheme adopted by BlueFS. We will briefly describe additional related work in Section 4. Section 5 concludes the paper.

2 The Design of the FlexFetch Scheme

There are three steps in FlexFetch: (1) profiling the execution of programs about their I/O and computation behaviors, (2) using the profiling information to determine which data source, disk or network, should be used for current data requests, and (3) constantly updating a program's current

execution profile and re-evaluating/adjusting the decision made in Step (2).

2.1 Profiling Program's Executions

The purpose of the profiling is to obtain the execution information that can be used to estimate execution times and energy costs with different I/O sources. A program execution alternates between I/O requests and computations. A program's life time consists of I/O times and compute times (or think times). The lengths of think times, as well as their contribution to the entire program's execution time and energy cost are not subject to I/O source. So we focus on profiling I/O behaviors in a program's execution. There are several requirements for the profiling: (1) A profile must be usable in estimating the I/O service times and energy costs for various storage devices. This means that the profile should be independent of I/O devices. (2) A profile must contain data access information that accommodates system dynamics in the data source evaluation. For example, applications' requests for data that are resident in system buffer cache should not incur accesses to storage devices. (3) A profile of a program should be divided into multiple periods, so that the profile for each period can be used separately to reflect changes from one execution to another execution, and the decision about I/O data source can be adaptively adjusted from one period to another period.

In the FlexFetch scheme, we monitor and track file read/write system calls made by a process, which provides file path-name, offset, and size of each request. If running a program involves only one process, we can associate all the file accesses from the process with the program. If there are multiple processes created when a program is running (such as *make*, which could generate multiple *gcc* processes concurrently to build executable), we leverage process group structure in Linux to associate all the file accesses requested by processes belonging to the same group with the program. Though profiling I/O system calls can precisely characterize applications' behavior, the profile cannot be directly used to estimate request service time in real systems. This is because operating systems usually conduct many optimizations on the I/O path, and the requests are not necessarily serviced in their arrival order. For example, most OS kernels prefetch sequential data in files, which helps translate interleaved accesses into multiple sequential streams [5]. Also, I/O schedulers attempt to re-arrange pending requests and merge requests for contiguous data blocks to achieve high throughput [9].

To estimate the service time of I/O requests, we adopt the concept of I/O burst, in which (1) we assume sequential data in a file that are requested in the same I/O burst are accessed in a device's peak bandwidth, even though these data may be requested by multiple system calls and these system calls may be interleaved with other system calls asking for data in other files. This is the expected consequence of I/O scheduling and I/O prefetching. (2) the (think) time gap between any two adjacent read/write system calls in an I/O burst can be masked by the prefetching I/O time or can be considered as negligible relative to the I/O cost. So we do not count small think times in an I/O burst into the total execution time. To this end, we define an I/O burst as a sequence of read/write system calls where the think time is less than the I/O burst threshold. In our experiments we set the threshold as the disk access time, i.e., the average time to receive the first byte of a random request on disk. We conservatively assume that when a think time exceeds the threshold, the time cannot be masked or neglected, so the following access belongs to a new I/O burst. Multiple requests that sequentially access the same file are merged into one request of size up to 128KB, the maximum prefetching window size in Linux, to simulate the prefetch effects. In the case of the local disk, we assume that sequential data in a file are usually contiguously laid out on disk [14]. In the case of the wireless card, we assume the access bottleneck is at the wireless connection rather than storage devices at the server.

2.2 Deciding Data Source based on Profile

By monitoring execution of a program,[1] we obtain its profile including a series of I/O bursts, and think times between these I/O bursts. To characterize the behaviors of a long running program in an appropriate granularity, we collect continuous I/O bursts, including think times between them, whose length just exceeds a pre-determined threshold, say 40 seconds used in our experiments, into an *evaluation stage*. Profile-based decision about data source can be made for each stage, which allows the profile accuracy and decision correctness to be evaluated in a timely manner and necessary adjustments can be made accordingly. When we have the profile and the program is to enter its first evaluation stage, we estimate the execution times and energy costs for the stage assuming disk or network is used, respectively, to determine I/O data source, using the profile that has been recorded for the program.

In order to estimate execution times and energy costs for servicing I/O requests on various data sources, we need to calculate the length of period of time when a device stays at each power mode. To this end, we maintain an on-line simulator for each device to emulate their power saving policies. Such simulation causes minimal overhead, since only a small amount of computation is needed in every 40-second stage.

[1] It does not matter where the I/O accesses are directed to disk, network, or both in the execution because we only need to record its read/write system calls and lengths of think times.

According to the power saving models as described in Section 1.1, if the idle period between two consecutive I/O requests is larger than a time-out threshold, the device switches from a high-power mode (the active/idle mode for the disk and the CAM mode for the wireless network card) to a low-power mode (the standby mode for the disk and the PSM mode for the wireless network card). For each request, its service time includes a latency and a transfer time. The disk latency is calculated using the average seek and rotation time, and an average network latency is used for the network card. The transfer time can be derived from the request size and device bandwidth. If a power mode transition is involved in servicing a request, the associated energy and time overhead is counted in the estimation. Notice that the mode-switch threshold as well as its time and energy cost for the network card are smaller than those for the disk, which makes wireless network a desired alternative data source in some I/O access patterns. In this way, we can estimate the execution time T_{disk} and the energy consumption E_{disk} for accessing data on the disk, and the execution time $T_{network}$ and energy consumption $E_{network}$ for servicing requests from a remote storage server via network.

FlexFetch optimizes energy usage for I/O operations by treating equally the I/O energy cost and performance and obeying a user-specified maximum tolerable I/O performance loss rate. A threshold loss rate of $m\%$ means that FlexFetch will not switch to the other I/O data source by saving $x\%$ energy consumption but extending I/O execution time by $n\%$ if $x < n$ or $n > m$. Asking for user to provide this kind of preference, such as loss rate, is common in today's battery-powered system, such as the "Power manager" in the IBM ThinkPad laptop. Using the estimated energy cost, execution time, and the user-specified performance loss rate, we summarize the rules determining I/O data source for an evaluation stage as follows:

1. If $T_{disk} < T_{network}$ and $E_{disk} < E_{network}$, choose the local disk as data source;
2. If $T_{network} < T_{disk}$ and $E_{network} < E_{disk}$, choose the wireless network as data source;
3. If $E_{network} < E_{disk}$ and $(E_{disk} - E_{network})/E_{disk} >= (T_{network} - T_{disk})/T_{disk}$ and $(T_{network} - T_{disk})/T_{disk} < loss\_rate$, choose the network as data source; otherwise, choose the disk.

2.3 Adapting to System Dynamics

When we have a profile of a program's prior execution and use the profile to determine the I/O data source for the program's current execution, there could be some dynamics that are involved in the on-going execution and make the initial decision about data source sub-optimal. These dynamics could occur in several scenarios, such as when the recorded profile does not reflect current execution behaviors, requested data hit in the system buffer cache and are not accessed from storage devices, wireless network bandwidth changes due to factors such as change of device location, and disk is spun up by other programs when network is selected. FlexFetch has been designed to adaptively accommodate these dynamics.

2.3.1 Re-evaluating I/O Data Source Decision

While an I/O data source is used in an evaluation stage according to the decision made based on the old profile, a new profile is being generated for the current execution. To determine whether the initial decision is still valid, we need to compare the (partial) profile that is being generated with the old profile. An intuitive way of the comparison is to try to match, between these two profiles, the files requested in each I/O burst and the length of think time between two adjacent I/O bursts, and then use the result to decide if the current data source should be kept. However, an exact match of the two profiles is not directly relevant to the validity of the decision. From one run to another run of a program, the data it processes might be different, and the distribution of data requests and think times might be changing, more or less. However, this may not necessarily invalidate the decision made by using the rule presented in Section 2.2. Our method is to replace the corresponding portion of the old profile with the current partial profile and use the assembled profile to run the rule. Specifically, we monitor the amount of data that have been requested in the current run. Whenever the amount just exceeds the amount of data requested in the first N I/O bursts, we use the new profile for this run to replace the N I/O bursts in the old profile and re-evaluate the rule using the assembled profile to decide if the data source should be changed. In this way, in the current run of a program, its initial behaviors are examined in a bigger history picture, and as time goes on, the current behaviors will have more weight in the re-evaluation. Finally, the new profile will be recorded to replace the old profile for future use at the end of this run.

To further reduce the interference from invalid profile, we conduct a periodical evaluation of the effectiveness of a profile-based decision at the end of each evaluation stage. If the energy cost incurred by choosing an I/O device based on the profile is larger than a scheme that chooses the alternative data source, then disk or network, whichever was more energy efficient, will be used in the next stage, disregarding the profile of the next stage. Only when the profile for the previous stage is proven more effective is the profile used for the next stage.

2.3.2 Reflecting Effect of System Buffer Cache

We know that I/O requests from applications are first attempted in the system buffer cache and only misses in the buffer incur accesses to storage devices. Because we record I/O system calls in the profile, the profile might overestimate the I/O requests if many requested data can be found in the buffer. To mitigate the cache effect, at the beginning of each evaluation stage we check the file data recorded in the history profile for the stage with the current cache content in memory, and remove the requests on data that are resident in the cache. The remaining requests are then used to evaluate the I/O data source.

2.3.3 Taking Advantage of a Spun-up Disk

Disk is a shared device and can be spun up by other programs requesting local data that are stored only on the disk or (periodic) system write-back. While disk is spun up and being kept in the active/idle states by other programs or system's activities, it is almost free to access data from the disk for a program that has selected the network. For this reason, we monitor the requests not from the profiled programs and see if their intervals are less than the disk spin-down threshold time. If true, which means that the disk will not be spun down anyhow, requests from the program can then use the disk as free riders.

When multiple programs concurrently issue I/O requests, FlexFetch merges these programs' profiles and forms evaluation stage on the aggregate profile to estimate the execution time and energy cost for the programs.

2.3.4 A Summary of the FlexFetch Scheme

In summary, for saving energy in mobile computing, FlexFetch is designed to proactively select the least costly data source to service applications' requests. To achieve this goal, FlexFetch profiles the device-independent behaviors of applications by tracking I/O-related system calls. The profile is then used to compare energy costs with different I/O devices in each evaluation stage. In addition, being adaptive to run-time dynamics, FlexFetch adopts a mechanism to re-evaluate the effectiveness of its decision and progressively update the profile on the fly.

3 Performance Evaluation

3.1 Experimental Settings

We built a simulator that is driven by real-life applications' execution traces to evaluate the performance of FlexFetch. It simulates the management of two storage devices (hard disk and wireless interface card) and the buffer cache in the memory. The simulator emulates the policies used for Linux buffer cache management, including the 2Q-like page replacement algorithm, the two-window read-ahead policy that prefetches up to 32 pages, the C-SCAN I/O request scheduling mechanism, and the asynchronous write-back scheme. We also simulate the policies adopted in the Linux laptop mode [1], such as eager writing back dirty blocks to active disks and delaying write-back to disks in the standby mode.

P_{active}	Active Power	2.0W
P_{idle}	Idle Power	1.6W
$P_{standby}$	Standby Power	0.15W
E_{spinup}	Spin up Energy	5.0J
$E_{spindown}$	Spin down Energy	2.94J
T_{spinup}	Spin up Time	1.6sec
$T_{spindown}$	Spin down Time	2.3sec

Table 1. The energy consumption parameters for the Hitachi-DK23DA hard disk drive

The disk simulated in our experiment is the Hitachi-DK23DA hard disk [8]. It has a 30GB capacity, 4200 RPM, and 35MB/second peak bandwidth. Its average seek time is 13ms, and its average rotation time is 7ms. Its energy-related parameters are listed in Table 1. The timeout threshold for disk spin-down is set as 20 seconds, the default value used in the Linux laptop mode.

The simulated wireless network interface card (WNIC) is the Cisco Aironet350 with a bandwidth of 11Mbps [4]. The card adopts an adaptive dynamic power management mechanism. It switches to the PSM mode from the CAM mode when WNIC has been idle for more than 800msec, and it switches back to the CAM mode if more than one packet is ready on the access point. Its power consumption parameters are shown in Table 2.

PSM(idle/recv/send)	0.39W /1.42W /2.48W
CAM(idle/recv/send)	1.41W /2.61W /3.69W
CAM to PSM(Delay/Energy)	0.41sec/0.53J
PSM to CAM(Delay/Energy)	0.40sec/0.51J

Table 2. The energy consumption parameters of the Cisco Aironet 350 wireless card

In addition to FlexFetch, we also simulated the policy adopted in BlueFS as it shares the same goal with FlexFetch. Moreover, we compare FlexFetch with the policies where only disk or WNIC is used, respectively. In the experiments about FlexFetch and BlueFS, we set maximum tolerable performance loss rate as 25%. The minimal size to form an evaluation stage is set as 40 seconds. We also assume that data sets of workloads are available on both local hard disk and remote server and synced, except as stated otherwise.

3.2 Traces Collection

Name	Description	# File	Size(MB)
Thunderbird	an email client	283	188.1
make	building Linux kernel	2579	72.5
grep	a text search tool	1332	50.4
xmms	a mp3 player	116	47.9
mplayer	a movie player	121	136.3
Acroread	a PDF file reader	10	200.0

Table 3. Trace description

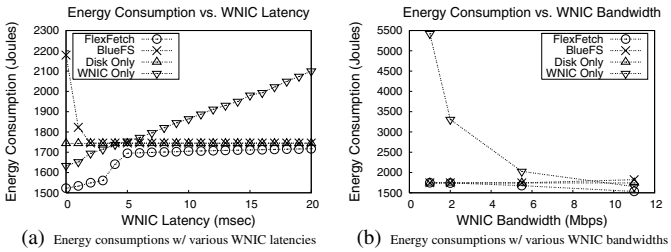
We modified the *strace* utility in Linux to collect traces to drive our simulator. The modified *strace* tool can intercept system calls related to file operations, such as open(), close(), read(), write(), lseek(), etc. For each system call, we collect the following information: pid, file descriptor, inode number, offset, size, type, timestamp, and duration. The blocks of the traced files are sequentially mapped to the local hard disk with a small random distance between files to simulate a real layout of files on the disk. We collected traces of six representative applications in a mobile computing environment, as listed in Table 3.

3.3 Experiment Results

Compared with accessing local hard disk, the performance and energy cost of accessing a remote storage via wireless card are sensitive to the networking environment. For example, the bandwidth of WNIC highly relies on the radio quality, which could be affected by many factors, such as distance and physical geometry. The 802.11b standard supports four bandwidths: 1Mbps, 2Mbps, 5.5Mbps, and 11Mbps, depending on the quality of radio signals. Moreover, the latency for accessing remote storage via WNIC is not constant, as it may be affected by many factors, such as server load and network congestion. Thus, accessing data via WNIC may experience significant change of latency and bandwidth, while accessing local hard disk has a constant performance. We present experiment results with various WNIC bandwidths and latencies. We vary the WNIC latency with a fixed 11Mbps bandwidth and vary the WNIC bandwidth with a fixed 1msec latency. Four policies are simulated: FlexFetch, the policy adopted by BlueFS (denoted as *BlueFS*), using the hard disk only (denoted as *Disk-only*), and using remote storage via WNIC only (denoted as *WNIC-only*).

3.3.1 A Programming Scenario: *grep* and *make*

This workload simulates a typical programming scenario, where a kernel programmer first searches the Linux source code using *grep* and then builds a kernel binary using *make*.



(a) Energy consumptions w/ various WNIC latencies (b) Energy consumptions w/ various WNIC bandwidths

Figure 1. *grep*+*make*: Energy consumptions with various WNIC bandwidths and latencies

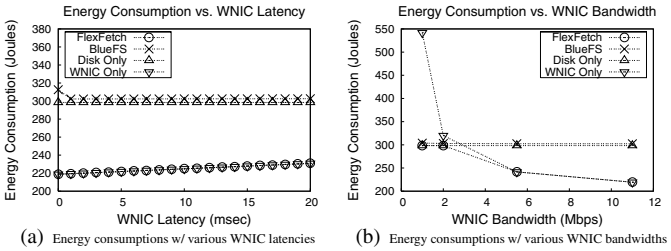
Figure 1(a) shows the energy consumption with various WNIC latencies. With a zero latency to access the remote storage via WNIC, *Disk-only* consumes 1743J energy, while *WNIC-only* requires 1634J. This is because of expensive head positioning operation in the hard disk. In this workload, *make* requests a large number of small files, where each request can incur a substantial latency and energy cost on the disk. So WNIC with a low latency is more cost effective than the hard disk. *BlueFS* consumes 2179J energy. In contrast *FlexFetch* consumes only 1522J, which is even less than that for *WNIC-only*. The difference of energy consumption between *BlueFS* and *FlexFetch* is due to the mixed access patterns exhibited in the workload. In this experiment, a large number of small files are first accessed in a very short period (*grep*), and then the Linux kernel is built (*make*), which takes several minutes. *FlexFetch* identifies different access patterns to select energy-efficient devices accordingly. Specifically, at the beginning *FlexFetch* spins up the hard disk to service the data set of *grep*, as the hard disk can complete the whole operation in a few seconds with small energy consumption. The data set of *make* is then mainly serviced by the WNIC, which is energy efficient for non-bursty workloads. *BlueFS*, however, has no knowledge of future accesses and solely relies on the recent history of data accesses and current storage device status to make a choice, which makes it lack a long-term view of energy consumption caused by different workload access patterns. As a result, it keeps switching between the hard disk and the remote storage, which incurs significant energy consumption for both devices.

With the increase of WNIC latency, the energy consumption of *WNIC-only* quickly increases and exceeds that of *Disk-only*. Meanwhile, *BlueFS* reduces its energy consumption by dispatching most of the requests to the hard disk, which now has not only comparable latency but also much higher bandwidth (35MB/s) than the WNIC (11Mb/s). Similarly, *FlexFetch* responds to the change of energy efficiency caused by the increased WNIC latency by issuing more requests to the hard disk, which makes its curve increasingly close to the curve of *Disk-only*.

Figure1(b) shows the energy consumptions with the variation of the WNIC bandwidth. *WNIC-only* is very sensitive to the bandwidth, as the I/O service time is highly dependent on the bandwidth. *FlexFetch* benefits from the increased WNIC bandwidth, which gives *FlexFetch* more flexibility to choose between the two devices and use WNIC to escape from the high cost involved in small file access on disk when it deems profitable. *BlueFS* is not sensitive to the change of bandwidth, as it selects device based on the current status of devices and accumulated opportunity costs.

3.3.2 A Media Streaming Scenario: *mplayer*

In this workload, a user uses *mplayer* to play a movie, which continuously accesses large movie files. As shown in Figure 2(a), the energy consumption for *FlexFetch* is almost the same as that for *WNIC-only*. *Mplayer* continuously accesses data, but only a small amount of data at a time, which makes it energy inefficient to use the hard disk. Using the profile of *mplayer*, *FlexFetch* can recognize the fact by estimating and comparing energy consumptions for the WNIC and the disk. *BlueFS*, however, can only see a recent history. *BlueFS* first issues requests to WNIC, as spinning up the hard disk incurs substantial energy overhead. However, when it sees more requests, it issues a 'ghost-hint' to the hard disk with a hope that servicing the following requests via an active disk could reduce energy consumption. Unfortunately, the following I/O requests are sparsely distributed — an access pattern that makes accessing the disk energy inefficient. Moreover, as *BlueFS* occasionally accesses data through WNIC, its energy consumption is even higher than *Disk-only*.



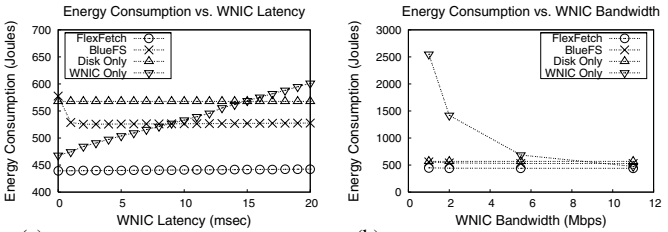
(a) Energy consumptions w/ various WNIC latencies (b) Energy consumptions w/ various WNIC bandwidths

Figure 2. *mplayer*: Energy consumptions with various WNIC bandwidths and latencies

Figure 2(b) demonstrates that *FlexFetch* can adaptively select a proper device. With a large WNIC bandwidth, say 5.5Mbps, *FlexFetch* accesses data via WNIC and achieves an energy consumption similar to that of *WNIC-only*. For WNIC with a bandwidth lower than 2Mbps, *FlexFetch* switches to the local hard disk, which is more efficient than a low-bandwidth WNIC, and consumes the amount of energy that is comparable to that for *Disk-only* and up to 45% less than that for *WNIC-only*.

3.3.3 An Email Search Scenario: *Thunderbird*

Thunderbird is a widely used email client application, which stores user's email in several large email files. It first reads several emails one after another with considerable think time in between, and then quickly searches the entire email files to locate user-specified emails.



(a) Energy consumptions w/ various WNIC latencies (b) Energy consumptions w/ various WNIC bandwidths

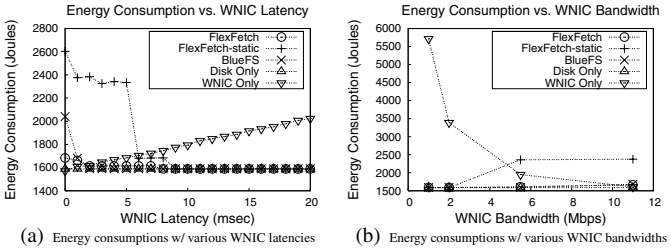
Figure 3. *Thunderbird*: Energy consumptions with various WNIC bandwidths and latencies

As shown in Figure 3(a), *Disk-only* consumes a considerable amount of energy, as servicing the small requests for initial several emails from the hard disk is energy inefficient. For WNIC with latency over 15msec, *WNIC-only* consumes even more energy than *Disk-only*, since the I/O latency becomes a dominant factor. *FlexFetch* consumes 17% less energy than *BlueFS* for most of WNIC latencies we examined. This is because *FlexFetch* dispatches the small requests at the beginning of the workload to WNIC, while the bursty search operations are then carried out at the high-bandwidth disk. Also, both *FlexFetch* and *BlueFS* are not sensitive to the change of WNIC bandwidth, since the amount of data serviced by the WNIC accounts for a small portion of the entire workload.

3.3.4 A Scenario with Forced Disk Spin-up: *grep, make, and xmms*

As we have discussed, a disk can be spun up for reasons beyond FlexFetch can predict in advance. One example is that some data may exist only on a particular storage device, which means the device has to be activated to service requests for the data. To test the effectiveness of FlexFetch in this situation, we adopt the workload that was used to represent programming scenario (*grep+make*). Concurrently, the user runs *xmms* to listen to MP3 music whose files are stored only on the local hard disk. The experiment results are shown in Figure 4. To emphasize the ability of FlexFetch to adapt to dynamics, we show the results of a FlexFetch alternative that does not have the capability to adapt to the run-time dynamics and denote it as *FlexFetch-static*.

Solely based on the profile of *make*, *FlexFetch* would behave similarly to what it did as described in Section 3.3.1, where *FlexFetch* serviced most requests from the remote



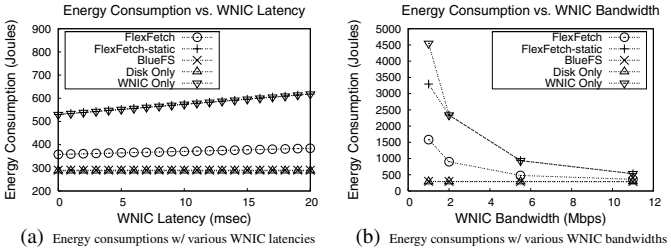
(a) Energy consumptions w/ various WNIC latencies (b) Energy consumptions w/ various WNIC bandwidths

Figure 4. *grep+make / xmms*: **Energy consumptions with various WNIC bandwidths and latencies**

storage via WNIC. However, in this experiment the concurrently running *xmms* keeps accessing the hard disk to make the disk stay in the active/idle states. *FlexFetch* observes the forced spin-up and responds by switching the data source to the local disk based on measured actual energy consumption. As shown in Figure 4(a), with a network latency less than 9msec, *FlexFetch* substantially avoids the high energy cost with *FlexFetch-static*, which lacks ability to recognize the run-time dynamics. Similar performance trend can also be observed in Figure 4(b) for different network bandwidths. With the increase of WNIC latency, both *FlexFetch-static* and *FlexFetch* choose the hard disk as the I/O device, so their curves merge eventually.

3.3.5 A Scenario about Invalid Profile: *Acroread*

A profile of one execution of a program can be substantially different from that of another execution, which is the case especially for interactive applications. To test how this would affect the effectiveness of FlexFetch for energy saving, we designed a workload in which a user searches multiple keywords in several 20MB PDF files continuously with a 10 seconds interval using Acroread. However, the profile previously collected is about an execution of Acroread where a set of 2MB PDF files are read with an interval of 25 seconds, which is longer than the disk time-out.



(a) Energy consumptions w/ various WNIC latencies (b) Energy consumptions w/ various WNIC bandwidths

Figure 5. *Acroread*: **Energy consumptions with various WNIC bandwidths and latencies**

At the beginning of this workload *FlexFetch* uses the out-of-date profile and chooses WNIC as the I/O device, while the hard disk is more energy efficient for servicing the bursty workload in the current execution. At the end of the first stage, *FlexFetch* evaluates how much energy could be consumed by using a different I/O device based on the I/O requests observed during the previous stage. By comparing the actual energy consumptions, *FlexFetch* realizes that the decision made based on the profile is incorrect. So it adopts the device that consumed less energy during the previous stage, the hard disk, as the data source for the next stage. As shown in Figure 5(a), for a network latency of 10msec *FlexFetch* consumes around 371J energy, which is 36% less than that for *FlexFetch-static* (574J). However, *FlexFetch* consumes around 15% more energy than BlueFS. This is because *FlexFetch* has to spend at least one stage to examine actual energy costs and correct its prior decision.

4 Additional Related Work

There have been extensive research work on reducing energy consumption for I/O data access on disk, wireless network, and both, in mobile computing.

Regarding disk energy saving, some work focuses on the selection of disk spin-down timeout threshold, which could be a fixed time period [6], or be adaptively adjusted at run-time [7]. These schemes passively monitor disk I/O operations without extending disk idle time. Other work customizes system or application software for disk energy-saving [16, 17]. The proposed scheme in [16] uses aggressive prefetching to *create* more busty disk accesses for long disk idle periods. This scheme needs to carefully decide the timing to carry out prefetching, how much and which data should be prefetched, and which data should be replaced. The scheme requires an intensive change of memory management in operating systems. There are schemes that use flash memory to further improve the effectiveness of caching and prefetching for disk energy saving [2, 13]. Our FlexFetch scheme is not involved in the system buffer cache management, so it can be implemented in the existing system in a non-intrusive manner. Moreover, FlexFetch treats the cached data and prefetch requests, either from applications or from systems, as part of the environment it accommodates, and is complementary to system-level energy-saving schemes.

Regarding energy-saving for wireless communication in mobile computers, Jung and Vaidya propose a power control MAC protocol for Ad Hoc networks [10] that allows nodes to vary transmit power level on a per-packet basis. In [12], Lufei and Shi propose to use energy as a QoS metric for application level protocol adaptation, which considers only the wireless network energy consumption and neglects the disk energy consumption.

Kuenning and Popek concern with the the mobile computing in the absence of a network, or disconnected operations [11]. They propose a scheme that observes user's behaviors and file accesses and predicts future needs, so that it can choose which files should be hoarded. Their experiments demonstrate that the scheme can hoard entire user working set with a high confidence using a semantic clustering technique, so that most of data that are needed in the mobile computing are resident in the local disk as well as on the server. The observation that user behavior and file access pattern are well predictable supports the use of profile-based approach in energy saving.

5 Conclusions and Future Work

Energy consumption remains a pressing issue in mobile computing, where on-demand data could be available in both local hard disk and remote storage via wireless network. In this paper, we propose an effective power-saving scheme, FlexFetch, to save energy by adaptively selecting the most cost-efficient data source for various workloads. By considering both I/O devices' characteristics and runtime dynamics, FlexFetch estimates and compares energy consumption of various data sources and redirects I/O requests to the least costly data source. Our experiments based on trace-driven simulations show that FlexFetch can achieve significant energy saving, compared to existing energy-saving schemes that use the local disk only or that select data source solely based on recently serviced requests.

There are several limitations in our work that are to be addressed in the future. First, in an environment where there are non-profiled programs running, the energy saving effort of FlexFetch for profiled programs could be foiled by unexpected I/O requests. In this work, we assume that usually a fixed set of programs are used in a personal mobile computer. Second, current evaluation is based on trace-driven simulation. An implementation of a FlexFetch prototype would reveal more insights on issues such as time, space, and energy overhead of applying the scheme. Third, in this work, we assume that the synchronization issue is taken care of by other system component, such as a hoarding system [11]. We leave it as a future work to study how synchronization would affect the performance of FlexFetch.

6 Acknowledgments

This research is sponsored in part by the Office of Advanced Scientific Computing Research; U.S. Department of Energy. Part of the work was performed at the Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC under Contract No. De-AC05-00OR22725.

References

[1] Linux laptop mode document. http://lxr.linux.no/source/Documentation/laptop-mode.txt.

[2] F. Chen, S. Jiang, and X. Zhang. SmartSaver: Turning flash memory into a disk energy saver for mobile computers. In *Proc. of 2006 International Symposium on Low Power Electronics and Design (ISLPED'06)*, 2006.

[3] X. Chen and X. Zhang. A popularity-based prediction model for web prefetching. In *IEEE Computer*, volume 36, March 2003.

[4] CISCO. http://www.cisco.com/application/pdf/en/us/guest/products/ps448/c1650/ccmigration_09186a0080088828.pdf.

[5] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang. DiskSeen: Exploiting disk layout and access history to enhance I/O prefetch. In *Proc. of USENIX'07*, 2007.

[6] F. Douglis, P. Krishnan, and B. Marsh. Thwarting the power-hungry disk. In *Proc. of 1994 Winter USENIX Conference*, January 1994.

[7] D. P. Helmbold, D. D. E. Long, and B. Sherrod. A dynamic disk spin-down technique for mobile computing. In *Proc. of the 2nd Annual International Conference on Mobile Computing and Networking*, 1996.

[8] HITACHI. http://www.hitachigst.com/tech/techlib.nsf/products/DK23DA_Series.

[9] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang. DULO: an effective buffer cache management scheme to exploit both temporal and spatial localities. In *Proc. of FAST'05*, 2005.

[10] E. Jung and N. Vaidya. A power control MAC protocol for Ad Hoc networks. In *Wireless Networks (ACM WINET)*, 2005.

[11] G. Kuenning and G. Popek. Automated hoarding for mobile computers. In *Proc. of SOSP'97*, 1997.

[12] H. Lufei and W. Shi. Energy-aware QoS for application sessions across multiple protocol domains in mobile computing. In *Computer Networks (COMNET)*, volume 51(11), 2007.

[13] B. Marsh, F. Douglis, and P. Krishnan. Flash memory file caching for mobile computers. In *Proc. of the 27th Hawaii Conference on Systems Science*, 1994.

[14] M. K. Mckusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. In *Transactions on Computer Systems*, volume 2(3), 1984.

[15] E. Nightingale and J. Flinn. Energy-efficiency and storage flexibility in the blue file system. In *Proc. of OSDI'04*, 2004.

[16] A. E. Papathanasiou and M. L. Scott. Energy efficient prefetching and caching. In *Proc. of USENIX'04*, June 2004.

[17] A. Weissel, B. Beutel, and F. Bellosa. Cooperative IO - a novel IO semantics for energy-aware applications. In *Proc. of OSDI'02*, December 2002.