A Report on Impact of Data Compression on Energy Consumption of Wireless-Networked Handheld Devices *

Rong Xu Zhiyuan Li Cheng Wang Peifeng Ni
Department of Computer Sciences, Purdue University
West Lafayette, IN 47907
{xur,li,wangc,npf}@cs.purdue.edu

Abstract

We investigate the use of data compression to reduce the battery consumed by handheld devices when downloading data from proxy servers over a wireless LAN. To make a careful trade-off between the communication energy and the overhead to perform decompression, we experiment with three universal lossless compression schemes, using a popular handheld device in a wireless LAN environment and we find interesting facts.

The results show that, from the battery-saving perspective, the `gzip` *compression software (based on LZ77) to be far superior to* `bzip2` *(based on BWT) and* `compress` *(based on LZW). We then present an energy model to estimate the energy consumption for the compressed downloading. With this model, we further reduce the energy cost of* `gzip` *by interleaving communication with computation and by using a block-by-block selective scheme based on the compression factor of each block. We also use a threshold file size below which the file is not to be compressed before transferring.*

1 Introduction

Wireless-networked handheld devices have the potential to become powerful mobile tools to access information and applications from anywhere at any time. However, with the current state of the art, effective use of wireless-networked handheld devices needs strong support from *proxy* servers, or *proxies*. Such servers are stationary networked computers which provide the handheld devices a range of backup resources and supporting services, including secondary storage, computing power, Internet data caching and information filtering [1, 2, 7, 6, 13, 8].

When proxies are employed in a wireless LAN environment, battery consumption by communication vs. computation becomes interesting on the handheld devices. Today's handheld devices are increasingly used for accessing information over the network, information in the form of text, graphics, photographs, data images, speech, sound, and so on. Compressing such information on the proxies, in advance or on demand, has the obvious potential advantage of reducing the battery consumed by the wireless network interface on the handheld device. On the other hand, considerable battery energy may be consumed by the processor for decompression. A similar tradeoff issue exists when the handheld device uploads information, e.g. lively captured voice and pictures. It is in this context that we investigate, in this paper, the impact of data compression on battery consumption of wireless-networked handheld devices. We focus, however, on data downloading only, as this constitutes the predominant uses of the handheld devices in current practice.

We aim to study a variety of aspects of the data compression issue. We pay most of our present attention on universal lossless compression schemes which can be applied to arbitrary data types with the ability to fully recover the contents by decompression.

In the rest of the paper, we first describe our experimentation set-up (Section 2) and compare the timing and energy data from using three different compression schemes (Section 3). For the winning scheme, namely LZ77, we present an energy consumption analysis and explain how the energy efficiency can be further improved by interleaving the communication and decompression (Section 4). Based on the energy model, we identify the threshold data size and the threshold compression

* Technique Report CSD-TR-03-003, Department of Computer Sciences, Purdue University, West Lafayette, IN 47907, February, 2003

Table 1. Power parameters

iPAQ	WaveLAN	Power Saving	Current(mA)
idle	sleep	-	90
busy	sleep	-	300 – 440 (310)
idle	idle	off	310
idle	idle	on	110
busy	idle	off	530 – 670 (570)
busy	idle	on	330 – 470 (340)
—	recv	off	430
—	recv	on	400
busy	recv	off	550 – 690
busy	recv	on	470 – 690

ratio to be energy worthy for the LZ77 scheme. With such thresholds in mind, we develop a LZ77-based algorithm that makes the compression decision dynamically, on a block-by-block base. This set of experiments consider both precompression and compression on demand. These are followed by a discussion of related work (Section 6) and a conclusion (Section 7).

2 Experimentation Setup

The handheld device used in our experiment is a Compaq iPAQ 3650 which has a 206MHz Intel StrongArm SA1110 processor and 32MB RAM. The proxy server is a Dell Dimension 4100 which has a 1GHz P-III processor. Both machines run Linux operating systems and communicate through TCP socket calls. The wireless connection is through a Lucent Orinoco (WaveLAN) Golden PCMCIA card which follows the IEEE 802.11b standard. Under the 11Mb/s nominal peak rate, the effective data rate of the WaveLAN card is measured as about 5Mb/s. The bit rate (for both send and receive) can be adjusted downward in a few different ways, by changing the settings of the access point, by increasing the communication distance, or by increasing structure obstacles between the two antennas. In Section 4.2, to validate our energy model, we manually change the nominal bit rate from 11Mb/s to 2Mb/s. However, for the main part the paper, all the data are collected under the nominal bit rate of 11Mb/s unless otherwise specified.

To measure the electrical current drawn by the handheld device, we connect it to an HP 3458a low-impedance (0.1Ω) digital multi-meter which takes several hundred samples per second and automatically records maximum, minimum and average electrical current. In order to get a reliable and accurate reading, we disconnect the batteries from both the iPAQ and the extension pack, using an external 5V DC power supply instead. The start and finish of the meter reading is controlled by software, using the built-in trigger mechanism of the multi-meter. According to our measurement, the overhead associated with the triggering interrupts is less than 0.5% and the readings are consistent over repeated runs.

We measure the electrical current drawn in different running modes. The reading is shown in Table 1. (All the numbers are measured with the screen turned off.) The first column shows two functioning modes of the iPAQ: *idle* when it does nothing and *busy* when it performs computation. The *WaveLAN* column indicates the status of the WaveLAN card, *sending*, *receiving*, *idle*, or *sleep*. In the sleep mode, the WaveLAN card is not receptive to the incoming signals and it draws significantly lower current than in the idle mode. Certain instructions, e.g. memory loads and stores, are more expensive than others. The current reading may fluctuate as a result, and the table shows the range of the current measured for iPAQ $busy$ modes. And in these modes, the number in the parentheses is the average reading for `gzip` decompression.

The *power saving mode* utilizes the hardware mechanism that periodically switches the WaveLAN card between the sleep mode and the idle mode. It reduces the electrical current drawn significantly when there are no network requests. When there exist send or receive requests, the effective data rate decreases by about 25% in the power-saving mode, due to the overhead to switch between the states. Heuristics have been proposed in literature to predict the optimal timing to wake-up from the sleep mode [11]. However the success rate of such methods highly depends on event predictability. Also, leaving the WaveLAN card in the sleep mode for an extended period of time may prevent timely reception of instant messages. In this paper, therefore, we simply use the hardware power-saving mechanism to take advantage of the sleep mode. The *Power Saving* column indicates whether the power saving mode is enabled for the WaveLAN card.

When the WaveLAN card is sending or receiving, the CPU spends time on the network interface. Hence the CPU is not idle even if it is not performing any computational tasks. In these cases, the *iPAQ* column is marked as '—'. The electrical current is the highest when computation is interleaved with communication.

3 Comparing Universal Compression Schemes

We consider three widely used universal compression methods [1], namely the Burrows-Wheeler Transform (BWT) [3], Lempel-Zib coding algorithms (LZ77 [14], and LZW [12]). Three widely circulated UNIX tools, based on the respective compression methods, are used in our experiments. Throughout the paper, the term *compression factor* refers to the ratio of the input size over the output size, and the term *compression ratio* refers to the reciprocal of the compression factor.

The `gzip` software uses LZ77 algorithm. The algorithm replaces the second occurrence of a string with a pointer to the previous string, in the form of a (*distance, length*) pair. The distances are limited to the size-sliding window (of 32K bytes), and the lengths are limited to the size of the look-ahead buffer. If a string does not occur anywhere in the slide window, it is emitted as a sequence of literal bytes. The input stream is divided into blocks. A block is terminated when the compression algorithm determines that it is better to start a new block to achieve a better compression. The version of `gzip` we use in this paper is `gzip 1.2.4`. The core algorithm of `gzip` is also used in `zlib`, a general purpose data compression library. The version of `zlib` used in this paper is `zlib 1.1.3`.

The `compress` software uses the LZW algorithm with a growing dictionary. Unlike LZ77, it does not use a search buffer, a look-ahead buffer, or a sliding window. Instead, it maintains a dictionary of previously encountered string. It starts with a small dictionary of $2^9 = 512$ entries (with the first 256 of them filled up). While this dictionary is being used, 9-bit pointers are written into the output. When the dictionary fills up, its size is doubled to 1024 entries, and 10-bit pointers are used. This process continues until the pointer size reaches 16 bits. The compression continues without making changes to the dictionary until the compression factor falls below a predefined threshold. At that time, the dictionary is deleted and a new 512-entry dictionary is started. The version of `compress` we use in this paper is `ncompress_4.2.4`.

The `bzip2` software compresses files using the Burrows-Wheeler block sorting compression algorithm. It first applies a reversible transformation to a block of data. The transformation groups characters together so that the probability of finding a character close to another instance of the same character is increased. The compression rate is generally considerably better than that achieved by more conventional Lempel-Ziv-based compressors [5]. However, it is also more computation intensive, by some constant factors, than Lempel-Ziv algorithms. The version of `bzip2` we use in this paper is `bzip2 1.0.1`.

3.1 Workload

We have experimented with a large number of files of different types, focusing on types typical for use on handheld devices, such as web pages, documents, binary machine codes (recall the limited storage and the need to download applications), and media files. For this paper, in order to have space to show results for individual files (with different compression factors), we select a representative subset of files, which are listed in Table 2. Column 1 lists the file names. Column 2 shows the file sizes. The rest of the columns show the compression factors by the individual compression schemes. We divide the files into two categories, the small sized (under 80K bytes) and the relatively large sized. Table 3 gives a brief description of the individual files.

All of the three compression tools provide options to choose the desired compression level. A higher level tends to result in a higher compression factor and a slower compression speed. On the other hand, for the same compression scheme, our experiments show that a high compression factor does not increase the *decompression* speed and energy much. Hence, we use the highest compression level for each tool throughout the experiments, i.e. level 9 for both `gzip` and `bzip2` and "-b 16" for `compress` (meaning a maximum of 16 bits for common substring codes).

From Table 2, we see that all of the three schemes can compress the program source or text files well (the compression factors are from 2 to 25). For binary machine codes, the compression factors range from 1.6 to 3.5. Certainly, no schemes can compress the already encoded data or random data well. Consistent to previous reports, `bzip2` usually achieves the highest compression factor, while `compress` gets the lowest in most cases.

3.2 Comparing Results

In this section, we examine experimental results to see when data compression can reduce total energy consumption on the handheld device and what compression tool reduces the most. We first assume that all downloaded files are compressed *a priori* and stored on the proxy server.

Assuming fix-sized packets, given the fixed data rate, the energy cost to transfer a data file over the wireless network is linearly proportional to the data size. For compression to benefit, the saving in the communication energy (due to reduced

[1] A data compression method is called universal if it has no prior assumption on the statistics of the input.

Table 2. Test files and compression factors

Name	Size	Gzip	Compress	Bzip2	Name	Size	Gzip	Compress	Bzip2
nes96.xml	2961063	18.23	6.51	24.59	pegwit	360788	2.57	1.73	2.48
M31C.xml	8391571	14.64	9.91	18.58	NTBACKUP.EXE	1162512	2.46	1.79	2.50
M31Csmall.xml	340668	12.90	6.63	17.52	input.program	3450558	2.30	1.66	2.41
input.log	4409036	11.24	5.92	18.37	startup.wav	1158380	2.24	2.26	3.25
langspec-2.0.html.tar	1762816	4.68	3.08	6.13	pp.au	920316	1.11	0.99	1.23
input.source	9553920	3.90	2.54	4.88	input.graphic	6656364	1.09	0.97	1.38
proxy.ps	2175331	3.80	3.06	6.87	image01.jpg	1833027	1.04	0.84	1.36
j2d-book.ps	5234774	3.66	2.75	4.70	loveoflife.mp3	4328513	1.02	0.83	1.02
java.ps	1698978	3.55	2.61	4.46	tens_015.m2v	2816594	1.01	0.85	1.02
localedef	330072	3.50	2.18	3.72	image01.gif	5075287	1.00	0.82	1.00
JavaCCParser.class	126247	3.06	2.00	3.17	input.random	4194309	1.00	0.81	1.00
langspec-2.0.pdf	4419906	2.79	1.98	3.00					
mail3	1438	1.82	1.47	1.67	tail	26240	2.07	1.59	2.11
mail1	1611	1.91	1.48	1.75	time_fig.eps	31290	3.22	1.95	3.17
PolyhedronElement.class	2271	1.79	1.42	1.64	intro.pdf	56840	1.77	1.23	1.80
nohup	2500	1.97	1.47	1.81	fscrib	57372	2.09	1.55	2.14
mail2	4285	2.16	1.66	2.06	intro.ps	64477	2.37	1.87	2.54
yahooindex.html	16709	3.64	2.22	3.66	JavaFiles.class	72004	2.93	1.82	2.97
Slate.class	22890	2.23	1.54	2.15	pa1.ps	79072	2.58	1.99	2.83

Table 3. Test files type information

Name	Description	Name	Description
nes96.xml	a xml webpage	pegwit	a program binary
M31C.xml	a xml webpage	NTBACKUP.EXE	a program binary
M31Csmall.xml	a xml webpage	input.program	a program binary (from `SPEC 2000`)
input.log	a webpage log (from `SPEC 2000`)	startup.wav	a data file in .wav format
langspec-2.0.html.tar	a tar file of Java language specification in html format	pp.au	a data file in .au format
input.source	a program source (from `SPEC 2000`)	input.graphic	a TIFF image(from `SPEC 2000`)
proxy.ps	a postscript document	image01.jpg	a jpeg image
j2d-book.ps	a postscript document	loveoflife.mp3	a mp3 music
java.ps	a postscript document	tens_015.m2v	a mpeg-2 movie
localedef	a program binary	image01.gif	a GIF file
JavaCCParser.class	a Java class file	input.random	random data (from `SPEC 2000`)
langspec-2.0.pdf	Java specification in pdf format		
mail3	a text mail	tail	a program binary
mail1	a text mail	time_fig.eps	an encapsulated postscript file
PolyhedronElement.class	a Java class file	intro.pdf	a pdf file
nohup	a shell script	fscrib	a program binary
mail2	a text mail	intro.ps	a postscript document
yahooindex.html	a html webpage	JavaFiles.class	a Java class file
Slate.class	a Java class file	pa1.ps	a postscript file

data size) needs to be greater than the energy cost spent on decompression on the iPAQ. Thus, the scheme which compresses the deepest does not necessarily saves the most energy.

The three schemes which we used are known to decompress much faster than to compress. However, `bzip2` performs more computation than the other two schemes, since it requires a reverse transformation. As we shall see, the difference is significant enough to put `bzip2` in energy disadvantage. Figure 1 compares the time spent to download and decompress data files using the three different compression schemes, The top two bar graphs are for relatively large-sized files, which are sorted in the decreasing order of the compression factor. The third bar graph is for the small sized data files, which are sorted in the increasing order of the file size.

The height of each vertical bar is relative to the time spent when downloading without compression. Each bar here is divided into two parts, the lower one is the time spent on downloading and the upper one spent on decompression. One observes that, for each particular scheme, the trend is a reduction in the overall time with the increased compression factor.

When the network card is idle, it can potentially save energy by entering the power-saving mode. The electrical current decreases from 570mA to 340mA if we switch WaveLAN card from idle to sleep mode when iPAQ is busy (refer to Table 1). We will further discuss this issue in Section 4.2. Our experiments show that such saving materializes for `bzip2` (as it takes long time to decompress) but not much for the other two schemes. Thus, we show the energy results with power-saving enabled for `bzip2` but not for the other two schemes.

Figure 2 compares the energy consumed. The bars are organized in the same way as in Figure 1. Figures 2 clearly shows

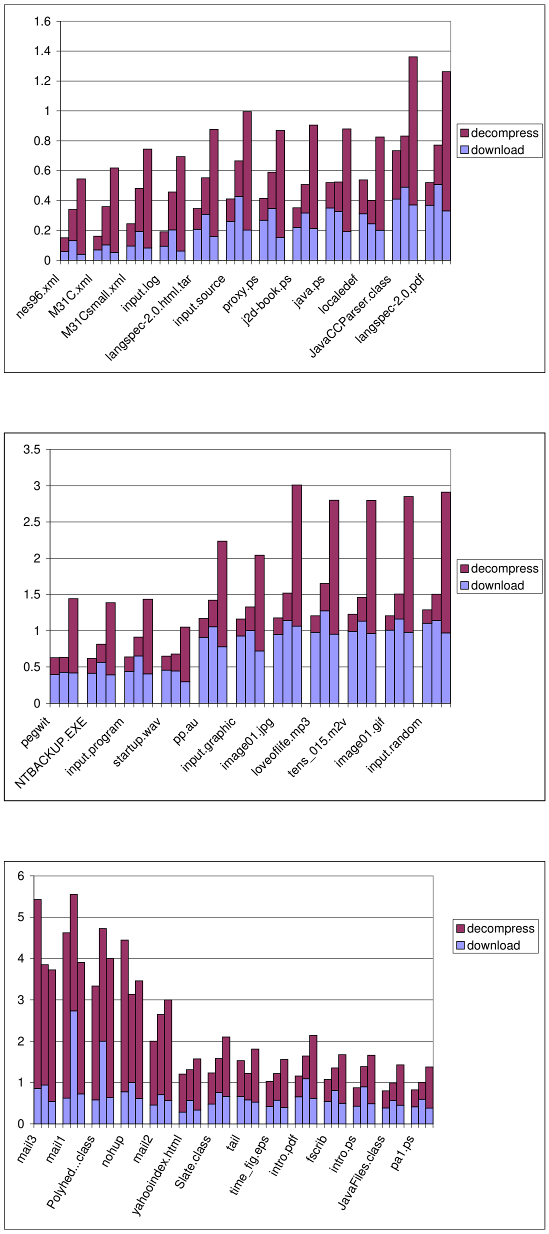


Figure 1. Time comparison for different compression schemes. (left bar: `gzip`; middle bar: `compress`; right bar: `bzip2`)

that if the raw file is large and compression factor is high, all compression schemes can save energy. However, if the input file is small, compression fares worse due to the start-up cost. If the compression factor is low, it is not beneficial either, because the communication cost is not reduced enough while the decompression cost remains.

The results also show that, for large data files, neither the scheme with the highest compression factor nor the one with the lowest factor gets the best energy result on the iPAQ. It is the decompression efficiency that matters the most. In this paper, we do not give a detailed analysis of the three different algorithms. Nevertheless, the experiments in our IEEE 802.11b Wireless LAN environment show that, except for a few very small data files, `gzip` balances the communication cost vs. the decompression cost the best. It is also clear, by comparing the communication time with the decompress time, that `gzip`

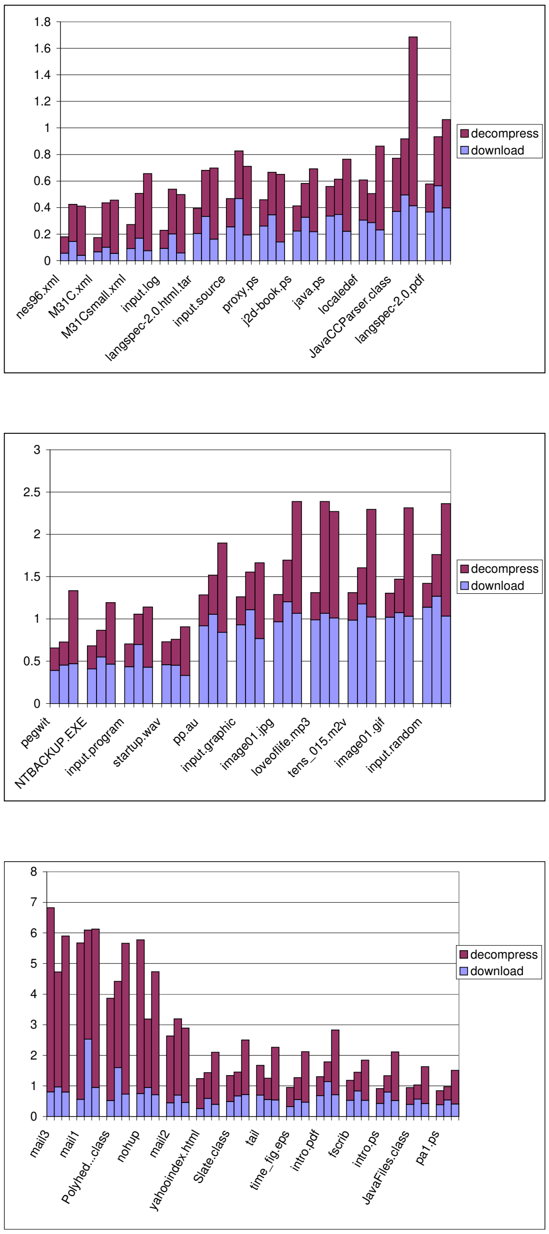


Figure 2. Energy comparison for different compression schemes. (left bar: `gzip`**; middle bar:** `compress`**; right bar:** `bzip2`**)**

will remain the strongest algorithm even with interleaving. In the next section, we will show the impact of interleaving for `gzip` only and leave out the similar data for the other two schemes.

4 Enhancements Based on Energy Modeling

From the previous section, `gzip` stands as the clearly superior choice from both the perspectives of time and energy on the handheld device. Nonetheless, we see that when the compression factor is not high enough, even `gzip` consumes more energy than when there is no compression. Furthermore, compression gets far worse results for small data due to the start-up cost of decompression. In this section, we analyze the energy consumption and propose methods to improve the energy performance for compressed data communication.

4.1 Interleaving Decompression and Receiving

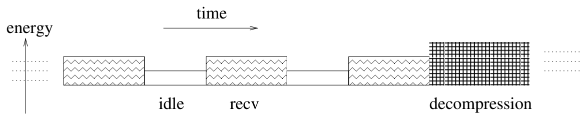
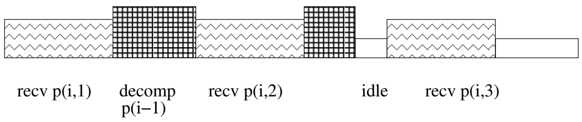


Figure 3. Energy consumption when download and decompress

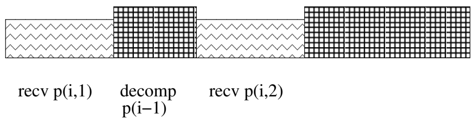
Handheld devices usually have simple hardware architectures which have a single processor. The processor is needed both for computation and for network communication (e.g writing to and reading from the network interface, defragmentation and assembly of packets). Figure 3 illustrates the energy consumption breakdown when we download before decompressing the data. The energy consumed by downloading consists of two parts, the energy to receive and copy data, and the energy consumed in the CPU idle intervals between packages arriving. The idle-time energy consumption can be substantial. In our experimental environment, even when we receive the packets at the full speed (602 KB/s), the idle time is about 40% of the total receiving time. Thus, about 30% of the total downloading energy is consumed when idling.

We can improve the performance and energy efficiency by filling these idle periods with decompression work. The interleaving takes place as follows. We decompress the i^{th} block of the data when downloading the $(i+1)^{th}$ block. The decompression starts as soon as the first block is received. The overall effect is decompressing and downloading at the same time.

Figure 4 shows two scenarios of interleaving. In (a), the decompression of i^{th} block is faster than downloading the $(i+1)^{th}$ block, which creates a CPU-idle period. In (b), the decompression is slower than downloading, and the system is processing at the full speed and there is no idle time.



(a) idle time greater than decompression time



(b) idle time less than decompression time

Figure 4. Energy consumption when interleaving downloading and decompressing; `p(i)` **is the i^{th} block and** `p(i,j)` **is the j^{th} packet of i^{th} block.**

Let s be the size of the data, the energy for receiving the data is

$$E = m * s + c_s + t_i * p_i \quad (1)$$

where m is the energy to receive a unit of data, c_s is the network communication start-up cost, p_i is the power during system idle time, while t_i is the total idle time between packets arriving.

If we perform compression when sending and decompression when receiving, the energy consumed is

$$E = m * s_c + c_s + (t_{i'} + t_{i1}) * p_i + t_d * p_d \quad (2)$$

where s_c is the compressed file size; t_d is the total time of decompression the whole data; p_d is the average power when decompressing. Further, t_{i1} is the idle time when receiving the packet for the first compressed block, while $t_{i'}$ is the idle time when receiving the remaining packets. We separate these two because we can not make use of the t_{i1} to perform decompression.

With interleaving, the energy equation becomes

$$
\begin{array}{l}
if\ t_{i'} > t_d \\
\qquad m * s_c + c_s + t_d * p_d + (t_{i'} - t_d + t_{i1}) * p_i \\
if\ t_{i'} \leq t_d \\
\qquad m * s_c + c_s + t_d * p_d + t_{i1} * p_i
\end{array}
\tag{3}
$$

Interleaving communication and decompression can be achieved in various ways. For example, we can implement the decompression in the TCP stack, so the interleaving can be managed explicitly by the kernel. In this work, we adopt a user-level method. We implement the decompression as a user process which decompresses the downloaded data block by block. Since the receiving process is in the kernel interrupt handler, the receiving of i^{th} block will interrupt the decompression of previous blocks.

During the whole compressed downloading, the CPU idle time $t_i = t_{i'} + t_{i1}$ depends on the system load and the receiving rate. If there exist other active processes, the CPU's idle time decreases. So does the benefit of interleaving. In our experiment, there exist no other active processes. In this case, we found that t_i usually is a fixed fraction of the whole downloading time. We measured t_i to be about 40% of data-receiving time at 0.6MB/s download rate. So, $t_i = 0.4 * s/0.6$. The following equations determine $t_{i'}$ and t_{i1}.

$$
\begin{array}{ll}
if\ s \geq 0.128 & t_{i'} = 0.4 * (s_c - 0.128 * s_c/s)/0.6 \\
 & t_{i1} = 0.4 * (0.128 * s_c/s)/0.6 \\
if\ s < 0.128 & t_{i'} = 0 \\
 & t_{i1} = 0.4 * s_c/0.6
\end{array}
\tag{4}
$$

where s and s_c are in Megabytes. Here we assume that the size of the compression buffer is 0.128MB.

To implement interleaving, we use the compression/decompression library `zlib`. There exist subtle differences between `gzip` and `zlib` even without interleaving. To give a fair assessment of the effect of interleaving, Figure 5 compares the time for `gzip` (the left bar), `zlib` without interleaving (the middle bar), and `zlib` with interleaving (the right bar). Figure 6 compares the energy, correspondingly. From these figures, we see that interleaving brings down the decompression overhead (both time-wise and energy-wise) rather substantially.

To validate the energy model for interleaving, we calculate the energy consumption based on Equation 3. Parameters such as p_i, p_d and t_d are obtained through direct measurement, while $t_{i'}$, t_{i1} are computed (in Equation 4). $m * s_c + c_s$ is calculated by subtracting the idle energy $t_i * p_i$ from the value of the measured total downloading energy (refer to Equation 1).

Figure 7 shows the error rate defined as $(calculated\ value - measured\ value)/measured\ value$. For large files, the error rates are less than 6.5% and the average error rate defined as $\frac{\sum |error\ rate|}{the\ number\ of\ data\ points}$ is 2.5%. For small files, the average error rate is 9.1%. This is mainly attributed to the inaccuracy for the five smallest files. Due to the extremely small energy value, even a small interference on the handheld device can affect the result substantially. If these five files are excluded, the average error rate for small files is 4.5%.

4.2 Estimating Energy Consumption

In last section, we have shown that interleaving decompression with receiving can reduce time and energy for most of the data files. Still, when the compression factor is low, even with interleaving, the energy saved in data communication does not compensate the energy used in decompression. The net energy loss ranges between 2%-14%, compared to no compression. Hence, we seek to derive a formula to estimate the energy consumption for a given file size and a given compression factor, such that we can make the compression decision adaptively.

To derive such a formula, We first need to know the coefficients m and c_s. As mentioned in last subsection, t_i is linearly proportional to s . Hence, the downloading energy E can be expressed as a linear function of s. Based on linear fitting of a large number of data points, we get the following equation to estimate the energy for downloading:

$$E = 3.519 * s + 0.012$$

with an average error of 7.2%. The result of the fitting is plotted in Figure 8 (b). Based on this fitting, we can derive m and c_s from Equations 1 and 4. In our experiments, we get $m = 2.486$ and $c_s = 0.012$.

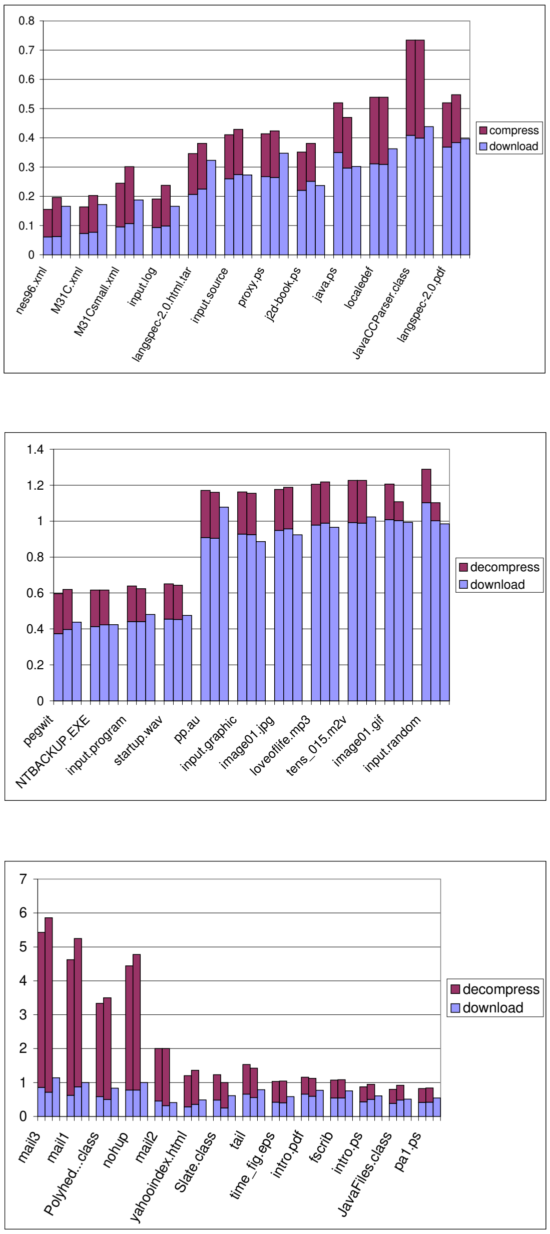


Figure 5. Effect of interleaving on time comparison. (left bar: `gzip`**; middle bar:** `zlib w/o interleaving`**; right bar:** `zlib w/ interleaving`**)**

We also use linear fitting to estimate the decompression time based on the file size and the compression factor,

$$t_d = 0.161 * s + 0.161 * s_c + 0.004$$

where s and s_c are in Megabytes. The result of fitting is plotted in Figure 8 (a). The average error rate is 3%, and the maximum error rate is 13%. The coefficient of determination of the estimation is 96.7%.

After plugging all the parameters obtained above into Equation 3, we have the following formulas to estimate energy

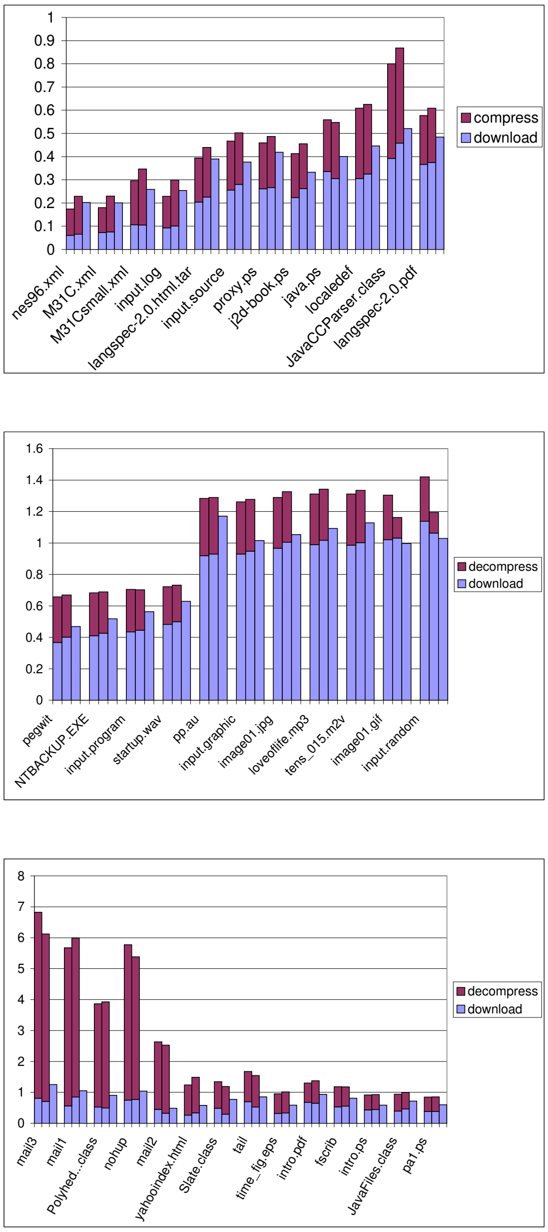
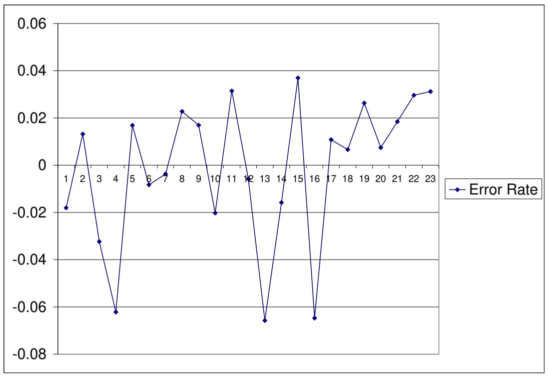


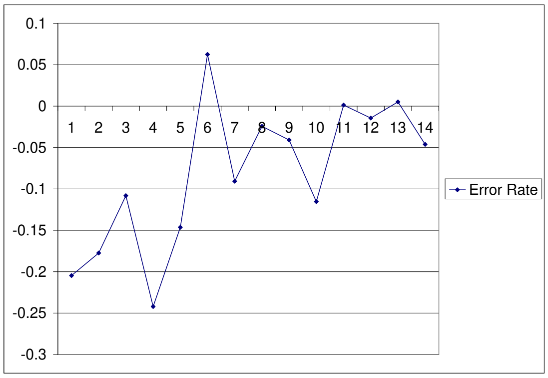
Figure 6. Effect of interleaving on energy comparison. (left bar: `gzip`; middle bar: `zlib w/o interleaving`; right bar: `zlib w/ interleaving`)

consumption in the case of interleaving,

$$\begin{aligned}
&if \;\; F > 3.14 - 0.265/s \;\&\; s > 0.128\,,\;\; E = \\
&\qquad 0.4589 * s + 2.945 * s_c + 0.132/F + 0.0234 \\
&if \;\; F \leq 3.14 - 0.265/s \;\&\; s > 0.128\,,\;\; E = \\
&\qquad 0.2093 * s + 3.729 * s_c + 0.0172 \\
&if \;\; s \leq 0.128\,,\;\; E = \\
&\qquad 0.4589 * s + 3.9784 * s_c + 0.0234
\end{aligned} \tag{5}$$



(a) Large files



(b) Small files

Figure 7. Error rate of energy estimation for interleaving (files have the same order as of the previous figures.)

where F is the compression factor.

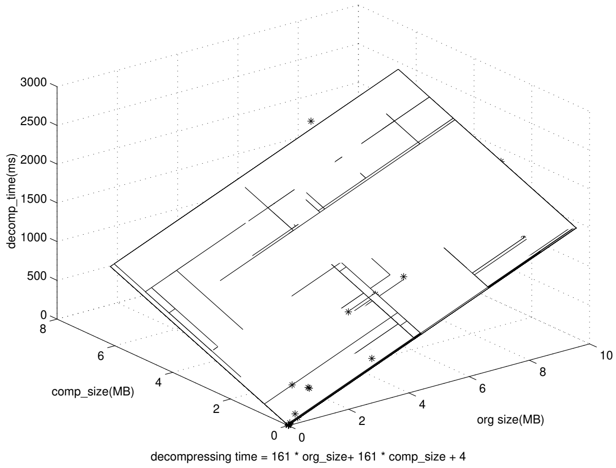
The curve in Figure 9 labeled as `11Mb/s` shows the error rate of energy estimation for `zlib` with interleaving under the 11Mb/s nominal bit rate. For large files, the prediction is rather accurate, with the average error-rate of 2.4%. For small files, the error-rate is relatively high. For the three smallest files, the error-rates range from -40% to 10%. But as file size increases, the precision increases. For the rest of the small files, the average error rate is 5.3%.

As we mentioned earlier, one can turn the WaveLAN card into sleep mode when doing the decompression if no interleaving is applied. We use Equation 2 to calculate the total energy, letting p_d equal to 1.70 (refer to Table 1). We can easily calculate that, in order for sleep mode to outperform interleaving, the compression factor must exceed 4.6 even if we do not count the energy overhead to switch between modes. This explains why the sleep mode does not have much impact on energy saving for `gzip` in the last section.

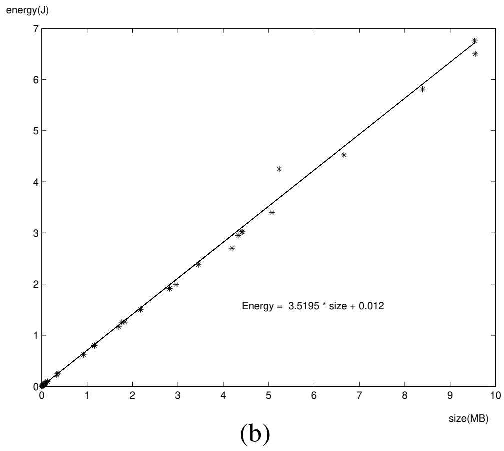
To further test the robustness of our interleaving energy model, we change the experimental setting in various ways. In one setting, we force the 802.11b network device to work at a lower transmission rate of 2Mb/s. We measure the effective receiving rate to be 180K bytes per second and the CPU idle time to be 81.5% of the total downloading time. Based on this information, we derive that, in order to completely fill the idle time with decompression work, one needs a compression factor at least of 27, which is not very likely in practice. For of compression factors lowers than 27, the energy estimation equation should be

$$2.0125 * s + 12.4291 * s_c + 0.0275$$

for the file size greater than 0.128MB. The curve in Figure 9 labeled as `2Mb/s` shows the error rate of energy estimation



(a)



(b)

Figure 8. (a) Fitting plot of decompression time (b)Energy estimation of original downloading

under 2Mb/s nominal bit rate, which indicates that the estimation agrees with the real measurement very well.

4.3 Selective Compression

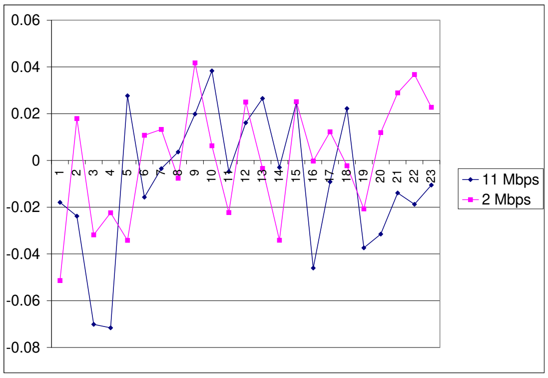
In this subsection, we develop a scheme to use the energy estimation model discussed in last section to further improve the compressed-data downloading efficiency. It is easy to derive that, in order to save energy by compressing the data before downloading, the following conditions must be met:

$$\begin{array}{lll} if\ \ s > 0.128 & : & 1.13/F < 1 - 0.00157/s \\ if\ \ s \leq 0.128 & : & 1.30/F < 1 - 0.00372/s \end{array} \tag{6}$$

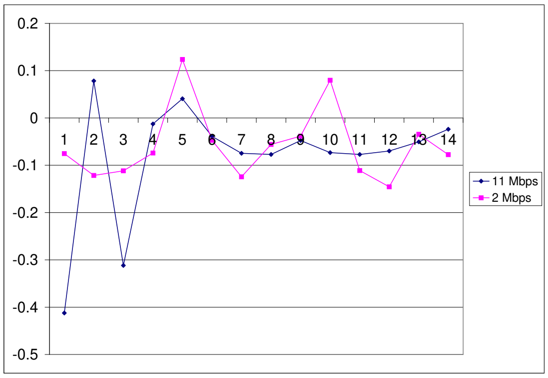
Based on the formula derived above, we do not compress the file if the original size is less than 3900 bytes (0.00372MB). Note that if the original file is much larger than 3900 bytes, only the compression-factor threshold matters.

The compression factors for different blocks (which is determined by the $zlib$ algorithm) within a file do not usually vary much. But certain types of files, for example, tar files, Power Point presentations, and PDF documents, may contain different types of data objects such as texts and graphics. The compression factors for different blocks may be rather different. To deal with such cases, we further modify the `zlib` implementation with a block-by-block adaptive scheme as in Figure 10. Note that here `"send"` means writing to the precompressed file.

Figure 11 shows the result of applying the block-by-block adaptive scheme. We only show the experimental results for files which may be affected by the scheme. For those not listed, the results remain the same as in the previous figures. The figures show that, by using the block-by-block adaptive scheme, we can further reduce the energy cost for the files with low



(a) Large files



(b) Small files

Figure 9. Error rate of energy estimation (files have the same order as of the previous figures.)

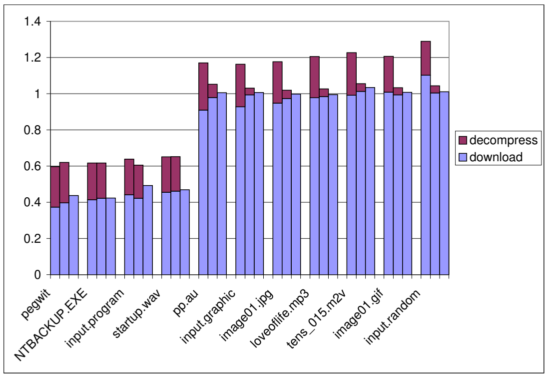
```
For each block
      if (block size < threshold size ) {
            send the raw data;
       }
      else {
            compress the block;
            if (Equation (6) test is negtive) {
                  send the raw data;
             }
            else
                  send the compressed data;
       }
```

Figure 10. A block-by-block adaptive scheme

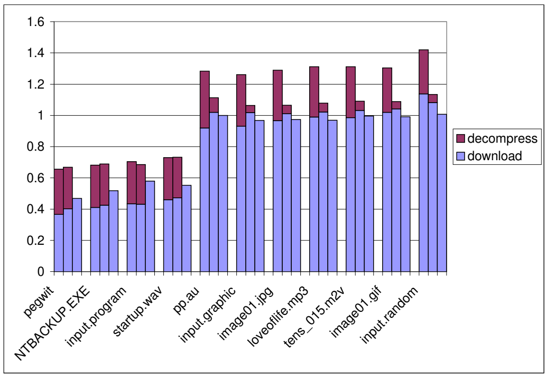
compression factors. We point out that, with this block-by-block adaptive scheme, the compression tool no longer incurs higher energy cost (than no compression) for any file used in our experiments,

5 Compression on demand

In previous sections, we assume that proxy server keeps a copy of the precompressed file. This is not always true. The server may only store the file in its original format. As a result, the decision on whether to compress needs to take the



(a) Time



(b) Energy

Figure 11. Effect of the block-by-block adaptive scheme (left bar: `gzip`**; middle bar:** `zlib w/o interleaving`**; right bar:** `zlib w/ interleaving`**)**

compression speed into account.

For most compression schemes, including three schemes we use in this paper, compression requires more computation resource than decompression. If the rate of compression is slower than the rate of transmitting on the proxy server, the CPU of handheld device may have more idle periods. This in turn potentially provides more opportunities for the interleaving to save the idle energy. However, in our experiments, as shown in the later figures, for those not-so-expensive compression schemes, the compression almost completely overlaps with data transmitting on the proxy server.

It is widely known that `Bzip2` compresses slower than `gzip` and `compress`, so it can be eliminated from the consideration. Here we compare `gzip` and `compress`. We also study the energy saving by interleaving compression with communication.

Figure 12 compares the time for relatively large files which are compressed by `gzip`, `compress` and `zlib`. (The implementation of `zlib` here adapts block by block, overlaps compression with communication, and interleaving downloading and decompression.) Note that we now may have three components for each bar. The upper one is the compress time. The middle one is the decompress time and the lower bar is the downloading time. Again, all quantities shown are relative to the time to download the original data file. Correspondingly, Figure 13 compares the energy for the large-sized files.

From the above figures, we see that `gzip` still fares better than `compress` in nearly all cases, although it takes longer time to compress for several files. The interleaving in the revised `zlib` completely masks the compression time and hence no energy cost is wasted on waiting for the compressed data to arrive. This is mainly because the desktop machine can interleave communication and compression much better.

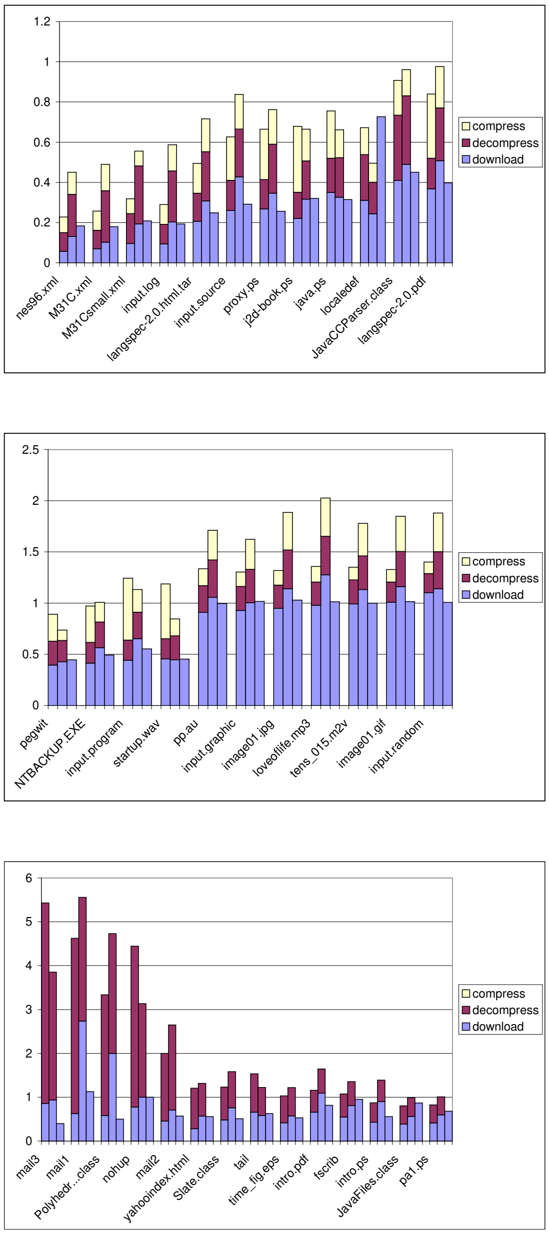


Figure 12. Time comparison when compressing on demand (left bar: `gzip`; middle bar: `compress`; right bar: `zlib with interleaving`)

6 Related Work

Extensive work has been reported in the literature on proxy services, which mostly focus on transcoding of multimedia files to reduce the bandwidth requirement. A number of references are cited in the introduction of this paper.

Previous comparisons between universal coding schemes, theoretically [5] or empirically [3], primarily focus on compression factors and compression speed. An interesting recent technical report examines the effect of `gzip` on *uploading* files from mobile computers [9]. The experiments were conducted on a wired LAN to study the uploading speed. Previous work comparing still-image coding schemes is quite extensive, focusing primarily on the compression factors, compression speeds

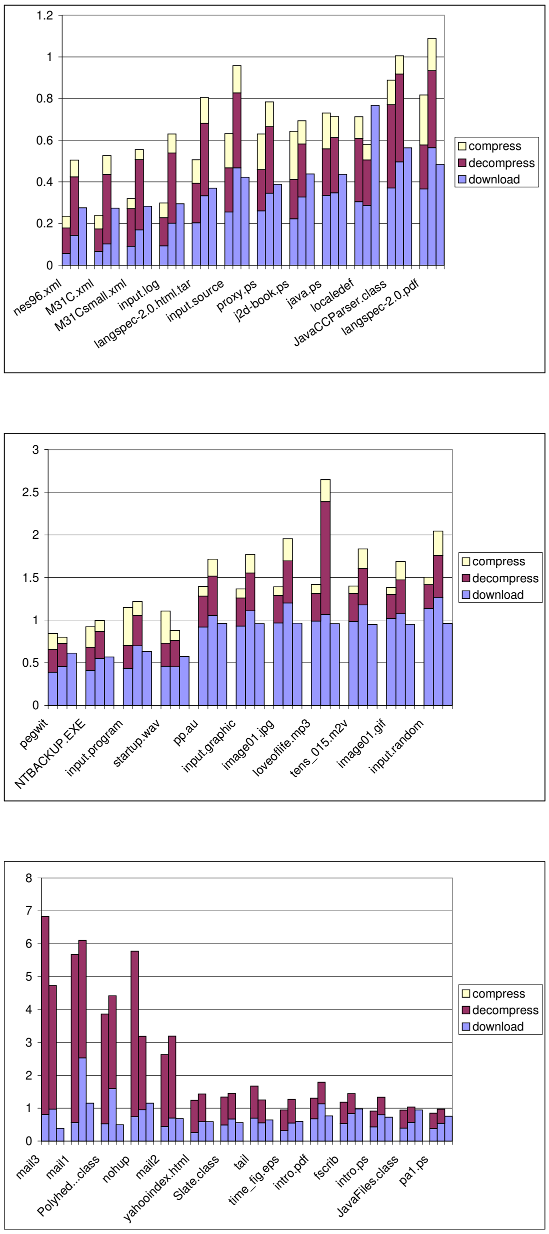


Figure 13. Energy comparison when compressing on demand (left bar: `gzip`**; middle bar:** `compress`**; right bar:** `zlib w/ interleaving`**)**

and the reproduction quality [4, 10]. With our best efforts, we have not found previous work that compares the effect of data compression on energy consumption on handheld devices in a wireless network environment.

7 Conclusion

We have shown that, when carefully chosen and carefully applied, data compression schemes can effectively reduce the energy cost on handheld devices in a wireless LAN environment. When carelessly chosen and applied, however, a data compression scheme can cause a significant energy loss instead of saving. We propose an energy model to estimate the energy consumption for compressed downloading. Based on the model, we develop methods that can make the decision automatically based on the data file size and the compression factor. The model-based enhancements improve the energy efficiency and incur virtually no energy cost for all data files.

The tradeoff is shown to depend on the network bandwidth and the ratio of communication energy over computation energy. With the advent of faster speed wireless LAN devices and the growing popularity of community-wide wireless LAN services, a wider range of experimental environments will become available to enrich the experimental results. Further studies are required for specialized compression schemes for video, music data and for uploading of multimedia data as well.

Acknowledgments

This work is sponsored in part by National Science Foundation through grants CCR-0208760, ACI/ITR-0082834, and CCR-9975309, and by Indiana 21st Century Fund.

References

[1] S. Ardon, P. Gunningberg, Y. Ismailov, B. Landfeldt, M. Portmann, A. Seneviratne, and B. Thai. Mobile aware server architecture: A distributed proxy architecture for content adaptation. In *proc. of INET2001*, Stockholm, Sweden, June 2001.

[2] H. Bharadvaj, A. Joshi, and S. Auephanwiriyakul. An active transcoding proxy to support mobile web access. *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, 1998.

[3] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Systems Research Center, Palo Alto, California, May 1994.

[4] S. Chandra and C. S. Ellis. JPEG compression metric as a quality-aware image transcoding. In *USENIX Symposium on Internet Technologies and Systems*, 1999.

[5] M. Effros. Universal lossless source coding with the burrows wheeler transform. In *Proceedings of the Data Compression Conference*, Snowbird, Utah, March 1999.

[6] A. Fox, I. Goldberg, S. D. Gribble, and D. C. Lee. Experience with top gun wingman: A proxy-based graphical web browser for the 3com palmpilot. In *Proceedings of Middleware '98, September 1998*, Lake District, England, September 1998.

[7] A. Fox, S. D. Gribble, Y. Chawathe, and E. A. Brewer. Adapting to network and client variation using active proxies: Lessons and perspectives. *IEEE Personal Communications*, Aug. 1998.

[8] R. Han and P. Bhagwat. Dynamic adaptation in an image transcoding proxy for mobile web browsing. *IEEE Personal Communications Magazine*, Dec. 1998.

[9] N. Hu. Network aware data transmission with compression. In *The 4th Annual CMU Symposium on Computer Systems*, October 2001.

[10] A. RB and T. TK. Comparison of international standards for lossless still image compression, 1994.

[11] M. Stemm and R. H. Katz. Measuring and reducing energy consumption of network interfaces in hand-held devices. *Proceeding of 3rd International Workshop on Mobile Multimedia Communications (MOMUC-3)*, September 1996.

[12] T. A. Welch. A technique for high-performance data compression. *Computer Magazine of the Computer Group News of the IEEE Computer Group Society*, 17(6), 1984.

[13] B. Zenel. A proxy based filtering mechanism for the mobile environment, 1998.

[14] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.