Performance and Energy Consumption of Lossless Compression/Decompression Utilities on Mobile Computing Platforms

Aleksandar Milenković, Armen Dzhagaryan
Department of Electrical and Computer Engineering
The University of Alabama in Huntsville
Huntsville, AL, U.S.A.
{milenka, aad002}@uah.edu

Martin Burtscher
Department of Computer Science
Texas State University-San Marcos
San Marcos, TX, U.S.A.
burtscher@txstate.edu

***Abstract*—Data compression and decompression utilities can be critical in increasing communication throughput, reducing communication latencies, achieving energy-efficient communication, and making effective use of available storage. This paper experimentally evaluates several such utilities for multiple compression levels on systems that represent current mobile platforms. We characterize each utility in terms of its compression ratio, compression and decompression throughput, and energy efficiency. We consider different use cases that are typical for modern mobile environments. We find a wide variety of energy costs associated with data compression and decompression and provide practical guidelines for selecting the most energy efficient configurations for each use case. The best performing configurations provide 6-fold and 4-fold improvements in energy efficiency for compressed uploads and downloads over WLAN, respectively, when compared to uncompressed data transfers.**

Keywords—mobile computing; measurement techniques; data compression; energy-aware systems

I. INTRODUCTION

The general goal of data compression is to reduce the number of bits needed to represent information. Data can be compressed losslessly or lossily. Lossless compression means that the original data can be reproduced exactly by the decompressor. In contrast, lossy compression, which often results in much higher compression ratios, can only approximate the original data. This is typically acceptable if the data are meant for human consumption such as audio and video. However, program code and input, medical data, email and other text generally do not tolerate lossy compression. We focus on lossless compression in this paper.

Lossless compression is achieved by replacing frequent bit or byte strings with shorter sequences and infrequent bit or byte strings with longer sequences, which tends to reduce the overall data size. For example, in Huffman compression, bit strings are assigned unique, variable-length code words whose length is inversely proportional to the frequency of the corresponding bit strings. Huffman coding [1], or the slower but more sophisticated arithmetic coding [2], is often preceded by a transformation stage whose purpose is to model (or predict) the data. If the model is accurate, then the difference sequence between the predicted and the actual data primarily consists of small values that cluster around zero, which are easy to encode effectively. Various models are in use, including dictionaries of expected or recently encountered "words," sliding windows that assume that recently seen data patterns will repeat, which are used in the Lempel-Ziv approach [3], as well as reversibly sorting data to bring similar values close together, which is the approach taken by the Burrows and Wheeler transform [4]. The data compression algorithms used in practice combine different models and coders, thereby favoring different types of inputs and representing different tradeoffs between speed and compression ratio. Moreover, they typically allow the user to select the dictionary, window, or block size through a command-line argument.

The choice of compression algorithm, the compression level, and the quality of the implementation affect performance and energy consumption. Whereas energy consumed for compression and decompression tasks is not critical on desktop PCs and workstations, it can be a decisive factor in battery-powered handheld devices. Achieving a higher compression ratio requires more computation and therefore energy, but better compression reduces the number of bytes, thus saving energy when transmitting the data. Hence, we believe it is important to take a close look at the *energy efficiency of lossless compression algorithms on state-of-the-art mobile computing platforms. In particular, we want to determine whether compression is useful for reducing energy consumption, which common compression algorithms should be used, what configurations result in the best energy efficiency, and whether parallel execution can save energy.*

In this paper, we perform a comparative, measurement-based study of the most recent versions of several popular compression utilities, including gzip, lzop, bzip2, xz, pigz (a parallel implementation of gzip) and pbzip2 (a parallel implementation of bzip2) on Pandaboard and Raspberry Pi, state-of-the-art mobile development platforms. For each utility, we analyze the effectiveness of all supported compression levels. We examine several performance metrics, such as compression ratio and compression and decompression throughputs. Using our experimental setup for energy measurements [5], we study the amount of energy consumed by compression and decompression tasks and report the energy efficiency.

We evaluate the compression utilities in three typical use scenarios. Local involves compression and decompression tasks performed locally on the platforms of interest. Wired and Wireless involve compression tasks that stream data to

and from a remote server over a secure communication channel. Wired uses an Ethernet network interface and Wireless uses a wireless LAN interface.

Our main findings are as follows.

- The effectiveness and energy efficiency of compression utilities varies widely across different utilities and compression levels, often spanning two orders of magnitude. We identify combinations of the utilities and their compression levels that achieve the best throughput and energy efficiency for typical use scenarios.
- In the Local experiment, lzop with compression levels -1 to -6 achieves the best throughput and energy efficiency in spite of having the lowest compression ratio among all the utilities. lzop with -6 to -9 achieves the best decompression throughput and energy efficiency.
- In the Wired experiment, we find that compressed uploads with lzop -1 to -6 and pigz -1 to -4 are the only combinations that are beneficial, achieving a higher throughput and energy efficiency than the uncompressed uploads. The best performing lzop provides 7.75 times higher energy efficiency than the uncompressed transfer. Similarly, the compressed downloads with gzip, lzop, and pigz provide up to 2.5-fold energy savings over the uncompressed downloads.
- In the Wireless experiment, compressed uploads with pigz and lzop with a low compression level perform the best, providing up to 6.4 times more energy efficient transfers when compared to the uncompressed uploads. For decompression after download, xz with the highest compression levels (-4 to -6) achieves the best decompression throughput and energy efficiency.

Our work complements a prior study that was conducted almost a decade ago [6], [7]. We consider the most recent compression utilities including some with parallel implementations, our setup supports more accurate energy measurements, and we use a state-of-the-art platform representative of modern mobile devices. In addition, we study performance and energy efficiency for all supported compression levels and in three typical use scenarios with representative modern datasets.

The rest of this paper is organized as follows. Section II describes the operation of the six compression utilities we have studied. Section III describes the experimental setup, including the platforms, the dataset, and the measurement setup used to obtain the results. Section IV explains what we measured and how we computed the derived metrics. Section V discusses the results. Section VI surveys related work. Section VII summarizes our findings and draws conclusions.

II. Lossless Compression Utilities

Table 1 lists the six lossless compression utilities we have studied along with the supported range of compression levels. We chose the relatively fast gzip utility and the slower but better compressing bzip2 utility because of their widespread use. lzop is included because of its high speed. xz is also gaining ground and is known for its high compression ratio, slow compression, and fast decompression. Since some handheld devices, including our Pandaboard, are already equipped with multicore CPUs, we also included pigz and pbzip2, which are parallel versions of gzip and bzip2, respectively. All of these utilities operate at byte granularity and support a number of compression levels that allow the user to trade off speed for compression ratio. Lower levels favor speed whereas higher levels result in better compression.

TABLE I. Compression Utilities

Utility	Compression levels	Version	Notes
gzip	1 – 9 (6)	1.4	DEFLATE (Ziv-Lempel, Huffman)
lzop	1 – 9 (3)	1.0.3	LZO (Lempel-Ziv-Oberhumer)
bzip2	1 – 9 (9)	1.0.6	RLE+BWT+MTF+RLE+Huffman
xz	0 – 9 (6)	5.1.0alpha	LZMA2
pigz	1 – 9 (6)	1.1.5	Parallel implementation of gzip
pbzip2	1 – 9 (9)	2.1.6	Parallel implementation of bzip2

gzip [8] implements the deflate algorithm, which is a variant of the LZ77 algorithm [3]. It looks for repeating strings, i.e., sequences of bytes, within a 32 kB sliding window. The length of the string is limited to 256 bytes. gzip uses two Huffman trees, one to compress the distances in the sliding window and another to compress the lengths of the strings as well as the individual bytes that were not part of any matched sequence. The algorithm finds duplicated strings using a chained hash table that is indexed with 3-byte strings. The selected compression level determines the maximum length of the hash chains and whether lazy evaluation should be used.

lzop [9] uses a block-based compression algorithm that favors speed over compression ratio and requires very little memory for decompression. It splits each block of data into sequences of matches (a sliding dictionary) and non-matching literals, which it then compresses.

bzip2 [10] implements a variant of the block-sorting algorithm described by Burrows and Wheeler (BWT) [4]. bzip2 applies a reversible transformation to a block of inputs, uses sorting to group bytes with similar contexts together, and then compresses them with a Huffman coder. The selected compression level adjusts the block size between 100 kB and 900 kB.

xz [11] is based on the Lempel-Ziv-Markov chain compression algorithm (LZMA) developed for 7-Zip [12]. It uses a large dictionary to achieve good compression ratios and employs a variant of LZ77 with special support for repeated match distances. The output is encoded with a range encoder, which uses a probability model for each bit (rather than whole bytes) to avoid mixing unrelated bits, i.e., to boost the compression ratio. We only use compression levels -0 to -6 as the memory requirement for levels -7 to -9 exceeds the available memory on the platforms.

pigz [13] is a parallel version of gzip for shared memory machines that is based on pthreads. It breaks the input up into 128 kB chunks and concurrently compresses multiple

chunks. The compressed data are outputted in their original order. Decompression operates mostly sequentially.

pbzip2 [14] is a multithreaded version of bzip2 that is also based on pthreads. It works by compressing multiple blocks of data simultaneously. The resulting blocks are then concatenated to form the final compressed file, which is compatible with bzip2. Decompression is also parallelized.

III. Experimental Setup

In subsection III.A we describe the platforms, in III.B the dataset, and in III.C the measuring setup.

A. Platforms

We use Pandaboard and Raspberry Pi as the target platforms in our experiments. Pandaboard is designed by Texas Instruments to support software development for smartphones and other mobile devices [15]. It features a Texas Instruments system-on-a-chip (SoC) OMAP4430 [16] with 1 GB of low-power DDR2 SDRAM. The OMAP4430 SoC includes a dual-core ARM Cortex-A9 MPCore processor, a 3D graphics accelerator, an image signal processor, and a rich set of standard peripherals. A number of commercial mobile devices, such as the Amazon Kindle Fire, BlackBerry Playbook, Motorola Droid RAZR, and Samsung Galaxy Tab are based on this chipset. Pandaboard also features an onboard 10/100 Ethernet port, wireless interfaces (802.11n and Bluetooth), DVI and HDMI video interfaces, an audio interface, and two USB ports. Unfortunately, it does not support mobile broadband Internet access. The platform can run mobile open-source operating systems that are based on Linux, including Ubuntu, Android, and Tizen. In our experiments, we use an Ubuntu distribution provided by Linaro, a non-profit organization that works on consolidating and optimizing open-source code for the ARM architecture [17].

Raspberry Pi is a low-cost platform developed by the Raspberry Pi Foundation [18] to promote education and software development. It features a Broadcom BCM2835 SoC, which contains an ARM1176JZFS running at 700 MHz, a Videocore 4 GPU, and 512 MB of RAM. Model B also includes an onboard 10/100 Ethernet port, GPIO pins, RCA and HDMI video interfaces, an audio interface, two USB ports and an SD card slot. Raspberry Pi has a large developer's community with projects ranging from entertainment centers to dedicated computers for photography, home automation, as well as medical and robotic applications. We use Raspbian, a Debian Linux distribution optimized for Raspberry Pi.

B. Datasets

In selecting the data to evaluate the effectiveness of compression algorithms, we compiled a set of diverse input files representative of mobile computing. The input file formats include text, an executable, an image, a file with comma-separated values from a wearable health monitor, and source code. Table 2 gives the input files, their types, size in bytes, and a description. The files are merged into a single archive file (tar) that is used as an input for the compression utilities.

C. Measuring setup

Fig. 1 illustrates the setup for measuring the energy expenditure during program execution on our mobile platforms. The platform is connected to a power supply (V_{SUPPLY} = 5 V) via a low-resistance shunt resistor (R = 0.1 Ω). The voltage over the shunt resistor (V_{SHUNT} = R×I) is sampled using a data acquisition (DAQ) device connected to a development workstation. The current can be calculated from the voltage drop over the shunt resistor as $I = V_{SHUNT}/R$.

The workstation runs a custom mPowerProfile program that controls both the platform (via a serial link terminal) and the DAQ (via a USB port). mPowerProfile starts collecting voltage samples and, after a predefined head delay, a Linux command is issued to the platform. It collects samples during application execution as well as for a predefined tail delay after the application has completed. mPowerProfile provides utilities for configuring the head and tail delays, the scaling factor for samples, and the sampling frequency.

The accuracy of the energy estimation increases with increasing sampling frequency. The maximum sampling frequency supported by the DAQ is 200,000 samples per second (200 Ksps). The processor cores are running at 1 GHz on Pandaboard, which means that we can sample the voltage drop over the shunt resistor every 5,000 CPU clock cycles. We experimented with different sampling frequencies in the range of 10 Ksps to 200 Ksps and evaluated their impact on the energy calculations. We found that the energy calculated using 20 Ksps is within 1% of the energy calculated using 200 Ksps, so for our experiments we use a sampling frequency of 20 Ksps.

TABLE II. Dataset

i	Name	Type	Raw size [bytes]	Notes
1	book	text (txt)	15,711,660	Project Gutenberg Works of Mark Twain
2	libso	exec. (so)	12,452,484	Open source web content engine libwebkit library
3	globe	image (bmp)	16,777,270	An image of Earth from space
4	health	table (csv)	9,988,982	~2 hours of health and physical activity data collected on a health monitor
5	perl	code (tar)	11,233,280	Perl 5.8.5 source code

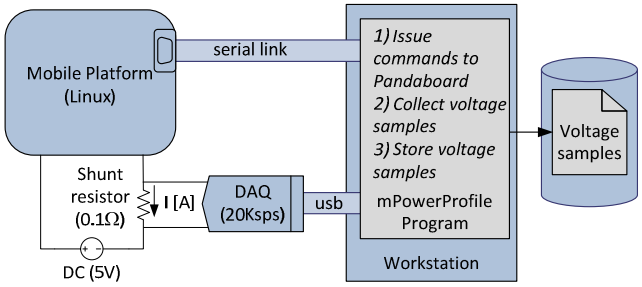


Figure 1. Measuring energy on mobile platforms.

Fig. 2 shows the measured current drawn by Pandaboard during compression of the health input file using gzip -1. The head and tail delays are set to 4 seconds each and the compression takes about 8.6 seconds. Fig. 2a shows the current drawn by Pandaboard during this period as it is used in our energy calculations. Fig. 2b shows the filtered signal, provided here only to enable easier visual inspection by a human of the changes in the current drawn during program execution.

Pandaboard with all unnecessary services turned off draws 0.565 A when idling (I_{idle}=0.565 A). The start of the compression is marked with a steep increase in the current, which remains high throughout the compression and goes down to the idle current once the application has completed. The number of samples during the execution of a compression utility is $n = T.C \times SF$, where $T.C$ is the compression time for a given file and SF is the sampling frequency. The total energy consumed ($ET.C$) is calculated as follows:

$$ET.C = \sum_{j=1}^{n} I_j \cdot V_{PLATFORM,j} \cdot \Delta t \qquad (1)$$

where $\Delta t = 1/SF$, and $V_{PLATFORM,j} = V_{SUPPLY} - I_j \times R$. Note that the calculation can be simplified by assuming $V_{PLATFORM}$ to be constant because the voltage drop over the shunt resistor is negligible. In addition to $ET.C$, we also calculate the energy overhead of the compression task alone, $ET.C(0)$, which excludes the energy needed to run the platform when idle. This energy overhead is calculated as:

$$ET.C(0) = ET.C - I_{idle} \cdot V_{PLARTFORMidle} \cdot T.C \qquad (2)$$

where $V_{PLATFORM,idle} = V_{SUPPLY} - I_{idle} \times R$. We similarly calculate the total energy and the overhead energy for decompression tasks using the decompression time $T.D$ instead of the compression time $T.C$. Once we determine the energy overheads $ET.C(0)$ and $ET.D(0)$, we can find the total energies $ET.C(I_{idle})$ and $ET.D(I_{idle})$ as a function of the idle current using (2), thus decoupling our findings from Pandaboard, which draws a relatively high idle current.

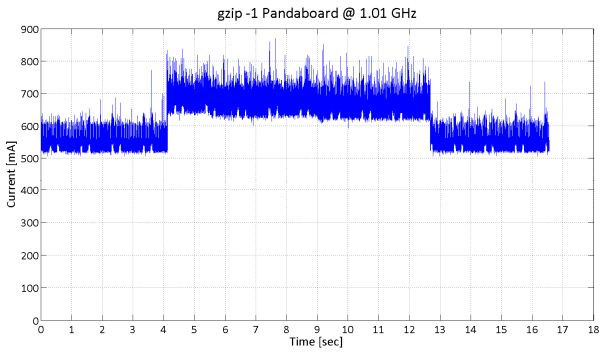
IV. Metrics and Experiments

In subsection IV.A, we describe the metrics used in the evaluation of the compression utilities, including the compression ratio, compression and decompression throughput, and energy efficiency of compression and decompression tasks. Subsection IV.B describes the type of experiments conducted. Table 3 summarizes the metrics used as well as their definitions.

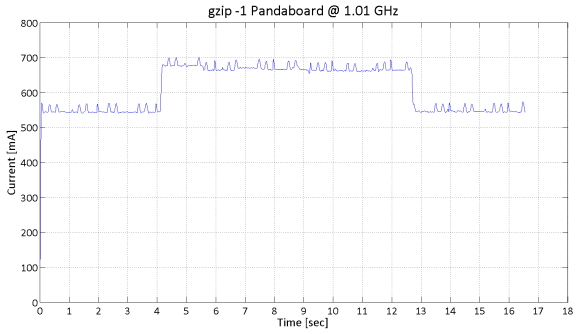
A. Metrics

Compression ratio. We use the compression ratio to evaluate the compression effectiveness of an individual utility and its levels of compression. The compression ratio CR is calculated as the size of the uncompressed input file (US) divided by the size of the compressed file (CS), CR=US/CS.

Performance. To evaluate the performance of individual compression utilities and compression levels, we measure the time to compress the raw input file (T.C) and the time to decompress (T.D) a compressed file generated by that utility



a.



b.

Figure 2. Current drawn by Pandaboard during execution of the gzip utility.

with the selected compression level. The times are measured using the Linux time utility that reports the elapsed time for a running task. Each compression and decompression task is repeated three times and the average time is calculated. Instead of reporting the execution times directly, we report the compression and decompression throughput (Th.C and Th.D), expressed in megabytes per second, which is calculated as the size of the uncompressed input file divided by the time to perform a compression or decompression task. Alternatively, the throughputs can be calculated as the number of bytes eliminated by compression, |US-CS|, divided by the time to perform a compression or decompression task.

The original throughput (Th.C, Th.D) captures the efficiency of data transfers from a user point of view – users produce and consume raw data and care more about the time it takes to transfer data than about what approach is used internally to make the transfer fast. In addition, this metric is suitable for evaluating networked data transfers and comparing compressed and uncompressed transfers. Whereas the alternative throughput metric captures the compression strength of the individual utilities directly, it is not suitable for the evaluation of networked transfers. Thus, we report performance using the throughputs Th.C and Th.D. However, we have considered both metrics and our findings hold for both.

Energy efficiency. For each compression task with a selected compression level, we calculate the energy overhead for compression (ET.C(0)) using the method described in (2). In addition, we calculate the total energy as a function of the idle current (ET.C(I_{idle}), where I_{idle}={0.25, 0.5} A). Simi-

larly, for each decompression task we calculate the total energy as a function of the idle current ((ET.D(I_{idle})). For each combination of a compression utility and a compression level, three measurements are conducted and the average energies are calculated. Instead of reporting the energy directly in joules, we report the energy efficiency (EE.C and EE.D) calculated as the size of the uncompressed input file divided by the total energy to perform a compression or decompression task. An alternative energy efficiency metric can be calculated as the number of bytes eliminated by compression divided by the total energy (|US-CS|/ET.C or |US-CS|/ET.D).

B. Experiments

To evaluate the throughput and energy efficiency of compression and decompression, we consider three typical usage scenarios as illustrated in Fig. 3.

The first experiment (Local) involves measuring the time and energy of compression and decompression tasks performed locally on Pandaboard and Raspberry Pi. To eliminate latencies and energy overheads caused by reading and writing files from the file system on the SD memory card, the input files for the compression and decompression tasks are read from tmpfs, a temporary Linux file system stored in main memory. The output of the compression and decompression tasks is re-directed into the Linux null device (/dev/null) – a special 'file' that discards all data written to it.

The second and third experiments (Wired and Wireless) involve measuring the time and energy of compression and decompression tasks performed on the platforms while communicating with a remote server. For the compression tasks, the raw input file (UF) is read from the local tmpfs, compressed on the platform, and streamed to the remote server over a secure channel. The output files are redirected to the null device of the remote server. For the decompression tasks, the compressed files (CF) are retrieved from the temporary file system of the remote server through a secure channel and decompressed on the platform. The output files are redirected to the null device of the platform. The communication between input, compression/decompression, and output operations is carried out through Linux pipes. The execution times include file transfer latencies as well as compression and decompression times. Similarly, the energies are measured for completing the entire tasks. These two scenarios correspond to typical file-transfer tasks on mobile platforms: compressing and uploading files to a remote server, and downloading files from a remote server and decompressing them. In addition to the transfers that involve compression and decompression operations, we evaluate the time and energy needed to upload and download the raw input file over a secure communication channel. Whereas ssh adds the extra task of data encryption/decryption, it reflects current practice and better represents realistic upload and download settings. We ran additional experiments to quantify the impact of the crypto operations in ssh on the transfer times and the energy consumed but found that their impact is not significant when compared to the netcat and wget utilities that do not use secure communication.

TABLE III. METRICS

Symbol	Description	Unit	Definition
US	Uncompressed file size	MB	Measured
CS	Compressed file size	MB	Measured
CR	Compression ratio	-	US/CS
T.C [T.D]	Time to [de]compress	s	Measured
T.UUP [T.UDW]	Time to upload [download] the raw file	s	Measured
ET.C [ET.D]	Total energy for [de]compression	J	Measured
ET.UUP [UDW]	Total energy for upload [download] of the raw file	J	Measured
ET.C(0) [ET.D(0)]	Overhead energy for [de]compression	J	ET.C – I_{idle}×V_S×T.C [ET.D – I_{idle}×V_S×T.D]
Th.C [Th.D]	[De]Compression throughput	MB/s	US/T.C [US/T.D]
Th.UUP [Th.UDW]	Raw upload [download] throughput	MB/s	US/T.UUP [US/T.UDW]
EE.C [EE.D]	[De]Compression energy efficiency	MB/J	US/ET.C [US/ET.D]

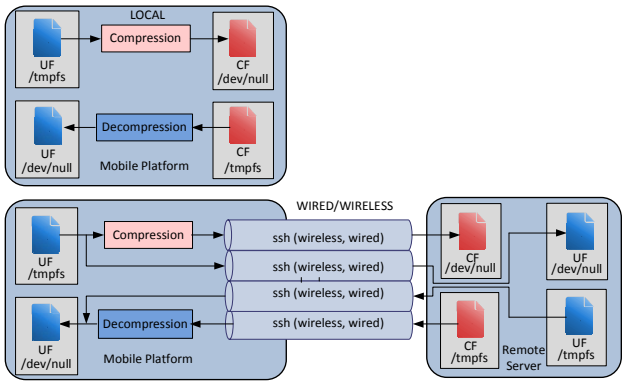


Figure 3. Experiments' data flow. Legend: blue file icons illustrate uncompressed raw files, and red file icons illustrate compressed files. Description: Local (top), and Wired and Wireless experiments (bottom).

In Wired, the platforms are connected to a local router using their Ethernet port. The remote server is also connected to the local router and no other nodes participate in any communication. In Wireless, Pandaboard uses its wireless LAN interface (802.11n) to connect to the local router and through it to the remote server. The remote server is a Dell workstation T1600 with a quad-core Intel Xeon E3-1200 processor (SandyBridge architecture) with 8 GB of main memory. It runs the Ubuntu 12.04 Linux distribution. The local router is a Linksys E900 Wireless N-300 with four 10/100 Ethernet ports.

V. RESULTS

This section discusses the results of our experimental analysis, including the compression ratio (subsection V.A), compression and decompression throughput (subsection V.B), and the energy efficiency as well as the overhead energy efficiency (subsection V.C). Finally, the energy efficiency findings are summarized in subsection V.D.

A. *Compression ratio*

Fig. 4 shows the compression ratio for the input dataset. (pigz and pbzip2 are equivalent to gzip and bzip2, respectively). Generally, the compression ratio increases with an increase in the compression level. The best overall compression ratio is achieved by xz, ranging from 3.38 with -0 to 4.29 with -6; and by bzip2/pbzip2 ranging from 3.49 with -1 to 3.91 with -9. The lowest compression ratio is achieved by lzop, ranging from 2.07 with -1 through -6 to 2.62 with -9.

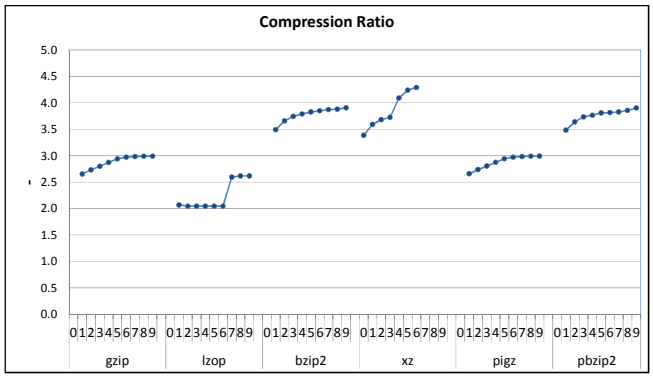
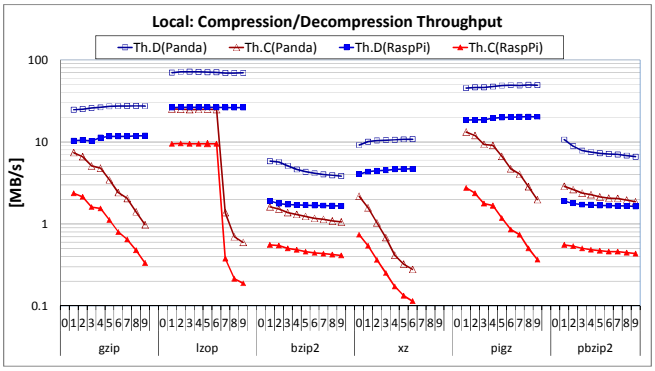


Figure 4. Overall compression ratio (CR).

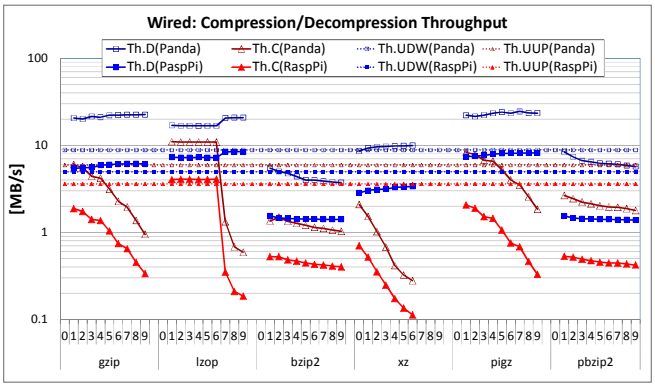
B. *Compression and decompression throughput*

Local. Fig. 5a shows the overall compression and decompression throughput on Pandaboard (Panda) and Raspberry Pi (RaspPi) for Local. The compression throughput varies widely across different compression utilities as well as across different compression levels of a single compression utility. For all compression utilities, the higher compression levels result in lower throughput because of the increased computational complexity. The throughput drop may exceed an order of magnitude, e.g., for lzop. By far the highest compression throughput is achieved by lzop -1 to -6, ~26 MB/s on Pandaboard and 9.5 MB/s on Raspberry Pi. The second highest compression throughput is achieved by pigz -1, 13.2 MB/s on Pandaboard and 2.7 MB/s on Raspberry Pi. pigz fully utilizes two processor cores on Pandaboard to almost double the compression throughput relative to gzip (~7.4 MB/s with -1). pigz gains over gzip even on the single-core Raspberry Pi (~2.7 MB/s vs. ~2.3 MB/s). xz and bzip2 achieve significantly lower compression throughputs (e.g., from 1.6 to 1.1 MB/s for bzip2, and from 2.2 to 0.3 MB/s for xz on Pandaboard). We observe an almost linear speedup in the pbzip2 compression throughput relative to bzip2's on Pandaboard and no speedup on Raspberry Pi.

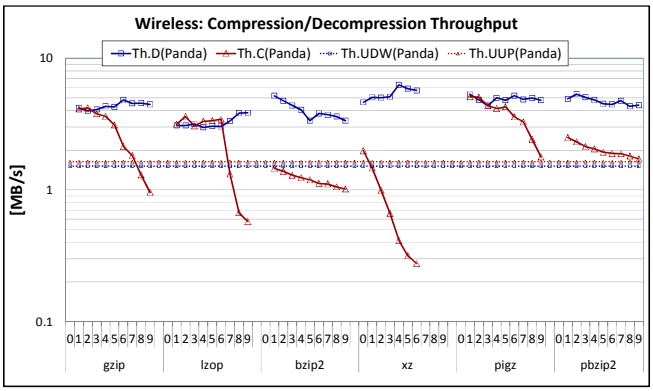
The decompression throughputs are much higher than the compression throughputs (from as low as ~3 times to over 112 times higher) and are only indirectly dependent on the compression level. The higher compression levels resulting in smaller compressed files may increase decompression throughputs because less time is needed to read the input files. Notable exceptions are bzip2 and pbzip2, where decompression throughputs slightly decrease for higher compression levels, in spite of smaller input files. This can be explained by the higher computational complexity of bzip2's decompression when input files are generated using higher compression levels. The highest decompression throughput is achieved by lzop (of 71.9 MB/s on Pandaboard, 26.5 MB/s on Raspberry Pi), followed by pigz and gzip. xz, bzip2, and pbzip2 achieve significantly lower decompression throughputs (below ~10.8 MB/s on Pandaboard and below 4.7 MB/s on Raspberry Pi). pigz decompression throughput almost doubles relative to the gzip on both platforms. Although decompression itself in pigz is not parallelized (it is single threaded), three other threads are created for reading, writing, and checking calculations that speed up decompression [13]. Our findings indicate that even single-core systems benefit from the parallelized implementation in pigz. pbzip2's implementation includes parallelized decompression, thus fully benefiting from the dual-core processor on Pandaboard, but no speedup is observed on Raspberry Pi.



a.



b.



c.

Figure 5. Compression and decompression throughput.

Wired. Fig. 5b shows the compression and decompression throughputs in Wired on Pandaboard and Raspberry Pi. The dashed lines represent the measured effective network throughput when the uncompressed input files are uploaded to the remote server (Th.UUP(Panda) = 5.95 MB/s, Th.UUP(RaspPi) = 3.6 MB/s) and downloaded from the remote server (T.UDW(Panda) = 8.8 MB/s, T.UDW(RaspPi) = 5 MB/s).

The compression throughput is limited by the effective network throughput and is therefore always below CR×Th.UUP. For example, lzop -1 (through -6) on Pandaboard plateaus at 11 MB/s, which is below 2.07×5.95 = 12.3 MB/s (the compression ratio for lzop -1 is 2.07). The effective compression throughput in this case is thus significantly below the 25 MB/s measured in Local. However, for lzop with -7, -8, and -9, where the original compression throughput is lower than the upload network throughput (Th.UUP=5.95 MB/s), the compression throughputs remain unchanged relative to those measured in Local. Similar observations can be made about the other compression utilities. For gzip -1 and pigz -1 on Pandaboard, the compression throughputs are 6 and 8.3 MB/s, respectively, well below the maximum achievable 15.8 MB/s (2.65×5.95, where 2.65 is the compression ratio for gzip and pigz with -1). In contrast, pbzip2 consistently offers a higher compression throughput relative to bzip2 because they both have a compression throughput that is below the effective network upload throughput. When compared to the throughput for uploading the uncompressed dataset, only lzop with -1 to -6, gzip with -1, and pigz with -1 to -4 provide an increased effective network throughput on Pandaboard, whereas the other combinations do not appear to be beneficial (i.e., it takes more time to compress and upload an input file than to just upload the raw input file). On Raspberry Pi, only lzop with -1 to -6 offers an increased effective network throughput, ~4.1 MB/s, slightly more than the 3.6 MB/s achieved for the uncompressed upload.

The decompression throughputs are also limited by the effective network throughput, resulting in lower effective decompression throughputs, which are below CR×(US/T.UDW). For example, lzop with -9 on Pandaboard achieves a decompression throughput of ~20.8 MB/s, which is very close to the maximum achievable (2.62×8.84 = 23.2 MB/s). gzip with -9 achieves ~22.6 MB/s and pigz with -9 achieves ~23.5 MB/s. They outperform lzop because they provide higher compression ratios – their achievable maximum decompression throughput is below 2.99×8.84 = 26.4 MB/s. These three utilities effectively increase the available network throughput (their throughputs are above the Th.UDW line) and decrease the download time relative to the time needed to download the uncompressed file from the remote server. pbzip2, bzip2, and xz are ineffective on both platforms (they fall below the uncompressed throughput).

Wireless. Fig. 5c shows the compression and decompression throughputs for Wireless on Pandaboard. The dashed lines represent the measured effective upload and download throughput when transferring uncompressed files wirelessly, Th.UUP = 1.64 MB/s and Th.UDW = 1.52 MB/s, respectively. Similar to the prior experiment, the effective compression throughput is limited by the network upload throughput and is always below CR×(US/T.UUP). In Wireless, compression effectively increases the upload throughput for gzip with -1 to -7, lzop with -1 to -6, xz with -0, pigz with -1 to -9, and pbzip2 with -1 to -9, whereas bzip2 falls below 1.52 MB/s. Lower effective network throughputs enable more compression configurations to be beneficial. Compression with lower compression levels is still preferred to higher levels. The highest compression throughput of ~5.1 MB/s is achieved by pigz with -1. It outperforms gzip -1 (4.1 MB/s) and lzop -1 (3.2 MB/s).

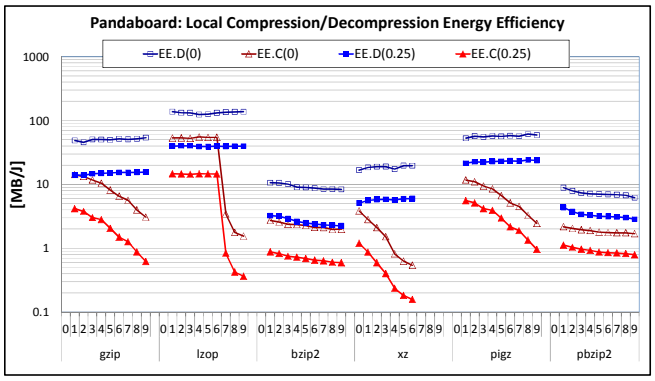
With the low effective throughput for downloads offered by the wireless interface, all decompression utilities increase the available bandwidth (Th.D > Th.UDW for all tested compression utilities with all compression levels). Again, the maximum achievable decompression throughput is limited to CR×Th.UDW. xz provides the highest decompression throughput, ranging from 4.6 with -0 to 6.3 MB/s with -4, followed by pigz (from 4.3 to 5.3 MB/s), and gzip (from 4 to 4.8 MB/s). Interestingly, pigz and pbzip2 offer only limited improvements in decompression throughput over their sequential counterparts due to very low network throughput.

C. *Energy efficiency*

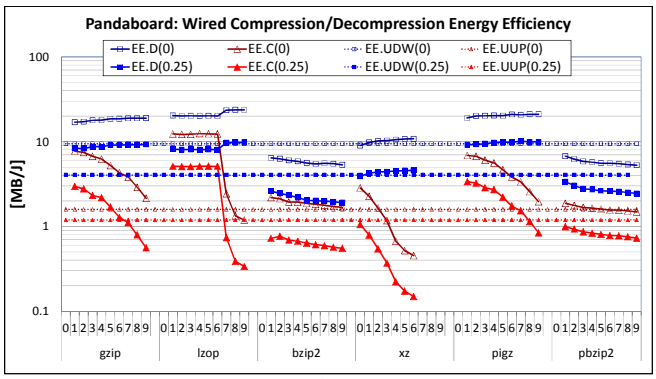
Local. Fig. 6a and Fig. 7a show the energy efficiency for the compression and decompression tasks on Pandaboard and Raspberry Pi, respectively, as a function of the idle current I_{idle} (I_{idle}={0, 0.25} A). Expectedly, the energy efficiency for compression varies widely for different utilities and for different compression levels within each utility (often by more than an order of magnitude). The most energy efficient compression utility by far is lzop with compression levels -1 to -6 regardless of the idle current; it achieves ~54 MB/J (Megabyte/joule) for I_{idle} = 0 A, ~14.5 MB/J for I_{idle} = 0.25 A on Pandaboard, and 45.5 MB/J for I_{idle} = 0 A and 6.5 MB/J for I_{idle} = 0.25 A on Raspberry Pi. Distant second and third are gzip and pigz with -1 achieving ~14 MB/J and ~11 MB/J for I_{idle} = 0 A on Pandaboard. Following the trends in compression throughputs, higher compression levels for gzip, pigz, and lzop result in a dramatic decrease in energy efficiency (e.g., down to 1.5 MB/J for lzop with -9). pigz and pbizp2 are more energy efficient than their sequential counterparts when $I_{idle} \neq 0$ A because they reduce the compression time. However, if we consider only the energy efficiency when I_{idle} = 0 A (EE.C(0)), the parallel implementations are slightly less energy efficient. pbzip2 and bzip2 exhibit low energy efficiencies as does xz, which is the least attractive choice with high compression levels.

The energy efficiencies of the decompression tasks (Fig. 6a and Fig. 7a) vary widely for different utilities. The decompression efficiency is relatively stable for individual utilities – it increases slightly for higher compression levels for all utilities except bzip2 and pbizp2. Thus, EE.D(0) on Pandaboard is ~136 MB/J for lzop, ~50 MB/J for gzip, ~55 MB/J for pigz, and just below ~10 MB/J for bzip2/pbzip2. lzop emerges as the most energy-efficient choice in spite of its lower compression ratio. It remains the most energy effi-

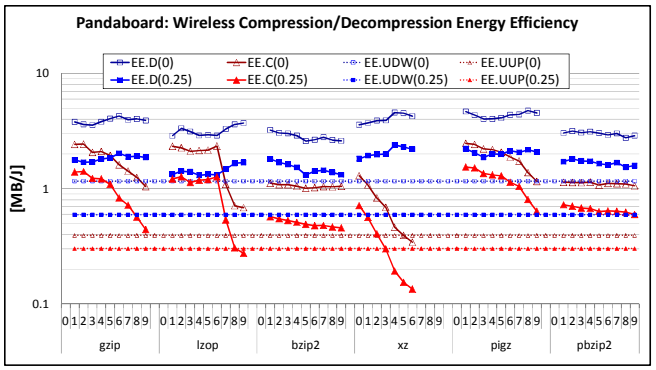
cient tool even when compression strength is taken into account by considering (|US-CS|/ET.D).



a.



b.



c.

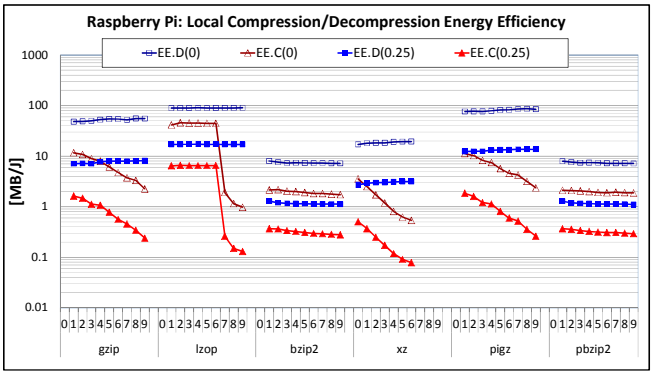
Figure 6. Energy efficiency for Pandaboard.

Wired. Fig. 6b and Fig. 7b show the energy efficiency for compression and decompression as a function of the idle current on Pandaboard and Raspberry Pi, respectively, in Wired. In addition, the graphs show the energy efficiency for uncompressed upload (EE.UUP) and uncompressed download (EE.UDW) as a function of the idle current. This way, one can easily identify cases when transfers with compression and decompression offer higher energy efficiency than raw uploads ($EE.C(I_{idle}) > EE.UUP(I_{idle})$ and raw downloads ($EE.D(I_{idle}) > EE.UDW(I_{idle})$).

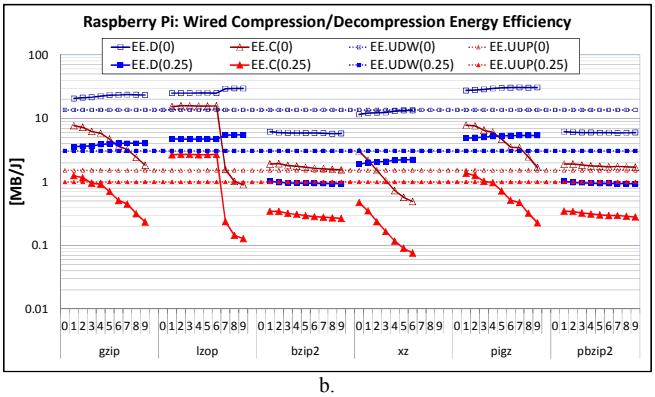
With $I_{idle} = 0$, gzip, pigz, and lzop with -1 to -7 and xz with -1 to -2 provide higher energy efficiency than the raw upload. However, only lzop with -1 to -6, gzip -1 to -4, and pigz with -1 to -5 provide higher energy efficiency for all considered idle currents. Again, the most energy efficient utility is lzop with -1 to -6 with ~12.5 MB/J when $I_{idle}=0$, ~ 5MB/J when $I_{idle} = 0.25$ A. bzip2, pbizp2, and xz exhibit rather low energy efficiency for compression. Similar conclusions can be drawn for Raspberry Pi: only lzop with -1 to -6, and pigz with -1 provide increased energy efficiencies over the uncompressed uploads for all idle currents considered.

The energy efficiency of decompression for gzip, lzop, and pigz exceeds the energy efficiency of the uncompressed download for all considered idle currents, whereas xz, bzip2 and pbzip2 are less energy efficient. lzop -7 to -9 (~23.7 MB/J on Pandaboard, ~29.5 MB/J on Raspberry Pi) and pigz -4 to -9 (21 MB/J on Pandaboard, 30 MB/J on Raspberry Pi) emerge as the most energy efficient utilities on both platforms when $I_{idle} = 0$ A. They also outperform others when $I_{idle} = \{0.25, 0.5\}$ A.

Wireless. Fig. 6c shows the energy efficiency for compression and decompression on Pandaboard as a function of the idle current, respectively. Similar to the previous experiment, the graphs also display the energy efficiency for uncompressed upload (EE.UUP) and uncompressed download (EE.UDW) as a function of the idle current.



a.



b.

Figure 7. Energy efficiency for Raspberry Pi.

The relatively low network throughput for upload results in all utilities having higher energy efficiency than the raw upload when $I_{idle}=0$ A (i.e., $EE.C(0) > EE.UUP(0)$ for all utilities except xz -5 and -6). pigz -1 is the most energy efficient with 2.5 MB/J, followed closely by gzip -1 and lzop -1.

pigz -1 remains the most energy-efficient compression utility when I_{idle} = {0.25, 0.5} A.

All decompression alternatives offer a total energy efficiency that exceeds the total energy efficiency of uncompressed download from the remote server, EE.UDW(0), which is 1.16 MB/J. Generally, downloading files that were compressed with higher compression levels increases the energy efficiency except for bzip2 and pbzip2. The total energy efficiency when I_{idle}=0 A, EE.D(0), is 3.6 to 4.3 MB/J for gzip, 2.9 to 3.7 MB/J for lzop, 4 to 4.8 MB/J for pigz, 2.9 to 3.1 MB/J for pbzip2, and 3.6 to 4.6 MB/J for xz. pigz and xz emerge as the most energy-efficient utilities when I_{idle} = {0.25, 0.5} A. xz benefits from providing a superior compression ratio in conditions where communication energy dominates the overall energy cost.

D. Putting it all together

Table 4 summarizes the results of our experimental study. It lists the most energy efficient utilities for compression and decompression tasks in the three experimental setups (Local, Wired, Wireless) for both platforms (Pandaboard/Raspberry Pi). We show the energy efficiency for compression and decompression tasks (EE.C and EE.D) as well as for raw file transfers (EE.UUP, EE.UDW). The energy efficiencies are reported for three idle currents.

For local compression tasks, we find lzop with -1 to -6 to be superior for both compression throughput and energy efficiency, in spite of yielding the lowest compression ratio. It outperforms the next best utilities pigz with -1 and gzip with -1 by more than a factor of four. Similarly, we find that lzop -6 to -9 outperforms the next best utilities pigz -9 and gzip -9 for local decompression tasks by a factor of 2.5.

For compression tasks in the wired experiment, we find lzop with -1 to -6 to offer superior throughput and energy efficiency. It outperforms pigz with -1 and gzip -1 as the next best alternatives. For decompression tasks, lzop with -7 to -9 and pigz with -6 to -9 are the most energy efficient, with pigz having slightly higher efficiency when the idle current is higher than zero. The most energy-efficient compression utility in the wireless experiment is pigz -1 (followed closely by gzip -1 and lzop -1 to -6), whereas xz with -4 to -6 and pigz with -7 to -9 stand out for decompression tasks.

In summary, we find a high throughput to be most important for achieving good energy efficiency for compression tasks, with improved compression ratios only slightly affecting the choice of utility as the available bandwidth becomes constrained. In contrast, decompression also favors throughput but only in combination with a reasonable compression ratio, and the balance rapidly tips towards more emphasis on compression ratio as the available bandwidth becomes limited.

VI. RELATED WORK

We are aware of two related studies that investigate data compression in the context of energy efficiency on embedded and mobile systems [6], [19]. Both studies examine the feasibility of using compression to reduce energy consumption and explore tradeoffs between time, compression ratio, and energy.

The most closely related work to ours is a study by Barr and Asanović [6], [7]. It also investigates the energy efficiency of lossless data compression on a wireless mobile device. Their excellent publications include details that are beyond the scope of our work, such as the frequency with which different types of instructions are executed, the branch prediction accuracy, and the performance of the memory hierarchy. Their experimental setup has several advantages over ours. For example, their Skiff platform, which mimics an iPAQ mobile device, enabled them to separately measure the energy drawn by the CPU, the memory subsystem, peripherals, and the wireless interface. However, our test environment is superior in other aspects. Some of them are simply a result of almost a decade of advances in technology. For instance, their now obsolete processor had a single core, a clock frequency of 233 MHz, and 32 MB of DRAM. The Skiff platform was further limited to 4 MB of nonvolatile flash memory. Thus, the root file system had to be mounted externally via an Ethernet port using NFS. In comparison, our OMAP4430 has two cores, runs at 1.01 GHz, and has 1 GB of DDR2 SDRAM. The OMAP SoC is one of the leading platforms for current mobile devices and features an integrated communication interface and supports higher transmission speeds. Another advantage of our test bed is the sampling frequency of 20 kHz, which is about 500 times

TABLE IV. THE MOST ENERGY-EFFICIENT UTILITIES

Experiment	**Compression**			**Raw UUP**		**Decompression**			**Raw UDW**	
LOCAL	Best Utility	EE.C [MB/J]		EE.UUP [MB/J]		Best Utility	EE.D [MB/J]		EE.UDW [MB/J]	
		Panda	RaspPi	Panda	RaspPi		Panda	RaspPi	Panda	RaspPi
I_{idle} = 0 A	lzop -1 to -6	55	44	-	-	lzop -6 to -9	137	90	-	-
I_{idle} = 0.25 A	lzop -1 to -6	14.5	6.5	-	-	lzop -6 to -9	40	17	-	-
I_{idle} = 0.5 A	lzop -1 to -6	8.5	3.5	-		lzop -1 to -9	23	9.5	-	-
WIRED					-					
I_{idle} = 0 A	lzop -1 to -6	12.4	15.5	1.6	1.53	lzop -7 to -9	23.5	30	9.4	13.5
I_{idle} = 0.25 A	lzop -1 to -6	5.1	2.7	1.2	0.74	pigz -6 to -9	10	5.5	4.0	3.1
I_{idle} = 0.5 A	lzop -1 to -6	3.2	1.5	0.95	0.6	pigz -6 to -9	6.5	3	2.6	1.7
WIRELESS										
I_{idle} = 0 A	pigz -1	2.5		0.39		pigz -4 to -7	4.6		1.16	
I_{idle} = 0.25 A	pigz -1	1.5		0.30		xz -4 to -6	2.4		0.59	
I_{idle} = 0.5 A	pigz -1	1.1		0.25		xz -4 to -6	1.6		0.40	

higher than theirs, presumably yielding more accurate measurements. Even when accounting for the difference in clock frequency, our hardware takes a sample every 50,000 CPU clock periods whereas theirs sampled once per five million clocks.

There are also substantial software differences between Barr and Asanović's study and ours. Whereas several of their compression utilities are predecessors of the utilities we evaluated, they only tested a few compression levels (we test all of them), and we include newer utilities such as xz as well as the parallel implementations pigz and pbzip2. Furthermore, their input data was limited to 1 MB of text and 1 MB of web data. We cover a wider range of relevant data types and our files are over an order of magnitude larger. Due to their hardware's low sampling rate, they were forced to run the same compression or decompression in an infinite loop to obtain sufficiently many samples. We are able to run our tests individually, that is, in a manner that is more representative of actual usage.

VII. Conclusion

This paper describes an experimental evaluation of recent implementations of common compression utilities on Pandaboard and Raspberry Pi, two state-of-the-art mobile development platforms. We measure compression and decompression times, total and overhead energies consumed by compression and decompression tasks and report metrics such as compression ratio, compression/decompression throughputs, and compression/decompression energy efficiencies across different compression levels. Our measurements mimic typical usage scenarios of mobile devices involving transfers of data over wired and wireless networks.

Based on the results of our analysis, we provide practical guidelines for selecting the most energy-efficient utilities depending on the usage scenario. For compression tasks, utilities that are fast and have low computational complexity, such as lzop with compression levels -1 to -6 and pigz with -1, are the most energy-efficient options in spite of their relatively low compression ratio. For decompression tasks, lzop, gzip, and pigz are good choices, as well as xz when transferring data over a low-throughput wireless network. The results of our study show that the common utilities with their default compression levels may not always be the most energy-efficient combinations. For example, the energy efficiency of compressed uploads over the wireless network using the widely used gzip with the default compression level -6 is 1.6 MB/J, whereas pigz -1 achieves 2.5 MB/J, a 50% improvement in energy-efficiency.

Our findings may guide energy optimizations of data transfers in mobile applications and encourage the development of data transfer frameworks that are conscientious of the mobile device's energy status. For example, a server could easily store multiple copies of the same file, compressed with different utilities and compression levels, to allow the mobile device to choose, based on its capabilities, currently available network bandwidth, energy status, and user preferences, which version of a file to download. Based on similar criteria, the mobile device could choose which format to use for uploading a file, and the server could then convert the file, if necessary, to the best download format(s).

Acknowledgment

This material is based upon work supported in part by the National Science Foundation under Grants No. 1141022, 1205439, 1217231 and 1217470. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] D. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *Proc. Ire*, vol. 40, no. 9, pp. 1098–1101, Sep. 1952.

[2] J. Rissanen and G. G. Langdon, "Arithmetic Coding," *IBM J. Res. Dev.*, vol. 23, no. 2, pp. 149–162, Mar. 1979.

[3] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inf. Theory*, vol. 23, pp. 337–343, 1977.

[4] M. Burrows and D. J. Wheeler, "A Block-sorting Lossless Data Compression Algorithm," Digital SRC, Report 124, May 1994.

[5] M. Milosevic, A. Dzhagaryan, E. Jovanov, and A. Milenkovic, "An Environment for Automated Power Measurements on Mobile Computing Platform," in *in the Proceedings of the ACM Southeast Conference (ACMSE'13)*, Savannah, GA, 2013.

[6] K. Barr and K. Asanović, "Energy aware lossless data compression," in *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services (MobiSys'03)*, 2003, pp. 231–244.

[7] K. C. Barr and K. Asanović, "Energy-aware lossless data compression," *ACM Trans. Comput. Syst.*, vol. 24, no. 3, pp. 250–291, Aug. 2006.

[8] "The gzip home page." [Online]. Available: http://www.gzip.org/. [Accessed: 25-May-2012].

[9] M. Oberhumer, "lzop file compressor (oberhumer.com Open-Source)." [Online]. Available: http://www.lzop.org/. [Accessed: 25-May-2012].

[10] "bzip2: Home." [Online]. Available: http://www.bzip.org/. [Accessed: 25-May-2012].

[11] "XZ Utils." [Online]. Available: http://tukaani.org/xz/. [Accessed: 25-May-2012].

[12] I. Pavlov, "7-Zip." [Online]. Available: http://www.7-zip.org/. [Accessed: 25-May-2012].

[13] "pigz - Parallel gzip." [Online]. Available: http://zlib.net/pigz/. [Accessed: 25-May-2012].

[14] J. Gilchrist, "Parallel BZIP2 (PBZIP2)." [Online]. Available: http://compression.ca/pbzip2/. [Accessed: 25-May-2012].

[15] "Pandaboard." [Online]. Available: http://pandaboard.org/. [Accessed: 28-May-2012].

[16] "OMAP™ 4 Platform - OMAP4430/OMAP4460." [Online]. Available: http://www.ti.com/omap4430. [Accessed: 02-Jun-2012].

[17] "Linaro: open source software for ARM SoCs." [Online]. Available: http://www.linaro.org/. [Accessed: 28-May-2012].

[18] "Raspberry Pi." [Online]. Available: http://www.raspberrypi.org/. [Accessed: 05-Feb-2013].

[19] C. M. Sadler and M. Martonosi, "Data compression algorithms for energy-constrained devices in delay tolerant networks," in *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, 2006, pp. 265–278.