Energy-Aware Lossless Data Compression

KENNETH C. BARR and KRSTE ASANOVIĆ
MIT Computer Science and Artificial Intelligence Laboratory

Wireless transmission of a single bit can require over 1000 times more energy than a single 32-bit computation. It can therefore be beneficial to perform additional computation to reduce the number of bits transmitted. If the energy required to compress data is less than the energy required to send it, there is a net energy savings and an increase in battery life for portable computers. This article presents a study of the energy savings possible by losslessly compressing data prior to transmission. A variety of algorithms were measured on a StrongARM SA-110 processor. This work demonstrates that, with several typical compression algorithms, there is a actually a net energy *increase* when compression is applied before transmission. Reasons for this increase are explained and suggestions are made to avoid it. One such energy-aware suggestion is *asymmetric compression*, the use of one compression algorithm on the transmit side and a different algorithm for the receive path. By choosing the lowest-energy compressor and decompressor on the test platform, overall energy to send and receive data can be reduced by 11% compared with a well-chosen symmetric pair, or up to 57% over the default symmetric *zlib* scheme.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]—*Performance attributes*

General Terms: Algorithms, Measurement, Performance

Additional Key Words and Phrases: Compression, energy-aware, lossless, low-power, power-aware

1. INTRODUCTION

Wireless communication is an essential component of mobile computing, but the energy required for transmission of a single bit has been measured to be over 1000 times greater than for a single 32-bit computation. Thus, if 1000 computation operations can compress data by even 1 bit, energy should be saved. Compression algorithms which once seemed too resource- or time-intensive might be valuable for saving energy. Implementations which made concessions in compression ratio to improve performance might be modified to provide an overall energy savings. Ideally, the effort exerted to compress data should be variable to allow a flexible tradeoff between speed and energy. Earlier work has considered lossy compression techniques which sacrifice the quality of compressed audio

This work was supported by MIT Project Oxygen, DARPA PAC/C award F30602-00-2-0562, NSF CAREER award CCR-0093354, and an equipment grant from Intel.
Authors' addresses: The Stata Center, Building 32-G776, 32 Vassar Street, Cambridge, MA 02139; email: {kbarr,krste}@mit.edu.

or video data streams to reduce the bit rate and energy required [Flinn 2001; Sinha et al. 2000; Taylor and Dey 2001; Noble and Satyanarayanan 1999]. In this work, we consider the challenge of reducing wireless communication energy for data that must be transmitted faithfully.

We provide an in-depth examination of the energy requirements of several lossless data compression schemes. We take energy measurements using the StrongARM-based *Skiff* platform developed by Compaq Cambridge Research Labs, which allows energy consumption of the CPU, memory, network card, and peripherals to be measured separately. The platform is similar to the popular iPAQ handheld computer, so the results are relevant to designers of embedded and mobile systems. Several families of compression algorithms are analyzed and characterized, and it is shown that compression prior to transmission may actually cause an overall energy increase. We highlight behaviors and resource-usage patterns which allow for energy-efficient lossless compression of data. Finally, a new energy-aware data compression scheme composed of these behaviors is presented and measured.

Section 2 contains a brief overview of important lossless compression concepts, algorithms, and the implementations that we study. Section 3 describes the experimental setup including benchmark selection and methodology. Section 4 begins with the measurement of an encouraging communication-computation gap, but shows that modern compression tools do not exploit the the low relative energy of computation versus communication. Factors which limit energy reduction are presented. Section 5 applies an understanding of these factors to reduce overall energy of transmission though hardware-conscious optimizations and asymmetric compression choices. Section 6 discusses related work, and Section 7 concludes the article.

2. LOSSLESS COMPRESSION OVERVIEW

In this section, we review the concepts and terminology of data compression to provide background for our discussion on what constitutes energy-aware lossless data compression. The descriptions in this section are simplified versions of those that appear in Lelewer and Hirschberg [1987] and Sayood [2002], each of which contains bibliographic references to seminal articles. Particular implementations of algorithms will be discussed as each algorithm is introduced in Section 2.3.

2.1 Terminology

In applications where some loss or degradation of data can be tolerated (such as the transmission of images or sounds) much work has been done to exploit this tolerance in order to reap higher *lossy* compression ratios. When transmitting text or a binary executable, one must be able to reconstruct every bit perfectly—hence the need for *lossless* data compression.

Compression is usually broken into two steps: *modeling* and *coding*. With a perfect, concise *model* that describes the generation of the input source which is to be compressed, one could reproduce the data without transmitting the source data. (i.e., if the sequence $1\ 1\ 2\ 3\ 5 \cdots 6765$ were to be transmitted, one

could express it with a "model" of Fibonacci numbers). In practice, one must approximate and construct an approximate mathematical model for the data. In English text, for example, one can model the probability of a letter occurring as a probability conditioned on letters that have already been transmitted. This probabilistic model is transmitted with a description of how the data differs from the model.

In a *coding* step, information is mapped to compact *codewords*. Obviously, a codeword must decode to a unique value so there can be no doubt of the original message. *Prefix codes* are used so that no codeword is the prefix of any other codeword. It has been proved [McEliece 1977] that, for any nonprefix code that may be uniquely decoded, a prefix code can be found with the same codeword lengths. Often the modeling and coding steps are tightly coupled. For instance, Lempel-Ziv codes can be constructed as an input source is parsed into a "dictionary" model. When it is difficult to extricate the coding from the modeling, the two will be discussed together.

2.2 Coding Methods

Coding maps *symbols* from the input *alphabet* into compact binary sequences. Discussion in this section is based on an alphabet made up of the set of 256 symbols that can be represented in an 8-bit byte. Though many coding schemes exist, we focus on the most popular schemes for data compression tools.

2.2.1 *Huffman Coding.* If the probability of each source symbol is known a priori (perhaps by scanning through the source), a procedure known as static *Huffman coding* can be used to build an optimal code in which the most frequently occurring symbols are given the shortest codewords. Huffman codes are established by storing the symbols of the alphabet in a binary tree according to their probability. As the tree is traversed from root to leaf, the code grows in length. When visiting the right child, a 0 is appended to the code. When visiting the left child, a 1 is appended. Thus, symbols which occur frequently are stored near the root of the tree and have the shortest codes. Since data compression tools rarely have the luxury of a priori knowledge and cannot afford two passes through the data source, variants of the Huffman algorithm have been developed that work dynamically and update the tree as source symbols are encountered.

2.2.2 *Arithmetic Coding.* Optimal compression ratio for a data source is traditionally described with respect to Claude Shannon's definition of source *entropy* [Shannon 1948]: a measure of the source's information and therefore the average number of bits required to represent it. Sometimes the most frequently occurring symbol can contain so little information that it would be ideal to represent it with less than 1 bit. Huffman codes are restricted to using an integral number of bits per symbol, increasing the coding overhead. Arithmetic codes have been designed to support a fractional number of bits per symbol to bring the average length of a codeword much closer to the optimal.

Knowing the probability of occurrence for each symbol, a unique identifier can be established for a series of symbols. This identifier is a binary fraction in

the interval [0,1). Unlikely symbols narrow this interval so that more bits are required to specify it, while highly likely symbols add little information to a message and require the addition of fewer bits as the interval refinement is coarser. As the fraction converges, the most significant bits become fixed, so the fraction can be transmitted most-significant-bit-first as soon as it is known. Arithmetic coding requires frequent division and multiplication, but our experiments show that an optimized implementation can run faster than the well-optimized Unix *compact* program, an adaptive Huffman encoder.

2.2.3 *Lempel-Ziv Codes.* A Lempel-Ziv codebook is made up of fixed-length codewords in which each entry has nearly the same probability of appearing, but in which longer groups of symbols are represented in the same length as single symbols. Thus, it may require extra bits to send the coded version of a single symbol, but a string of frequently occurring symbols can be represented with a fraction of the bits ordinarily required. Since only n codewords can be represented with $lg(n)$ bits, systems for dynamically increasing the length of codewords exist.

2.3 Lossless Compression Algorithms

The coding techniques described above are used in the algorithm families introduced below. There are two fundamental methods for constructing Lempel-Ziv codes. Introduced in the late 1970s, these methods are known by the initials of their creators and the year of introduction: LZ77 and LZ78. Prediction with Partial Match (PPM) uses Markov modeling followed by arithmetic coding. The Burrows-Wheeler Transform (BWT) reversibly permutes a block of source data so that it can easily be compressed. After introducing each algorithm, an implementation is presented. The implementations (*bzip2*, *compress*, *LZO*, *PPMd*, and *zlib*) are the benchmarks used for the investigation in Section 4.

2.3.1 *Sliding Window—LZ77.* LZ77 maintains a current pointer into the source data, a search buffer, and a look-ahead buffer [Ziv and Lempel 1977]. The search buffer is made up of symbols encountered prior to the current symbol, and the look-ahead buffer contains symbols which appear after the current symbol. Together, the buffers comprise a "window" which specifies the section of the input source under consideration. As the current pointer advances, the window "slides" over the input. As symbols are encountered in the look-ahead buffer, the algorithm searches backward for the longest match in the search buffer. Instead of transmitting the matched symbols, they can be encoded with a triple: <offset from pointer, length of match, next codeword>. The "next codeword" is the codeword corresponding to the symbol in the look-ahead buffer following the match. It is necessary in case a match for the look-ahead buffer cannot be found (in which case <0,0,s> is transmitted where s is the codeword of the current symbol).

This scheme can be enhanced by using a variable length coder (e.g., Huffman coding) to reduce the size of the fixed-length triples. Another popular enhancement involves a more efficient way to represent a single character without an entire triple, using a flag to indicate whether a literal or match is being transmitted.

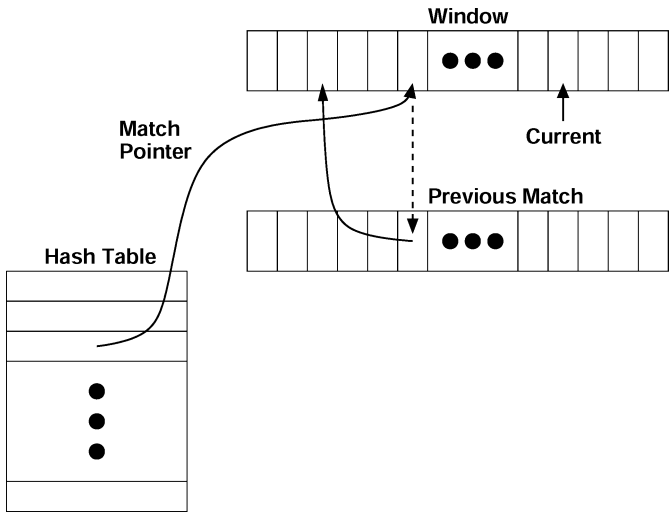

Fig. 1. Hash table implementation of LZ77.

As one of our representative implementations for LZ77, we chose the *zlib* library form of the popular *gzip* utility [Gailly and Adler 2002]. The library form was chosen because it provides more options to trade off memory for performance. Unless specified, it is configured with similar parameters as *gzip* including a default 32-KB sliding window. Most of the window is a search buffer; the rest is a fixed-size, 262 symbol look-ahead buffer. Literals and offsets are encoded with Huffman trees. These trees are compacted with another round of Huffman coding using either a tree built in to the library or an adaptive one that must be sent with the compressed data, with *zlib* choosing the optimal scheme on a block-by-block basis. The LZ77/Huffman algorithm in this form is called *deflate*. Window size and memory size may be set by the user. A larger window improves the ability to find a match. More memory allows for less collisions in an internal hash table. Users may also set an "effort" parameter which dictates how hard the compressor should try to extend matches it finds in its history buffer.

zlib implements its longest-match search with the three arrays depicted in Figure 1. As the *current* pointer moves through the window, a hash of the current symbol and the two that follow is computed. This hash is used to index into a table. If the entry is empty, a pointer to the current symbol is added. If a corresponding *match* pointer into the window is already present, the program scans forward from *current* and *match* in an attempt to extend the match. To further extend the match, a chain of previous matches is maintained for each index into the window. The chain is followed, and the longest match is selected. In the interest of speed, the user may limit traversal of the chain, settling for *a* match rather than the *longest* match. To decompress the data, no searching is needed as the compressor has issued an explicit stream of literals, locations, and match lengths. Note that the process becomes even more efficient if the window is contained entirely in the cache, so that retrieving a match is fast no matter where it occurs in the window.

As an alternative LZ77 implementation, we examine the *LZO* compression library designed for "real-time" compression [Oberhumer 2000]. Like *zlib*, it uses LZ77 with a hash table to perform searches. *LZO* is unique in that its hash table dictionary fits in 16 KB of memory so it can remain in cache. Its small footprint, coding style (it is written completely with macros to avoid function call overhead), and ability to read and write data "in-place" without additional copies make *LZO* extremely fast. In the interest of speed, its hash table can only store pointers to 4096 matches, and no effort is made to find the longest match. Match length and offset are coded more simply than in *zlib*; large offsets are represented by combining their least significant bits with short markers.

2.3.2 *Dictionary—LZ78.* The LZ78 scheme was introduced to account for cases in which a nearby match cannot be found [Ziv and Lempel 1978]. Instead of the sliding search-buffer, LZ78 uses a separate dictionary, which also serves as a codebook. As each group of symbols is encountered, the dictionary is checked. An <index, code> pair is output where *index* corresponds to the longest prefix (if any) that matches the current input, and *code* is the unmatched symbol which follows. The pair is then added to the dictionary. The decompressor builds its dictionary in a corresponding fashion so that received indices refer to the same symbol as they did in the compressor. A popular improvement to LZ78 is called LZW [Welch 1984]. It seeds the dictionary with letters from the source alphabet which eliminates the need to send the second element of the pair, shortening the number of bits that must be sent for a single character. With every symbol present in the dictionary, only the index need be sent. Since each new dictionary entry contains a pointer to a previous entry, decoding occurs recursively, requiring decompression to buffer symbols in a stack and reverse them before output.

Such a system results in the quick accumulation of long patterns which can be stored indefinitely, but has several drawbacks. Until the dictionary is filled with longer frequently seen patterns, "compressed" output will be larger than in its original form. Since the dictionary can grow without bound, implementations of LZ78 must erase the dictionary when it gets too large, freeze the dictionary and continue in a nonadaptive fashion, or adopt another policy to limit memory usage.

We chose the popular Unix utility *compress* as a representative implementation of the LZW algorithm. It implements LZW with codewords beginning at nine bits. When all 9-bit codes have been used, the codebook size is doubled and 10-bit codes are used. This doubling continues until codes are 16 bits long, inclusive. The dictionary becomes static once it is entirely full. Whenever *compress* detects a decreasing compression ratio, the dictionary is cleared and the process begins anew. Dictionary entries are stored in a hash table.

Each hash table entry contains its code, the code of its immediate predecessor, and a symbol. Figure 2 shows the table entries for the word *baseball*. The blank space serves as a reminder that since the entries are in a hash table, they are not stored consecutively. As each symbol from the input string is encountered, it is hashed with the previous code to determine its location in the table. The hashing repeats until a symbol (rather than another hash index)

Code	Previous code	Symbol	Equivalent string
0-255	n/a	Literals (every 8-bit ASCII character)	
258	115 ('s')	'e'	"base"
257	97 ('a')	's'	"bas"
256	98 ('b')	'a'	"ba"
259	101 ('e')	'b'	"baseb"
260	256 ('ba')	'l'	"basebal"
261	108 ('l')	'l'	"baseball"

Fig. 2. Hash table implementation of LZW.

is found in the "previous code" field. Hashing allows constant access time on average to any <prefix, symbol> pair, but has the disadvantage of poor spatial locality when combining multiple entries to form a string. To reduce collisions, the table should be sparsely filled, which results in wasted memory. During decompression, each pair may be inserted into a table in the location specified by its code, so no collisions will occur and no space is wasted. Despite the random dispersal of codes throughout the table, common strings will benefit from temporal locality.

2.3.3 *Prediction with Partial Match—PPM.* The fact that a certain string of symbols has appeared can aid in predicting which symbol will come next. For instance, if the letters *compr* appear in this article, there is a strong probability they will be followed by an *e*. The PPM scheme maintains such *context* information to estimate the probability of the next input symbol to appear [Cleary and Witten 1984]. An arithmetic coder can use this stream of probabilities to code the source efficiently. Clearly, longer contexts will improve the probability estimation, but require more time to arise (this is similar to the startup effect in LZ78). To account for this, "escape symbols" exist to progressively step down to shorter context lengths. This introduces a tradeoff in which encoding a long series of escape symbols can require more space than is saved by the use of large contexts. Much effort has gone into choosing probabilities for the escape symbols to minimize their overhead. Storing and searching through each context accounts for the large memory requirements of PPM schemes.

PPMd is a recent implementation of the PPM algorithm [Shkarin 2002b]. Windows users may unknowingly be using *PPMd* as it is the text compression engine in the popular *WinRAR* program. The length of the maximum context can be varied, but defaults to four. When the context tree fills up, *PPMd* can clear and start from scratch, freeze the model and continue statically, or prune sections of the tree until the model fits into memory.

2.3.4 *Burrows-Wheeler Transform—BWT.* The newest technique among those examined, the Burrows-Wheeler Transform, converts a block S of length n into a pair consisting of a permutation of S (call it L) and an integer in the

interval $[0..n-1]$. Though the transformation is simple to describe, it is not obvious how it may be reversed. Curious readers are referred to Burrows and Wheeler [1994]. More important than the method is its effect. The transform collects groups of identical input symbols such that the probability of finding a symbol *ch* in a region of L is very high if another instance of *ch* is nearby. Such an L can be processed with a *move-to-front* (MTF) coder which will yield a series consisting of a small alphabet: runs of zeros punctuated with low numbers which in turn can be processed with the coders seen above (Huffman or Arithmetic). For processing efficiency, long runs can be filtered with a *run length encoder* (RLE) which replaces them with a <symbol, run-length> pair. As block size is increased, compression ratio improves. Diminishing returns (with English text) do not occur until block size reaches several tens of megabytes. Unlike the other algorithms, one could consider BWT to take advantage of symbols which appear in the *future,* not just those that have passed.

In latency-critical single-threaded applications, the block-based processing of BWT could be a bottleneck. Several distinct operations must be performed in series (transform, move to front, run-length encode, entropy coding) and entire blocks of data must be processed before moving on to the next. Sorting is the critical operation. Although BWT-based compression could be performed in very little memory with in-place sorting, common implementations use fast sort algorithms and/or structures such as the suffix tree which require substantial memory to provide speed.

We use the popular *bzip2* application as a representative Burrows Wheeler Transform code. It reads in blocks of data, using run-length-encoding to improve sort speed. It then applies the BWT and uses a variant of move-to-front coding to produce a compressible stream. Though the alphabet may be large, codes are only created for symbols in use. This stream is run-length encoded to remove any long runs of zeros. Finally Huffman encoding is applied. To speed sorting, *bzip2* applies a modified quicksort which has memory requirements over five times the size of the block.

2.4 Performance and Implementation Concerns

The original Lempel-Ziv-inspired methods have remained popular since their newer competitors require more time and memory to achieve compression. PPM variants have been recognized as the leader in compression ratios since their introduction in 1984, but these ratios come at a tremendous time and memory expense. BWT has grown in popularity because its implementations, based on efficient sorting, lead to greater speed than PPM implementations while giving similar excellent compression ratios. Recently, BWT has been recast as a problem similar to PPM, inspiring PPM programs to exploit advances in BWT implementations. It has taken nearly 20 years for implementations of PPM to approach the speed of the LZ77, LZ78, and BWT methods [Effros 2000; Shkarin 2002a].

A compression algorithm may be implemented with many different, yet reasonable, data structures (including binary tree, splay tree, trie, hash table, and list) and yield vastly different performance results [Bell and Kulp 1989]. The

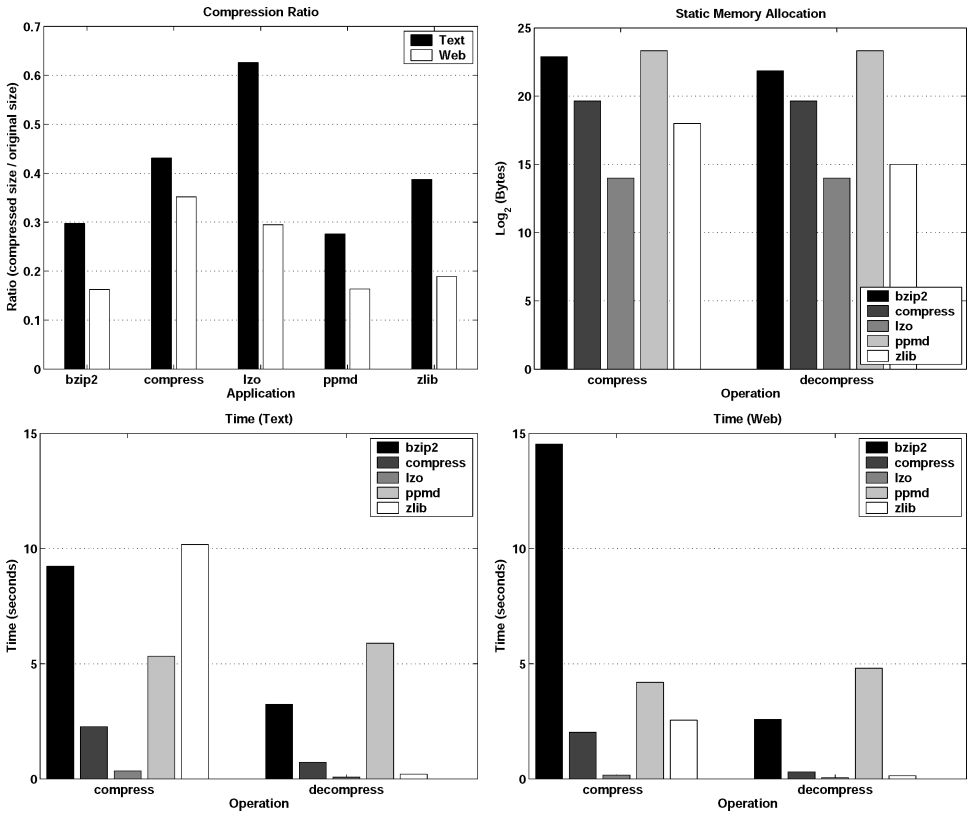

Fig. 3. Benchmark comparison by traditional metrics.

quality and applicability of the implementation is as important as the underlying algorithm. This section has presented example implementations from each algorithmic family. Choosing the top representative in each family is how we level the implementation playing field, making it easier to gain insight into the underlying algorithm and its influence on energy. Nevertheless, software may be continually optimized, as we show in Section 5.1, so our evaluation focuses on inherent patterns in addition to a direct quantitative comparison.

3. EXPERIMENTAL SETUP

This section begins with brief demonstration of the differences among our chosen applications. Next the Skiff platform is introduced along with an explanation of how it can be used to make energy measurements of algorithms. We discuss the error inherent in these measurements, but show that hardware measurements are more accurate than a naive simulation.

3.1 Benchmark Selection

Figure 3 shows the performance of several lossless data compression applications using metrics of compression ratio, execution time, and static memory

Table I. Compression Applications and Their Algorithms

Application Version	Algorithm	Notes Defaults
bzip2 [Seward 1999] 0.1pl2	BWT	RLE→BWT→MTF→RLE→Huffman 900k block size
compress [Jannesen et. al 1996] 4.2.4	LZW	Unix Compress program 16-bit codes (maximum), fast hashing
LZO [Oberhumer 2000] 1.07	LZ77	Favors speed over compression lzo1x_12 (4K entry hash table uses 16 KB)
PPMd [Shkarin 2002b] variant I	PPM	Used in "rar" compressor Order 4, 10 MB memory, restart model
zlib [Gailly and Adler 2002] 1.1.4	LZ77	Library form of gzip Chaining level 6/32 k Window/32 k Hash Table

allocation. The datasets are the first megabyte (English books and a bibliography) from the Calgary Corpus [Bell et al. 1989] and 1 MB of easily compressible Web data (mostly HTML, Javascript, and CSS) obtained from the homepages of the Internet's most popular Web sites [Lycos 2002; Nielsen NetRatings Audience Measurement Service 2002]. Graphics were omitted as they are usually in compressed form already and can be recognized by application-layer software via their file extensions. Most popular repositories for comparison of data compression do not examine the memory footprint required for compression or decompression [Bell et al. 1997; Gailly 1999; Gilchrist 2002]. Though static memory usage may not always reflect the size of the application's working set, it is an essential consideration in mobile computing where memory is a more precious resource. A detailed look at the memory used by each application, and its effect on time, compression ratio, and energy will be presented in Section 4.3.

Figure 3 confirms that the chosen array of applications spans a range of compression ratios and execution times. Each application represents a different family of compression algorithms as noted in Table I. The table includes the default parameters used with each program. To avoid unduly handicapping any algorithm, it is important to work with well-implemented code. Mature applications such as *compress*, *bzip2*, and *zlib* reflect a series of optimizations that have been applied since their introduction. While *PPMd* is an experimental program, it is effectively an optimization of the Prediction by Partial Match (PPM) compressors that came before it. *LZO* represents an approach for achieving great speed with LZ77. Consideration was also given to popularity and documentation, as well as quality, parameterizability, and portability of the source code.

3.2 Methodology

This section describes the hardware platform we used to take energy measurements and the test harness used for running compression programs. The energy measurement methodology is described, along with an analysis of sources of error. Finally, the results of a simulation-based study are

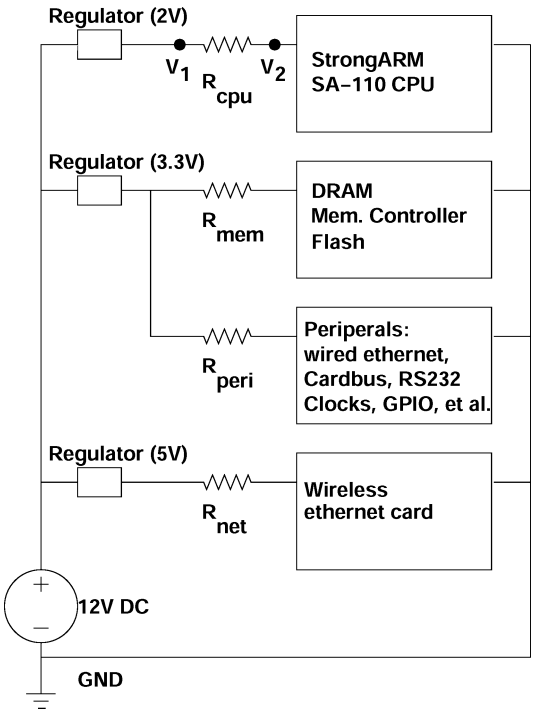

Fig. 4. Simplified Skiff power schematic.

presented which motivate the use of the hardware-only technique for measuring long-running programs.

3.2.1 *Equipment.* The Compaq Personal Server, codenamed *Skiff*, is essentially an initial, "spread-out" version of the Compaq iPAQ built for research purposes [Hicks et al. 1999]. Powered by a 233-MHz StrongARM SA-110 [Montanaro et al. 1996; Intel Corporation 2000], the Skiff is functionally similar to the popular Compaq iPAQ handheld (based on the SA-1110 [Intel Corporation 2001]). For wireless networking, we add a 5-V Enterasys 802.11b wireless network card (part number CSIBD-AA). The Skiff has 32 MB of DRAM, support for the Universal Serial Bus, a RS232 Serial Port, Ethernet, two Cardbus sockets, and a variety of general-purpose I/Os. The Skiff PCB boasts separate power planes for its CPU, memory and memory controller, and other peripherals, allowing each to be measured in isolation (Figure 4). With a Cardbus extender card, one can isolate the power used by a wireless network card as well. A programmable multimeter and sense resistor provide a convenient way to examine energy in a active system with error less than 5% [Viredaz and Wallach 2001].

The Skiff runs ARM/Linux 2.4.2-rmk1-np1-hh2 with PCMCIA Card Services 3.1.24. The Skiff has only 4 MB of nonvolatile flash memory to contain a file system, so the root filesystem is mounted via NFS using the wired Ethernet port. For benchmarks which require file system access, the executable and input dataset is brought into RAM before timing begins. This is verified

by observing the cessation of traffic on the network once the program completes loading. I/O is conducted in memory using a modified SPEC harness [Standard Performance Evaluation Corporation 2000] to avoid the large cost of accessing the network filesystem.

3.2.2 *Energy Calculations.* To minimize resource contention and the effect of context-switching, all unnecessary user-level programs are stopped, leaving only kernel threads. No modular kernel drivers are present. The application under test is placed in an infinite loop, and a digital multimeter is used to statistically sample the supply voltage (e.g., the voltage supplied to the CPU in Figure 4 is $V_{cpu} = V_2 - GND$). In a subsequent experiment, the application runs in a loop while current is determined by measuring the voltage across the known sense resistance ($I_{cpu} = \frac{V_1 - V_2}{R_{cpu}}$). The multimeter internally acquires the average of 285 samples over the course of 6.5 s, sending five such acquisitions back to the host PC. The five acquisitions are averaged. This measured voltage and current comprise average power as $P_{cpu} = I_{cpu} V_{cpu}$. The multimeter also reports the maximum and minimum observed voltages which can be used to bound the error (Section 3.2.3). Network energy was measured in a one-time experiment (Section 4.1). This leaves six distinct measurements which must be made per experiment to obtain the total nonnetwork system power ($P = P_{cpu} + P_{mem} + P_{peripheral}$).

To compute total energy, the running time of each application is required. The application is run n times in a row, and total time is measured with the Skiff's real-time clock. Applications which complete quickly must be run for large n to minimize timing error and eliminate one-time effects. The duration, t, is $(t_1 + t_2 + \cdots + t_n)/n$ where $\{t_i | i = 1..n\}$ is the set of times for each individual run of the application. We can now calculate the energy of the application as $Energy = P \times t$.

3.2.3 *Error Analysis.* This method of measurement involves two sources of error: hardware and averaging. Hardware error may effect the measured value of the sense resistor as well as any voltages that are measured. While the precision sense resistors on the Skiff board have a tolerance of 1%, short leads ($\approx$3 in) must be soldered to the board so that the multimeter may be attached. The resistance, rated at 0.20 Ω, increased to as much as 0.48 Ω with the addition of the leads and solder. This resistance is measured using the four-wire ohmmeter capability of the multimeter as it is most accurate for low resistances. Resistance measured by the multimeter includes error stated as a percentage of reading and percentage of the 100 -Ω range [Agilent Technologies 2000].

Voltage measurement error takes a similar form, consisting of error in reading (dependent on the input level) and an error inherent to the range. Integration time of an analog-to-digital converter, the time required to charge or discharge its capacitor, affects its sample rate. By choosing an integration period matched to the period of the wall supply's alternating current, the integration noise is mimimized. Operating with 60 integrations/s, we add a small additional noise error. The sample rate could be made faster, but this would

increase the integration error to as much as 0.12 mV per volt or 0.03 mV to a measured millivolt due to poorer noise reduction. To increase sampling speed, the multimeter's auto-zero functionality is turned off. We allow the multimeter to warm up to reduce auto-zero error due to temperature variation.

Hardware error also includes the problem that the Skiff is not observed continuously, only during an analog-to-digital integration cycle. Since the clock period of the Skiff is much shorter than the sample period and overhead time of the multimeter, many cycles may pass in between measurements. We rely on the uniform, repetitive nature of each application combined with several 6.5-s acquisition periods to increase the probability of observing all parts of the application. Repeated acquisitions are especially important for the few applications which take greater than 6.5 s to complete. Each acquisition period is separated by a upload to the host computer. This upload takes a varying amount of time which prevents the acquisitions from being synchronized with the application being measured. These effects decrease the probability that important events are going unobserved. Observing *too much* energy is another source of hardware error. For example, the Skiff's wired Ethernet controller is enabled for the duration of the benchmark even though the network is not required. Inability to isolate such components leads to an inflated peripheral energy. While this energy is indeed consumed on the Skiff, it tells us little about the compression algorithm itself.

Since the methodology involves the averaging of discrete voltage samples within the multimeter and multiplying them by the average of another set of current samples, we can only estimate the true average power over a particular integration period. The formula for maximum error due to combination of uncorrelated samples is stated in Viredaz and Wallach [2001] (Equation 12). It is derived from the number of samples and the maximum and minimum observed voltages.

System level effects (e.g., broadcast network traffic and OS maintenance tasks) can vary the runtime of an application. Thus, each application is run multiple times in a loop amortizing any timing error across each iteration. The hardware timer granularity is about 20 ns, but software rounds off times to the nearest microsecond. Nevertheless, looped applications run on the order of seconds, so any error in timing is negligible. It should be noted that the "real time clock" of the Skiff is not real time at all since it runs at 48 MHz while Linux treats it as a 50-MHz clock. Thus, the Skiff overestimates the number of seconds in a wall-clock minute. This only effects absolute timing and is constant across all experiments; thus comparisons between applications are unaffected.

Energy error is comprised of the product of current, voltage, and time, so the total error for an acquisition is the sum of each component's relative error. By this method, the experiments that follow have energy measurement error less than 1%, as shown in Tables II and III. The applications with the highest margin of error are hurt mostly by the error-in-averaging component of total error. The CPU is most affected since it draws the least current; any change in current causes a large relative change. In addition, the CPU clocks more slowly while waiting for reads from memory, so applications which alternate

Table II. Maximum Measurement Error: Compression

	CPU (percent)	Memory (percent)	Peripheral (percent)
bzip2	0.36	0.10	0.11
compress	0.31	0.09	0.06
lzo	0.15	0.09	0.06
PPMd	0.18	0.09	0.07
zlib	0.60	0.09	0.12

Table III. Maximum Measurement Error: Decompression

	CPU (percent)	Memory (percent)	Peripheral (percent)
bzip2	0.53	0.10	0.13
compress	0.28	0.09	0.08
lzo	0.13	0.09	0.06
PPMd	0.19	0.10	0.08
zlib	0.12	0.10	0.06

memory access with computation have less uniform power profiles, increasing the error due to averaging. Network card energy error is omitted from the tables, but can be expected to be very small as the network energy benchmark is very uniform; in addition, a larger sense resistance is used which decreases the voltage measurement error.

3.2.4 *Simulation.* One popular approach for modeling power in software is to multiply event counts generated by an architectural simulator by the power they consume (as measured offline on real hardware). Unfortunately, such simulators are highly tied to the hardware on which they were calibrated, and they can require a very detailed architectural model to be of any use. To examine the feasibility of a simulator-based approach, we used the SimpleScalar execution-driven simulator [Burger and Austin 1997]. Though SimpleScalar is inherently an out-of-order, superscalar simulator, it has been modified to read statically linked ARM binaries and model the five-stage, in-order pipeline of the SA-110x. The modified simulator reports IPC within 4% of real hardware [Austin and Burger 2001]. No attempt was made to verify cycle counts produced by the simulator against the Skiff, as SimpleScalar relies on the host as a proxy for OS calls. As such, cycle counts are underestimated by the simulator. Using Skiff execution time as baseline, the simulated cycle count is off by a factor of $1.8\times - 2.3\times$ depending on the benchmark. Events such as taken branches and cache hits are more closely related to the instructions executed and the layout of the cache and may be used more reliably.

We predicted the energy that would be consumed by several applications by multiplying event counts generated by the simulator with the actual measured CPU and memory energy of operations. Events were grouped into the following classes: computation, load hit, load miss, store hit, buffered store miss, unbuffered store miss, and network. The number of instruction misses was negligible for all five compressors so we do not attempt to quantify instruction cache energy. The energy-per-operation is measured as in Section 4.1 and 4.3.2.

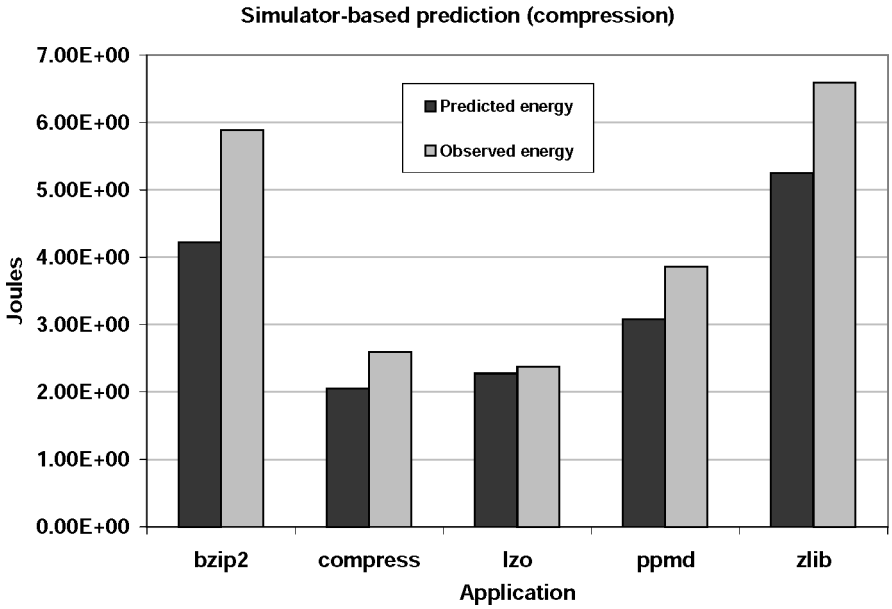

Fig. 5. Using a simulator to predict energy.

The following formula was used:

$$
\begin{aligned}
E_{predicted} = {} & E_{compute}(\text{computes} + \text{predicted branches} + 2 * \text{mispredicted branches}) \\
& + E_{load\ hit} * \text{load hits} \\
& + E_{load\ miss} * \text{load misses} \\
& + E_{writeback} * \text{writebacks} \\
& + E_{store\ hit} * \text{store hits} \\
& + E_{store\ miss\ combined\ in\ write\ buffer\ (estimated)} * \text{store misses near} \\
& + E_{uncombined\ store\ miss} * \text{store misses far} \\
& + E_{send\ a\ bit} * \text{bits sent.}
\end{aligned}
$$

The factor of 2 represents the extra cycle required by a mispredicted branch. *Near* and *far* refer to the spatial locality of stores; near misses are those which should be found in the combining store buffer, while far addresses are spatially restricted from being present in the buffer. We approximate these near and far counts by instrumenting loads and stores in the simulator. Figure 5 shows that the difference between observed energy (the sum of memory and CPU energy) and predicted energy varies from about 4% for the simple, fast *LZO* compressor to 28% for the slow, memory-intensive *bzip2*. Adding 1.12 nJ, per executed instruction to account for bias in the model (e.g., unsimulated operating system instructions) lowers the difference between observed and predicted energy to 0.0%–18%. This bias amount was chosen to completely remove the error from one application while minimizing the error of the application with the largest error.

We see that generalizing a system's operations can lead to inaccuracy. For example, the Skiff has a relatively simple datapath, but its unique memory hierarchy is not accurately modeled by the simulator. Measurement error, simulator inaccuracies, and error due to generalization of instruction classes are compounded over long periods of time to produce significant error. For short programs, however, the ≈4% error in Figure 5 is close to other studies of small programs [Sinha and Chandrakasan 2001]. As discussed in Section 3.2.3, energy measured with hardware may be inaccurate as well, but has the advantage of corresponding more closely to reality.

Executing programs on a simulator requires more time than running them on hardware. Hosted on a 1.2-GHz Athlon, the simulator operates around 500 kHz, 467 times slower than running applications on the Skiff. However, the current experimental setup in the hardware laboratory requires six distinct measurements per application, each taking roughly 30 s (not including the time it takes to rearrange the multimeter probes). Whether or not to use a simulator is dependent on desired accuracy and the relative convenience of obtaining actual measurements. For greatest accuracy, pure energy measurements were made exclusively with hardware. While simulators may be tuned to provide reasonably accurate estimations of a particular system's energy, observing real hardware ensures that complex interactions of components are not overlooked or oversimplified.

4. OBSERVED ENERGY OF COMMUNICATION, COMPUTATION, AND COMPRESSION

In this section, we observe that over 1000 32-bit ADD instructions can be executed by the Skiff with the same amount of energy it requires to send a single bit via wireless Ethernet. This fact motivates the investigation of pretransmission lossless compression of data to reduce overall energy. However, initial experiments revealed that reducing the number of bits to send does not always reduce the total energy of the task. This section elaborates on both of these points, which necessitate the in-depth experiments of Section 4.3. After examining the performance of common lossless compression schemes, we derive guidelines to minimize the energy consumption of compressed data transmission.

4.1 Raw Communication-to-Computation Energy Ratio

To quantify the smallest gap between wireless communication and computation, we measured 802.11b wireless idle, send, and receive energies on the Skiff platform. To eliminate competition for wireless bandwidth from other devices in the lab, we established a dedicated channel and ran the network in ad hoc mode consisting of only two wireless nodes. UDP packets were streamed from one node to the other; UDP was used to eliminate the effects of waiting for a transport layer ACK. This also ensures that receive tests measure only receive energy and send tests measure only send energy at the transport layer and above. Clear-to-send/ready-to-send is disabled, but it is unknown whether media access control (MAC-layer) retransmissions are enabled; our wireless drivers do not support the retry setting. This setup is intended to mimimize the impact of components of wireless communication that we can control so

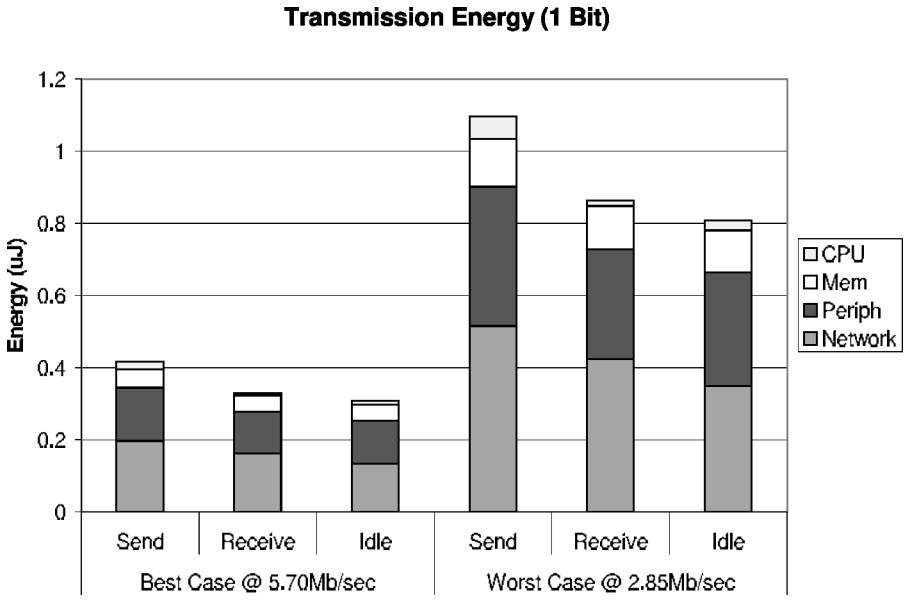

Fig. 6. Measured communication energy of Enterasys wireless NIC.

that we do not artificially exaggerate the cost of communication. In fact, others have used detailed power simulations to show that TCP and UDP have similar energy profiles in both low-noise and high-noise environments [Banerjee and Misra 2004].

With the measured energy of the transmission and the size of data file, the energy required to send or receive a bit can be derived. The results of these network benchmarks appear in Figure 6 and are consistent with other studies [Jamieson 2002]. The card was set to its maximum speed of 11 Mb/s and two tests were conducted. In the first, the Skiff communicated with a wireless card mere inches away and achieved 5.70 Mb/s. In the second, the second node was placed as far from the Skiff as possible without losing packets. Only 2.85 Mb/s was achieved. These two cases bounded the performance of our 11-Mb/s wireless card; typical performance should be somewhere in between. For each of the two transfer rates, we showed sending a bit requires more energy than receiving one. The energy required for the platform to sit idle for 1 bit's worth of transmission time is shown as well to highlight the marginal cost of communication. Recall that our platform allowed us to separate the cost of four components of our system. In Figure 6, it is mostly the network card's energy variation that leads to the total energy variation.

Next, a microbenchmark was used to determine the minimum energy for an ADD instruction. Drawing on earlier work [Nathuji 2000], we used Linux boot code to bootstrap the processor; select a cache configuration; and launch assembly code unencumbered by an operating system. One thousand ADD instructions were placed in a loop body closed by a single unconditional branch. Once the program had been loaded into instruction cache, the energy used by the processor for a single add was 0.86 nJ. From these initial network and ADD

Table IV. Total Energy of an ADD

Network card	0.43 nJ
CPU	**0.86 nJ**
Mem	1.10 nJ
Periph	4.20 nJ
Total	6.59 nJ

measurements, we can conclude that sending a single bit is roughly equivalent to performing 485–1267 ADD operations depending on the quality of the network link ($\frac{4.17\times10^{-7}\,\mathrm{J}}{0.86\times10^{-9}\,\mathrm{J}} \approx 485$ or $\frac{1.09\times10^{-6}\,\mathrm{J}}{0.86\times10^{-9}\,\mathrm{J}} \approx 1267$). This gap of two to three orders of magnitude, even with limited wireless communication use, suggests that much additional effort can be spent trying to reduce a file's size before it is sent or received. But the issue is not so simple.

4.2 Application-Level Communication-to-Computation Energy Ratio

On the Skiff platform, memory, peripherals, and the network card remain powered on even when they are not active, consuming a fixed energy overhead. They may even switch when not in use in response to changes on shared buses. The energy used by these components during the ADD loop is significant and is shown in Table IV. Once a task-switching operating system is loaded and other applications vie for processing time, the communication-to-computation energy ratio will decrease further. Finally, the applications we examined contained more than a series of ADDs; the variety of instructions (especially Loads and Stores) in compression applications shrank the ratio further.

The first rows of Figures 7 and 8 show the energy required to compress our text and Web dataset and transmit it via wireless Ethernet. To avoid punishing the benchmarks for the Skiff's high idle power, idle energy has been removed from the peripheral component so that it represents only the amount of *additional* energy (due to bus toggling and arbitration effects) over and above the energy that would have been consumed by the peripherals remaining idle for the duration of the application. Idle energy is not removed from the memory and CPU portions as they are required to be active for the duration of the application. The network is assumed to consume no power until it is turned on to send or receive data, and all data is sent or received in a single batch. The popular compression applications discussed in Section 3.1 are used with their default parameters, and the right-most bar shows the energy of merely copying the uncompressed data over the network. Along with energy due to default operation (labeled *bzip2-900*, *compress-16*, *lzo-16*, *ppmd-10240*, and *zlib-6*), the figures include energy for several invocations of each application with varying parameters. *bzip2* is run with both the default 900 KB block sizes as well as its smallest 100 KB block. *compress* is also run at both ends of its spectrum (12-bit and 16-bit maximum codeword size). *LZO* runs in just 16 KB of working memory. *PPMd* uses 10 MB, 1 MB, and 32 KB memory with the cutoff mechanism for freeing space (as it is faster than the default "restart" in low-memory configurations). *zlib* is run in a configuration similar to *gzip*. The numeric suffix (9, 6, or 1) refers to effort level and is analogous to *gzip*'s commandline option. These various invocations will be studied in Section 4.3.3.

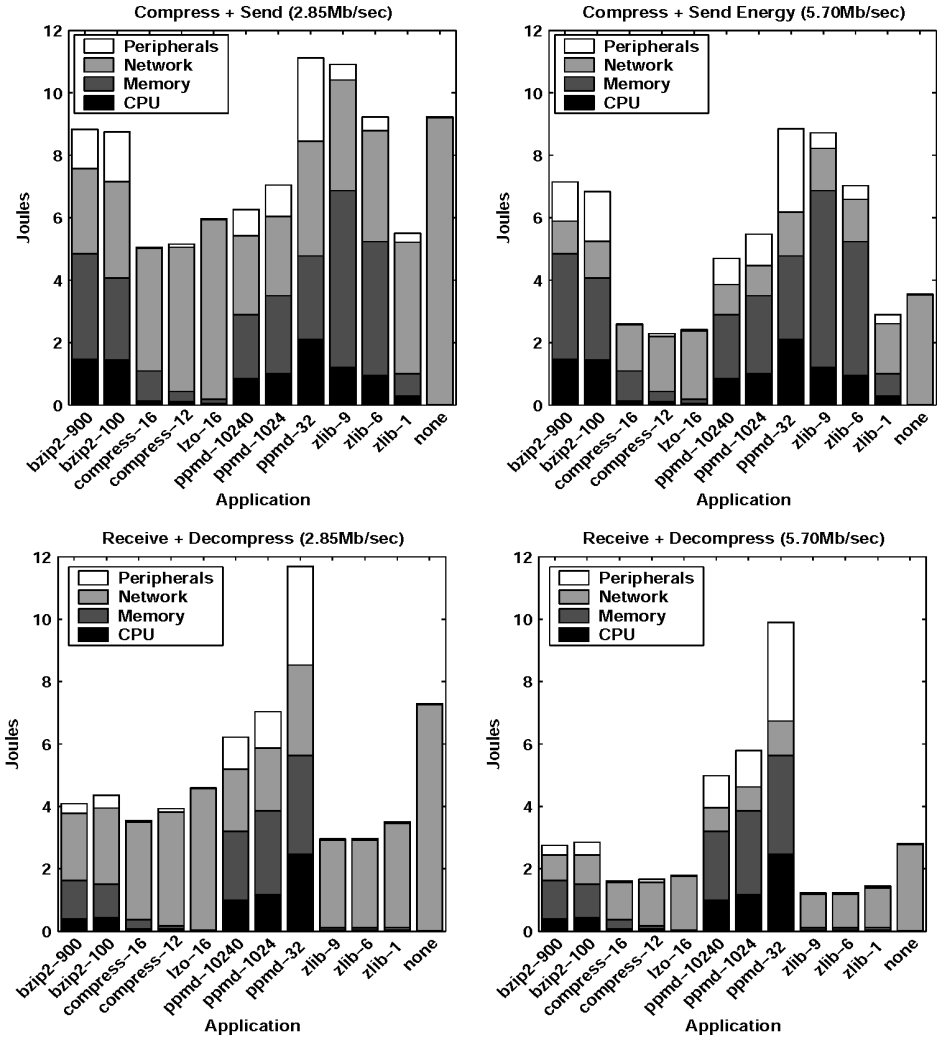

Fig. 7. Energy required to transmit 1 MB of compressible text data.

While most compressors do well with the Web data, in several cases the energy to compress the file approaches or outweighs the energy to transmit it. This problem is even worse for the harder-to-compress text data. The second rows of Figures 7 and 8 show the reverse operation: receiving data via wireless Ethernet and decompressing it. The decompression operation is usually less costly than compression in terms of energy, a fact which will be helpful in choosing a low-energy, asymmetric, lossless compression scheme. Section 4.3 will discuss how such high net energy is possible despite the motivating observations.

4.3 Energy Analysis of Popular Compressors

We will look deeper into the applications to discover why they cannot exploit the communication-computation energy gap. To perform this analysis, we rely

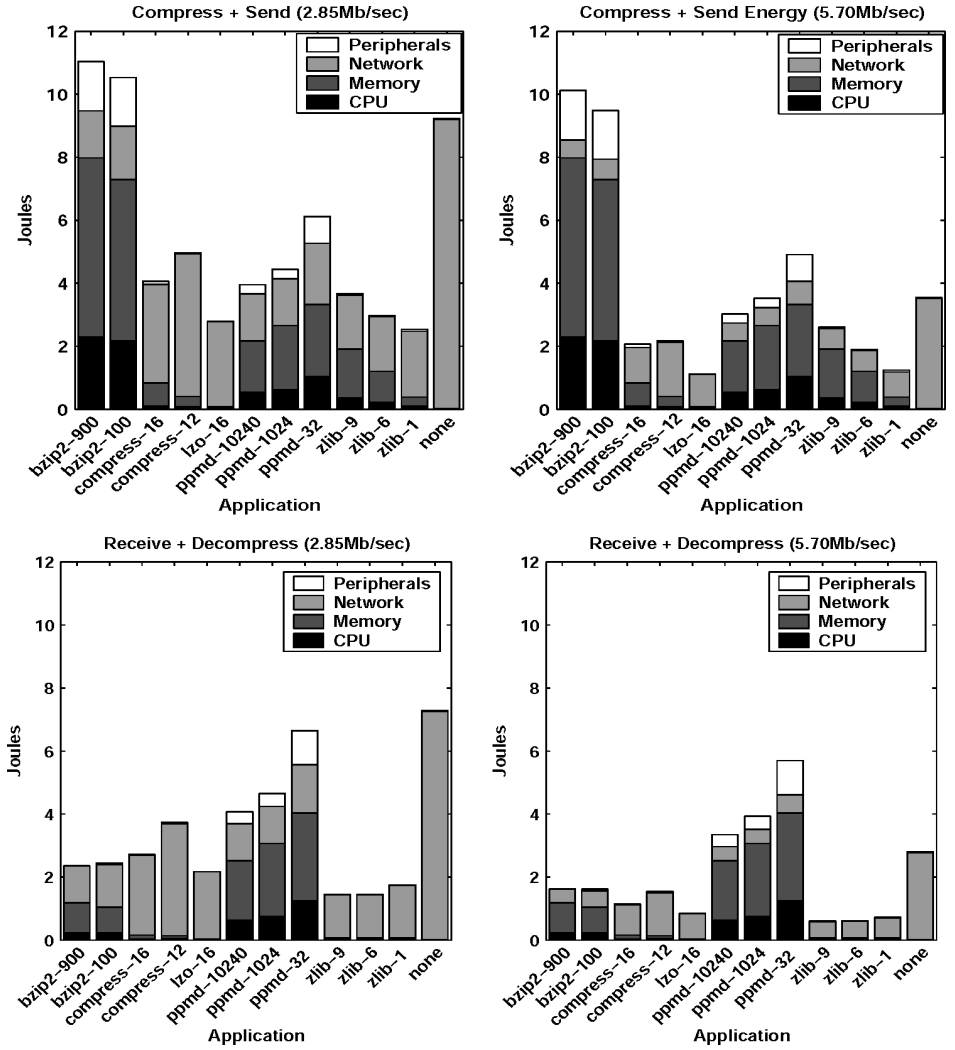

Fig. 8. Energy required to transmit 1 MB of compressible web data.

on empirical observations on the Skiff platform as well as the SimpleScalar simulator. Since this version of SimpleScalar is beta software we will handle the statistics it reports with caution, using them to explain the *traits* of the compression applications rather than to describe their precise execution on a Skiff. Namely, high instruction counts and high cost of memory access lead to poor energy efficiency.

4.3.1 *Instruction Count.* We begin by looking at the number of instructions each requires to remove and restore a bit (Table V). The range of instruction counts is one empirical indication of the applications' varying complexity. The excellent performance of *LZO* is due in part to its implementation as a single function; thus there is no function call overhead. In addition, *LZO* avoids

Table V. Instructions per Bit

	bzip2	compress	LZO	PPMd	zlib
Compress: instructions per bit removed (Text)	116	10	7	76	74
Decompress: instructions per bit restored (Text)	31	6	2	10	5
Compress: instructions per bit removed (Web)	284	9	2	60	23
Decompress: instructions per bit restored (Web)	20	5	1	79	3

superfluous copying due to buffering (in contrast with *compress* and *zlib*). As we will see, the number of memory accesses plays a large role in determining the speed and energy of an application. Each program contains roughly the same percentage of loads and stores, but the great difference in dynamic number of instructions means that programs such as *bzip2* and *PPMd* (each executing over 1 billion instructions) execute more total instructions and therefore have the most memory traffic.

4.3.2 *Memory Hierarchy*. One noticeable similarity of the bars in Figures 7 and 8 is that the memory requires more energy than the processor. To pinpoint the reason for this, microbenchmarks were run on the Skiff memory system. The microbenchmarks fit in the instruction cache.

The SA-110 data cache is 16 KB. It has 32-way associativity and 16 sets. Each block is 32 bytes. Data is evicted at half-block granularity and moves to a 16-entry $\times$ 16-byte write buffer. The write buffer also collects stores that miss in the cache (the cache is writeback/non-write-allocate). The store buffer can merge stores to the same entry.

The hit benchmark accesses the same location in memory in an infinite loop. The miss benchmark consecutively accesses the entire cache with a 32-byte stride followed by the same access pattern offset by 16 KB. Writebacks are measured with a similar pattern, but each load is followed by a store to the same location that dirties the block forcing a writeback the next time that location is read. Store hit energy is subtracted from the writeback energy. The output of the compiler is examined to ensure the correct number of load or store instructions is generated. Address generation instructions are ignored for miss benchmarks as their energy is minimal compared to that of a memory access. When measuring store misses in this fashion (with a 32-byte stride), the worse-case behavior of the SA-110's store buffer is exposed as no writes can be combined. In the best case, misses to the same buffered region can have energy similar to a store hit, but in practice, the majority of store misses for the compression applications are unable to take advantage of merging writes in the store buffer.

Table VI shows the energy of the CPU and off-chip memory for various operations. Hitting in the data cache requires energy similar to an ADD, but cache misses require up to 63 times the energy of an ADD. Store misses are less expensive as the SA-110 has a store buffer to batch accesses to memory. To minimize energy, then, we must seek to minimize cache misses, which require prolonged access to higher voltage components.

4.3.3 *Minimizing Memory Access Energy*. One way to minimize misses is to reduce the memory requirements of the application. Figure 9 shows the effect

Table VI. Measured Memory Energy Versus ADD Energy

	Cycles	Energy (nJ)
Load hit	1	2.72
Load miss	80	124.89
Writeback	107	180.53
Store hit	1	2.41
Store miss	33	78.34
ADD	1	1.96

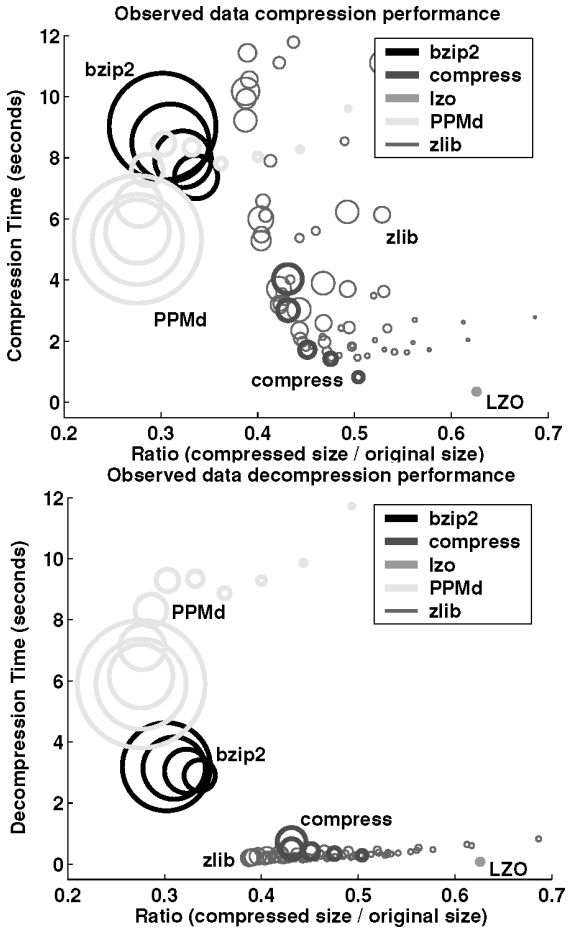

Fig. 9. Memory, time, and ratio (Text data). Memory footprint is indicated by area of circle; footprints shown range from 3 KB to 8 MB.

of varying memory size on compression/decompression time and compression ratio. Looking back at Figures 7 and 8, we see the energy implications of choosing the right amount of memory. Most importantly, we see that merely choosing the fastest or best-compressing application does not result in lowest overall energy.

In the case of *compress* and *bzip2*, a larger memory footprint stores more information about the data and can be used to improve compression ratio. However, storing more information means less of the data fits in the cache leading to more misses, longer runtime and hence more energy. This tradeoff need not apply in the case where more memory allows a more efficient data structure or algorithm. For example, *bzip2* uses a large amount of memory, but for good reason. When we replaced the provided sort routine with the standard C library quicksort routine, we saved significant memory but compression took over 2.5 times longer due to large constants in the runtime of the library quicksort. This slowdown occurred even when 16-KB block sizes [Seward 2000] were used to further reduce memory requirements.

PPMd has three performance regions. Without enough memory, it has no room to model source data and is inefficient. Adding memory buys both improved speed and greater compression because the work became more productive, but there were points at which adding memory had the expected effect of slowing down compression as a deeper tree had to be examined. Finally, compression ratio approached a limit, and additional memory served to improve speed. This behavior was due to the complexity in handling escapes (situations in which the symbol has not been seen in the current context). With more memory, more context information can be stored and less complicated escape handling is necessary.

The widely scattered performance of *zlib*, even with similar footprints, suggest that one must be careful in choosing parameters for this library to achieve the desired goal (speed or compression ratio). Increasing window size affects compression; for a given window, a larger hash table improves speed. Thus, the net effect of more memory is variable. The choice is especially important if memory is constrained as certain window/memory combinations are inefficient for a particular speed or ratio.

The decompression half of the figure underscores the valuable asymmetry of some of the applications. Often decompressing data is a simpler operation than compression and requires less memory (as in *bzip2* and *zlib*). The simple task requires a relatively constant amount of time as there is less work to do: no sorting for *bzip2* and no searching though a history buffer for *zlib*, *LZO*, and *compress* since all the information to decompress a file is explicit. The contrast between compression and decompression for *zlib* is especially large. PPM implementations must go through the same procedure to decompress a file, undoing the arithmetic coding and building a model to keep probability counts in sync with the compressor's. The arithmetic coder/decoder used in *PPMd* requires more time to perform the decode operation than the encode.

Each of the applications examined allocates fixed-size structures regardless of the input data length. Thus, in several cases more memory is set aside than is actually required. However, a large memory footprint may not be detrimental to an application if its current working set fits in the cache. The simulator was used to gather cache statistics. PPM and BWT were known to be quite memory intensive. Indeed, *PPMd* and *bzip2* accessed the data cache on to two orders of magnitude more often than the other benchmarks. *zlib* accessed data cache almost as much as *PPMd* and *bzip2* during compression, but dropped from

Table VII. Application Throughputs (Mb/s)

	bzip2	compress	LZO	PPMd	zlib
Compress read throughput (Text data)	0.91	3.70	24.22	1.57	0.82
Decompress write throughput (Text data)	2.59	11.65	109.44	1.42	41.15
Compress read throughput (Web data)	0.58	4.15	50.05	2.00	3.29
Decompress write throughput (Web data)	3.25	27.43	150.70	1.75	61.29

150 million accesses to 8.2 million during decompression. Though LZ77 is local by nature, the large window and data structures hurt its cache performance for *zlib* during the compression phase. *LZO* also uses LZ77, but is designed to require just 16 KB of memory and went to main memory over five times less often than the next fastest application. The followup to the SA-110 (the SA-1110 used in Compaq's iPAQ handheld computer) has only an 8 KB data cache, which would exaggerate any penalties observed here. Though large, low-power caches are becoming possible (the X-Scale has two 32-KB caches), as long as the energy of going to main memory remains so much higher, we must be concerned with cache misses.

As an aside, while Figure 9 plots speed, Table VII notes the absolute throughput of each application. We see that, with the Skiff's processor, several applications have difficulty meeting the line rate of the network which may preclude their use in latency-critical applications. A faster processor would permit more choices.

4.3.4 *Instruction Mix.* One rough characterization of any application is its instruction mix. Figure 10 shows the static and dynamic instruction mix of the five compression applications. In the figure, "compute" includes ALU, Logical, Compare, and register transfer operations. The "other" set of instructions, which is negligible, includes software traps, and (in the static image) floating point operations in library code. The absolute number of instructions is shown in parenthesis below the graph. Static instructions include both the those for compression and decompression as they are commonly contained in the same program.

As we have seen, the number of memory accesses plays a large role in determining the speed and energy of an application. Each program contains roughly the same percentage of loads and stores, but the great difference in dynamic number of instructions means that programs such as *bzip2* and *PPMd* (each executing over 1 billion instructions) execute more total instructions and therefore have the most memory traffic. During compression, the LZ-based schemes and *PPMd* involve mostly searching and thus execute more loads than stores. *bzip2*'s has a slightly greater percentage of stores and it is based on sorting.

Branches are not a large cause of stalls in the StrongARM's short pipeline, but the static not-taken prediction scheme is rather poor (Figure 11). As embedded processor pipelines grow, effective branch prediction will be needed to minimize the number of flushes that must occur.

4.4 Summary

On the Skiff, compression and decompression energy are roughly proportional to execution time. Figure 12 shows that CPU and memory power is

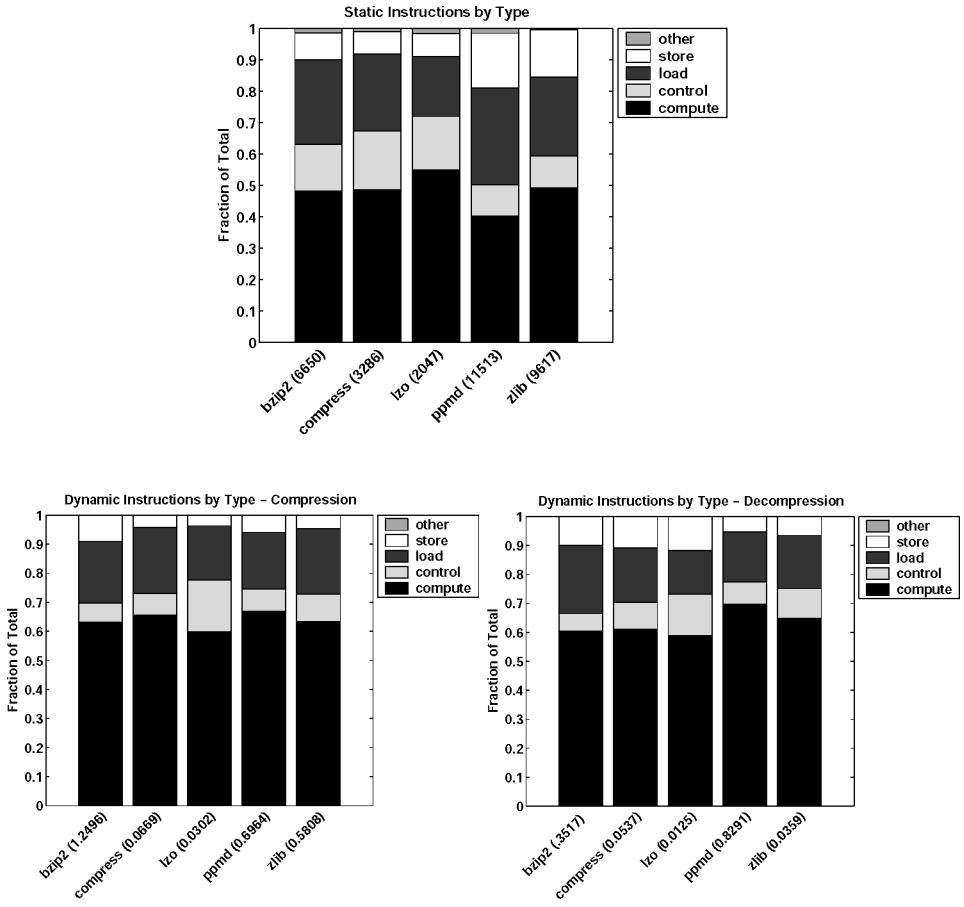

Fig. 10. Instruction mix. Number in parenthesis shows absolute number of instructions (static) and billions of absolute instructions (dynamic).

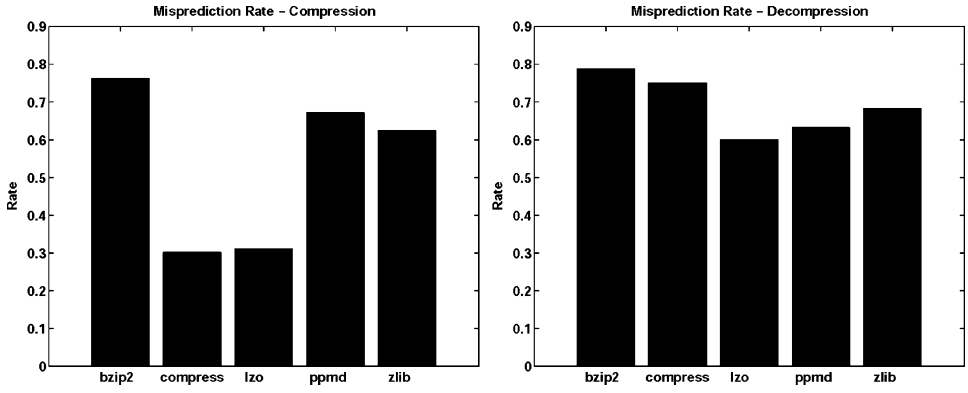

Fig. 11. Branch behavior.

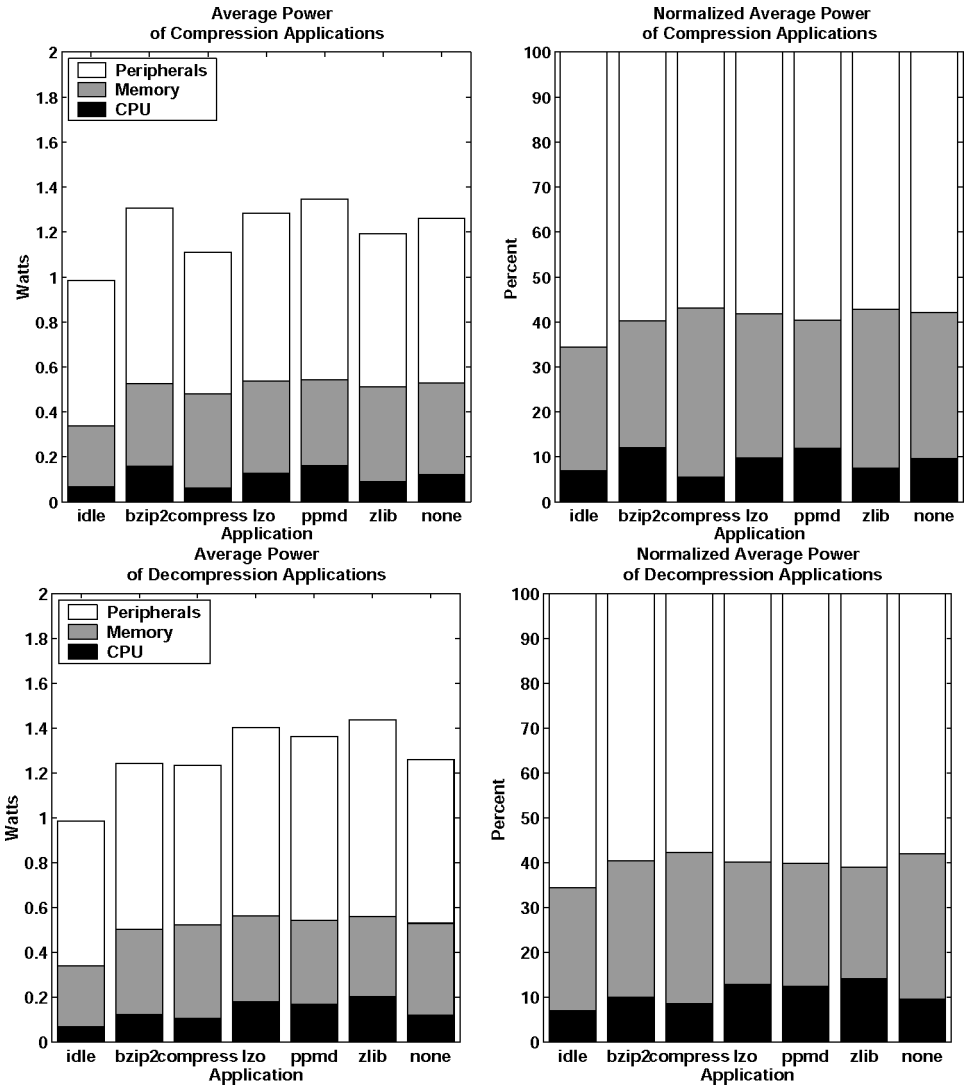

Fig. 12. Average power of compression and decompression applications.

relatively constant for each application, making energy largely time-dependent, though the power is divided in different ways among CPU, memory, and peripherals.

Table VIII ranks our compression applications on a 5.70-Mb/s system with more desirable attributes (smaller files, faster speeds, lower memory, and lower energy) at the top of each column. When we neglect to consider the power and time costs of sending the data, energy perfectly mirrors speed. However, we have shown that there is a delicate balance between computation energy (including memory energy) and communication energy. While we have access to impressive lossless compression algorithms, the communication reduction they provide

Table VIII. Ranking Compression Applications by Four Metrics

	Compress	Decompress	Compress	Decompress	Compress	Decompress
File Size	**Speed**		**Memory**		**Energy**	
ppmd	none	none	none	none	none	none
bzip2	lzo	lzo	lzo	lzo	lzo	lzo
zlib	compress	zlib	zlib	zlib	compress	zlib
compress	ppmd	compress	compress	compress	ppmd	compress
lzo	bzip2	bzip2	bzip2	bzip2	bzip2	bzip2
none	zlib	ppmd	ppmd	ppmd	zlib	ppmd

Table IX. Ranking Total Time (Computation plus Communication Time)

Compress	Decompress
lzo	zlib
none	lzo
compress	compress
ppmd	none
bzip2	bzip2
zlib	ppmd

Table X. Ranking Energy of Compression Applications Including Network Energy

Compress	Decompress
lzo	zlib
compress	compress
none	lzo
ppmd	bzip2
zlib	none
bzip2	ppmd

is not always helpful given high computation cost. Likewise, merely choosing the fastest-running compressor or decompressor does not necessarily minimize *total* transmission energy. Even were we to include network transmission time (Table IX), the speed ranking does not match the total energy ranking due to the different power of each system component.

Table X reorders the applications in terms of *total* energy cost of the compression and transmission task. We see that during decompression both *zlib* and *compress* run slower than *LZO*, but they receive fewer bits due to better compression so total energy is less than *LZO*. These applications successfully balance computation versus communication cost. Despite the greater energy needed to decompress the data, the decrease in receive energy makes the net operation a win. More importantly, we have shown that reducing energy is not as simple as choosing the fastest or best-compressing program or even compressing the data at all.

We can generalize the results obtained on the Skiff in the following fashion. Memory energy is some multiple of CPU energy. Network energy (send and receive) is a far greater multiple of CPU energy. It is difficult to predict how

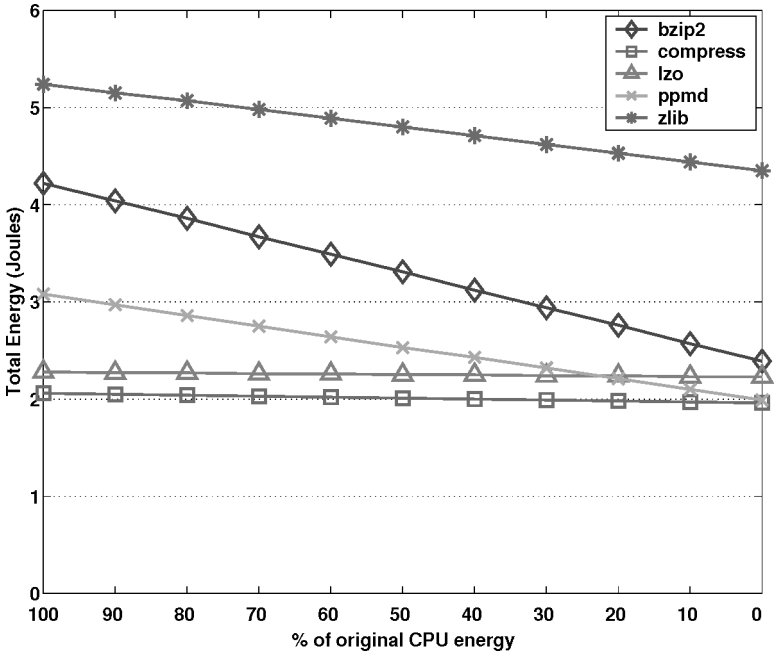

Fig. 13. Total energy as CPU energy decreases.

quickly energy of components will change over time. Even predicting whether a certain component's relative energy usage will grow or shrink can be difficult. Many researchers envision ad hoc networks made of nearby nodes [Chang and Tassiulas 2000; Hohlt et al. 2004]. Such a topology, in which only short-distance wireless communication is necessary, could reduce the energy of the network interface relative to the CPU and memory. On the other hand, for a given mobile CPU design, planned manufacturing improvements may lower power and energy relative to wireless technology with fixed, physical lower bounds. This represents the current trend in "smartphones": consumers want smaller devices that can perform more impressive computational tasks. In order to stay cool in a user's pocket and have acceptable battery life, mobile devices must stay within a fixed or shrinking power budget and find architectural ways to provide more computation [Hicks 2005].

Figures 13 through 15 show the effect on overall compression energy as the ratio between component energies vary. The graphs are produced by multiplying the compute, memory, and network energy (measured by microbenchmarks in Section 4.3.2) with simulated event counts as in Section 3.2.4. The leftmost point of each graph represents the energy of the Skiff platform. Moving left to right, we reduce the energy of one or more components until its contribution disappears.

If one subscribes to the belief that CPU and/or memory energy will steadily decrease while network energy remains constant, then current low-energy compression tools (*compress* and *LZO*) face competition from their computation and memory intensive peers (Figures 13 and 14). However, if only network energy

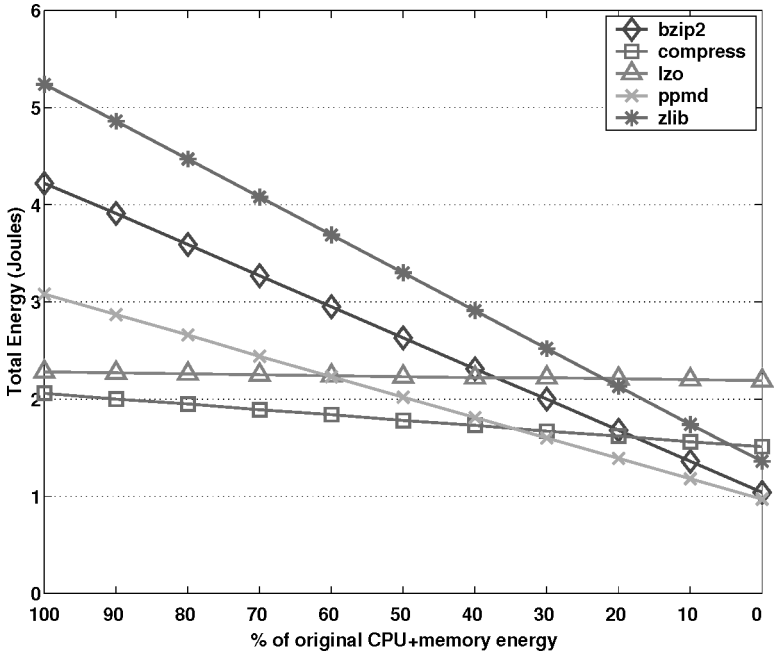

Fig. 14. Total energy as both CPU and memory energy decreases.

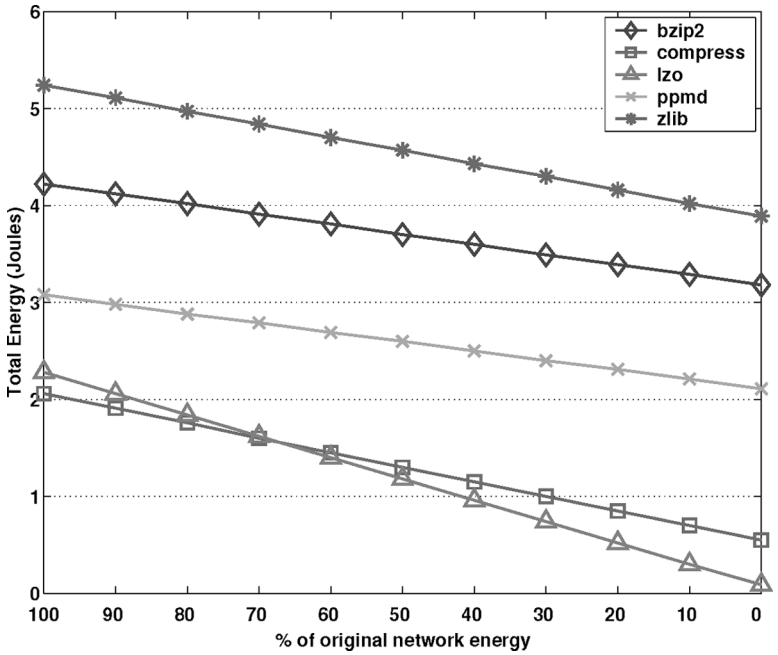

Fig. 15. Total energy as network energy decreases.

decreases while the CPU and memory systems remain static, energy-conscious systems may forgo compression altogether as it now requires more energy than transmitting raw data (Figure 15). Thus, it is important for software developers to be aware of such hardware effects if they wish to keep compression energy as low as possible. Awareness of the type of data to be transmitted is important as well. For example, transmitting our World Wide Web data required less energy in general than the Text data. Trying to compress precompressed data (not shown) requires significantly more energy and is usually futile.

Finally, we have seen that as transmission speed increases, the value of reducing wireless energy through data compression is less. Thus, even when compressing and sending data appears to require the same energy as sending uncompressed data, it is beneficial to apply compression for the greater good: more shared bandwidth will be available to all devices, allowing them to send data faster and with less energy.

5. REDUCING TRANSMISSION ENERGY WITH ENERGY-AWARE LOSSLESS DATA COMPRESSION

We have seen that energy can be saved by compressing files before transmitting them over the network, but one must be mindful of the energy required by this compression. Compression and decompression energy may be minimized through wise use of memory (including efficient data structures and/or sacrificing compression ratio for cacheability). One must be aware of evolving hardware's effect on overall energy. Finally, knowledge of compression and decompression energy for a given system permits the use of asymmetric compression in which the lowest-energy application for compression is paired with the lowest-energy application for decompression.

5.1 Understanding Cache Behavior

Figure 16 shows the compression energy of several successive optimizations of the *compress* program. The baseline implementation is itself an optimization of the original *compress* code. The number preceding the dash refers to the maximum length of codewords. The graph illustrates the need to be aware of the cache behavior of an application in order to minimize energy. The data structure of *compress* consists of two arrays: a hash table to store symbols and prefixes, and a code table to associate codes with hash table indexes. The tables are initially stored back-to-back in memory. When a new symbol is read from the input, a single index is used to retrieve corresponding entries from each array. The 16-merge version combines the two tables to form an array of structs. Thus, the entry from the code table is brought into the cache when the hash entry is read. The reduction in energy is negligible: though one type of miss has been eliminated, the program is actually dominated by a second type of miss: the probing of the hash table for free entries. The Skiff data cache is small (16 KB) compared to the size of the hash table ($\approx$270 KB); thus the random indexing into the hash table results in a large number of misses. A more useful energy and performance optimization is to make the hash table more sparse. This admits fewer collisions, which results in fewer probes and thus a smaller

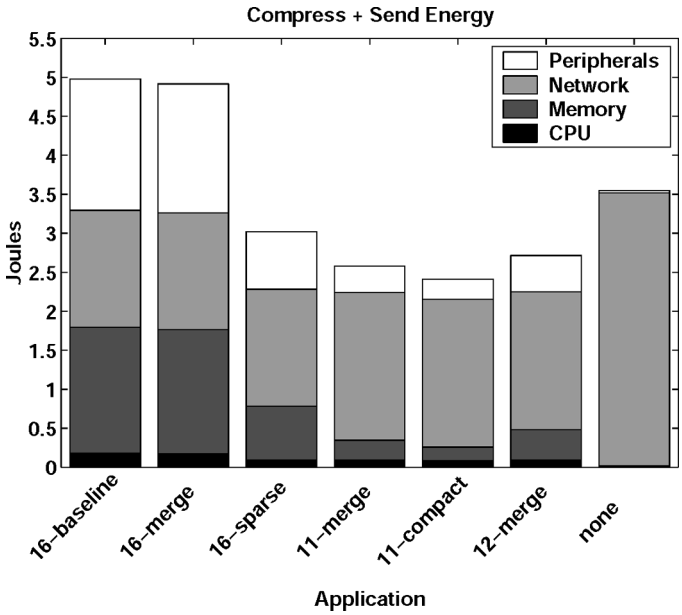

Fig. 16. Optimizing *compress* (Text data).

number of cache misses. As long as the extra memory is available to enable this optimization, about 0.53 J are saved compared with applying no compression at all. This is shown by the 16-sparse bar in the figure. The baseline and 16-merge implementations require more energy than sending uncompressed data, but 16-sparse has made compression worthwhile. A 12-bit version of compress is shown as well. Even when peripheral overhead energy is disregarded, it outperforms or ties the 16-bit schemes as the reduced memory energy due to fewer misses makes up for poorer compression.

Another way to reduce cache misses is to fit both tables completely in the cache. Compare the following two structures:

```
struct entry{           struct entry{
  int fcode;              signed fcode:20;
  unsigned short code;    unsigned code:12;
}table[SIZE];           }table[SIZE];
```

Each `entry` stores the same information, but the array on the left wastes 4 bytes per entry. Overly wide types cause 12 wasted bits in `fcode` and 4 bits wasted in `code`, while 2 bytes are used just to align the short `code`. Using bitfields, the layout on the right contains the same information yet fits in half the space. If the entry were not 4 bytes, it would need further padding for alignment. Code becomes more complex as C does not support arrays of bitfields, but unless the additional code introduces significant instruction cache misses, the change is low-impact. A bitwise AND and a shift are all that is needed to determine the offset into the compact structure. By allowing the whole table to fit in the cache, the program with the compacted array has just 56,985 data

cache misses compared with 734,195 in the unpacked structure: a 0.26% miss rate versus 2.88%. The energy benefit for *compress* with the compact layout is negligible because there is so little CPU and memory energy to eliminate by this technique. The 11-merge and 11-compact bars illustrate the similarity. Nevertheless, 11-compact runs 1.5 times faster due to the reduction in cache misses, and such a strategy could be applied to any program which needs to reduce cache misses for performance and/or energy. Despite a dictionary with half the size, the number of bytes to transmit increases by just 18% compared to 12-merge. Energy, however, is lower with the smaller dictionary due to less energy spent in memory and increased speeds, which reduce the overhead of powering peripheral devices.

Similarly, data structures which contain groups of pointers (which are 4 bytes long regardless of what they point to) can be converted to groups of 16-bit integers to halve storage space when indices do not exceed 16 bits. As long as network energy continues to dominate total energy, the effective doubling of storage space should be used to improve compression ratios.

The fully associative sets of the SA-110 cache are useful for eliminating conflict misses, but the poor locality and large data structures of the compression applications cannot exploit this. Redesigning the caching policy could play a large role in reducing the energy of lossless compression applications. Disabling the data cache can save over 50% of the energy of a load miss [Flinn et al. 2000]. Thus, when it is known that many load misses are bound to occur, disabling the data cache may be wise. Fetching eight-word blocks is another inefficient use of hardware for compression applications. Unless the benchmark can be restructured with spatial locality at this granularity, single-word block fetches would be just as useful and require less power.

5.2 Exploiting the Sleep Mode

When a platform has a low-power idle state, it may be sensible to sacrifice energy in the short-term in order to complete an application quickly and enter the low-power idle state [Miyoshi et al. 2002]. Figure 17 shows the effect of this analysis for compression and sending of text. Receive/decompression exhibits a similar, but less-pronounced variation for different idle powers. It is interesting to note that, assuming a low-power idle mode can be entered once compression is complete, the choice of compression strategies will vary. With its 1 W of idle power, the Skiff would benefit most from *zlib* compression. A device which used negligible power when idle would choose the *LZO* compressor. While *LZO* does not compress data the most, it allows the system to drop into low-power mode as quickly as possible, using less energy when long idle times exist. For Web data (Figure 18), the compression choice is *LZO* when idle power is low. When idle power is 1 W, *bzip2* is over 25% more energy-efficient than the next best compressor.

5.3 Asymmetric Compression

Consider a wireless client similar to the Skiff exchanging English text with a server. All requests by the client should be made with its minimal-energy compressor, and all responses by the server should be compressed in such a

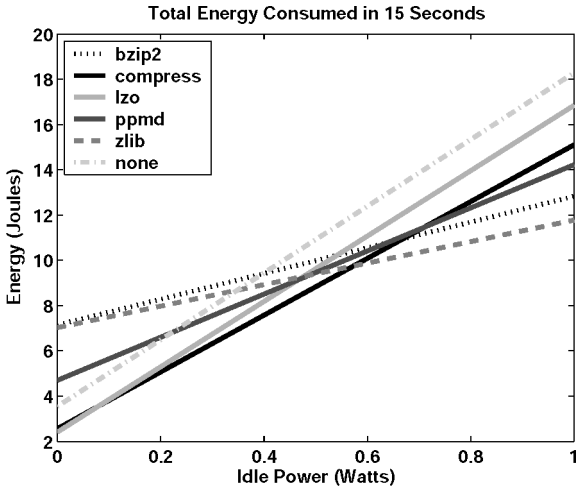

Fig. 17. Compression + Send energy consumption with varying sleep power (Text data).

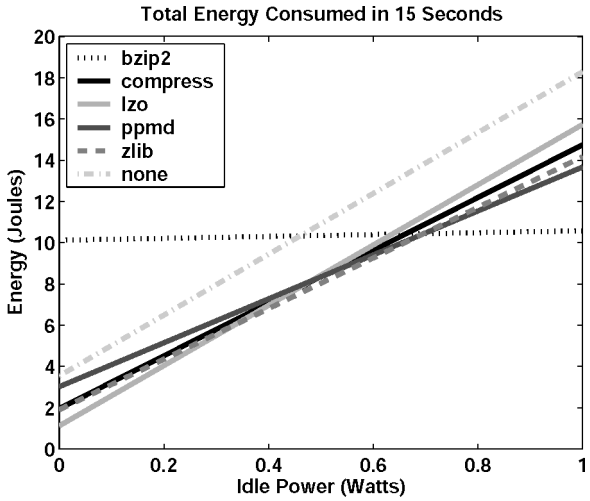

Fig. 18. Compression + Send energy consumption with varying sleep power (Web data).

way that they require minimal decompression energy at the client. Recalling Figures 7 and 8, and recognizing that the Skiff has no low-power sleep mode, we choose compress-12 (the 12-bit codeword LZW compressor) for our text compressor as it provides the lowest total compression energy over all communication speeds.

To reduce decompression energy, the client can request data from the server in a format which facilitates low-energy decompression. If latency is not critical and the client has a low-power sleep mode, it can even wait while the server converts data from one compressed format to another. On the Skiff, *zlib* is the lowest energy decompressor for both text and Web data. It exhibits the property that, regardless of the effort and memory parameters used to compress data, the

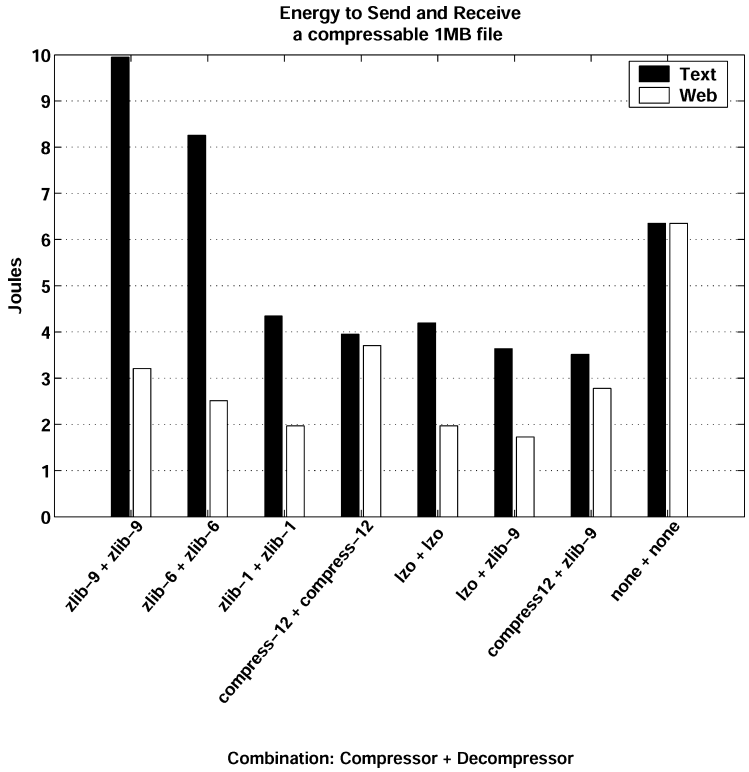

Fig. 19. Choosing an optimal compressor-decompressor pair.

resulting file is quite easy to decompress. The decompression energy difference between *compress*, *LZO*, and *zlib* is minor at 5.70 Mb/s, but more noticeable at slower speeds.

Figure 19 shows several combinations of compressor and decompressor at 5.70 Mb/s. zlib-9 + zlib-9 represents the symmetric pair with the least decompression energy, but its high compression energy makes it unlikely to be used as a compressor for devices which must limit energy usage. compress-12 + compress-12 represents the symmetric pair with the least compression energy. If symmetric compression and decompression is desired, then this "old-fashioned" Unix compress program can be quite valuable. Choosing zlib-1 at both ends makes sense as well—especially for programs already linked with the ubiquitous zlib library. Compared with the minimum symmetric compressor-decompressor, asymmetric compression on the Skiff saves only 11% of energy. However, modern applications such as ssh and mod_gzip use zlib-6 at both ends of the connection. Compared to this common scheme, the optimal asymmetric pair yields a 57% energy savings—mostly while performing compression.

It is more difficult to realize a savings over symmetric zlib-6 for Web data as all compressors do a good job compressing it and zlib-6 is already quite fast. Nevertheless, by pairing lzo and zlib-9, we save 12% of energy over symmetric lzo and 31% over symmetric zlib-6.

6. RELATED WORK

While related work can be found in fields ranging from coding theory to operating system design, this section is restricted to the three most relevant areas: energy measurement and estimation; data compression for low-bandwidth devices; and optimizing algorithms for low energy. Though much work has gone into these fields individually, there has been little work that combines these areas to analyze the system-level effects of lossless compression for wireless transmission. Computation-to-communication energy ratio has been been examined before [Havinga 1999], but our work adds physical energy measurements and applies the results to lossless data compression.

6.1 Energy Measurement and Estimation

To quantify energy savings, it is necessary to have an accurate measurement methodology. Sometimes hardware can be measured in the lab directly or with software-controlled tools. Simulators can allow for a quicker or more detailed breakdown of energy consumption though they may due so at reduced accuracy.

Compaq's Western Research Laboratory published a series of technical notes outlining a methodology with bounded error for measuring the power consumed by an actual handheld system based on the StrongARM SA-1100 [Flinn et al. 2000; Viredaz and Wallach 2000, 2001]. These technical notes examined power usage corresponding to various idle and sleep states, and various cache and buffer configurations. They observed a great energy difference between memory references that hit in the cache versus those that missed. When the data cache was enabled, a read cache miss cost twice the energy of a read from DRAM without caching.

Disabling clock-switching, an implementation-specific function of the SA-110, improves energy usage when done before a lengthy write to memory. The trends observed should be similar to those presented in Section 4 as the systems are very similar in design.

Powerscope is a portable tool for statistically sampling power consumption, which requires markers be placed into the application code [Flinn and Satyanarayanan 1999]. The bench equipment used in the original Powerscope configuration allowed sampling at 1.6-ms intervals. The samples were analyzed offline to associate each with a particular function in source code. Powerscope measurement was used to characterize the various Odyssey applications (see Section 6.3) before optimization work began [Flinn 2001]. An energy-driven sampling technique was presented to improve the accuracy of Powerscope-style tools [Chang et al. 2002]. Although it boasts an energy-driven interrupt scheme to help hone in on energy hotspots and to avoid perturbing the system-under-test during periods of lower energy consumption, initial results were mostly within 4% of Powerscope.

Many simulators and activation models exist for estimating energy consumption by counting events and applying an energy model. Jouletrack [Sinha and Chandrakasan 2001] is one such tool which has been calibrated with the SA-1100 and Hitachi SH-4. In designing the Jouletrack tool, it was discovered that most StrongARM instructions fall into five classes in terms of

average current consumption corresponding to a range of 0.255 to 0.344 W. No class varies more than 38% from the average, and the intrainstruction variation—as a result of various addressing modes and data—is even smaller. Sinha and Chandrakasan noted that most applications have similar power so that their energy usage is roughly proportional to their execution time. To refine this model, Jouletrack groups all StrongARM cycles into four classes based on current consumption: instruction, sequential memory access, nonsequential memory access, and internal cycle. This refined model shows less than 2% error. A good summary of other popular simulators such as SimplePower, Wattch, Millywatt, Joulewatcher, etc., is contained in Chang et al. [2002].

6.2 Lossless Data Compression for Low-Bandwidth Devices

Like any optimization, compression can be applied at many points in the hardware-software spectrum. When implemented in hardware, the benefits and costs propagate to all aspects of the system. Compression in software may have a more dramatic effect, but, for better or worse, its effects will be less global. This section presents related work ranging from silicon solutions to World Wide Web applications.

Modems have implemented the V.42bis standard in hardware since 1990. The algorithm is simple and has low resource requirements; it was implemented on existing 10-MHz Z80-powered modems with as little as 8 KB of additional RAM [Thomborson 1992]. Compression is valuable in modems as they are used on low-bandwidth telephone links. IBM has incorporated hardware data compression in its disk arrays to increase capacity [Craft 1998]. By using content-addressable memory (CAM) arrays for single-cycle dictionary lookups and the density of CMOS technology to implement large history buffers on a chip, gigabyte-per-second throughput has been achieved. An earlier design found CAMs to be power-hungry and a fast, low-power systolic cell has been designed which achieved over 5.5× speedup compared with software implementations [Jung and Burleson 1994, 1995].

The introduction of RISC sparked interest in executing compressed code in the instruction cache to reduce the memory overhead of fixed-length instructions [Wolfe and Chanin 1992]. Code compression and bus compaction, reducing the switching of used bits or sending fractions of words when possible, are related ways to reduce energy in hardware [Lekatsas et al. 2000]. IBM recently introduced Memory Expansion Technology which uses hardware compression and decompression to effectively double the size of main memory for most applications [IBM 2001]. A large L3 cache of uncompressed data is added to hide the latency of the decompression operation, so there is negligible performance loss.

The introduction of low-power, portable, low-bandwidth devices has brought about new (or rediscovered) uses for data compression. Van Jacobson introduced TCP/IP Header Compression in RFC1144 to improve interactive performance over low-speed (wired) serial links [Jacobson 1990], but it is equally applicable to wireless. By taking advantage of uniform header structure and self-similarity over the course of a particular networked conversation, 40-byte headers can be compressed to 3–5 bytes. Three-byte headers are the common case. An

all-purpose header compression scheme (not confined to TCP/IP or any particular protocol) appears in Lilley et al. [2000]. TCP/IP payloads can also be compressed with IPComp [Shacham et al. 2001], but this can be wasted effort if data has already been compressed at the application layer.

The Low-Bandwidth File System (LBFS) exploits similarities between the data stored on a client and server, to exchange only data blocks which differ [Muthitacharoen et al. 2001]. Files are divided into blocks with content-based fingerprint hashes. Blocks can match any file in the file system or the client cache; if client and server have matching block hashes, the data itself need not be transmitted. Despite the complexity of the scheme, much of its bandwidth savings comes from simply applying *gzip* compression to its streams. Rsync [Tridgell 2000] is a protocol for efficient file transfer which preceded LBFS. Rather than content-based fingerprints, Rsync uses its rolling hash function to account for changes in block size. Block hashes are compared for a pair of files to quickly identify similarities between client and server. Rsync block sharing is limited to files of the same name.

A protocol-independent scheme for text compression, NCTCSys, is presented in Motgi and Mukherjee [2001]. NCTCSys involves a common dictionary shared between client and server. The scheme chooses the best compression method it has available for a dataset (or none at all), based on parameters such as file size, line speed, and available bandwidth.

Along with remote proxy servers which may cache or reformat data for mobile clients, splitting the proxy between client and server has been proposed to implement certain types of network traffic reduction for HTTP transactions [Housel and Lindquist 1996; Krashinsky 2003]. Because the delay required for manipulating data can be small in comparison with the latency of the wireless link, bandwidth can be saved with little effect on user experience. Alternatively, compression can be built into servers and clients as in the *mod_gzip* module available for the Apache Webserver and HTTP 1.1-compliant browsers (Hyperspace Communications, Inc; go online to `http://sourceforge.net/projects/mod-gzip/`). Delta encoding, the transmission of only parts of documents which differ between client and server, can also be used to compress network traffic [Hunt et al. 1996; Mogul 1999; Mogul et al. 1997; Santos and Wetherall 1998].

6.3 Optimizing Algorithms for Low Energy

Advanced RISC Machines (ARM) provides an application note which explains how to write C code in a manner best-suited for its processors and ISA [Advanced RISC Machines Ltd (ARM) 1998]. Suggestions include rewriting code to avoid software emulation and working with 32-bit quantities whenever possible to avoid a sign-extension penalty incurred when manipulating shorter quantities. To reduce energy consumption and improve performance, the OptAlg tool represents polynomials in a manner most efficient for a given architecture [Peymandoust et al. 2002]. As an example, cosine may be expressed using two MAC instructions and an MUL to apply a “Horner transform” on a Taylor Series rather than making three calls to a cosine library function.

Besides architectural constraints, high-level languages such as C may introduce false dependencies which can be removed by disciplined programmers. For instance, the use of a global variable implies loads and stores which can often be eliminated through the use of register-allocated local variables. Both types of optimizations are used as guidelines by PHiPAC [Bilmes et al. 1997], an automated generator of optimized libraries. In addition to these general coding rules, architectural parameters are provided to a code generator by search scripts which work to find the best-performing routine for a given platform.

Yang et al. [2001] measured the power and energy impact of various compiler optimizations, and reached the conclusion that energy can be saved if the compiler can reduce execution time and memory references. Šimunić et al. [2000] found that floating point emulation requires much energy due to the sheer number of extra instructions required. They also discovered that instruction flow optimizations (such as loop merging, unrolling, and software pipelining) and ISA specific optimizations (e.g., the use of a multiply-accumulate instruction) were not applied by the ARM compiler and had to be introduced manually. Writing such energy-efficient source code saves more energy than traditional compiler speed optimizations [Šimunić et al. 1999].

The CMU Odyssey project studied "application-aware adaptation" to deal with the varying and often limited resources available to mobile clients. Odyssey trades data quality for resource consumption as directed by the operating system. By placing the operating system in charge, Odyssey balances the needs of all running applications and makes the choice best suited for the system. Application-specific adaptation continues to improve. When working with a variation of the Discrete Cosine Transform and computing first with DC and low-frequency components, an image may be rendered at 90% quality using just 25% of its energy budget [Sinha et al. 2000]. Similar results have been shown for FIR filters and beamforming using a most-significant-first transform. Parameters used by JPEG lossy image compression can be varied to reduce bandwidth requirements and energy consumption for particular image quality requirements [Taylor and Dey 2001]. Research to date has focused on situations where energy-fidelity tradeoffs are available. Lossless compression does not present this luxury because the original bits must be communicated in their entirety and reassembled in order at the receiver.

7. CONCLUSION AND FUTURE WORK

We have examined the energy implications of lossless compression of data before transmission over a wireless network. The value of this research is not merely to show that one can optimize a given algorithm to achieve a certain reduction in energy, but to show that the choice of how and whether to compress is not obvious. It is dependent on hardware factors such as relative energy of CPU, memory, and network, as well as software factors including compression ratio and memory access patterns. These factors can change, so techniques for lossless compression prior to transmission/reception of data must be reevaluated with each new generation of hardware and software. On our StrongARM computing platform, measuring these factors allows an energy savings of up

to 57% compared to a popular default compressor and decompressor. Compression and decompression often have different energy requirements. When one's usage supports the use of asymmetric compression and decompression, up to 12% of energy can be saved compared with a system using a single optimized application for both compression and decompression.

On platforms which support low-power sleep mode, one's choice of compression changes to favor schemes which best balance compression energy with low sleep energy. Choices may change further when the relative energy of system components change with technological advancement. When looking at an entire system of wireless devices, it may be reasonable to allow some to individually use more energy in order to minimize the total energy used by the collection. Designing a low-overhead method for devices to cooperate in this manner would be a worthwhile endeavor.

This work reminds hardware and software developers that committing to one particular compression/decompression scheme is unlikely to be wise in terms of energy. As portable, networked, battery-powered computers evolve and become more popular, extended battery life will grow in importance. Careful, perhaps automated, evaluation of a platform's relative component energy can help choose the most energy-aware lossless compression scheme.

ACKNOWLEDGMENTS

Thanks to John Ankcorn, Christopher Batten, Jamey Hicks, Ronny Krashinsky, Hari Balakrishnan, and the anonymous reviewers for their comments and assistance.

REFERENCES

Advanced RISC Machines Ltd. (ARM). 1998. *Writing Efficient C for ARM*. Application note 34. Go online to www.arm.com/pdfs.

AGILENT TECHNOLOGIES. 2000. *Agilent 34401A Multimeter: User's Guide*, 5th ed. Palo Alto, CA.

AUSTIN, T. M. AND BURGER, D. C. 2001. SimpleScalar version 4.0 release (tutorial). In *Proceedings of the 34th Annual International Symposium on Microarchitecture*.

BANERJEE, S. AND MISRA, A. 2004. Power adaptation based optimization for energy efficient reliable wireless paths. Tech. rep. Department of Computer Sciences, University of Wisconsin-Madison, Madison, WI.

BELL, T. AND KULP, D. 1989. Longest match string searching for Ziv-Lempel compression. Tech. Rep. 06/89. Department of Computer Science, University of Canterbury, Christchurch New Zealand.

BELL, T., POWELL, M., HORLOR, J., AND ARNOLD, R. 1997. The Canterbury Corpus. Go online to `http://www.corpus.canterbury.ac.nz/`.

BELL, T., WITTEN, I. H., AND CLEARY, J. G. 1989. Modeling for text compression. *ACM Comput. Surv. 21*, 4, 557–591.

BILMES, J., ASANOVIĆ, K., CHIN, C.-W., AND DEMMEL, J. 1997. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. In *Proceedings of the 11th ACM International Conference on Supercomputing*.

BURGER, D. C. AND AUSTIN, T. M. 1997. The SimpleScalar tool set, version 2.0. Tech. Rep. CS-TR-97-1342. University of Wisconsin, Madison, Madison, WI.

BURROWS, M. AND WHEELER, D. J. 1994. A block-sorting lossless data compression algorithm. Tech. Rep. 124. Digital Systems Research Center, Palo Alto, CA.

CHANG, F., FARKAS, K., AND RANGANATHAN, P. 2002. Energy-driven statistical profiling: Detecting software hotspots. In *Proceedings of the 2nd Workshop on Power-Aware Computer Systems* (HPCA-8).

CHANG, J.-H. AND TASSIULAS, L. 2000. Energy conserving routing in wireless ad-hoc networks. In *Proceedings of IEEE INFOCOM*. 22–31.

CLEARY, J. G. AND WITTEN, I. H. 1984. Data compression using adaptive coding and partial string matching. *IEEE Trans. Commun. 32*, 4 (Apr.), 396–402.

CRAFT, D. J. 1998. Data compression in ASIC cores. *IBM J. Res. Devel. 42*, 6.

EFFROS, M. 2000. PPM performance with BWT complexity: A new method for lossless data compression. In *Proceedings of the Data Compression Conference*.

FLINN, J. 2001. Extending mobile computer battery life through energy-aware adaptation. Ph.D. dissetation. Carnegie Mellon University, Pittsburgh, PA. Also Tech. rep. TR No. CMU-CS-01-171, Computer Science Depatment, Carnegie Mellan University.

FLINN, J., FARKAS, K. I., AND ANDERSON, J. 2000. Power and energy characterization of the Itsy pocket computer (version 1.5). Tech. Rep. TN-56. Compaq Computer Corporation, Houston, TX.

FLINN, J. AND SATYANARAYANAN, M. 1999. Powerscope: A tool for profiling the energy usage of mobile applications. In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*.

GAILLY, J. 1999. Go online to `comp.compression` Internet newsgroup: Frequently Asked Questions.

GAILLY, J. AND ADLER, M. 2002. zlib. Go online to `http://www.gzip.org/zlib`.

GILCHRIST, J. 2002. Archive comparison test. Go online to `http://compression.ca`.

HAVINGA, P. J. 1999. Energy efficiency of error correction on wireless systems. In *Proceedings of the IEEE Wireless Communications and Networking Conference*.

HICKS, J. 2005. Director, MIT-Quanta T-Party Project. Personal communication.

HICKS, J. ET AL. 1999. Compaq personal server project. Go online to `http://crl.research.compaq.com/projects/personalserver/default.htm`.

HOHLT, B., DOHERTY, L., AND BREWER, E. 2004. Flexible power scheduling for sensor networks. In *Proceedings of the IEEE and ACM Third International Symposium on Information Processing in Sensor Networks*.

HOUSEL, B. C. AND LINDQUIST, D. B. 1996. Webexpress: A system for optimizing Web browsing in a wireless environment. In *Proceedings of the Second Annual International Conference on Mobile Computing and Networking*. 108–116.

HUNT, J. J., VO, K.-P., AND TICHY, W. F. 1996. An empirical study of delta algorithms. In *Software Configuration Management: ICSE 96 SCM-6 Workshop*. Springer, Berlin, Germany, 49–66.

IBM. 2001. *IBM J. Res. Devel. 45*, 2. Preface by Richard E. Harper, Guest Editor.

INTEL CORPORATION. 2000. *SA-110 Microprocessor Technical Reference Manual*. Intel Corporation, Santa Clara, CA.

INTEL CORPORATION. 2001. *Intel StrongARM SA-1110 Microprocessor Developer's Manual*. Intel Corporation, Santa Clara, CA.

JACOBSON, V. 1990. RFC 1144: Compressing TCP/IP headers for low-speed serial links. Available online at `www.rfe-editer.org`.

JAMIESON, K. 2002. Implementation of a power-saving protocol for ad hoc wireless networks. M.S. thesis. Massachusetts Institute of Technology, Cambridge, MA.

JANNESEN, P. ET. AL 1996. (n)compress. Available, among other places, in Redhat 7.2 distribution of Linux.

JUNG, B. AND BURLESON, W. P. 1994. A VLSI systolic array architecture for Lempel-Ziv based data compression. In *Proceedings of the International Symposium on Circuits and Systems*.

JUNG, B. AND BURLESON, W. P. 1995. Real-time VLSI compression for high-speed wireless local area networks. In *Proceedings of the Data Compression Conference*.

KRASHINSKY, R. 2003. Efficient Web browsing for mobile clients using HTTP compression. Tech. Rep. MIT-LCS-TR-882. MIT Laboratory for Computer Science, Combridge, MA.

LEKATSAS, H., HENKEL, J., AND WOLF, W. 2000. Low-power techniques for code compression in embedded systems. In *Proceedings of the Design Automation Conference*.

LELEWER, D. A. AND HIRSCHBERG, D. S. 1987. Data compression. *ACM Comput. Serv. 19*, 3, 261–297.

LILLEY, J., YANG, J., BALAKRISHNAN, H., AND SESHAN, S. 2000. A unified header compression framework for low-bandwidth links. In *Proceedings of the 6th ACM MOBICOM*.

LYCOS. 2002. Lycos 50. Top 50 searches on Lycos for the week ending September 21, 2002.

McEliece, R. 1977. The theory of information and coding. In *Encyclopedia of Mathematics and Its Application*. Vol. 3. Addison-Wesley, Reading, MA.

Miyoshi, A., Lefurgy, C., Hensbergen, E. V., Rajamony, R., and Rajkumar, R. 2002. Critical power slope: Understanding the runtime effects of frequency scaling. In *Proceedings of the International Conference on Supercomputing*.

Mogul, J. C. 1999. Trace-based analysis of duplicate suppression in HTTP. Tech. Rep. 99.2. Compaq Computer Corporation, Houston, TX.

Mogul, J. C., Douglis, F., Feldmann, A., and Krishnamurthy, B. 1997. Potential benefits of delta encoding and data compression for HTTP. Tech. Rep. 97/4a. Compaq Computer Corporation, Houston, TX.

Montanaro et al., J. 1996. A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor. *IEEE J. Sol.-State Circ. 31*, 11 (Nov.), 1703–1714.

Motgi, N. and Mukherjee, A. 2001. Network conscious text compression systems (NCTCSys). In *Proceedings of the International Conference on Information and Theory: Coding and Computing*.

Muthitacharoen, A., Chen, B., and Mazières, D. 2001. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01*, Chateau Lake Louise, Banff, Alta., Canada). 174–187.

Nathuji, R. 2000. Characterization of DRAM. MIT Advanced Undergraduate Project. Massachusetts Institute of Technology, Cambridge, MA.

Nielsen NetRatings Audience Measurement Service. 2002. Top 25 U.S Properties; Week of Sept. 15th. Go online to `www.nielsen-netratings.com`.

Noble, B. D. and Satyanarayanan, M. 1999. Experience with adaptive mobile applications in odyssey. *Mobile Netw. Appl. 4*, 4, 245–254.

Oberhumer, M. F. 2000. Lzo. Go on line to `http://www.oberhumer.com/opensource/lzo/`.

Peymandoust, A., Šimunić, T., and Micheli, G. D. 2002. Low power embedded software optimization using symbolic algebra. In *Proceedings of the Conference on Design, Automation and Test in Europe*.

Santos, J. and Wetherall, D. 1998. Increasing effective link bandwidth by suppressing replicated data. In *Proceedings of the USENIX Annual Technical Conference*. 213–224.

Sayood, K. 2002. *Introduction to Data Compression*, 2nd ed. Morgan Kaufman San Francisco, CA.

Seward, J. 1999. bzip2. Go online to `http://www.spec.org/osg/cpu2000/CINT2000/256.bzip2/docs/256.bzip2.html`.

Seward, J. 2000. e2comp bzip2 library. Go online to `http://cvs.bofh.asn.au/e2compr/index.html`.

Shacham, A., Monsour, B., Pereira, R., and Thomas, M. 2001. RFC 3173: IP payload compression protocol. Available online at `www.rfc-editor.org/`.

Shannon, C. E. 1948. A mathematical theory of communication. *Bell Syst. Tech. J. 27*, 379–423 and 623–656.

Shkarin, D. 2002a. PPM: One step to practicality. In *Proceedings of the Data Compression Conference*.

Shkarin, D. 2002b. PPMd. Go online to `ftp://ftp.elf.stuba.sk/pub/pc/pack/ppmdi1.rar`.

Šimunić, T., Benini, L., and Micheli, G. D. 1999. Energy-efficient design of battery-powered embedded systems. In *Proceedings of the IEEE International Symposium on Low Power Electronics and Design*.

Šimunić, T., Benini, L., Micheli, G. D., and Hans, M. 2000. Source code optimization and profiling of energy consumption in embedded systems. In *Proceedings of the International Symposium on System Synthesis*.

Sinha, A. and Chandrakasan, A. 2001. Jouletrack—a Web based tool for software energy profiling. In *Proceedings of the 38th Design Automation Conference*.

Sinha, A., Wang, A., and Chandrakasan, A. 2000. Algorithmic transforms for efficient energy scalable computation. In *Proceedings of the IEEE International Symposium on Low Power Electronics and Design*.

Standard Performance Evaluation Corporation. 2000. CPU2000. Go online to `www.spec.org`.

Taylor, C. N. and Dey, S. 2001. Adaptive image compression for wireless multimedia communication. In *Proceedings of the IEEE International Conference on Communication*.

THOMBORSON, C. 1992. The V.42bis standard for data-compressing modems. *IEEE Micro 12*, 5.
TRIDGELL, A. 2000. Efficient algorithms for sorting and synchronization. Ph.D. dissertation. Australian National University, Canberra, Australia.
VIREDAZ, M. A. AND WALLACH, D. A. 2000. Power evaluation of Itsy version 2.3. Tech. Rep. TN-57. Compaq Computer Corporation, Houston, TX.
VIREDAZ, M. A. AND WALLACH, D. A. 2001. Power evaluation of Itsy version 2.4. Tech. Rep. TN-59. Compaq Computer Corporation, Houston, TX.
WELCH, T. A. 1984. A technique for high-performance data compression. *IEEE Comput. 17*, 6, 8–19.
WOLFE, A. AND CHANIN, A. 1992. Executing compressed programs on an embedded RISC architecture. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*.
YANG, H., GAO, G. R., MARQUEZ, A., CAI, G., AND HU, Z. 2001. Power and energy impact of loop transformations. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power 2001, Parallel Architecture and Compilation Techniques*.
ZIV, J. AND LEMPEL, A. 1977. A universal algorithm for data compression. *IEEE Trans. Inform. Theor. 23*, 3 (May), 337–343.
ZIV, J. AND LEMPEL, A. 1978. Compression of individual sequences via variable rate coding. *IEEE Trans. Inform. Theor. 24*, 5 (Sep.), 530–536.

Received February 2005; revised November 2005; accepted November 2005