Energy Aware Lossless Data Compression

by

Kenneth C. Barr

B.S.E. Computer Engineering
University of Michigan, April 2000

Submitted to the Department of
Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2002

Author .....
Department of
Electrical Engineering and Computer Science
August 16, 2002

Certified by .....
Krste Asanović
Assistant Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Energy Aware Lossless Data Compression

by

Kenneth C. Barr

Submitted to the Department of
Electrical Engineering and Computer Science
on August 16, 2002, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

Wireless transmission of a bit can require over 1000 times more energy than a single 32-bit computation. It may therefore be desirable 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 consequently, a longer battery life for portable computers. This thesis is a study of the energy profiles of lossless data compression algorithms. Several distinct algorithms have been selected and are measured on a StrongARM SA-110 processor. This work demonstrates that with several typical compression tools, there is a net energy *increase* when compression is applied before transmission. Reasons for this increase are explained and suggestions are made to avoid it. Compression and decompression need not be performed by the same algorithm. By choosing the lowest-energy compressor and decompressor on the test platform, rather than using default levels of compression, overall energy to send data can be reduced 57%. Compared with a system using the same optimized application for both compression and decompression, the asymmetric scheme saves 11% of the total energy.

Thesis Supervisor: Krste Asanović
Title: Assistant Professor

Acknowledgments

This thesis would not have been possible without the help, generosity, and kindness of several people. My advisor, Krste Asanović, not only suggested the intriguing topic of this thesis, but unfailingly provided support, direction, and advice as I worked to complete it. He is an extremely responsible and accessible advisor, and I am thankful for the opportunity to work with him and the Assam group. Scott Ananian and Jamey Hicks came to the rescue with Skiff help early on, allowing me to have a stable platform on which to evaluate the energy of applications. David Wentzlaff, Nathan Shnidman, and John Ankcorn helped me build confidence and acquire equipment in the hardware lab. Ronny Krashinsky patiently explained ways to look at energy and how to measure it. His work in compressing HTTP traffic was the jumping-off point for this thesis. Christopher Batten, my officemate, went above and beyond the call of duty. I appreciate his suggestions, curiosity, creativity, idealism, and perspective. His help made a big difference in the quality of this thesis. Caroline and my family provided an equally important form of help through unflagging encouragement, motivation, and love.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis is motivated by previous work in energy-aware computing and the communication - computation energy gap: 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 a single 32-bit computation. Thus, if 1000 computation operations can compress data by even one bit, energy should be saved. Algorithms which once seemed too resource or time intensive might be valuable for saving energy. Implementations which made compression concessions to gain performance might be modified to provide an overall energy savings. Ideally, the effort exerted to compress data should be variable so one may trade speed for energy. While some types of data (e.g., audio and video) may accept some degradation in quality, other data must be transmitted faithfully with no loss of information. Such data presents a unique challenge in that, unlike related work in lossy compression for reducing energy, one cannot sacrifice fidelity of the data to achieve energy goals.

This work provides an in-depth examination of the energy requirements of several lossless data compression schemes. The "Skiff" platform developed by Compaq Cambridge Research Labs is a StrongARM-based platform designed with energy measurement in mind. Energy requirements of the CPU, memory, network card, and peripherals can be measured separately in a laboratory. The platform is similar to the popular Compaq iPAQ handheld computer, so the results are relevant to handheld hardware and developers of embedded software. Several families of compression algorithms are analyzed and

characterized, and it is shown that compression prior to transmission may cause an overall energy increase. Behaviors and resource-usage patterns are highlighted 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.

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 losslessly compress data 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. Since these factors can change, techniques for lossless compression prior to transmission/reception of data must be re-evaluated with each new generation of hardware and software.

This thesis begins with a discussion of related work: Chapter 2 discusses relevant work in a variety of related fields, and Chapter 3 explains data compression terminology and the lossless compression algorithms examined in this thesis. Chapter 4 contains detailed results and analysis of several popular data compression applications as measured in the lab. Armed with these results, Chapter 5 presents a new energy-aware, lossless data compression strategy. Chapter 6 concludes and suggests future research in this area.

Chapter 2

Related work

While work related to this thesis can be found in fields ranging from coding theory to operating system design, this chapter is limited to 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, it is difficult to find any which combines them to examine lossless data compression from an energy standpoint. Computation-to-communication energy ratio has been been examined before [19], but this work adds physical energy measurements and applies the results to lossless data compression.

2.1 Energy measurement and estimation

To quantify reduction in energy, 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 quick estimation though they may due so at reduced accuracy.

Compaq's Western Research Laboratory has 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 [15, 56, 57]. These technical notes examine power usage corresponding to various idle and sleep states, and various cache and buffer configurations. They observe a great energy difference between memory references that hit in the cache versus those that miss. When the data cache is enabled, a

read cache miss costs 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 shown should be similar to those presented in Chapter 4 as the system described in the reports is quite similar to that used for evaluation in this thesis.

Powerscope is a portable tool for statistically sampling power consumption which requires markers be placed into the application code [16]. The bench equipment used in the original Powerscope configuration allowed sampling at 1.6 ms intervals. The samples are analyzed offline to associate each with a particular function in source code. Powerscope measurement was used to characterize the various Odyssey applications (see Section 2.3) before optimization work began [14]. An energy-driven sampling technique was presented to improve the accuracy of Powerscope-style tools [11]. Though 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 are mostly within 4% of Powerscope.

Many simulators and activation models exist for estimating energy consumption by counting events and applying an energy model. Jouletrack [48] 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 Watts. No class varies more than 38% from the average, and the intra-instruction variation – as a result of various addressing modes and data – is even smaller. Sinha notes 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, non-sequential 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 [11].

2.2 Data compression for low-bandwidth devices

Like any optimization, compression can be applied at many points in the hardware-software spectrum. When applied 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 10MHz Z80-powered modems with as little as 8 KB additional RAM [52]. 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 [12]. 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 designed a fast, low-power systolic cell which achieved over 5.5X speedup compared with software implementations [28, 29].

The introduction of RISC sparked interest in executing compressed code in the instruction cache to reduce the memory overhead of fixed-length instructions [58]. 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 [31]. IBM recently introduced Memory Expansion Technology which uses hardware compression and decompression to effectively double the size of main memory for most applications [22]. An 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 [25], 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 [33]. TCP/IP payloads can be compressed as well with IPComp [45], 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 [38]. 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 [53] 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 [37]. NCTCSys involves a common dictionary shared between client and server. The scheme chooses the best compression method it has available (or none at all) for a dataset based on parameters such as file size, line speed, and available bandwidth.

A split proxy system for compression of HTTP transactions with mobile clients is proposed in [30]. Since the delay required for compression is small in comparison with the latency of the wireless link, bandwidth can be saved with little effect on user experience. To address the additional energy requirements, the author suggests that less compute-intensive algorithms be used – my thesis seeks to produce such algorithms. 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 [21]. Delta encoding, the transmission of only parts of documents which differ between client and server, can also be used to compression HTTP. A review of these schemes (as well as other proxy based schemes) can be found in [30].

2.3 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 their ISA [1]. For example, since the ARM has no division instruction, modular arithmetic specified `a % b` will result in a call to a C library function which divides `a` by `b` and returns the remainder. Depending on its usage, the programmer might be able to replace the `%` operator with simpler arithmetic statements. The document also mentions that working with values less than 32 bits incurs a sign extension penalty and suggests that `chars` and `shorts` be replaced by 32 bit quantities if possible.

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 [8], 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. measured the power and energy impact of various compiler optimizations, and reached the conclusion that power can be saved if the compiler can reduce execution time and memory references [59]. Šimunić found that floating point emulation requires much energy due to the sheer number of extra instructions required [55]. It was 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 [54].

The CMU Odyssey project studied “application-aware adaptation” to deal with the varying, 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 [49]. Similar results are 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 [51].

To reduce energy consumption and improve performance, the OptAlg tool represents polynomials in a manner most efficient for a given architecture [41]. 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. Research to date has focused on situations where energy-fidelity tradeoffs are available. Lossless compression does not present this luxury since the original bits must be communicated in their entirety and re-assembled in order at the receiver.

Though most of these optimizations could just as easily be billed as speed optimizations, energy-only optimizations are possible. Energy-only savings are possible whenever non-critical path (parallel) useless work can be eliminated, however this is not likely on the single-issue StrongARM processor with blocking caches examined in this thesis. This thesis will focus on situations in which the mixture of high energy network operations and low energy processor operations can be adjusted so that overall energy is lower. This is possible even if the number of total operations, or time to complete them, increases. Understanding what lossless compression algorithms are available and how they work is important to achieve such optimization. Chapter 3 introduces the reader to several compression strategies and highlights relevant implementation details.

Chapter 3

Lossless data compression overview

A complete treatment of the discipline of data compression is outside the scope of this thesis, but before exploring what makes a lossless data compression scheme "energy aware," one must be familiar with the concepts and terminology of data compression. The descriptions below are simplified versions of those that appear in [32] and [42], each of which contains bibliographic references to seminal papers. Particular implementations of each algorithm will be discussed as each algorithm is introduced in Section 3.3.

3.1 Terminology

In the 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 an binary executable, one must be able to reconstruct every bit perfectly – hence the need for *lossless* data compression. 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.

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 each symbol. (i.e., if the sequence 1 1 2 3 5 ... 6765 was 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 prob-

ability of a letter occurring as a probability conditioned on letters that have already been transmitted. Next, the model is transmitted with a description of how the data differs from the model. In the *coding* step, this 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 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.

3.2 Coding

Coding maps symbols from the input alphabet into compact binary sequences. Though many coding schemes exist, I focus on the most popular schemes for data compression tools.

3.2.1 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, the Huffman algorithm has been adapted to work dynamically as source symbols are encountered. In the dynamic scheme, the tree is updated as each symbol is encountered.

Since the length of a Huffman code depends on the magnitude of the probability of the

most-frequent symbol, shorter codes can be obtained through the Huffman procedure if the alphabet is larger. This can be accomplished by "blocking" together symbols of the source alphabet. Unfortunately, this increases the size of the tree exponentially. If the alphabet has m symbols (a, b, ...), when they are considered in pairs, it is as if a new alphabet of size m^2 is being used (aa, ab, bb, ba, ...).

3.2.2 Arithmetic coding

Optimal compression ratio for a data source is traditionally described with respect to Shannon's definition of source *entropy*: a measure of the source's information and therefore, the average number of bits required to represent it. According to Shannon, the ideal number of bits per symbol (for a sequence of n independent and identically distributed (iid) symbols) is expressed: $Entropy = -\sum_{i=1}^{n} p_i \log p_i$ where p_i is the probability of occurrence of the ith symbol. The entropy is expressed as a limit for non-iid sources. Sometimes, the most frequently occurring symbol can contain so little information that it would be ideal to represent it with less than one bit. In these cases, an arithmetic code can be used to bring the average length of a codeword much closer to optimal than is practical with Huffman Coding.

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.

As larger groups of symbols are considered, the code approaches optimality in terms of bits need to represent a stream of bytes. Though both Huffman and Arithmetic schemes converge to optimal, it is more feasible to achieve near-optimal results with arithmetic coding since symbols can be joined into larger groups of length m without the need to build a codeword for each sequence of length m.

Arithmetic coding requires frequent division and multiplication, but can be implemented to run faster than the well-optimized Unix *compact* program, an adaptive Huffman encoder.

3.2.3 Lempel-Ziv codes

A Lempel-Ziv codebook is made up of fixed-length words 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 $log(n)$ bits, systems for gradually increasing the length of codewords exist.

3.3 Lossless compression algorithms

These coding techniques are used in the algorithm families introduced below. There are two fundamental methods for constructing Lempel-Ziv codes. Introduced in the late 1970's, 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 Chapter 4.

3.3.1 Sliding window - LZ77

LZ77 maintains a current pointer into the source data, a search buffer, and a look-ahead buffer. The search buffer is made up of symbols encountered prior to the current symbol, and the look-ahead buffer contains symbols which appear after 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.

zlib is based on LZ77 defaulting to a 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. *zlib* chooses the optimal 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 the hash table. Users may also set the an "effort" parameter which dictates how hard the compressor should try to extend matches it finds in its history buffer. *zlib* is the library form of the popular *gzip* utility (the library form was chosen since it provides more options for trading off memory and performance). Unless specified, it is configured with similar parameters as *gzip*.

zlib implements its longest-match search with the three arrays depicted in Figure 3-1. 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.

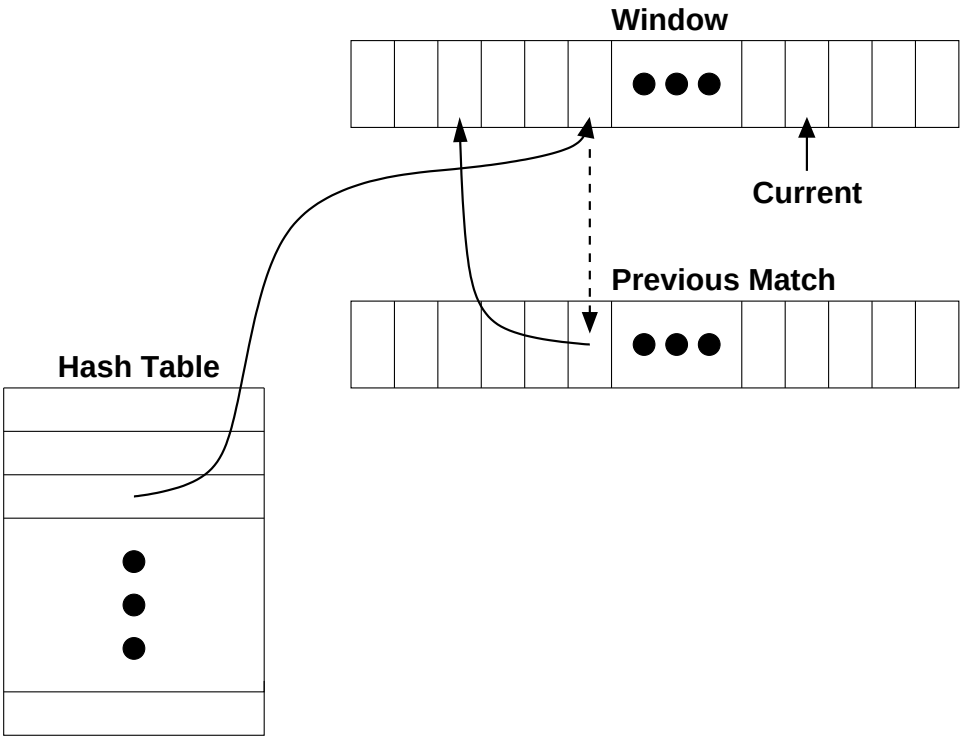

Figure 3-1: Hash table implementation of LZ77

LZO is a compression library meant for "real-time" compression. Like *zlib*, it uses LZ77 with a hash table to perform searches. *LZO* is unique in that its hash table dictionary fits in 16KB 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.

3.3.2 Dictionary - LZ78

The LZ78 scheme was introduced to account for cases in which a nearby match cannot be found. Instead of the sliding search-buffer, LZ78 uses a dictionary. 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. 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. Since every symbol exists 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 non-adaptive fashion, or adopt another policy to limit memory usage.

compress is a popular Unix utility. It implements the LZW algorithm with codewords beginning at nine bits. When all nine-bit codes have been used, the codebook size is doubled and the use of ten-bit codes begins. This doubling continues until codes are sixteen bits long, inclusive. The dictionary becomes static once it is entirely full. Whenever *compress* detects decreasing compression ratio, the dictionary is cleared and the process beings 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. Table 3.1 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.

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"

Table 3.1: Hash table implementation of LZW

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) is found in the "previous code" field. Hashing allows an average constant-time access 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 to the table, common strings will benefit from temporal locality.

3.3.3 Prediction with Partial Match - PPM

The fact that a certain string of symbols has appeared can aide in predicting what symbol will come next. For instance, if the letters "COMPR" appear in this thesis, 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. An arithmetic coder can use this stream of probabilities to efficiently code the source. Clearly, longer contexts will improve the probability estimation, but it requires time to amass large contexts (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 trade-off 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. 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.

3.3.4 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 and reversible, it is not obvious how the original S can be reconstructed. Curious readers are referred to [10]. 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" 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" 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.

BWT grew in popularity since its implementations, based on efficient sorting, lead to greater speed than PPM implementations available at the time and gave similar excellent compression ratios. In latency-critical applications, the block-based processing of BWT

could be a bottleneck. Also, 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. 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.

bzip2 is based on the Burrows Wheeler Transform. It reads in blocks of data, run-length-encoding them 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.

3.4 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. 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 that of the LZ77, LZ78, and BWT methods [13, 47].

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 [5]. The quality and applicability of the implementation is as important as the underlying algorithm. This chapter has presented example implementations from each algorithmic family. By choosing the top representative in each family, the implementation playing field is leveled, making it easier to gain insight into the underlying algorithm and its influence on energy. Nevertheless, it is likely that each application can be optimized further (Section 5.1 shows the benefit of optimization) or use a uniform style

of I/O. Thus, the evaluation of Chapter 4 focuses on inherent patterns rather than making a direct quantitative comparison.

Chapter 4

Evaluation of compression applications

This chapter begins by describing the choice of applications and data used to measure compression energy. The applications are chosen to provide a variety of algorithms and performance examples. Next the "Skiff" platform is introduced along with an explanation of how it can be used to make energy measurements of algorithms. I motivate the investigation of low-energy data compression by comparing the measured energy of communication with that of computation. After examining the performance of common lossless compression applications, guidelines are derived for those seeking to minimize energy consumption of compressed data transmission.

4.1 Benchmark selection

I have collected and compiled several benchmarks for the Skiff which have been described in Section 3.3. For input datasets, I have chosen the popular Calgary Corpus [7]. Though more modern and/or, methodically-chosen corpora exist, compression ratios for a given compressor have remained nearly identical over a range of well-chosen input datasets [3]. The Calgary Corpus remains the most popular reference for comparison despite its age. It consists of several varieties of English text (bibliography, book, paper, etc.) and several non-English sources (a picture, object files, geophysical data, etc.). All applications were cross-compiled (x86 host, ARM target) with GCC version 2.95.3. Level two optimizations were applied in addition to any optimizations already present in a given application's Makefile.

Application Version	Algorithm	Notes Defaults
bzip2 [43] 0.1pl2	BWT	RLE→BWT→MTF→RLE→HUFF 900k block size
compress [27] 4.2.4	LZW	Unix Compress program 16 bit codes (maximum), fast hashing
LZO [40] 1.07	LZ77	Favors speed over compression lzo1x_12 (4K entry hash table uses 16KB)
PPMd [46] variant I	PPM	used in "rar" compressor Order 4, 10MB memory, restart model
zlib [34] 1.1.4	LZ77	library form of gzip Chaining level 6 / 32K Window / 32K Hash Table

Table 4.1: Compression applications and their algorithms

Figures 4-1 - 4-4 show the performance of the lossless data compression applications using metrics of compression ratio, execution time, and static memory allocation. The data is repeated for clarity in Tables 4.2 - 4.3. Though such characterization has been performed before ([6, 17, 18]), the results are included here to correct for any implementation changes since previously published data was last obtained. Most popular repositories for comparison of data compression do not examine the memory footprint required for compression or decompression. 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 4.4.

Figures 4-1 - 4-4 confirm that I have chosen an array of applications that span a range of compression ratios and execution times. Each application represents a different family of compression algorithms as noted in Table 4.1. Consideration was also given to the popularity, quality, parameterizability, and portability of the source code and documentation. The table includes the default parameters used with each program.

As stated earlier, the role of implementation with respect to performance cannot be overlooked. 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 PPM compressors that came before it. *LZO* represents an approach for achieving great speed with LZ77.

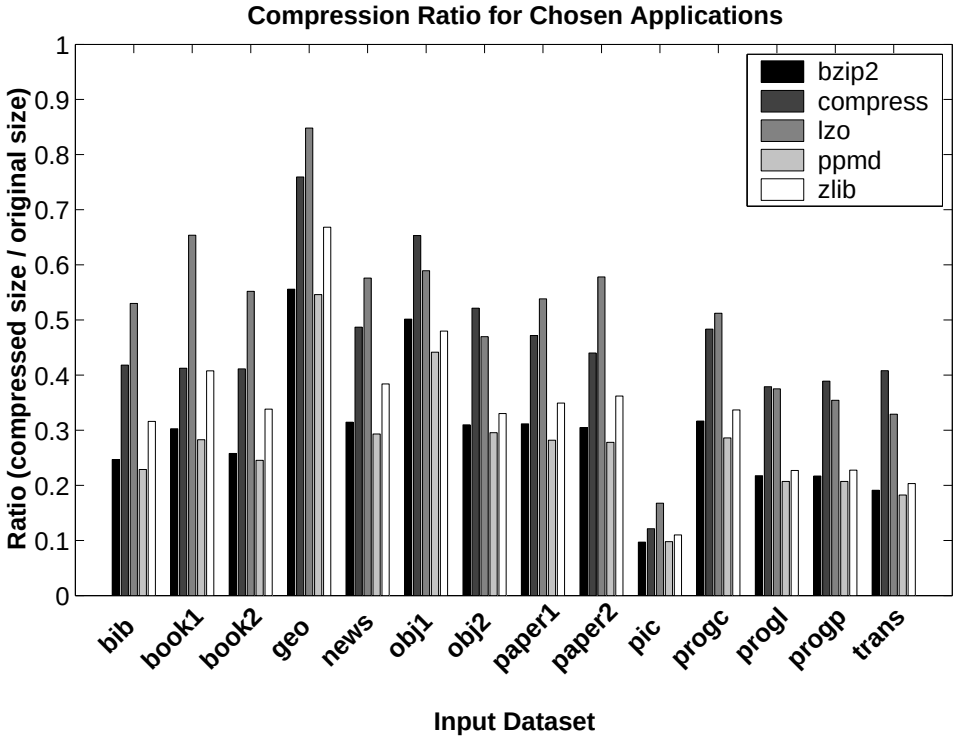

Figure 4-1: Compression Ratio

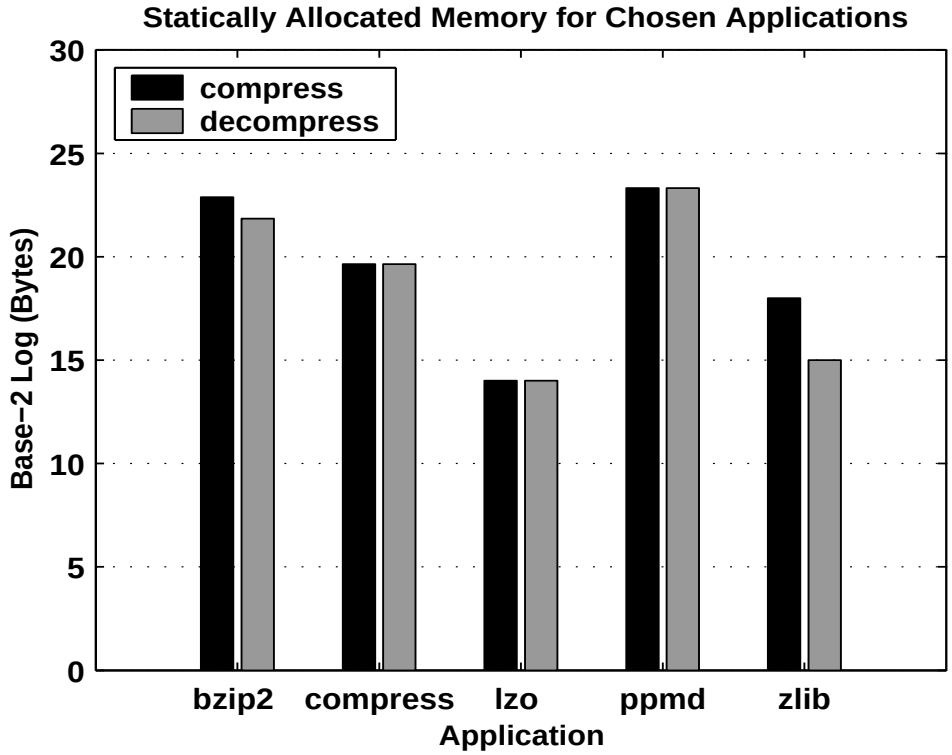

Figure 4-2: Statically allocated memory (KB)

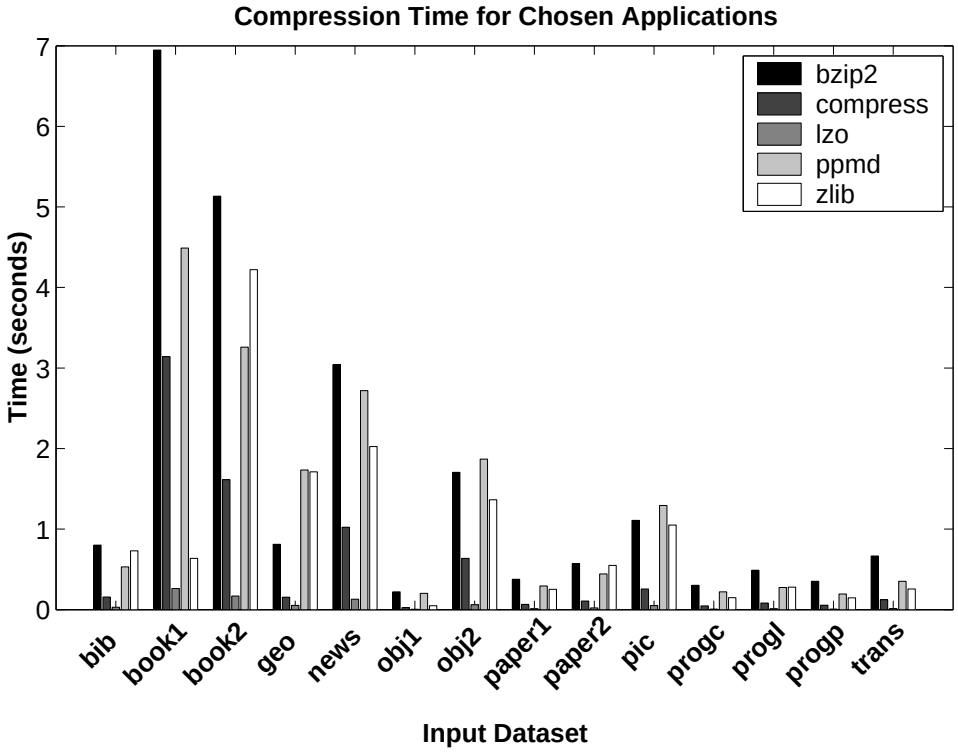

Figure 4-3: Compression Time

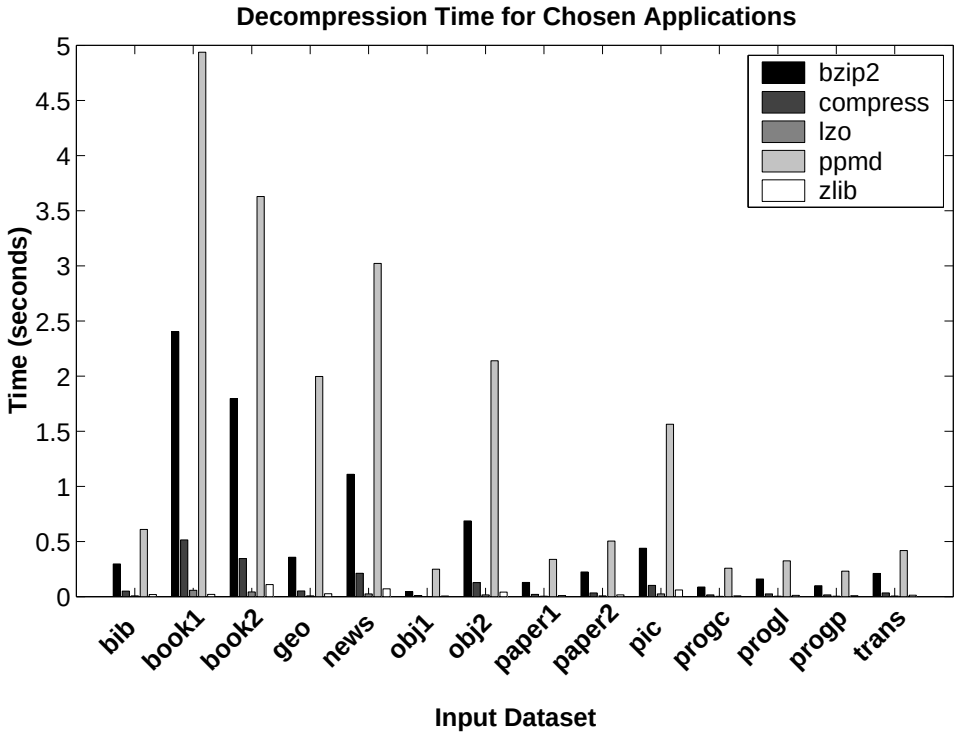

Figure 4-4: Decompression Time

	Application				
	ppmd	bzip2	zlib	compress	lzo
bib	0.229	0.247	0.316	0.418	0.530
book1	0.283	0.303	0.408	0.413	0.654
book2	0.245	0.258	0.338	0.411	0.552
geo	0.546	0.556	0.668	0.760	0.848
news	0.293	0.314	0.384	0.487	0.576
obj1	0.441	0.502	0.480	0.653	0.589
obj2	0.296	0.310	0.330	0.521	0.470
paper1	0.282	0.311	0.349	0.472	0.538
paper2	0.278	0.305	0.362	0.440	0.578
pic	0.098	0.097	0.110	0.121	0.168
progc	0.286	0.317	0.337	0.483	0.512
progl	0.207	0.217	0.227	0.379	0.375
progp	0.207	0.217	0.227	0.389	0.354
trans	0.182	0.191	0.203	0.408	0.329

Table 4.2: Compression ratio

	Application				
	lzo	zlib	compress	bzip2	ppmd
compress	16	256	800	7434	10240
decompress	16	32	800	3614	10240

Table 4.3: Statically allocated memory (KB)

	Application				
	lzo	compress	ppmd	zlib	bzip2
bib	0.030	0.156	0.531	0.731	0.802
book1	0.263	3.140	4.490	0.637	6.946
book2	0.170	1.613	3.259	4.221	5.134
geo	0.053	0.154	1.733	1.711	0.811
news	0.130	1.023	2.719	2.025	3.041
obj1	0.005	0.026	0.203	0.050	0.221
obj2	0.062	0.637	1.869	1.365	1.705
paper1	0.012	0.065	0.293	0.252	0.378
paper2	0.021	0.108	0.444	0.550	0.570
pic	0.051	0.256	1.294	1.050	1.108
progc	0.008	0.047	0.221	0.149	0.302
progl	0.011	0.083	0.276	0.280	0.489
progp	0.007	0.056	0.194	0.147	0.353
trans	0.014	0.123	0.353	0.256	0.666

Table 4.4: Compression time

	Application				
	lzo	zlib	compress	bzip2	ppmd
bib	0.007	0.019	0.049	0.296	0.609
book1	0.057	0.019	0.514	2.404	4.938
book2	0.043	0.109	0.345	1.797	3.630
geo	0.007	0.026	0.051	0.357	1.997
news	0.024	0.070	0.211	1.110	3.023
obj1	0.001	0.005	0.009	0.046	0.248
obj2	0.015	0.041	0.127	0.687	2.139
paper1	0.003	0.010	0.020	0.129	0.337
paper2	0.006	0.016	0.034	0.224	0.504
pic	0.024	0.060	0.101	0.439	1.564
progc	0.002	0.007	0.015	0.087	0.257
progl	0.004	0.011	0.024	0.159	0.324
progp	0.003	0.007	0.016	0.099	0.231
trans	0.005	0.013	0.033	0.210	0.418

Table 4.5: Decompression time

4.2 Methodology

Measuring energy at a fine grain is difficult. Proper equipment and methodology are helpful, but not a panacea. This section describes a hardware platform which facilitates energy measurement and a test harness for running compression programs. A programmable multimeter and sense resistor provide a convenient, accurate way to examine energy in a running system [57]. The energy measurement methodology is described along with an analysis of sources of error. Finally the results of a simulation-based study are presented which motivate the use of the hardware-only technique for measuring long-running programs.

4.2.1 Equipment

The Compaq Personal Server, codenamed "Skiff," is "a simple, configurable, StrongARM-based embedded computing platform designed to support a wide variety of applications in a very small (5×8 inch) footprint" [20]. The Skiff has 32 MB of DRAM and runs at 233 MHz. It has support for the Universal Serial Bus, a RS232 Serial Port, Ethernet, two Cardbus sockets, and a variety of general purpose I/O. A five volt Enterasys 802.11b wireless network card (part number CSIBD-AA) is used in one of the Cardbus sockets. Based on the Intel SA-110 [36, 23], the Skiff is computationally similar to the popular Compaq iPAQ handheld (an SA-1110 [24] based device). 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 4-5). With a Cardbus extender card, one can isolate the power used by a wireless network card as well. While designed with distinct power planes in mind, the CAD tools and an overwire fix joined these planes in several places. Thus, some rework was necessary before beginning measurements.

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 non-volatile 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

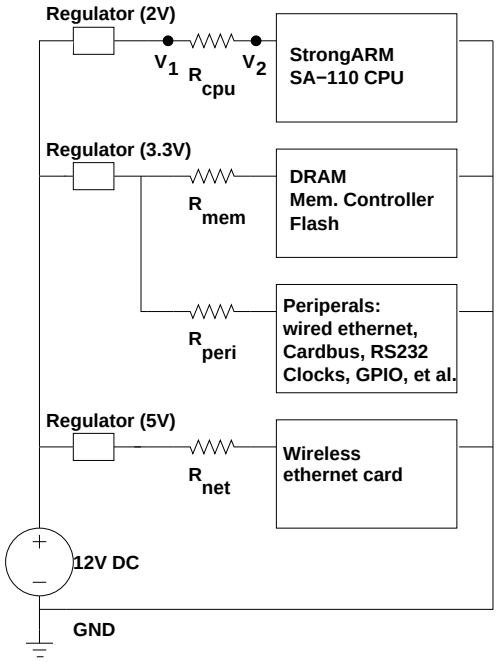

Figure 4-5: Simplified Skiff power schematic

program completes loading. I/O is conducted in memory using a modified SPEC harness [50] to avoid the large cost of accessing the network filesystem.

4.2.2 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 4-5 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 ($I_{cpu} = \frac{V_2 - V_1}{R_{cpu}}$). The multimeter internally averages 285 samples over the course of 6.5 seconds, 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 4.2.3). Network energy was measured in a one-time experiment (Section 4.3.1). This leaves six distinct measurements which must be made per experiment to obtain the total non-network system power ($P = P_{cpu} + P_{mem} + P_{peripheral}$).

To compute total energy, one must know the duration of the application. 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 $(t_1 + t_2 + ... + t_n)/n$ 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 $Energy = P * t$.

4.2.3 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 (≈ 3 in) must be soldered to the board so that the multimeter may be attached. The resistance, rated at $0.20\,\Omega$, increased as much as $0.46\,\Omega$ with the addition of the leads and solder. This resistance is measured using the 4-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 $100\,\Omega$ range [2].

Voltage measurement error takes a similar form, consisting of error in reading (dependent on the input level) and an error inherent to the range. Operating with sixty integrations per second, we add an additional noise error. The sample rate could be made faster, but this would add as much as 0.12 mV error per volt or 0.03 mV to a measured millivolt. This additional error, due to poorer noise reduction in the multimeter at high speeds, was overlooked in [57]. To increase sampling speed, the multimeter's auto-zero functionality is turned off, and we must compensate for this error as well.

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 second 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 seconds 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, one cannot know the true average power over a particular integration period, only an approximation. The formula for maximum error due to combination of uncorrelated samples is stated in [57] (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 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 "realtime clock" of the Skiff is not realtime 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 4.6 and 4.7. When larger error occurs, it is due mostly to the error-in-averaging component of total error. The CPU is most effected 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 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.

	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 4.6: Maximum measurement error: compression

	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

Table 4.7: Maximum measurement error: decompression

4.2.4 Simulation

While various techniques exist for estimating energy consumption by simulation, these tools are calibrated only on specific systems. Furthermore, without an accurate custom simulator, it can be difficult to obtain accurate energy estimates. To examine the feasibility of a simulator-based approach, I obtained the execution-driven simulator known as SimpleScalar [9]. 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 [4]. 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. As such, cycle counts are underestimated by the simulator. Using Skiff execution time as baseline, simulated cycle count is off by a factor of 1.8 - 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.

Multiplying event counts generated by the simulator with the actual measured CPU and memory energy of operations, I predicted the energy that would be consumed by several applications. Events were grouped into the following classes: computation, load hit, load miss, store hit, buffered store miss, unbuffered store miss, and network. The following formula was used:

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

Figure 4-6 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 1.1 nJ per executed instruction to account for error in the model lowers the difference between observed and predicted energy to 0.4% - 18%.

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 4-6 is close to other studies of small programs [48]. As discussed in section 4.2.3, energy measured with

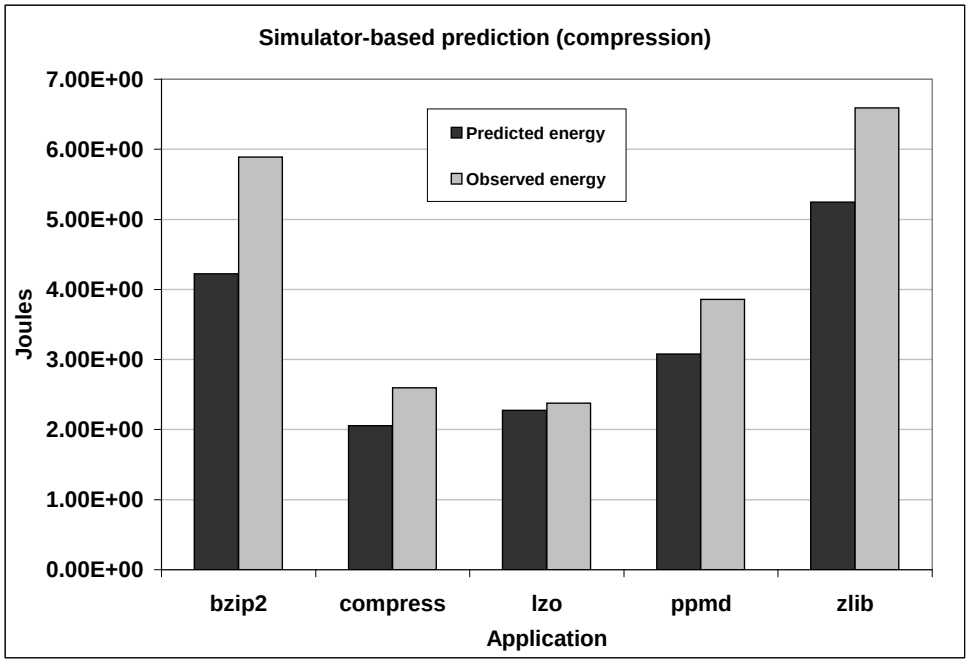

Figure 4-6: Using a simulator to predict energy

hardware may be inaccurate as well, but has the advantage of corresponding more closely with reality.

Executing programs on a simulator requires more time than running them on hardware. Hosted on a 1Ghz 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 seconds (not including the time it takes to rearrange the multimeter probes). Whether or not to use a simulator, then, is dependent on desired accuracy and the relative convenience of obtaining actual measurements. For greatest accuracy, energy measurements in this thesis will be made exclusively with hardware.

4.3 Motivation and misconception

With the above energy measurement methodology we can 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 pre-transmission compression of data to reduce overall energy. Initial experiments reveal 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 4.4.

4.3.1 High communication-to-computation ratio...

To quantify the gap between wireless communication and computation, I have measured wireless idle, send, and receive energies on the Skiff platform. Unfortunately, in the default "managed" configuration, all 802.11b cards using a particular channel must arbitrate to share a single wireless access point and maximum bandwidth cannot be realized. To minimize the effect of this arbitration phase, I created an ad-hoc network of two wireless nodes. I streamed UDP packets from one node to the other; UDP was used to eliminate the effects of waiting for an ACK. This also insures that receive tests measure only receive energy and send tests measure only send energy. This setup is intended to find the minimum network energy by removing arbitration delay and the energy of TCP overhead.

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 4-7 and are consistent with other studies [26]. The card is set to its maximum speed of 11 Mb/s and two tests are conducted. In the first, the Skiff communicates with a wireless card mere inches away and achieves 5.70 Mb/sec. In the second, the second node is placed as far from the Skiff as possible without losing packets. Only 2.85 Mb/sec is seen. These two cases bound the performance of my 11 Mb/sec wireless card; typical performance should be somewhere between them.

Next, a microbenchmark is used to determine the minimum energy for an ADD instruction. I drew on the work of [39], using Linux boot code to bootstrap the processor; select a cache configuration; and launch assembly code unencumbered by an operating system. I followed one thousand ADD instructions by an unconditional branch which repeats them. This code was chosen and written in assembly language to minimize effects of the branch. Once the program has been loaded into instruction cache, the energy used by the processor for a single add is 0.86 nJ.

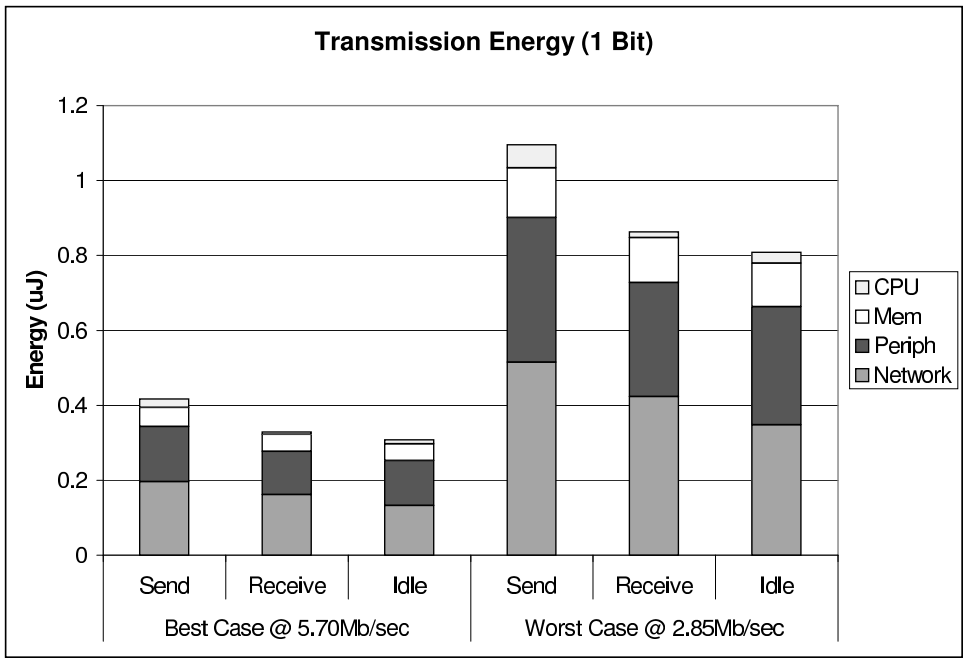

Figure 4-7: Communication energy

From these initial network and ADD 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 is consistent with the communication/computation gap reported in [30] and is likely to grow as radio requirements remain relatively static while processors and memory becomes faster and more energy-efficient. This gap of 2-3 orders of magnitude 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.

4.3.2 ...is not exploited by popular compressors

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 4.8. 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 examined in this

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

Table 4.8: Total Energy of an ADD

thesis are more than a mere series of ADDs; the variety of instructions (especially Loads and Stores) in compression applications shrinks the ratio further.

Figure 4-8 shows the energy required to compress the first (easily compressible) megabyte of the Calgary Corpus and transmit it via wireless ethernet. In the figures, 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. The popular compression applications discussed in Section 4.1 are used with their default parameters, and the right-most bar shows the energy of merely copying the uncompressed data over the network. In several cases the energy to compress the file approaches or outweighs the energy to transmit it! We also see that the 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. Figure 4-9 shows 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 designing a low-energy lossless compression scheme. Though the applications are "allowed" to execute 485-1267 additions for every bit removed from the file, they execute far less and still approach the energy of sending uncompressed data. Section 4.4 will discuss how such high net energy is possible despite the motivating observations.

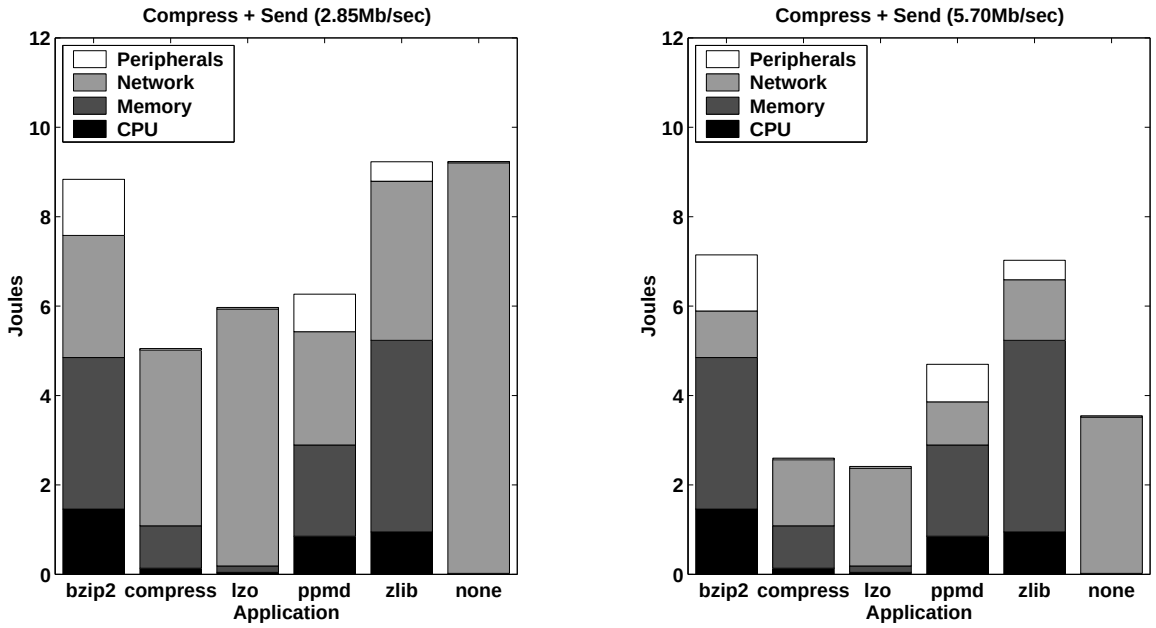

Figure 4-8: Energy required to send compressible 1MB file

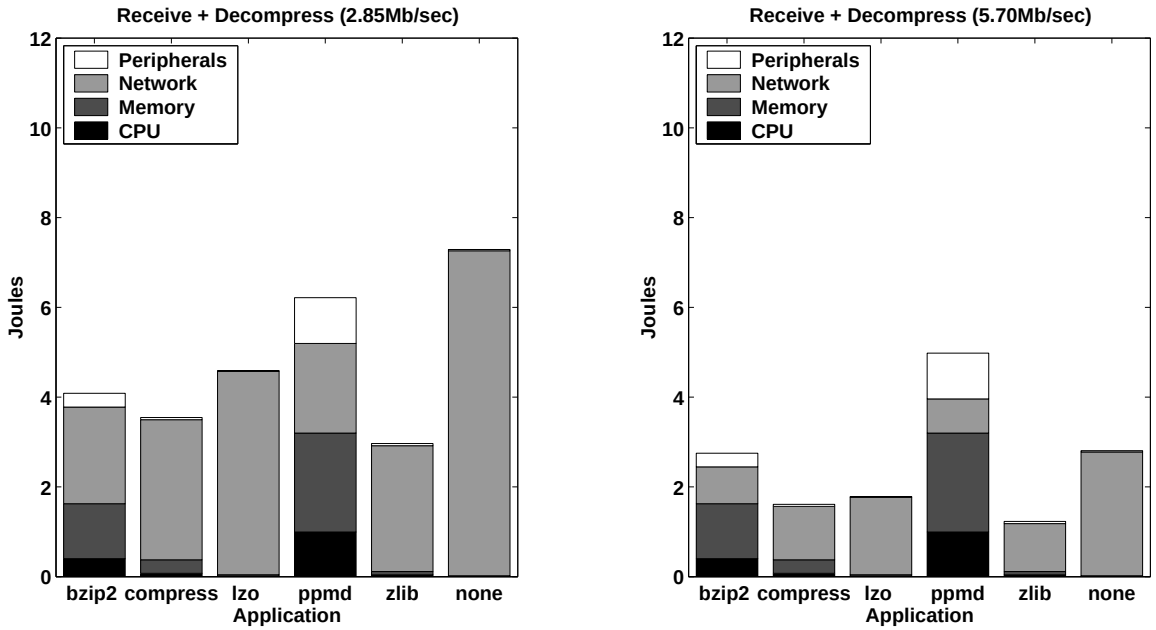

Figure 4-9: Energy required to receive a compressible 1MB file

Application	bzip2	compress	lzo	PPMd	zlib
Compress: instructions per bit removed	116	10	7	76	74
Decompress: Instructions per bit restored	31	6	2	10	5

Table 4.9: Instructions per bit

4.4 Energy analysis of popular compressors

We will look deeper into the applications to discover why they cannot exploit the communication - computation gap. To perform this analysis, we rely on empirical observations on the Skiff platform as well as the SimpleScalar simulator. Since SimpleScalar is beta software, we will handle the statistics it reports with caution, using them to explain the *traits* of the compression benchmarks rather than to describe their precise execution on a Skiff.

4.4.1 Instruction count

Since the applications do not realize their energy-saving potential as predicted in Section 4.3.1, we begin by looking at the number of instructions each requires to remove and restore a bit (Table 4.9). 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 superfluous copying due to buffering (in contrast with *compress* and *zlib*). Since all five are well within the 485-1267 computation instructions allowed for each bit removed or restored, it is clear that a compression application does more than simple addition.

4.4.2 Memory hierarchy

One noticeable similarity of the bars in Figures 4-8 and 4-9 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 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-by-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 in a similar pattern; instead of a series of loads or stores, however, each load is followed by a store to the same location which dirties the block which forces 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 insure the correct number of load or store instructions is generated. Address generation instructions are ignored 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 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 batching writes in the store buffer.

	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	0.86

Table 4.10: Measured memory energy vs ADD energy

Table 4.10 shows that hitting in the cache requires little more energy than an ADD (Table 4.8), while a load miss requires over 145 times as much energy as 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.

4.4.3 Minimizing memory access energy

One way to minimize misses is to reduce the memory requirements of the application. Figure 4-10 shows the effect of varying memory size on compression/decompression time

and compression ratio ("effort" level in *zlib* remains constant). In a system with unlimited resources, one would choose the application in the lower left corner to minimize time and maximize compression, favoring either the compression or decompression operation as needed. This figure allows one to see the memory implications of such a decision.

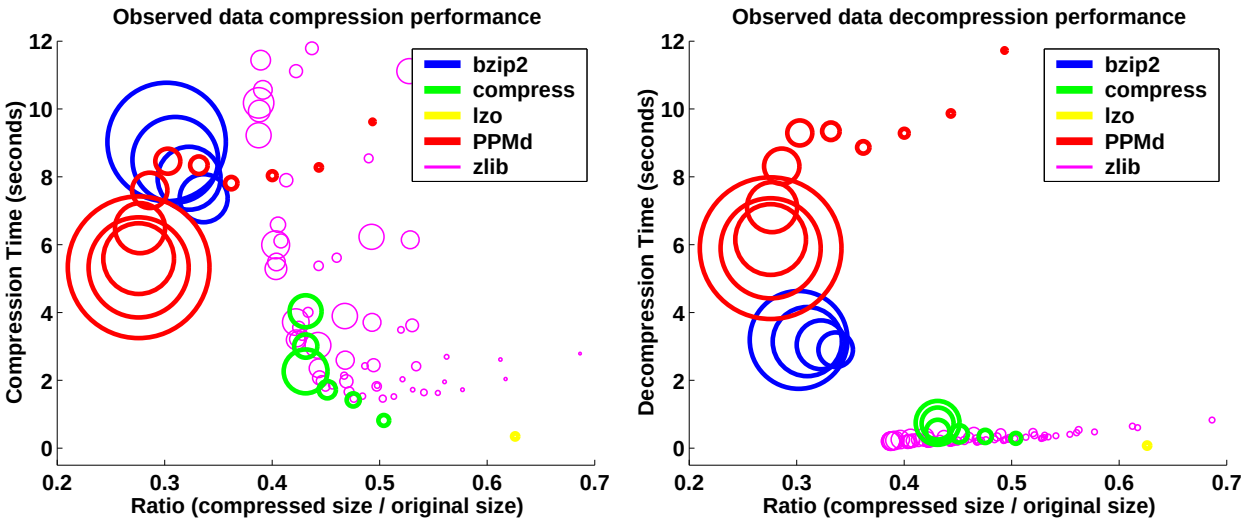

Figure 4-10: Memory, time, and ratio. Memory footprint is indicated by area of circle; footprints shown range from 3KB - 8MB

Alternatively, if memory is constrained, one's selection must be made from a reduced set of the datapoints. Figures 4-11 and 4-12, an expanded version of Figures 4-8 and 4-9, show the energy implications of varying the footprint of a given application. Along with energy due to default operation (labeled "bzip2-900," "compress," "lzo-16," "ppmd-10240," and "zlib-6"), Figures 4-11 and 4-12 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*, the application to beat, 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.

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, with more memory

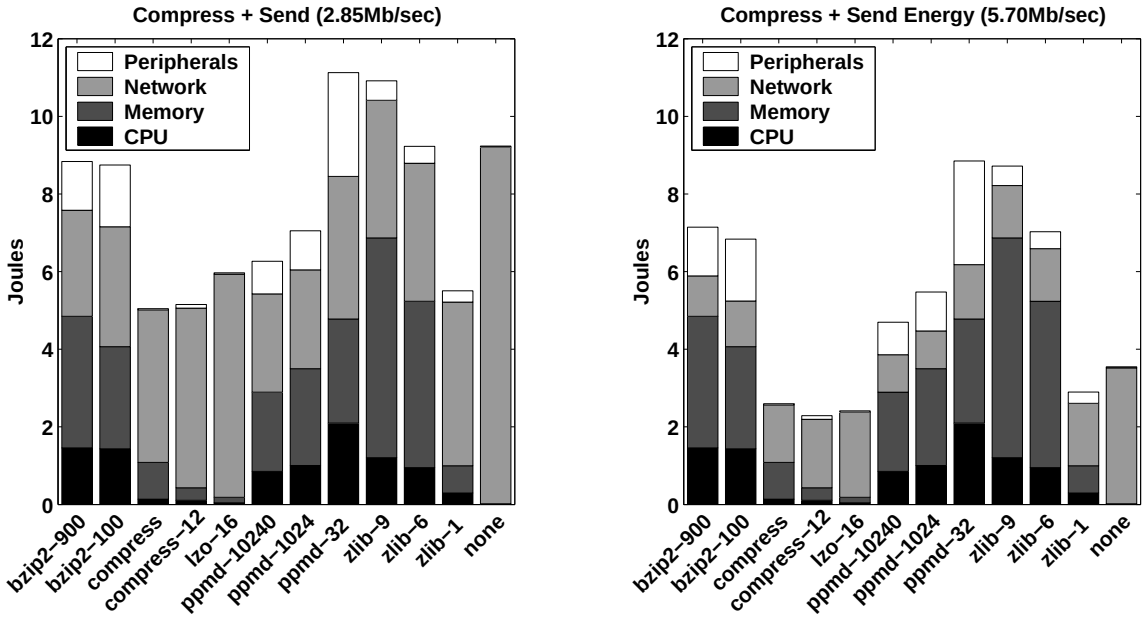

Figure 4-11: Energy required to send compressible 1MB file

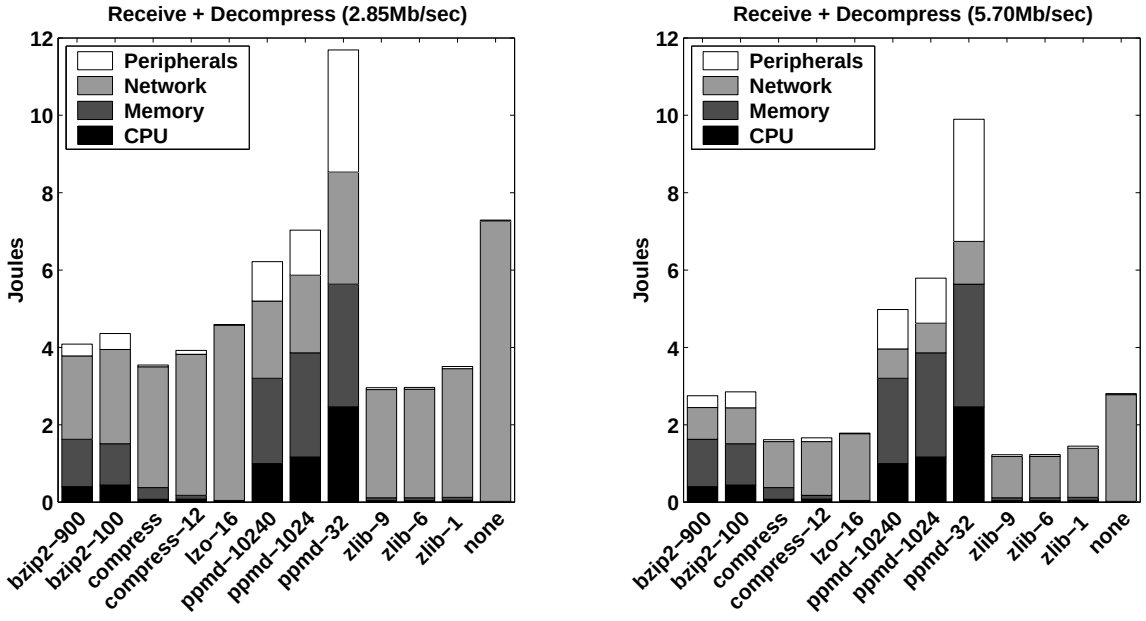

Figure 4-12: Energy required to receive a compressible 1MB file

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. For example, when *compress* has at least 800 KB, the implementation is able to use a more efficient hashtable with less collision and gains a marked speed improvement. This implementation is represented by the biggest circle in Figure 4-10 and by the "compress" bar in Figures 4-11 and 4-12.

bzip2 uses a large amount of memory, but for good reason. While I was able to implement its sort with the quicksort routine from the standard C library to save significant memory, the compression takes over 2.5 times as long due to large constants in the runtime of the more traditional quicksort in the standard library. This slowdown occurs even when 16 KB block sizes[44] are 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 as the work becomes more productive, but there are points at which adding memory has the expected effect of slowing down compression as a deeper tree must be examined. Finally, compression ratio approaches a limit, and additional memory serves to improve speed. This behavior is 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 effects 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 side of the figure underscores the valuable asymmetry of some of the applications. Often decompressing data is a simpler operation than compression which 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 its 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, so decompression requires more time.

Each of the benchmarks 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. Figures 4-14 and 4-13 look beyond the footprint of the application and examine its locality, but the rates in Figure 4-14 belie the absolute numbers in Figure 4-13. PPM and BWT are known to be quite memory intensive and this is evident in their large number of cache accesses. Though LZ77 is local by nature, the large window and data structures hurt its cache performance for *zlib*. *LZO* also uses LZ77, but is designed to require just 16KB of memory and goes to main memory orders of magnitude less than its slower competitors.

The followup to the SA-110 (the SA-1110 used in Compaq's iPAQ handheld computer) has only an 8KB data cache which would exaggerate any penalties observed here. Though large, low-power caches are becoming possible (the X-Scale has two 32KB caches), as long as the energy of going to main memory remains so much higher, we must be concerned with cache misses.

Both *compress* and *zlib* read data into a buffer before processing it and buffer the data again before output. If we are compressing a source that exists in memory, there is no need for such an input buffer – the memory saved can be used to increase the cacheable history available to the algorithm and improve compression ratio with little effect on speed. Output buffers are needed to assemble bits for output, but an overly large buffer also cuts into the cacheable memory available to the compression engine.

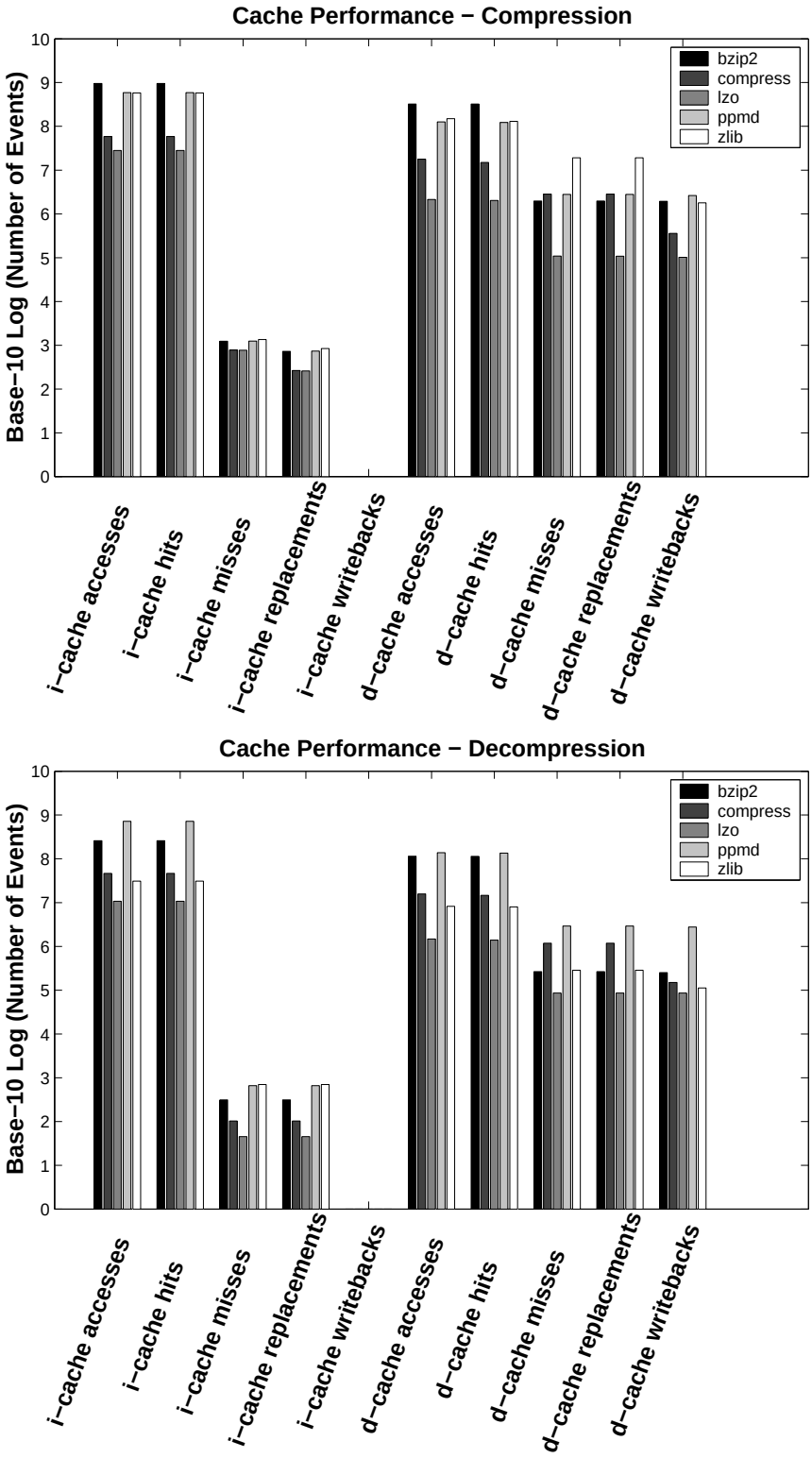

Figure 4-13: Cache performance: absolute counts

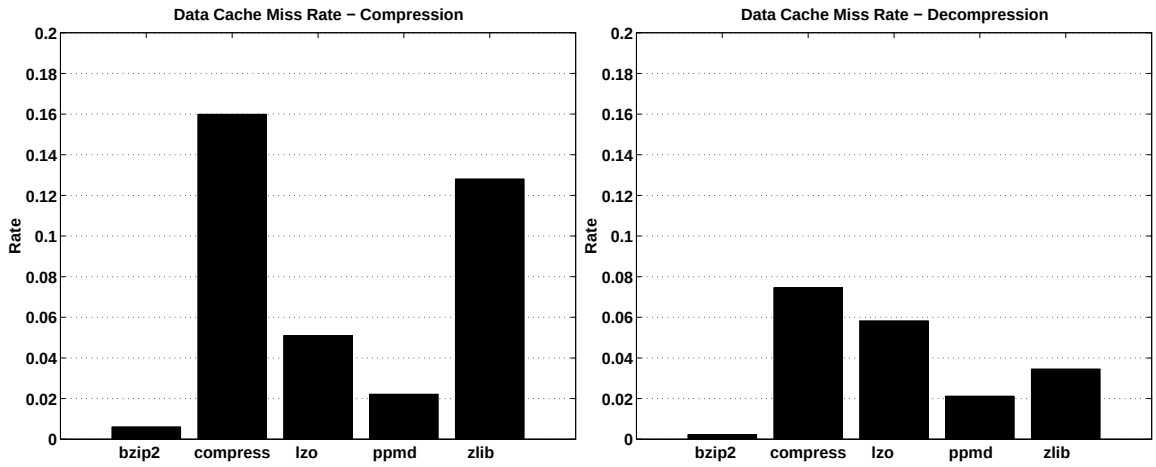

Figure 4-14: Cache performance: data cache miss rate

4.4.4 Instruction mix

One rough characterization of any application is its instruction mix. Figure 4-15 shows the static and dynamic instruction mix of the five compression applications. In the figure, "computation" 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 sort-based.

Branches are not a big problem in the StrongARM's short pipeline, but the not-taken prediction is rather poor (Figure 4-16). As embedded pipelines grow, effective branch prediction will be needed to minimize the number of flushes that must occur.

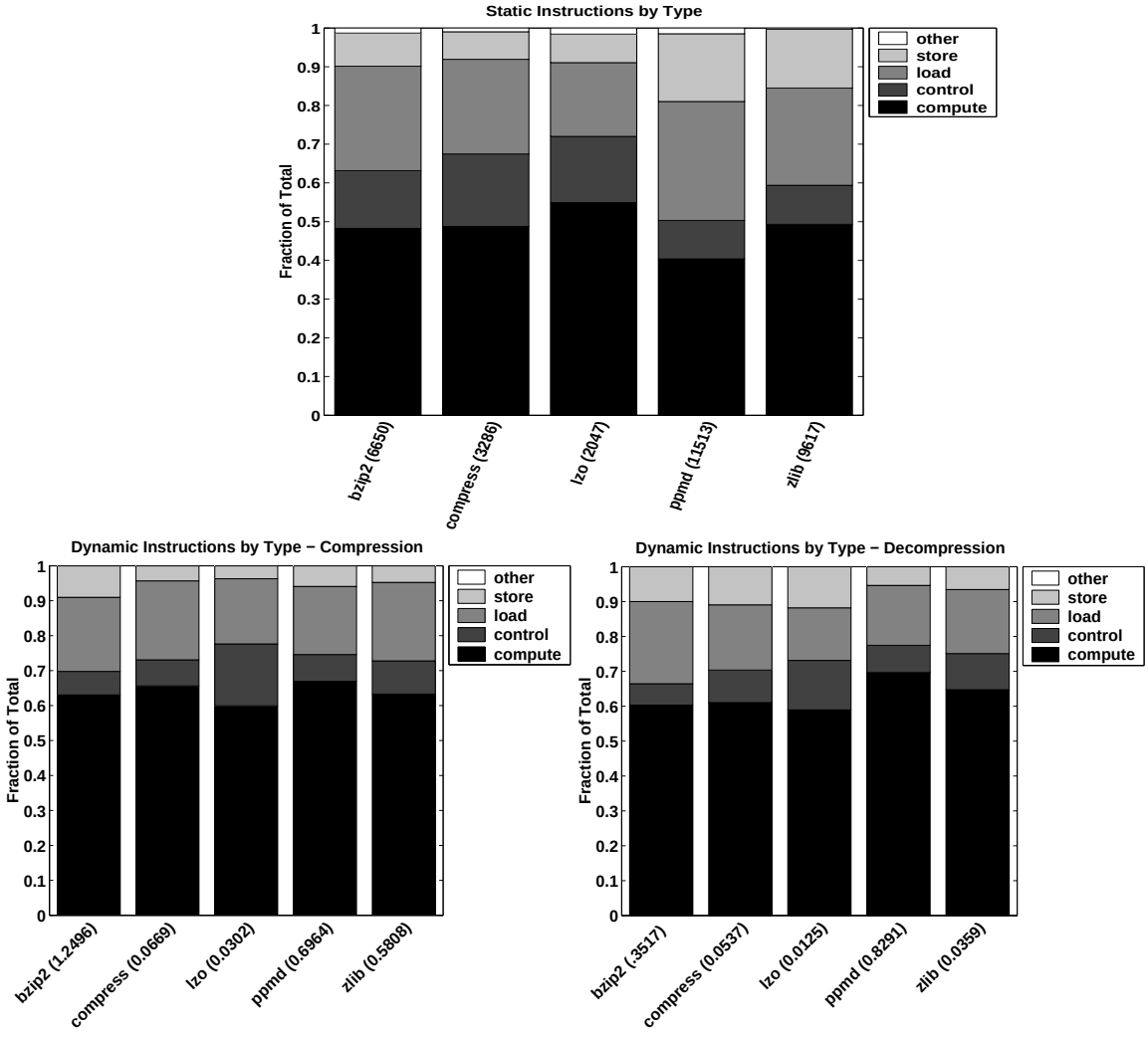

Figure 4-15: Instruction Mix. Number in parenthesis shows absolute number of instructions (static) and billions of absolute instructions (dynamic)

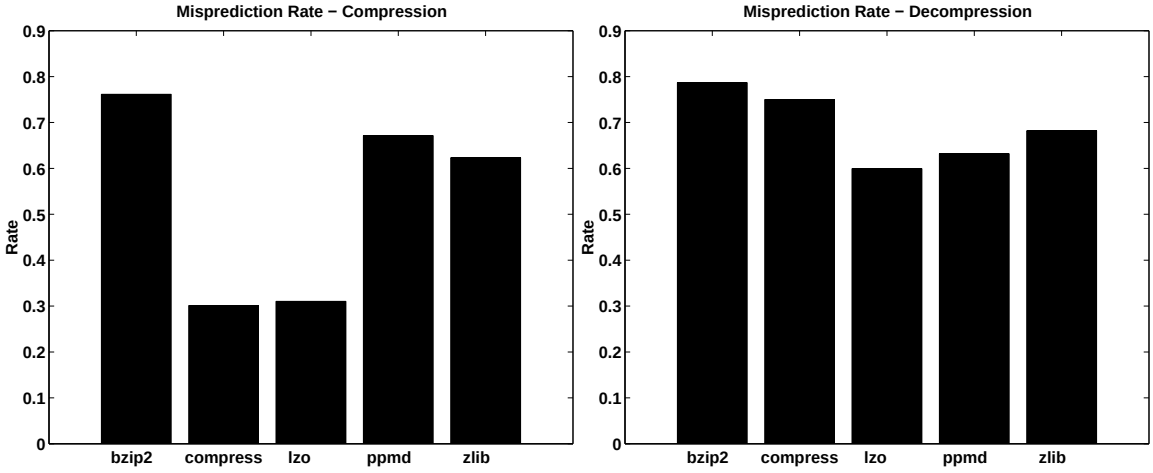

Figure 4-16: Branch behavior

4.5 Summary

We can now add energy to the metrics used to rank compression applications. Table 4.11 contains this ranking for a 5.70 Mb/sec system with more desirable attributes (more compression, faster speeds, lower memory, and lower energy) at the top of each column. Note that energy perfectly mirrors speed. This is not surprising when we note the power used by each application. Figures 4-17 and 4-18 show that CPU and memory power is relatively constant for each application, making energy largely time-dependent, though the power is divided in different ways among CPU, memory, and peripherals. They also show the substantial power of the Skiff in an idle state, waiting at a shell prompt for user input.

	Compress	Decompress	Compress	Decompress	Compress	Decompress
Ratio	**Speed**		**Memory**		**Energy**	
ppmd	lzo	lzo	lzo	lzo	lzo	lzo
bzip2	compress	zlib	zlib	zlib	compress	zlib
zlib	ppmd	compress	compress	compress	ppmd	compress
compress	zlib	bzip2	bzip2	bzip2	bzip2	bzip2
lzo	bzip2	ppmd	ppmd	ppmd	zlib	ppmd

Table 4.11: Ranking compression applications by four metrics

Compress	Decompress
lzo	zlib
compress	compress
ppmd	lzo
zlib	bzip2
bzip2	ppmd

Table 4.12: Ranking energy of compression applications including network energy

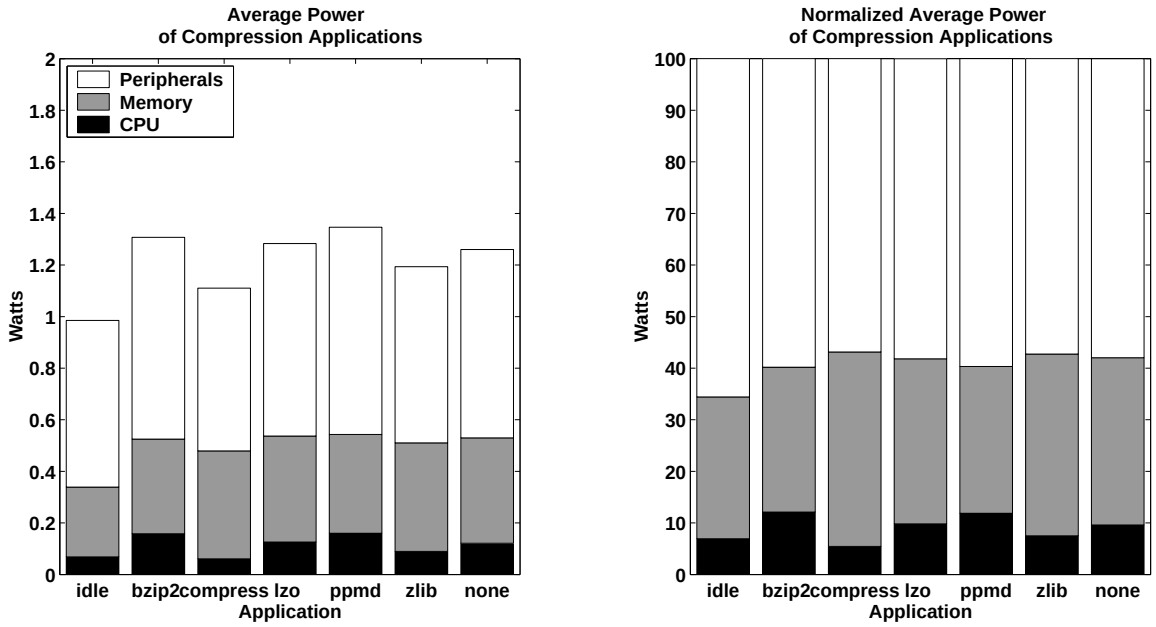

Figure 4-17: Average power of compression applications

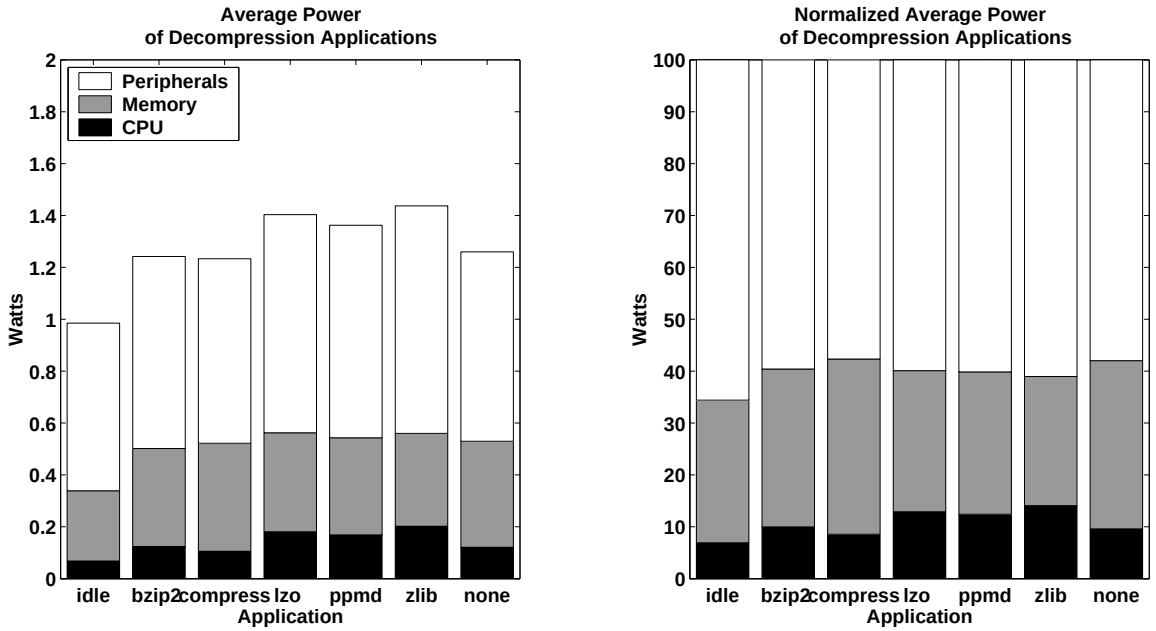

Figure 4-18: Average power of decompression applications

When these five applications are used to compress data before network transmission or do the opposite (receive and decompress data), the ranking of total energy is as shown in Table 4.12. These rankings are similar to the speed rankings in 4.11, but differences appear due to compression. For example, decompression energy of *zlib*, *LZO*, and *compress* does not mirror speed since fewer bits do indeed decrease overall energy. This change, which successfully walks the tightrope of computation versus communication cost is an encouraging result. Despite the greater energy needed to decompress the data, the decrease in receive energy makes the net operation a win. More importantly, it shows that reducing energy is not as simple as choosing the fastest or best-compressing program.

We can generalize the results obtained by 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 quickly energy of components will change over time. Even predicting whether a certain component's energy usage will grows or shrink can be difficult. Many researchers envision ad-hoc networks made of nearby nodes. 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. For a given mobile CPU design, planned manufacturing improvements may lower its relative power and energy. On the other hand, processors once used only in desktop computers are being recast as mobile processors. Though their power may be much larger than that of the Skiff's StrongARM, higher clock speeds may reduce energy.

Figures 4-19 through 4-22 show the effect on overall compression energy as the ratio between component energies vary. The trends are produced by beginning with energy measured by microbenchmarks. These energy measurements are multiplied by simulated event counts as in Section 4.2.4. Scaling is displayed as a ratio to de-emphasize the absolute energy and stress the relative energy. Each successive ratio represents an additional 10% reduction from the original energy. Figure 4-19 shows that reduction in CPU energy brings two of the applications closer to their peers in overall energy, becoming candidates for low-energy compression where they once were ignored. *zlib* benefits more from a reduction in memory access energy as shown by Figure 4-20. This figure and Figure 4-21 show that with significant energy scaling in the right system components, the lowest-energy application

does not remain fixed. 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. Figure 4-22 shows that if network energy decreases relative to the rest of the system, the choice of applications remains almost the same – determined by their processing and memory needs.

The next chapter will apply the results of this experimentation to create a data compression system which is more energy efficient than using these popular tools in their current form. We have seen energy can be saved by compressing files before transmitting them over the network, but one must be mindful of the energy required to do so. Compression and decompression energy may be minimized through wise use of memory (including efficient data structures and/or sacrificing compression ratio for cacheability). Finally, one must be aware of evolving hardware's effect on overall energy.

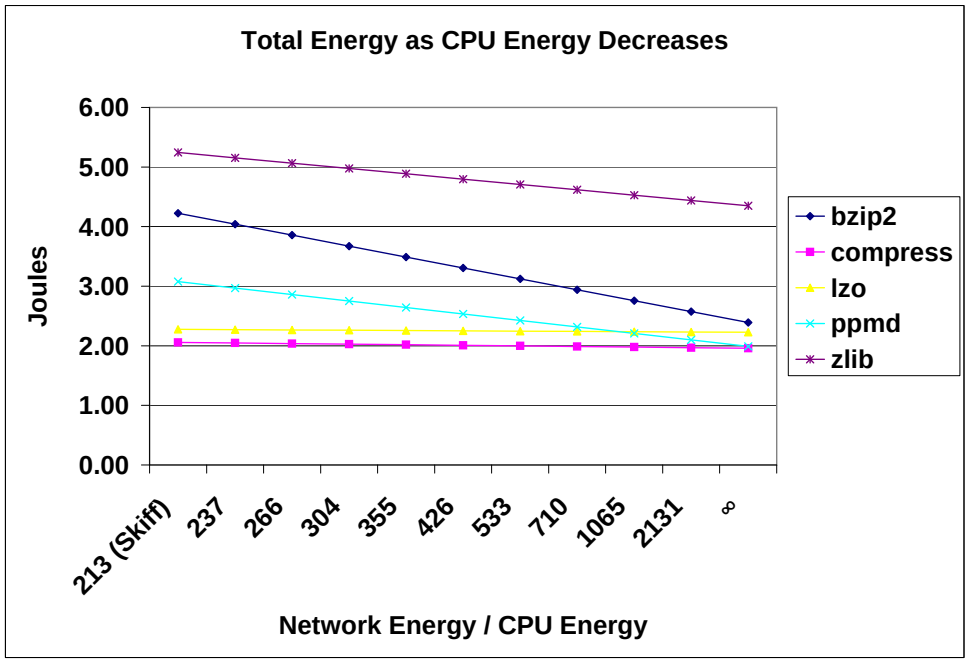

Figure 4-19: Total energy as CPU energy decreases

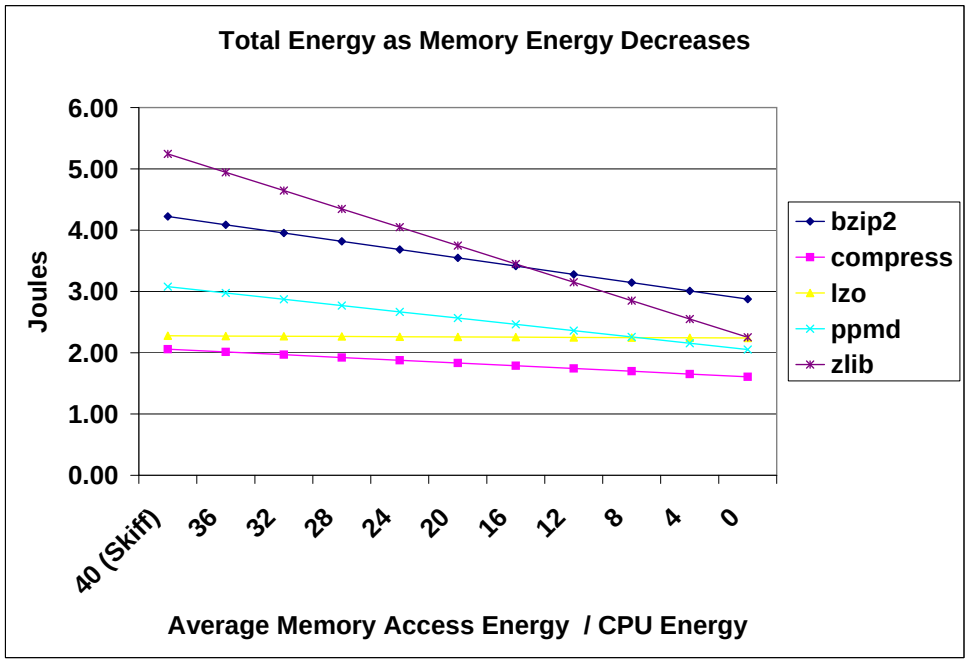

Figure 4-20: Total energy as memory energy decreases

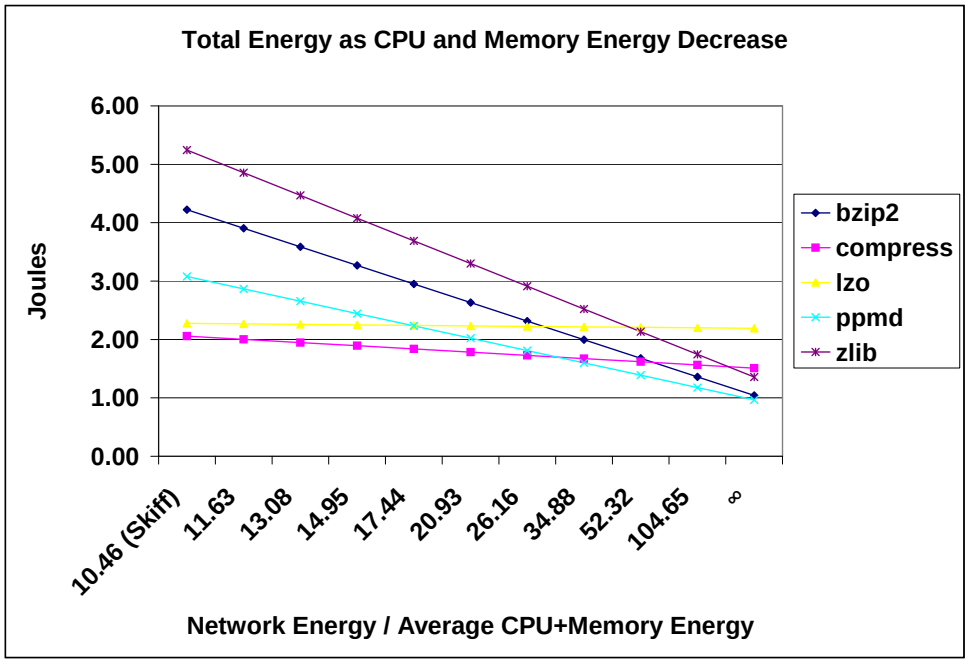

Figure 4-21: Total energy as both CPU and memory energy decreases

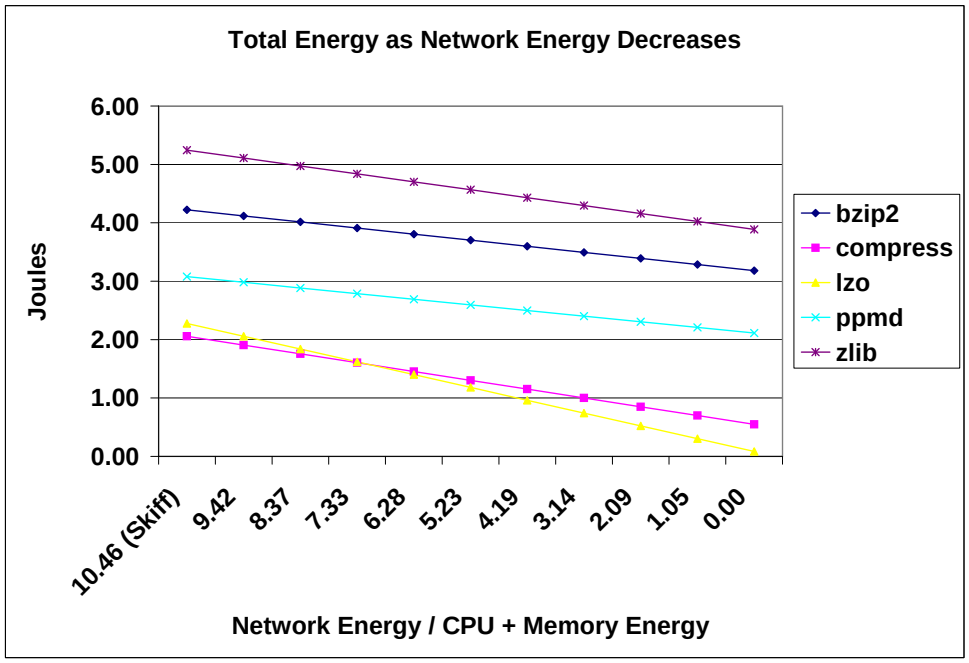

Figure 4-22: Total energy as network energy decreases

Chapter 5

Reducing the energy of transmitting compressed data

Using insight gained from the experiments in Chapter 4, one can synthesize a data compression system which strikes a balance between energy required to compress or decompress data and energy required to send that data on a wireless network.

5.1 Understanding cache behavior

It has been shown that load and store cache misses on the Skiff require a large amount of memory compared to computation. Thus, it is important to understand the cache behavior of a compression application so that it can be made more efficient.

Figure 5-1 shows the compression energy of several successive optimizations of the *compress* program. The baseline implementation is itself an optimization of *compress*. This optimized version is based on *compress* version 4.0 rather than version 4.2.4 which was studied in Chapter 4. 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. Since the Skiff data cache is small (16KB) compared to the size of the hash table (≈270KB) 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 number of cache misses. As long as the extra memory is available to enable this optimization, about 0.53 Joules 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. 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 its 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{
   int fcode;
   unsigned short code;
}table[SIZE];
```

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

Each `entry` stores the same information, but the array on the left wastes four bytes per entry. Two bytes are used only to align the short `code`, and overly-wide types result in twelve wasted bits in `fcode` and four bits wasted in `code`. Using bitfields, the layout on the right contains the same information yet fits in half the space. If the entry were not four bytes, it would need to contain more members for alignment. Code would become 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 56985 data cache misses compared with 734195 in the un-packed structure; a 0.0026% miss rate versus 0.0288%. The energy benefit for *compress* with the compact layout is negligible since 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. Eleven bit codes are necessary even with the compact layout in order to reduce the size of the data structure. Despite a dictionary with half the size, the number of bytes to transmit increases by just 18%. Energy, however, is lower with the smaller dictionary due to less energy spent in memory and increased speeds which reduce peripheral overhead.

Similarly, data structures which contain groups of pointers (which are four bytes long regardless of what they point to) can be converted to groups of 16-bit integers to halve storage space when as addressability needs 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 saves up more than 50% of the energy of a load miss [15]. 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-block fetches would be just as useful and much faster.

5.2 Exploiting the sleep mode

It has been noted that 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

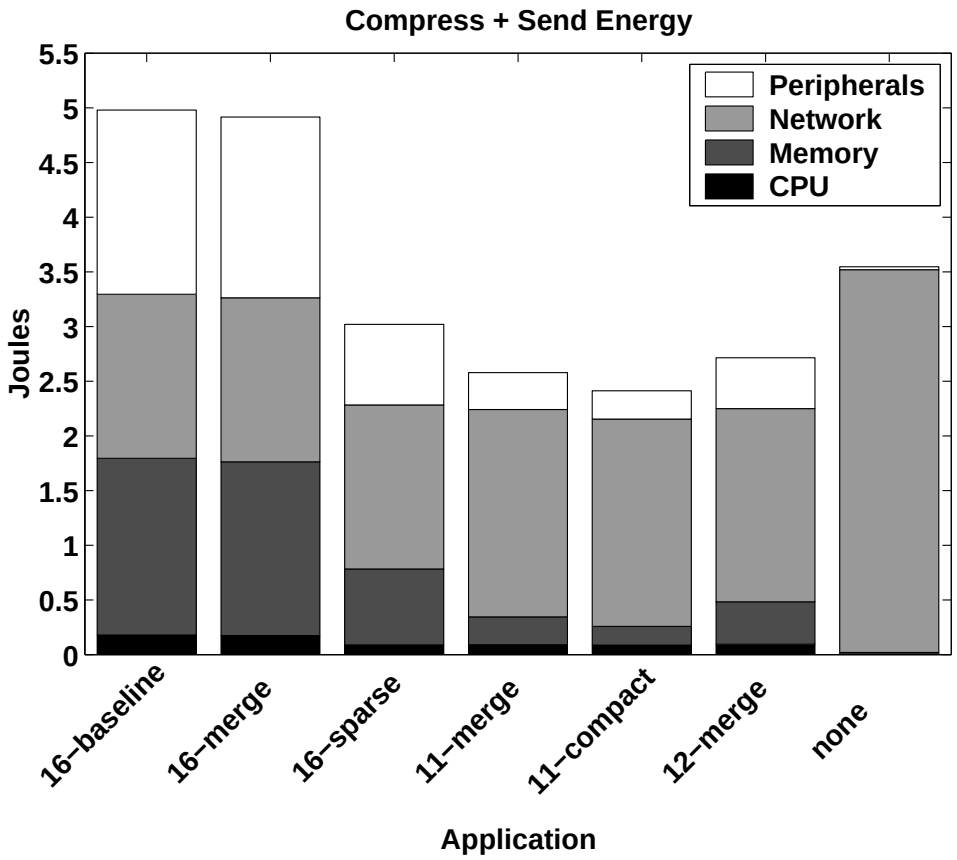

Figure 5-1: Optimizing *compress*

the low-power idle state [35]. Figures 5-2 and 5-3 show the effect of this analysis. It is interesting to note that, assuming a low-power idle mode can be entered once compression is complete, one's choice of compression strategies will vary. With its high 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.

5.3 Reducing energy on the Skiff

Consider a wireless client similar to the Skiff communicating with a server. Recalling Figures 4-11 and 4-12, and recognizing that the Skiff has no low-power sleep mode, we choose "compress-12" (the twelve-bit codeword LZW compressor) as it provides the lowest total compression energy over all communication speeds.

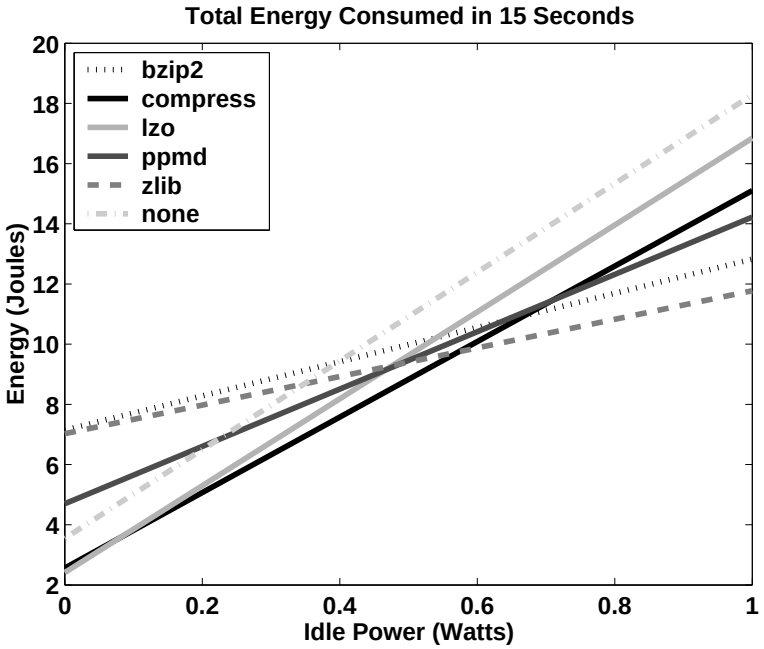

Figure 5-2: Compression + Send energy consumption with varying sleep power

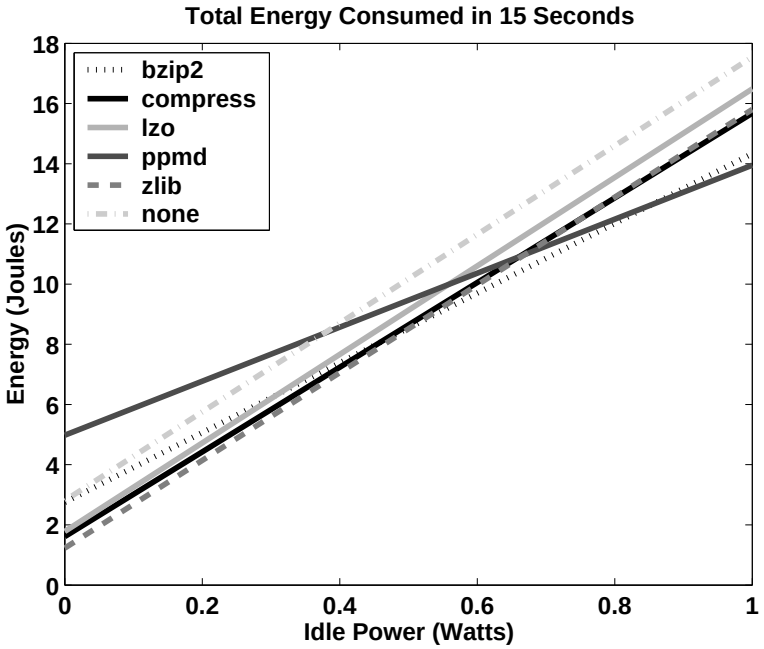

Figure 5-3: Receive + Decompression energy consumption with varying sleep power

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. Figure 5-4 reminds us that regardless of the effort and memory parameters used by *zlib* to compress data, the scheme is quite easy to decompress. In the figure, each invocation is specified by the code x-y-z corresponding to a window size of 2^x, memory of 2^y bits, and effort level z. It is as easy to decompress the 15-9-9 data as the rest (and usually faster since less bits are received).

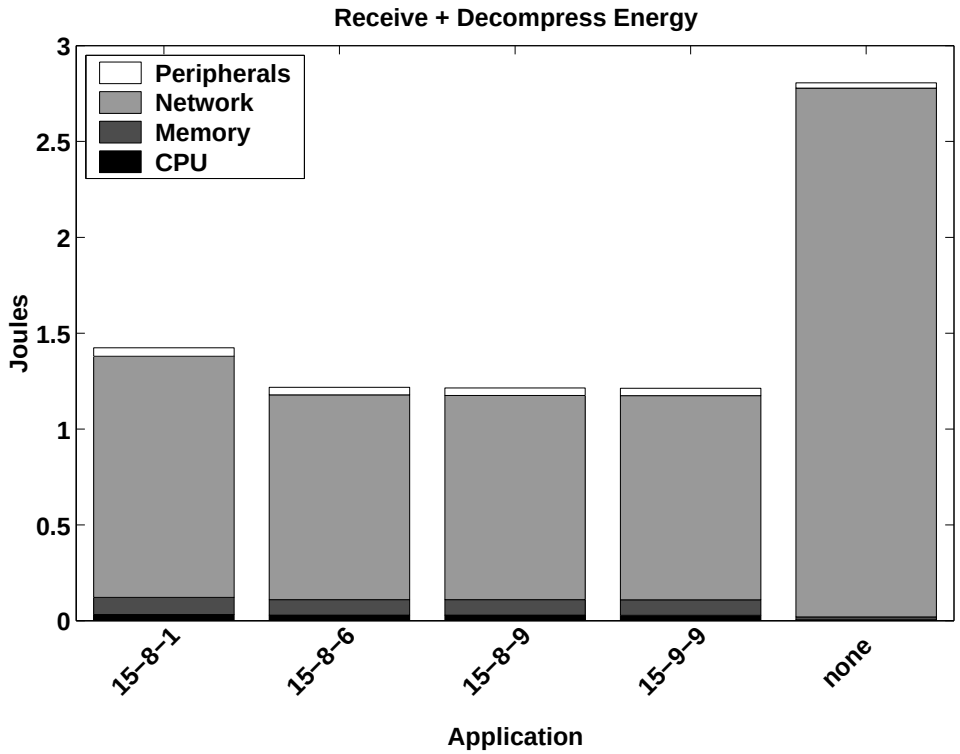

Figure 5-4: Receive + Decompression energy stays constant across *zlib* parameters

The decompression energy difference between *compress*, *LZO*, and *zlib* is minor at 5.70 Mb/sec, but more noticeable at slower speeds. Figure 5-5 shows several other combinations of compressor and decompressor at 5.70 Mb/sec. "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. 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.

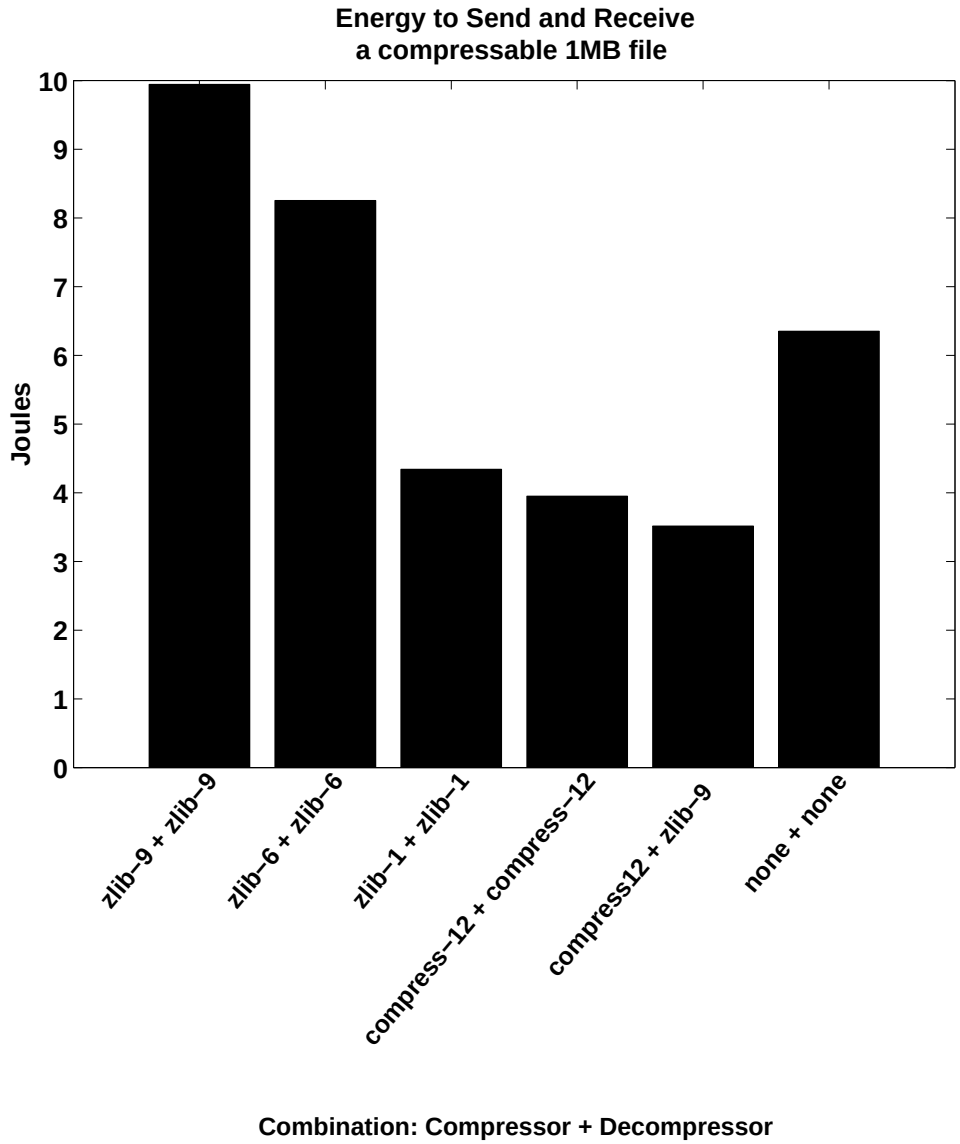

Figure 5-5: Choosing an optimal compressor-decompressor pair

Chapter 6

Conclusion and future work

This thesis has examined the energy implications of compressing data before transmitting it over a network. Specifically, lossless compression is the focus since related work in sacrificing quality for compression energy does not apply to data which must be transmitted without loss.

An understanding of both computer architecture and lossless compression algorithms is essential to minimizing the energy of the compress-transmit and receive-decompress tasks. On the Skiff, choosing an optimal, asymmetric compressor-decompressor pair reduces energy by 11% compared to the lowest energy symmetric pair. When comparing against the common *zlib* compression library, energy is reduced by 57%. 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 relative energy of system components change with technological advancement.

Future work in this area should examine sensitivity to the type of data. If one knows *a priori* that data is uncompressable (or can determine this fact dynamically), it is likely to change one's choice of compression schemes. Sensitivity to the latency requirements of a given task are crucial as well. The results presented in this work are most applicable to the transfer of large files for which one may be willing to tolerate latency. Interactive work requires elimination of perceived delay, and short realtime messages are unlikely to compress well unless they are correlated to provide extensive history. Thus, algorithms which require long warm-up times or large history structures are not likely to be useful.

Further optimization and quantification is possible. Newer ARM processors provide a lock-down area of cache which might be used to store frequently occurring patterns, reducing costly cache misses. More complicated programs such as *PPMd* and *bzip2* spend most of their energy in the memory and CPU. If an optimization were discovered in these implementations, it is likely to have a greater effect than those optimizations presented for compress and *zlib*. Distributed systems research into protocols and systems for negotiating low-energy data transfer and server-side translation to easily-decompressable formats is another potential future line of research.

Optimizing an entire network of devices is another possible desire. Perhaps the sender is not a wall-powered server but another handheld device. Perhaps a poor or crowded communication channel limits the size or speed of a transmission. Many combinations exist for which optimal energy and performance points must be found. How collections of devices might find their desired operating point is another area for research.

Most importantly, 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 lives will grow in importance. Careful, perhaps automated, evaluation of a platform's relative component energy can help one choose the most energy aware lossless compression scheme.

Bibliography

[1] Advanced RISC Machines Ltd (ARM). *Writing Efficient C for ARM*, Jan. 1998. Application Note 34.

[2] Agilent Technologies. *Agilent 34401A Multimeter: User's Guide*. Fifth edition, Apr. 2000.

[3] R. Arnold and T. Bell. A corpus for the evaluation of lossless compression algorithms. In *Data Compression Conference*, 1997.

[4] T. M. Austin and D. C. Burger. SimpleScalar version 4.0 release. *Tutorial in conjunction with 34th Annual International Symposium on Microarchitecture*, Dec. 2001.

[5] T. Bell and D. Kulp. Longest match string searching for Ziv-Lempel compression. Technical Report 06/89, Department of Computer Science, University of Canterbury, New Zealand, 1989.

[6] T. Bell, M. Powell, J. Horlor, and R. Arnold. The Canterbury Corpus. http://www.corpus.canterbury.ac.nz/.

[7] T. Bell, I. H. Witten, and J. G. Cleary. Modeling for text compression. *ACM Computing Surveys*, 21(4):557–591, 1989.

[8] J. Bilmes, K. Asanović, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *11th ACM International Conference on Supercomputing*, July 1997.

[9] D. C. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

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

[11] F. Chang, K. Farkas, and P. Ranganathan. Energy-driven statistical profiling: Detecting software hotspots. In *2nd Workshop on Power-Aware Computer Systems, HPCA-8*, February 2002.

[12] D. J. Craft. Data compression in ASIC cores. *IBM Journal of Research and Development*, 42(6), 1998.

[13] M. Effros. Ppm performance with bwt complexity: A new method for lossless data compression. In *Data Compression Conference*, 2000.

[14] J. Flinn. *Extending Mobile Computer Battery Life through Energy-Aware Adaptation*. PhD thesis, Carnegie Mellon University, Dec. 2001. TR No. CMU-CS-01-171.

[15] J. Flinn, K. I. Farkas, and J. Anderson. Power and energy characterization of the Itsy pocket computer (version 1.5). Technical Report TN-56, Compaq Computer Corporation, February 2000.

[16] J. Flinn and M. Satyanarayanan. Powerscope: A tool for profiling the energy usage of mobile applications. In *2nd IEEE Workshop on Mobile Computing Systems and Applications*, February 1999.

[17] J. Gailly, Maintainer. `comp.compression` Internet newsgroup: Frequently Asked Questions, Sept. 1999.

[18] J. Gilchrist. Archive comparison test. http://compression.ca.

[19] P. J. Havinga. Energy efficiency of error correction on wireless systems. In *IEEE Wireless Communications and Networking Conference*, Sept. 1999.

[20] J. Hicks et al. Compaq personal server project, 1999. http://crl.research.compaq.com/projects/personalserver/default.htm.

[21] Hyperspace Communications, Inc. Mod_gzip. Available at: http://www.ehyperspace.com/htmlonly/products/mod_gzip.html.

[22] *IBM Journal of Research and Development*, 45(2), 2001. Preface by Richard E. Harper, Guest Editor.

[23] Intel Corporation. *SA-110 Microprocessor Technical Reference Manual*, December 2000.

[24] Intel Corporation. *Intel StrongARM SA-1110 Microprocessor Developer's Manual*, October 2001.

[25] V. Jacobson. RFC 1144: Compressing TCP/IP headers for low-speed serial links, Feb. 1990.

[26] K. Jamieson. Implementation of a power-saving protocol for ad hoc wireless networks. Master's thesis, Massachusetts Institute of Technology, Feb. 2002.

[27] P. Jannesen et. al. (n)compress. available, among other places, in Redhat 7.2 distribution of Linux.

[28] B. Jung and W. P. Burleson. A VLSI systolic array architecture for Lempel-Ziv based data compression. In *Proceedings of the International Symposium on Circuits and Systems*, June 1994.

[29] B. Jung and W. P. Burleson. Real-time VLSI compression for high-speed wireless local area networks. In *Data Compression Conference*, March 1995.

[30] R. Krashinsky. Efficient web browsing for mobile clients using HTTP compression. Distributed Operating Systems term project, Massachusetts Institute of Technology, December 2000.

[31] H. Lekatsas, J. Henkel, and W. Wolf. Low-power techniques for code compression in embedded systems. In *Proceedings of Design Automation Conference*, 2000.

[32] D. A. Lelewer and D. S. Hirschberg. Data compression. 19(3):261–297, 1987.

[33] J. Lilley, J. Yang, H. Balakrishnan, and S. Seshan. A unified header compression framework for low-bandwidth links. In *Proceedings of 6th ACM MOBICOM*, Aug. 2000.

[34] J. loup Gailly and M. Adler. zlib. http://www.gzip.org/zlib.

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

[36] J. Montanaro et al. A 160-mhz, 32-b, 0.5-w CMOS RISC microprocessor. *IEEE Journal of Solid-State Circuits*, 31(11), Nov. 1996.

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

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

[39] R. Nathuji. Characterization of DRAM. MIT Advanced Undergraduate Project, June 2000.

[40] M. F. Oberhumer. Lzo. http://www.oberhumer.com/opensource/lzo/.

[41] A. Peymandoust, T. Šimunić, and G. D. Micheli. Low power embedded software optimization using symbolic algebra. In *Design, Automation and Test in Europe*, 2002.

[42] K. Sayood. *Introduction to data compression*. Morgan Kaufman Publishers, second edition, 2002.

[43] J. Seward. bzip2. http://www.spec.org/osg/cpu2000/CINT2000/256.bzip2/docs/256.bzip2.html.
[44] J. Seward. e2comp bzip2 library. http://cvs.bofh.asn.au/e2compr/index.html.
[45] A. Shacham, B. Monsour, R. Pereira, and M. Thomas. RFC 3173: IP payload compression protocol, Sept. 2001.
[46] D. Shkarin. Ppmd. ftp://ftp.elf.stuba.sk/pub/pc/pack/ppmdi1.rar.
[47] D. Shkarin. PPM: one step to practicality. In *Data Compression Conference*, 2002.
[48] A. Sinha and A. Chandrakasan. Jouletrack - a web based tool for software energy profiling. In *38th Design Automation Conference*, June 2001.
[49] A. Sinha, A. Wang, and A. Chandrakasan. Algorithmic transforms for efficient energy scalable computation. In *IEEE International Symposium on Low Power Electronics and Design*, August 2000.
[50] Standard Performance Evaluation Corporation. CPU2000, 2000.
[51] C. N. Taylor and S. Dey. Adaptive image compression for wireless multimedia communication. In *IEEE International Conference on Communication*, June 2001.
[52] C. Thomborson. The V.42bis standard for data-compressing modems. *IEEE Micro*, 12(5), 1992.
[53] A. Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, Apr. 2000.
[54] T. Šimunić, L. Benini, and G. D. Micheli. Energy-efficient design of battery-powered embedded systems. In *IEEE International Symposium on Low Power Electronics and Design*, 1999.
[55] T. Šimunić, L. Benini, G. D. Micheli, and M. Hans. Source code optimization and profiling of energy consumption in embedded systems. In *International Symposium on System Synthesis*, 2000.
[56] M. A. Viredaz and D. A. Wallach. Power evaluation of Itsy version 2.3. Technical Report TN-57, Compaq Computer Corporation, October 2000.
[57] M. A. Viredaz and D. A. Wallach. Power evaluation of Itsy version 2.4. Technical Report TN-59, Compaq Computer Corporation, February 2001.
[58] A. Wolfe and A. Chanin. Executing compressed programs on an embedded RISC architecture. In *25th Annual International Symposium on Microarchitecture*, Dec. 1992.
[59] H. Yang, G. R. Gao, A. Marquez, G. Cai, and Z. Hu. Power and energy impact of loop transformations. In *Workshop on Compilers and Operating Systems for Low Power 2001, Parallel Architecture and Compilation Techniques*, Sept. 2001.