Appears in: The First International Conference on Mobile Systems, Applications, and Services, San Francisco, CA, May 2003. Received Best Paper Award.

Energy Aware Lossless Data Compression

Kenneth Barr and Krste Asanović

MIT Laboratory for Computer Science
200 Technology Square, Cambridge, MA 02139
E-mail: {kbarr,krste}@lcs.mit.edu

Abstract

Wireless transmission of a bit can require over 1000 times more energy than a single 32-bit computation. It would therefore seem desirable to perform significant 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 paper reports on the energy of lossless data compressors as measured on a StrongARM SA-110 system. We show 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 hardware-aware programming optimizations are demonstrated. When applied to Unix *compress*, these optimizations improve energy efficiency by 51%. We also explore the fact that, for many usage models, 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 compressible web data can be reduced 31%. Energy to send harder-to-compress English text can be reduced 57%. Compared with a system using a single optimized application for both compression and decompression, the asymmetric scheme saves 11% or 12% of the total energy depending on the dataset.

1 Introduction

Wireless communication is an essential component of mobile computing, but the energy required for transmission of a single bit has been measured to be over 1000 times greater than a single 32-bit computation. Thus, if 1000 computation operations can compress data by even one bit, energy should be saved. However, accessing memory can be over 200 times more costly than computation on our test platform, and it is memory access that dominates most lossless data compression algorithms. In fact, even moderate compression (e.g. `gzip -6`) can require so many memory accesses that one observes an *increase* in the overall energy required to send certain data.

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. Fidelity can not be sacrificed to reduce energy as is done in related work on lossy compression. Fortunately, an understanding of a program's behavior and the energy required by major hardware components can be used to reduce energy. The ability to efficiently perform efficient lossless compression also provides second-order benefits such as reduction in packet loss and less contention for the fixed wireless bandwidth. Concretely, if n bits have been compressed to m bits ($n > m$); c is the cost of compression and decompression; and w is the cost per bit of transmission and reception; compression is energy efficient if $\frac{c}{n-m} < w$. This paper examines the elements of this inequality and their relationships.

We measure the energy requirements of several lossless data compression schemes using the "Skiff" platform developed by Compaq Cambridge Research Labs. The Skiff is a StrongARM-based system designed with energy measurement in mind. Energy usage for CPU, memory, network card, and peripherals can be measured individually. 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 carelessly applying 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 by applications or network drivers. We 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 opera-

tions, or time to complete them, increases. Finally, a new energy-aware data compression strategy composed of an asymmetric compressor and decompressor is presented and measured.

Section 2 describes the experimental setup including equipment, workloads, and the choice of compression applications. Section 3 begins with the measurement of an encouraging communication-computation gap, but shows that modern compression tools do not exploit the the low relative energy of computation versus communication. Factors which limit energy reduction are presented. Section 4 applies an understanding of these factors to reduce overall energy of transmission though hardware-conscious optimizations and asymmetric compression choices. Section 5 discusses related work, and Section 6 concludes.

2 Experimental setup

While simulators may be tuned to provide reasonably accurate estimations of a particular system's energy, observing real hardware ensures that complex interactions of components are not overlooked or oversimplified. This section gives a brief description of our hardware and software platform, the measurement methodology, and benchmarks.

2.1 Equipment

The Compaq Personal Server, codenamed "Skiff," is essentially an initial, "spread-out" version of the Compaq iPAQ built for research purposes [13]. Powered by a 233 MHz StrongARM SA-110 [29, 17], the Skiff is computationally similar to the popular Compaq iPAQ handheld (an SA-1110 [18] based device). For wireless networking, we add a five volt Enterasys 802.11b wireless network card (part number CSIBD-AA). The Skiff has 32 MB of DRAM, support for the Universal Serial Bus, a RS232 Serial Port, Ethernet, two Cardbus sockets, and a variety of general purpose I/O. 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 1). With a Cardbus extender card, one can isolate the power used by a wireless network card as well. A programmable multimeter and sense resistor provide a convenient way to examine energy in a active system with error less than 5% [47].

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

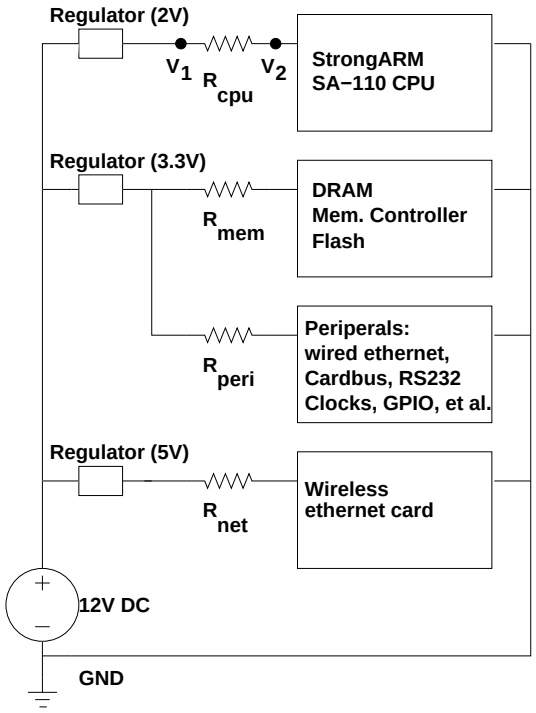

Figure 1. Simplified Skiff power schematic

the cessation of traffic on the network once the program completes loading. I/O is conducted in memory using a modified SPEC harness [42] to avoid the large cost of accessing the network filesystem.

2.2 Benchmarks

Figure 2 shows the performance of several lossless data compression applications using metrics of compression ratio, execution time, and static memory allocation. The datasets are the first megabyte (English books and a bibliography) from the Calgary Corpus [5] and one megabyte of easily compressible web data (mostly HTML, Javascript, and CSS) obtained from the homepages of the Internet's most popular websites [32, 25]. Graphics were omitted as they are usually in compressed form already and can be recognized by application-layer software via their file extensions. Most popular repositories ([4, 10, 11]) 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 3.3.

Figure 2 confirms that we have chosen an array of ap-

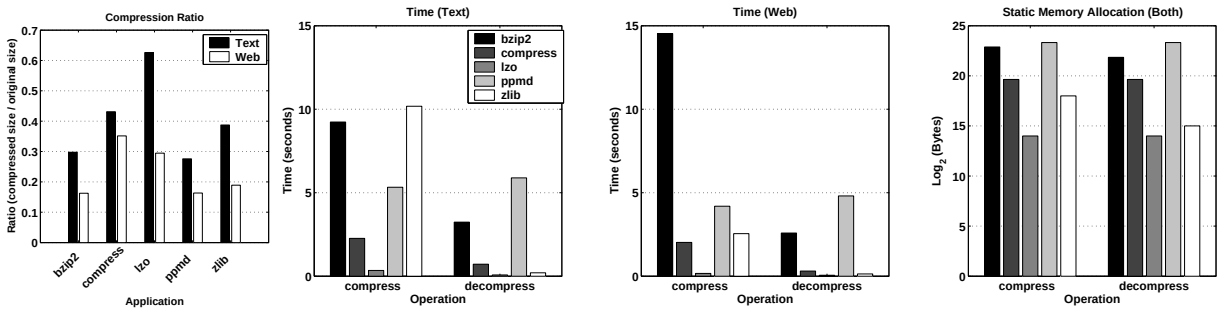

Figure 2. Benchmark comparison by traditional metrics

plications that span a range of compression ratios and execution times. Each application represents a different family of compression algorithms as noted in Table 1. Consideration was also given to popularity and documentation, as well as quality, parameterizability, and portability of the source code. The table includes the default parameters used with each program. To avoid unduly handicapping any algorithm, it is important to work with well-implemented code. Mature applications such as *compress*, *bzip2*, and *zlib* reflect a series of optimizations that have been applied since their introduction. While *PPMd* is an experimental program, it is effectively an optimization of the Prediction by Partial Match (PPM) compressors that came before it. *LZO* represents an approach for achieving great speed with LZ77. Each of the five applications is summarized below assuming some familiarity with each algorithm. A more complete treatment with citations may be found in [36].

zlib combines LZ77 and Huffman coding to form an algorithm known as "deflate." The LZ77 sliding window size and hash table memory size may be set by the user. LZ77 tries to replace a string of symbols with a pointer to the longest prefix match previously encountered. A larger window improves the ability to find such a match. More memory allows for less collisions in the *zlib* hash table. Users may also set an "effort" parameter which dictates how hard the compressor should try to extend matches it finds in its history buffer. *zlib* is the library form of the popular *gzip* utility (the library form was chosen as it provides more options for trading off memory and performance). Unless specified, it is configured with parameters similar to *gzip*.

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 can be sized to fit 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 encoded more simply than in *zlib*.

compress is a popular Unix utility. It implements the LZW algorithm with codewords beginning at nine bits. Though a bit is wasted for each single 8-bit character, once longer strings have been seen, they may be replaced with short codes. 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. 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. Hashing allows an average constant-time access to any entry, but has the disadvantage of poor spatial locality when combining multiple entries to form a string. Despite the random dispersal of codes to the table, common strings may benefit from temporal locality. To reduce collisions, the table should be sparsely filled which results in wasted memory. During decompression, each table entry may be inserted without collision.

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. PPM takes advantage of the fact that the occurrence of a certain symbol can be highly dependent on its context (the string of symbols which preceded it). The PPM scheme maintains such context information to estimate the probability of the next input symbol to appear. An arithmetic coder uses this stream of probabilities to efficiently code the source. As the model becomes more accurate, the occurrence of a highly likely symbol requires fewer bits to encode. 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. Stor-

Application (Version)	Source	Algorithm	Notes (defaults)
bzip2 (0.1pl2)	[37]	BWT	RLE→BWT→MTF→RLE→HUFF (900k block size)
compress (4.0)	[21]	LZW	modified Unix Compress based on Spec95 (16 bit codes (maximum))
LZO (1.07)	[33]	LZ77	Favors speed over compression (lzo1x_12. 4K entry hash table uses 16KB)
PPMd (variant I)	[40]	PPM	used in "rar" compressor (Order 4, 10MB memory, restart model)
zlib (1.1.4)	[9]	LZ77	library form of gzip (Chaining level 6 / 32K Window / 32K Hash Table)

Table 1. Compression applications and their algorithms

ing and searching through each context accounts for the large memory requirements of PPM schemes. The length of the maximum context can be varied by *PPMd*, 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.

bzip2 is based on the Burrows Wheeler Transform (BWT) [8]. The BWT 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 $[0..n-1]$. More important than the details of the transformation is its effect. The transform collects groups of identical input symbols such that the probability of finding a symbol s in a region of L is very high if another instance of s 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 a Huffman or Arithmetic coder. For processing efficiency, long runs can be filtered with a run length encoder. 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. *bzip2* 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.

2.3 Performance and implementation concerns

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 [3]. The quality and applicability of the implementation is as important as the underlying algorithm. This section has presented implementations from each algorithmic family. By choosing a 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 4.1 shows the benefit of optimization) or use a more uniform style of I/O. Thus, evaluation must focus on inherent patterns rather than making a direct quantitative comparison.

3 Observed Energy of Communication, Computation, and Compression

In this section, we observe that over 1000 32 bit ADD instructions can be executed by the Skiff with the same amount of energy it requires to send a single bit via wireless ethernet. This fact motivates the investigation of 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 3.3.

3.1 Raw Communication-to-Computation Energy Ratio

To quantify the gap between wireless communication and computation, we have measured wireless idle, send, and receive energies on the Skiff platform. To eliminate competition for wireless bandwidth from other devices in the lab, we established a dedicated channel and ran the network in ad-hoc mode consisting of only two wireless nodes. We 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 to avoid biasing our results.

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 3 and are consistent with other studies [20]. 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 achieved. These two cases bound the performance of our 11 Mb/sec wireless card; typical performance should be somewhere between them.

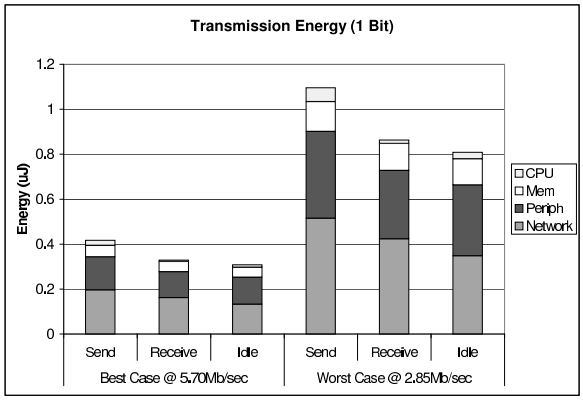

Figure 3. Measured communication energy of Enterasys wireless NIC

Next, a microbenchmark is used to determine the minimum energy for an ADD instruction. We use Linux boot code to bootstrap the processor; select a cache configuration; and launch assembly code unencumbered by an operating system. One thousand ADD instructions are followed 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.

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 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.

3.2 Application-Level Communication-to-Computation Energy Ratio

On the Skiff platform, memory, peripherals, and the network card remain powered on even when they are not active, consuming a fixed energy overhead. They may even switch when not in use in response to changes on shared buses. The energy used by these components during the ADD loop is significant and is shown in Table 2. 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 paper are more than a mere series of ADDs; the variety of instructions (especially Loads and Stores) in compression applications shrinks the ratio further.

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

Table 2. Total Energy of an ADD

The first row of Figures 4 and 5 show the energy required to compress our text and web dataset and transmit it via wireless ethernet. To avoid punishing the benchmarks for the Skiff's high power, idle energy has been removed from the peripheral component so that it represents only the amount of *additional* energy (due to bus toggling and arbitration effects) over and above the energy that would have been consumed by the peripherals remaining idle for the duration of the application. Idle energy is not removed from the memory and CPU portions as they are required to be active for the duration of the application. The network is assumed to consume no power until it is turned on to send or receive data. The popular compression applications discussed in Section 2.2 are used with their default parameters, and the rightmost bar shows the energy of merely copying the uncompressed data over the network. Along with energy due to default operation (labeled "bzip2-900," "compress-16," "lzo-16," "ppmd-10240," and "zlib-6"), the figures include energy for several invocations of each application with varying parameters. *bzip2* is run with both the default 900 KB block sizes as well as its smallest 100 KB block. *compress* is also run at both ends of its spectrum (12 bit and 16 bit maximum codeword size). *LZO* runs in just 16 KB of working memory. *PPMd* uses 10 MB, 1 MB, and 32 KB memory with the cutoff mechanism for freeing space (as it is faster than the default "restart" in low-memory configurations). *zlib* is run in a configuration similar to *gzip*. The numeric suffix (9, 6, or 1) refers to effort level and is analogous to *gzip*'s commandline option. These various invocations will be studied in section 3.3.3.

While most compressors do well with the web data, in several cases the energy to compress the file approaches or outweighs the energy to transmit it. This problem is even worse for the harder-to-compress text data. The second row of Figures 4 and 5 shows the reverse operation: receiving data via wireless ethernet and decompressing it. The decompression operation is usually less costly

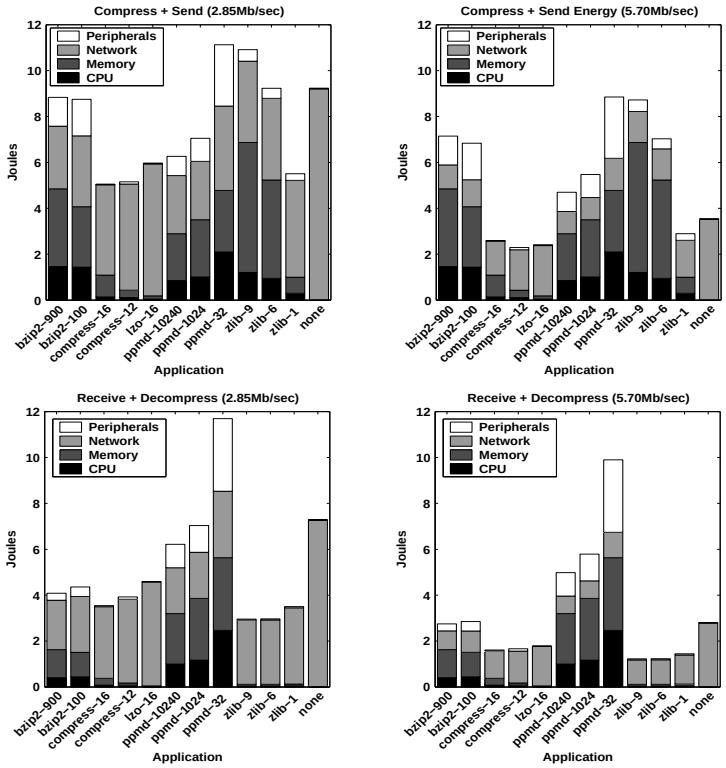

Figure 4. Energy required to transmit 1MB compressible text data

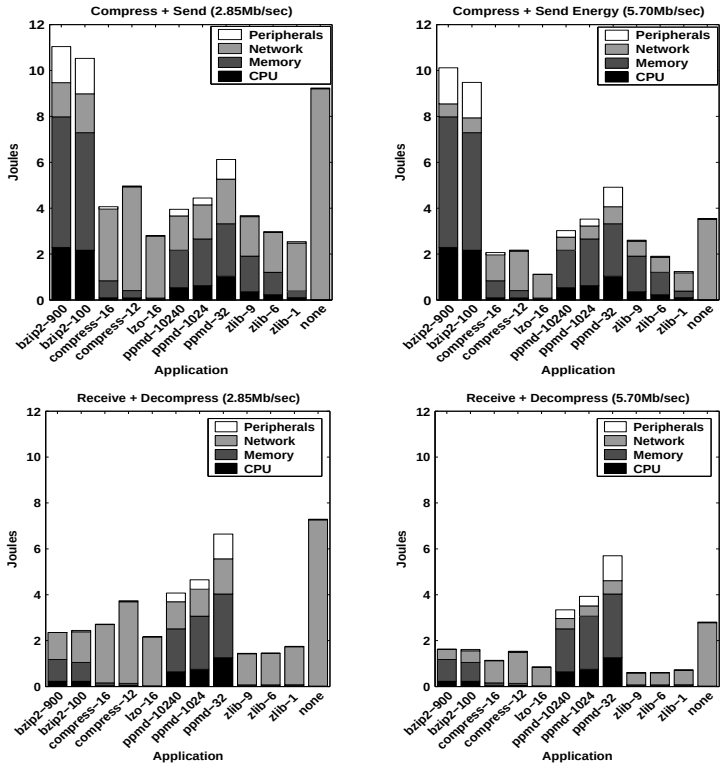

Figure 5. Energy required to transmit 1MB compressible web data

than compression in terms of energy, a fact which will be helpful in choosing a low-energy, asymmetric, lossless compression scheme. As an aside, we have seen that as transmission speed increases, the value of reducing wireless energy through data compression is less. Thus, even when compressing and sending data appears to require the same energy as sending uncompressed data, it is beneficial to apply compression for the greater good: more shared bandwidth will be available to all devices allowing them to send data faster and with less energy. Section 3.3 will discuss how such high net energy is possible despite the motivating observations.

3.3 Energy analysis of popular compressors

We will look deeper into the applications to discover why they cannot exploit the communication - computation energy gap. To perform this analysis, we rely on empirical observations on the Skiff platform as well as the execution-driven simulator known as SimpleScalar [7]. 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 [2]. As SimpleScalar is beta software we will handle the statistics it reports with caution, using them to explain the *traits* of the compression applications rather than to describe their precise execution on a Skiff. Namely, high instruction counts and high cost of memory access lead to poor energy efficiency.

3.3.1 Instruction count

We begin by looking at the number of instructions each requires to remove and restore a bit (Table 3). 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*). As we will see, the number of memory accesses plays a large role in determining the speed and energy of an application. Each program contains roughly the same percentage of loads and stores, but the great difference in dynamic number of instructions means that programs such as *bzip2* and *PPMd* (each executing over 1 billion instructions) execute more total instructions and therefore have the most memory traffic.

3.3.2 Memory hierarchy

One noticeable similarity of the bars in Figures 4 and 5 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 with a similar pattern, but each load is followed by a store to the same location that dirties the block forcing a writeback the next time that location is read. Store hit energy is subtracted from the writeback energy. The output of the compiler is examined to insure the correct number of load or store instructions is generated. Address generation instructions are ignored for miss benchmarks as their energy is minimal compared to that of a memory access. When measuring store misses in this fashion (with a 32 byte stride), the worse-case behavior of the SA-110's store buffer is exposed as no writes can be combined. In the best case, misses to the 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.

Table 4 shows that hitting in the cache requires more energy than an ADD (Table 2), and a cache miss requires up to 145 times the energy of an ADD. Store misses are less expensive as the SA-110 has a store buffer to batch accesses to memory. To minimize energy, then, we must seek to minimize cache-misses which require prolonged access to higher voltage components.

3.3.3 Minimizing memory access energy

One way to minimize misses is to reduce the memory requirements of the application. Figure 6 shows the effect of varying memory size on compression/decompression time and compression ratio. Looking back at Figures 4 and 5, we see the energy implications of choosing the right amount of memory. Most importantly, we see that merely choosing the fastest or best-compressing application does not result in lowest overall energy. Table 5 notes the throughput of each application; we see that with the Skiff's processor, several applications have difficulty meeting the line rate of the network which may preclude their use in latency-critical applications.

In the case of *compress* and *bzip2*, a larger memory footprint stores more information about the data and can be used to improve compression ratio. However, storing more information means less of the data fits in the cache leading to more misses, longer runtime and hence more

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

Table 3. Instructions per bit

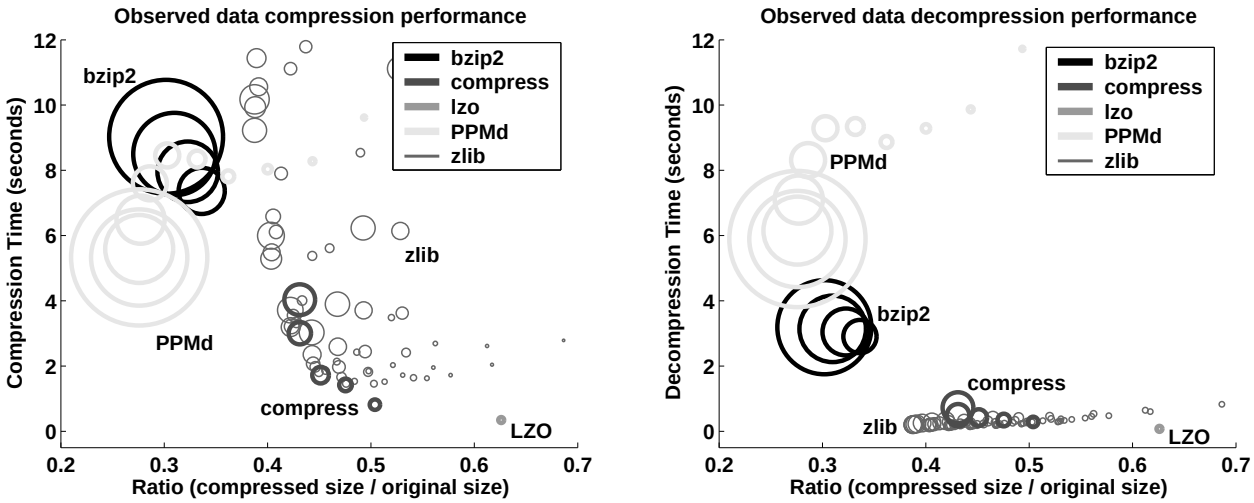

Figure 6. Memory, time, and ratio (Text data). Memory footprint is indicated by area of circle; footprints shown range from 3KB - 8MB

	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. Measured memory energy vs. ADD energy

energy. This tradeoff need not apply in the case where more memory allows a more efficient data structure or algorithm. For example, *bzip2* uses a large amount of memory, but for good reason. While we were 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 [38] are used to further reduce memory requirements. Once *PPMd* has enough memory to do useful work, 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* because 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 decode than encode, so decompression requires more time.

Each of the applications examined allocates fixed-size

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

Table 5. Application throughputs (Mb/sec)

structures regardless of the input data length. Thus, in several cases more memory is set aside than is actually required. However, a large memory footprint may not be detrimental to an application if its current working set fits in the cache. The simulator was used to gather cache statistics. PPM and BWT are known to be quite memory intensive. Indeed, *PPMd* and *bzip2* access the data cache 1–2 orders of magnitude more often than the other benchmarks. *zlib* accesses data cache almost as much as *PPMd* and *bzip2* during compression, but drops from 150 million accesses to 8.2 million during decompression. Though LZ77 is local by nature, the large window and data structures hurt its cache performance for *zlib* during the compression phase. *LZO* also uses LZ77, but is designed to require just 16KB of memory and goes to main memory over five times less often than the next fastest application. The followup to the SA-110 (the SA-1110 used in Compaq's iPAQ handheld computer) has only an 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.

3.4 Summary

On the Skiff, compression and decompression energy are roughly proportional to execution time. We have seen that the Skiff requires lots of energy to work with aggressively compressed data due to the amount of high-latency/high-power memory references. However using the fastest-running compressor or decompressor is not necessarily the best choice to minimize *total* transmission energy. For example, during decompression both *zlib* and *compress* run slower than *LZO*, but they receive fewer bits due to better compression so total energy is less than *LZO*. These applications successfully walk the tightrope of computation versus communication cost. Despite the greater energy needed to decompress the data, the decrease in receive energy makes the net operation a win. More importantly, we have shown that reducing energy is not as simple as choosing the fastest or best-compressing program.

We can generalize the results obtained on the Skiff in the following fashion. Memory energy is some multiple of CPU energy. Network energy (send and receive) is a far greater multiple of CPU energy. It is difficult to predict how quickly energy of components will change over time. Even predicting whether a certain component's energy usage will grow 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. On the other hand, for a given mobile CPU design, planned manufacturing improvements may lower its relative power and energy. 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. If one subscribes to the belief that CPU energy will steadily decrease while memory and network energy remain constant, then *bzip2* and *PPMd* become viable compressors. If both memory and CPU energy decrease, then current low-energy compression tools (*compress* and *LZO*) can even be surpassed by their computation and memory intensive peers. However, if only network energy decreases while the CPU and memory systems remain static, energy-conscious systems may forego compression altogether as it now requires more energy than transmitting raw data. Thus, it is important for software developers to be aware of such hardware effects if they wish to keep compression energy as low as possible. Awareness of the type of data to be transmitted is important as well. For example, transmitting our world-wide-web data required less energy in general than the text data. Trying to compress pre-compressed data (not shown) requires significantly more energy and is usually futile.

4 Results

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). One must be aware of evolving hardware's effect on overall energy. Finally, knowledge of com-

pression and decompression energy for a given system permits the use of asymmetric compression in which the lowest energy application for compression is paired with the lowest energy application for decompression.

4.1 Understanding cache behavior

Figure 7 shows the compression energy of several successive optimizations of the *compress* program. The baseline implementation is itself an optimization of the original *compress* code. The number preceding the dash refers to the maximum length of codewords. The graph illustrates the need to be aware of the cache behavior of an application in order to minimize energy. The data structure of *compress* consists of two arrays: a hash table to store symbols and prefixes, and a code table to associate codes with hash table indexes. The tables are initially stored back-to-back in memory. When a new symbol is read from the input, a single index is used to retrieve corresponding entries from each array. The "16-merge" version combines the two tables to form an array of structs. Thus, the entry from the code table is brought into the cache when the hash entry is read. The reduction in energy is negligible: though one type of miss has been eliminated, the program is actually dominated by a second type of miss: the probing of the hash table for free entries. The Skiff data cache is small (16KB) compared to the size of the hash table (≈270KB), thus the random indexing into the hash table results in a large number of misses. A more useful energy and performance optimization is to make the hash table more sparse. This admits fewer collisions which results in fewer probes and thus a smaller 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

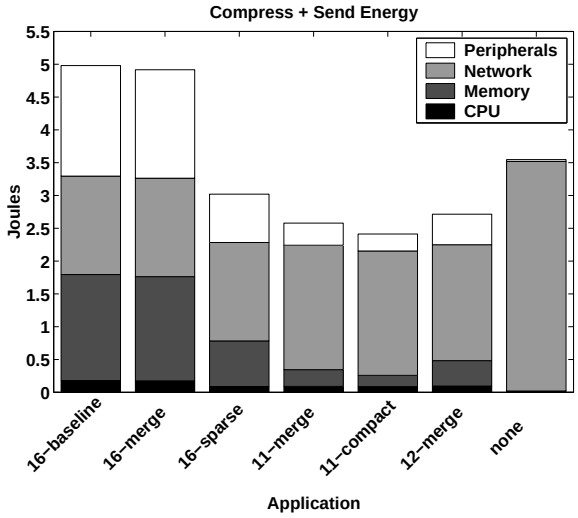

Figure 7. Optimizing *compress* (Text data)

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 with such structures 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 56,985 data cache misses compared with 734,195 in the un-packed structure; a 0.0026% miss rate versus 0.0288%. The energy benefit for *compress* with the compact layout is negligible because there is so little CPU and memory energy to eliminate by this technique. The "11-merge" and "11-compact" bars illustrate the similarity. Nevertheless, 11-compact runs 1.5 times faster due to the reduction in cache misses, and such a strategy could be applied to any program which needs to reduce cache misses for performance and/or energy. 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% compared to "12-merge." Energy, however, is lower with the smaller dictionary due to less energy spent in memory and increased speeds which reduce peripheral overhead.

4.2 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 the low-power idle state [26]. Figure 8 shows the effect of this analysis for compression and sending of text. Receive/decompression exhibits similar, but less-pronounced variation for different idle powers. It is interesting to note that, assuming a low-power idle mode can be entered once compression is complete, one's choice of compression strategies will vary. With its 1 Watt of idle power, the Skiff would benefit most from *zlib* compression. A device which used negligible power when idle would choose the *LZO* compressor. While *LZO* does not compress data the most, it allows the system to drop into low-power mode as quickly as possible, using less energy when long idle times exist. For web data (not shown due to space constraints) the compression choice is *LZO* when idle power is low. When idle power is one Watt, *bzip2* energy is over 25% more energy efficient than the next best compressor.

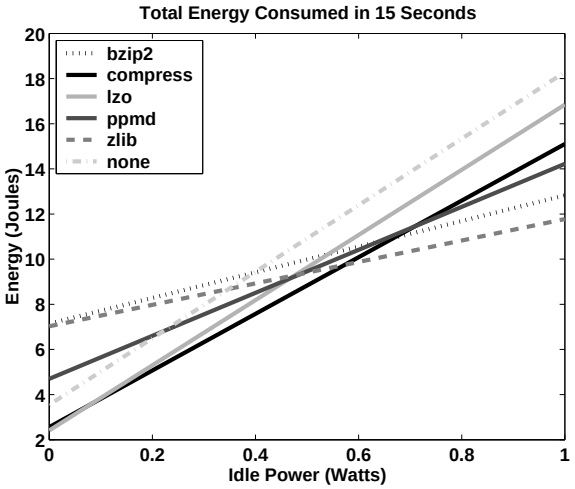

Figure 8. Compression + Send energy consumption with varying sleep power (Text data)

4.3 Asymmetric compression

Consider a wireless client similar to the Skiff exchanging English text with a server. All requests by the client should be made with its minimal-energy compressor, and all responses by the server should be compressed in such a way that they require minimal decompression energy at the client. Recalling Figures 4 and 5, and recognizing that the Skiff has no low-power sleep mode, we choose "compress-12" (the twelve-bit codeword LZW compressor) for our text compressor as it provides the lowest total compression energy over all communication speeds.

To reduce decompression energy, the client can request data from the server in a format which facilitates low-energy decompression. If latency is not critical and the client has a low-power sleep mode, it can even wait while the server converts data from one compressed format to another. On the Skiff, *zlib* is the lowest energy decompressor for both text and web data. It exhibits the property that regardless of the effort and memory parameters used to compress data, the resulting file is quite easy to decompress. The decompression energy difference between *compress*, *LZO*, and *zlib* is minor at 5.70 Mb/sec, but more noticeable at slower speeds.

Figure 9 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 – especially for programs linked with the zlib library. Compared with the minimum symmetric compressor-decompressor, asymmetric compression on the Skiff saves only 11% of energy. However, modern applications such as ssh and mod_gzip use "zlib-6" at both ends of the connection. Compared to this common scheme, the optimal asymmetric pair yields a 57% energy savings – mostly while performing compression.

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

5 Related work

This section discusses 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 [12], but this work adds physical energy measurements and applies the results to lossless data compression.

5.1 Lossless Data compression for low-bandwidth devices

Like any optimization, compression can be applied at many points in the hardware-software spectrum. When

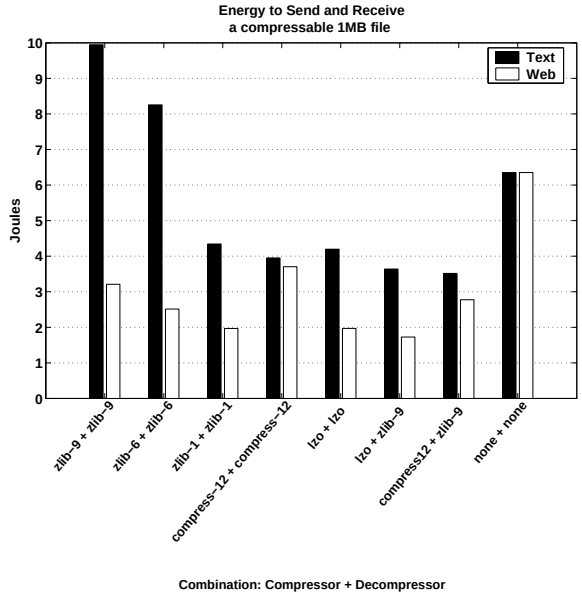

Figure 9. Choosing an optimal compressor-decompressor pair

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.

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 [19], 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 [24]. TCP/IP payloads can be compressed as well with IP-Comp [39], 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 [31]. 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. Compression is applied before the data is transmitted. Rsync [44] 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 [30]. 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.

Along with remote proxy servers which may cache or reformat data for mobile clients, splitting the proxy between client and server has been proposed to implement certain types of network traffic reduction for HTTP transactions [14, 23]. Because the delay required for manipulating data can be small in comparison with the latency of the wireless link, bandwidth can be saved with little effect on user experience. Alternatively, compression can be built into servers and clients as in the *mod_gzip* module available for the Apache webserver and HTTP 1.1 compliant browsers [16]. Delta encoding, the transmission of only parts of documents which differ between client and server, can also be used to compress network traffic [15, 27, 28, 35].

5.2 Optimizing algorithms for low energy

Advanced RISC Machines (ARM) provides an application note which explains how to write C code in a manner best-suited for its processors and ISA [1]. Suggestions include rewriting code to avoid software emulation and working with 32 bit quantities whenever possible to avoid a sign-extension penalty incurred when manipulating shorter quantities. To reduce energy consumption and improve performance, the OptAlg tool represents polynomials in a manner most efficient for a given architecture [34]. As an example, cosine may be expressed using two MAC instructions and an MUL to apply a “Horner transform” on a Taylor Series rather than making three calls to a cosine library function.

Besides architectural constraints, high level languages such as C may introduce false dependencies which can be removed by disciplined programmers. For instance, the use of a global variable implies loads and stores which can often be eliminated through the use of register-allocated local variables. Both types of optimizations are used as guidelines by PHiPAC [6], 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 energy can be saved if the compiler can reduce

execution time and memory references [48]. Šimunić found that floating point emulation requires much energy due to the sheer number of extra instructions required [46]. 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 [45].

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 [41]. 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 [43]. Research to date has focused on situations where energy-fidelity tradeoffs are available. Lossless compression does not present this luxury because the original bits must be communicated in their entirety and re-assembled in order at the receiver.

6 Conclusion and Future Work

The value of this research is not merely to show that one can optimize a given algorithm to achieve a certain reduction in energy, but to show that the choice of how and whether to compress is not obvious. It is dependent on hardware factors such as relative energy of CPU, memory, and network, as well as software factors including compression ratio and memory access patterns. These factors can change, so techniques for lossless compression prior to transmission/reception of data must be re-evaluated with each new generation of hardware and software. On our StrongARM computing platform, measuring these factors allows an energy savings of up to 57% compared with a popular default compressor and decompressor. Compression and decompression often have different energy requirements. When one's usage supports the use of asymmetric compression and decompression, up to 12% of energy can be saved compared with a system using a single optimized application for both compression and decompression.

When looking at an entire system of wireless devices, it may be reasonable to allow some to individually use more energy in order to minimize the total energy used by the collection. Designing a low-overhead method for devices to cooperate in this manner would be a worthwhile endeavor. To facilitate such dynamic energy adjustment, we are working on EProf: a portable, realtime, energy profiler which plugs into the PC-Card socket of a portable device [22]. EProf could be used to create feedback-driven compression software which dynamically tunes its parameters or choice of algorithms based on the measured energy of a system.

7 Acknowledgements

Thanks to John Ankcorn, Christopher Batten, Jamey Hicks, Ronny Krashinsky, and the anonymous reviewers for their comments and assistance. This work is supported by MIT Project Oxygen, DARPA PAC/C award F30602-00-2-0562, NSF CAREER award CCR-0093354, and an equipment grant from Intel.

References

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

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

[3] 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.

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

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

[6] 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.

[7] 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.

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

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

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

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

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

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

[14] B. C. Housel and D. B. Lindquist. Webexpress: a system for optimizing web browsing in a wireless environment. In *Proceedings of the Second Annual International Conference on Mobile Computing and Networking*, 1996.

[15] J. J. Hunt, K.-P. Vo, and W. F. Tichy. An empirical study of delta algorithms. In *Software configuration management: ICSE 96 SCM-6 Workshop*. Springer, 1996.

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

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

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

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

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

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

[22] K. Koskelin, K. Barr, and K. Asanović. Eprof: An energy profiler for the iPaq. In *2nd Annual Student Oxygen Workshop*. MIT Project Oxygen, 2002.

[23] R. Krashinsky. Efficient web browsing for mobile clients using HTTP compression. Technical Report MIT-LCS-TR-882, MIT Lab for Computer Science, Jan. 2003.

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

[25] Lycos. Lycos 50, Sept. 2002. Top 50 searches on Lycos for the week ending September 21, 2002.

[26] 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.

[27] J. C. Mogul. Trace-based analysis of duplicate suppression in HTTP. Technical Report 99.2, Compaq Computer Corporation, Nov. 1999.

[28] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. Technical Report 97/4a, Compaq Computer Corporation, Dec. 1997.

[29] 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.

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

[31] 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.

[32] Nielsen NetRatings Audience Measurement Service. Top 25 U.S Properties; Week of Sept 15th., Sept. 2002.

[33] M. F. Oberhumer. LZO. http://www.oberhumer.com/opensource/lzo/.

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

[35] J. Santos and D. Wetherall. Increasing effective link bandwidth by suppressing replicated data. In *USENIX Annual Technical Conference*, June 1998.

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

[37] J. Seward. bzip2. http://www.spec.org /osg/cpu2000/CINT2000/256.bzip2/docs/256.bzip2.html.

[38] J. Seward. e2comp bzip2 library. http://cvs.bofh.asn.au/e2compr/index.html.

[39] A. Shacham, B. Monsour, R. Pereira, and M. Thomas. RFC 3173: IP payload compression protocol, Sept. 2001.

[40] D. Shkarin. PPMd. ftp://ftp.elf.stuba.sk/pub/pc/pack/ppmdi1.rar.

[41] 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.

[42] Standard Performance Evaluation Corporation. CPU2000, 2000.

[43] C. N. Taylor and S. Dey. Adaptive image compression for wireless multimedia communication. In *IEEE International Conference on Communication*, June 2001.

[44] A. Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, Apr. 2000.

[45] 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.

[46] 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.

[47] M. A. Viredaz and D. A. Wallach. Power evaluation of Itsy version 2.4. Technical Report TN-59, Compaq Computer Corporation, February 2001.

[48] 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.