Adaptive Online Data Compression

Emmanuel Jeannot
LORIA, INRIA-Lorraine
Université H. Poincaré, Nancy I
France
ejeannot@loria.fr

Björn Knutsson*
Distributed System Laboratory
University of Pennsylvania
USA
bjornk@dsl.cis.upenn.edu

Mats Björkman
Department of Computer Engineering
Mälardalens högskola
Sweden
Mats.Bjorkman@mdh.se

Abstract

Quickly transmitting large datasets in the context of distributed computing on wide area networks can be achieved by compressing data before transmission. However, such an approach is not efficient when dealing with higher speed networks. Indeed, the time to compress a large file and to send it is greater than the time to send the uncompressed file. In this paper, we explore and enhance an algorithm that allows us to overlap communications with compression and to automatically adapt the compression effort to currently available network and processor resources.

1 Introduction

Recent developments in the area of grid-computing have focused on the need to efficiently transfer very large amounts of data. Indeed, when designing Problem Solving Environments (PSE) such as NetSolve [3], Ninf [17], or Scilab$_{//}$ [2], data (files, internal objects, etc.) need to be sent. Performance of PSEs greatly depends on the ability to transmit the data efficiently. However, a meta-computing environment is composed of a heterogeneous set of machines interconnected by heterogeneous networks. Available computing resources can be old sequential machines as well as brand new parallel computers. These machines can be interconnected by slow WANs (Internet), as well as fast backbones (e.g. 2.5 Gbit/s VTHD[1]) or 100 Mbit/s LANs, etc.

Since no assumptions can be made on the hardware infrastructure we need a general transmission algorithm that is efficient whatever the speed of the network and the speed of the machines.

When the server is a parallel machine and the data to be transmitted are distributed among the nodes of the server, one possibility to transfer these data efficiently is to use multi-socket techniques. This solution has the advantage of using the maximum bandwidth of the backbone, but is limited when a bottleneck exists (such as a client connected to a slow network). Another solution is to compress data prior to sending them. However, this solution is inefficient when the network is so fast that the time needed to compress the data and send it is greater than the time needed to send the uncompressed data.

The solution to this problem is to adapt compression to the currently available CPU and network resources in such a way as to produce compressed data at the rate it can be sent. This approach was first explored in [11] and developed into an algorithm and implemented in [12].

This paper revisits this algorithm in the context of distributed computing, and explores alternatives in the implementation strategies to improve performance and portability of the original algorithm.

This paper presents the *AdOC* (Adaptive Online Compression) algorithm which is designed to be a general purpose *portable* application layer algorithm suited not only for grid computing, but for any application data transfer, regardless of network bandwidths or type of application. We provide a library with a well-defined API which allows applications a greater degree of control and input on how the algorithm is applied to their data.

*Supported by the DoD University Research Initiative (URI) program administered by the Office of Naval Research under Grant N00014-01-1-0795.

[1] http://www.vthd.org

The rest of the paper is organized as follow. Section 2 presents some background on communication and compression. The AdOC algorithm is presented in section 3. Section 4 presents our experimental results. Section 5 discusses possible optimizations of the algorithm. Concluding remarks are given in Section 6.

2 Analyzing Compression

In this section we discuss some properties and background on compression and communication that will help motivate the proposed algorithm.

A popular way to trade one resource for another is compression. By compression, we mean codings that represent the same information using fewer bits. It is used for many different purposes in different contexts with different trade-offs:

- Trade CPU and memory usage for disk space, as seen in compressed file systems and compressed files on regular file systems.
- Trade CPU usage for memory space — designs for conserving memory in systems with memory size constraints, such as embedded systems. For example it is possible to compress the contents of the main memory and letting the CPU uncompress it on-the-fly as instructions are executed.
- Trade CPU usage for faster paging — instead of paging directly to disk, schemes have been devised where pages are compressed and put into a compressed page-cache in memory[8]. If not reused, the compressed pages can be paged out to disk. This will require a shorter I/O operation than handling the uncompressed page.
- Trade CPU and memory usage for more bandwidth, by compressing data before sending them over a communications link, as seen in V.42bis[4] modems and PPP[16].

When doing compression to reduce bandwidth requirements, one of the key observations is that without adapting, compression cannot be used efficiently across different systems and networks.

2.1 Trade-offs

Compression uses the CPU and memory to achieve compression, which means that the use of compression always is a trade-off between the resources used and the reduction in space.

Time is also a factor — while the system may have the CPU and memory to spare, it may not be able to wait while the data is being compressed or subsequently decompressed, i.e. time in itself is a tangible resource. For e.g. storing data on a disk, the CPU and memory used, and the time to compress and decompress the data, is traded off against less need for static storage space.

It may seem like a paradox that compression also can be used to trade for shorter access time. If compressed data can be produced faster than a communication channel can transmit it, the time needed to get data to the other end of the channel is reduced because fewer bits need to be transmitted. This is the scenario we are already familiar with for modem- and other low bandwidth communication. In these cases, it may make sense to compress data before sending it and decompressing it on the other side provided that the compression speed is greater than the bandwidth.

Compression can also be used to reduce CPU consumption for other data manipulations functions. E.g. if data will be encrypted, then by reducing the size of the data, the amount of CPU needed to encrypt it is reduced. Assume that it takes $t_{crypt}(b)$ seconds to encrypt b bytes and time to compress b bytes into b' bytes is $t_{compress}(b, b')$. Then if $t_{compress}(b, b') + t_{crypt}(b') < t_{crypt}(b)$ time will be saved by compressing data before encrypting it.

In summary, the reduction in size of the original data may translate into a gain in one or more of time, CPU, memory or static storage. But we see that when the gain is a side effect of the reduction in size, the system must be very carefully tuned to achieve that gain, since it relies on the speed of compression relative to other factors. If we e.g. misjudge the bandwidth of a channel or the CPU available, then compression can slow down the transfer instead of speeding it up because compression cannot keep up with the network.

2.2 Compression algorithms

There exists algorithms that are highly tuned to specific areas such as images and audio, and there are general algorithms that only exploit that "interesting" data often contains redundancy (as opposed to random data)[13].

A distinction is made between lossless and lossy algorithms, i.e. those algorithms that will preserve the original data exactly, and those that will discard parts of the data, reducing the quality. The latter type is typically domain specific, i.e. knowledge about what type of data is being compressed is needed to determine what to discard.

The development and analysis of compression algorithms is a research field all in itself, with its own conferences and journals, but the problems of compression algorithms are largely outside the scope of the work presented here. This paper will only deal with lossless general compression algorithms. Using other compression algorithms with the schemes presented here should be possible, but this has so far not been investigated by us.

For general compression three of the most often used algorithms are:

- Run length coding — A simple and fast scheme that replaces repeating patterns with the patterns and number of repetitions.
- Huffman[9] coding — Huffman coding analyzes the frequency of different fixed length symbols in a data set, and to the symbols assigns codes whose lengths correspond to the frequency of the respective symbol in the data set, i.e. frequent symbols get short codes, infrequent get long codes.
- Lempel-Ziv[21, 22] — These algorithms basically replace strings (variable length symbols) found in a dictionary with codes representing those strings. The efficiency of these algorithms is determined by the size of the dictionary and how much effort is spent searching in the dictionaries.

The algorithms used in GIF, several PPP[18, 20, 15] compressors, V.42bis[4], and many popular compression programs such as UNIX Compress, GZip[6] and others use general compression algorithms based on the two Lempel-Ziv algorithms LZ77[21] and LZ78[22] (or Lempel-Ziv-Welch[19], which is based on LZ78). These algorithms are sometimes augmented with Huffman[9] coding. Huffman coding on its own is typically faster than Lempel-Ziv based algorithms, although it will typically yield less compression.

Run length encoding is extremely fast, but the gain is often small compared to Lempel-Ziv or Huffman coding.

2.2.1 CPU

The actual CPU needed to compress data varies with the algorithms, and in many algorithms it will also vary with the data fed to the compression algorithm because different patterns will trigger different cases in the algorithm.

In algorithms that use a static scheme, like run length encoding or Huffman coding with a static table, the time needed to compress an arbitrary file can be predicted with fairly good precision without prior knowledge of the contents of the file. But in algorithms which use the content of the file to adapt the compression, like Lempel-Ziv, the CPU needed to compress an arbitrary file cannot be predicted.

Parameters to the algorithms will also determine the amount of CPU needed. In Lempel-Ziv based algorithms, the size of the dictionary and the depth of searches for matches in the dictionary can be set. Deeper searches or bigger dictionaries to search require more CPU, but often also translates into a better compression ratio. With Huffman coding, the width (in bits) of one symbol can be modified. With longer symbols, more CPU will be spent on bookkeeping, but the compression ratio will go up.

2.2.2 Memory

The amount of memory needed for compression varies with the algorithm used.

A run length encoding scheme typically only needs a counter and a small (couple of bytes) buffer. Huffman coding requires enough memory to store the symbol translation table. Lempel-Ziv schemes are more memory hungry since they compress by "knowing" what patterns have previously been seen in the data stream, and their efficiency is often a function of how much memory they are allowed to consume for buffering previously seen data.

The amount needed for all these algorithms can be predicted and/or limited in advance. Thus the compression algorithm can be assumed to have a static, predictable, memory footprint.

As an example of the effect memory constraints have on compression efficiency, look at telephone modems. The algorithms used (v.42) are efficient, but to keep costs down, they are not fitted with much memory. This is why compression in e.g. PPP often improve performance over compressing modems.

2.3 The time aspect

In some cases the compression time aspect is of little importance, e.g. it does not matter much if a file takes five or fifty seconds to compress, when the only goal is to fit it into a cramped storage space. In these cases the resources needed to compress are only relevant if the shortage is permanent. If e.g. the CPU is very slow, all we need to do is wait, eventually the compression will complete. If the compression algorithm needs more than the total available memory, i.e. it cannot run, we have to manually intervene to reduce the memory consumption by choosing parameters or algorithms that require less memory.

If compression is done on a general purpose computer which may run multiple applications and communicate at varying speeds over shared communication media, the parameters of the trade-off cannot be known beforehand. CPU speed, memory availability and media bandwidth may vary during one session.

For most applications, the compression time is essential. If we e.g. compress data to increase transmission speeds, and lack of CPU resources make compression so slow that it would be faster to send it uncompressed, then there is no point in compressing data in the first place.

We could express the trade-off between compressing by doing the following definitions:

T is the type of data we want to send. If we know T, we could choose a compression algorithm designed for this type of data.

P is the parameter set controlling the compression algorithm.

$C = f(CPU, T, P)$ is time to compress one byte, given the currently available CPU, type of data T and parameter set P.

τ is the time to send one bit, i.e. $1/bandwidth$.

$\rho = g(T, P)$ is the compression ratio we achieve for the type of data T and parameter set P.

$C' = f'(CPU', \rho)$ is time to decompress one byte, given the currently available CPU and the compression ratio ρ of the input data.

b is the amount of data we want to send.

The time we have available to compress data is the reduction in sending time between the uncompressed and compressed data $(\tau \times b) - (\tau \times b')$.

Thus, if $Cb + \tau \times b/\rho + C'b < \tau \times b$, then compressing data before sending it would lower the total transfer time.

The parameters CPU, P, T and τ controls the relation. Assuming that available CPU resources (CPU and CPU'), bandwidth ($1/\tau$) and type of data (T) are governed by outside factors and may change over time, this leaves us with adapting P to ensure that the relation remains true.

If we base P on a one-time measurement, and then keep it constant, we run a very real risk of either increasing the transfer time compared to not compressing, or losing opportunities to further reduce the transfer time.

2.4 Summary

Compression can, and is often used, to compensate for low bandwidth. This can be done by using a scheme like PPP[16] in which the host systems CPU is utilized to perform compression. Unfortunately, this does not take into account variations in bandwidth and local resources — what might be a reasonable setting for communicating over 9600 bps GSM on an otherwise idle system with a full battery charge, will most likely be suboptimal on a busy system with low batteries communicating over an 11 Mbps wireless LAN.

To use compression to improve perceived communications performance in an environment where resource availability and bandwidth can vary unpredictably, a system that will constantly monitor and adjust compression parameters to match the current situation, possibly turning compression off if it would lower performance, will be needed.

Such a monitoring system must be designed to respond fast to changes, but must also be cheap in terms of resources, since all resources dedicated to monitoring and controlling compression are resource that could have been used to improve compression.

2.5 Examples

	`oilpann.hb`			`bin.tar`		
level	c. time	ratio	d. time	c. time	ratio	d. time
1	10.2	4.88	9.6	16.9	2.23	12
2	11	5.13	9.3	17.2	2.27	12
3	11.3	5.52	8.9	18.3	2.31	13.4
4	11.9	5.83	9	21.7	2.38	13.7
5	13.6	6.32	8.7	23.6	2.43	13.6
6	16.7	6.64	8.3	28.3	2.44	13.7
7	20.4	6.75	8.7	32.1	2.45	13.14
8	47.6	6.99	7.7	44	2.45	13.1
9	76	7.02	7.8	65.8	2.46	12.3

Table 1. *Compression timings on bench files using gzip*

To illustrate how different parameter sets and characteristics can affect the achieved compression ratio, examine Table 1. Here two different files, `oilpann.hb`, a sparse matrix in the Harwell/Boeing format (ASCII) and an archive of binaries, (`bin.tar`) are compared[2]. The files are compressed with the popular `gzip` program.

As can be seen, the latter file does not compress nearly as well as the former, given the same compression *level*[3].

We see that with increasing compression level, compression time and ratio increase, but not linearly, whereas the decompression time is roughly constant. After level 6, we see low increases in compression ratio while compression time increases very fast.

3 AdOC Algorithm

3.1 Overview

1. **For each** data block to be sent.
2. Update the compression level according to the size of the output queue.
3. Compress data at current compression level.
4. Add compressed data to the output queue.

Figure 1. *Original Basic Algorithm*

[2]From traces taken on a Pentium III 500MHz running LINUX kernel 2.4.7-10

[3]The compression level of `gzip` can be seen as an abstraction of the compression parameter set discussed previously

The AdOC algorithm is based on the adaptive compression algorithm presented in [11, 12]. The core of this algorithm is illustrated in Figure 1. This basic algorithm automatically adapts the level of compression to the speed of the network — higher compression levels are only used if we have tried lower levels, and still managed to produce data faster than the network can send it. The existing queue also serves as a safety margin — if the increased compression level causes compression to take too long, packets will drain from the queue, but it will not go empty.

The original implementation was put into the transport layer (TCP), in order to be transparent to applications, and utilized functions in TCP and networking stack internal data for adaptation, and was implemented inside the Linux 2.0 kernel.

The advantages of an application level implementation were recognized both at the time, and in later work by one of the original authors [10], but to our knowledge, AdOC is the first attempt to implement it at the application level. The observation that placement inside the kernel was not ideal was also made in contemporary work[14], but with the processors available at the time, less options were available.

Being an in-kernel modification, the original implementation was not very portable. In fact, if needed even for a more modern Linux kernel, it would have to be reimplemented from scratch, since the kernel internals have changed.

The user level implementation described in this paper is based on TCP sockets and is written in C. Hence, it provides a portable set of functions that can be integrated into a library without modifying the kernel. It uses the following two features:

1. *Multi-threading*. The sender is made up of two threads. One thread (called the compression thread) is in charge of reading the file and compressing data when useful. Another thread (called the communication thread) is in charge of sending data. Such an architecture allows for compression/communication overlap.

2. *FIFO*. A FIFO data structure is used to store the data to be sent. The compression thread writes data to the FIFO. The communication thread reads the FIFO.

The most important parameter of the AdOC algorithm is the size of the FIFO, like the queue length of the original algorithm. However, the FIFO size is monitored for changes instead of absolute size. If it is shrinking, this means that the sending thread consumes data faster than it is produced by the compression thread, and the compression level is reduced. Conversely, if the FIFO is growing, this means that we can increase the compression level. The logic and motivation is thus the same as the original algorithm, but the mechanism differs.

The original implementation had to work for arbitrary applications, so it was designed to be conservative and work for arbitrary networks and sending patterns. AdOC, on the other hand, can afford to pursue optimizations more aggressively, since running it is an active choice by the application. Since AdOC was designed for distributed systems exchanging large data sets, this means that AdOC can assume that the application can tell it how much data is being sent at the outset, and that the application will send data continuously, and optimize for this case.

3.2 Compression/Communication Overlap

When a process/thread is doing some IO on a device (such as writing data to a file on a disk or to a socket on a network), it may be blocked waiting for the device to become ready. This is especially common when writing data to a socket because memory is faster than the network. When the sending process/thread is blocked, it yields the processor to other processes/threads. The idea in AdOC is to use this "free" CPU to compress data about to be sent.

The original algorithm did not divide compression/sending explicitly, but relied on the TCP implementation for this. In fact, modifications to the kernel were needed to keep the OS from completely suspending the system under some conditions. In AdOC, the separation is clean and explicit, and completely avoids the risk of accidentally blocking the compression and allows compression and sending to completely overlap. It also reduces the risk that the network goes idle because compressed data has not been produced fast enough.

Since the FIFO is a shared object between two threads we use a mutex and a semaphore to access it.

3.3 Compression Thread

The compression thread is in charge of reading the file and of compressing data if possible. As in the original implementation, we use the *zlib* [7] library, which is a compression library that implement the same algorithm as gzip, discussed above, and the same compression level abstraction of compression parameters.

Where the original adaptation algorithm treated the data being sent as a single stream, AdOC compresses data into independent chunks. This difference is significant for a number of reasons. First, because it completely eliminates the overhead of going through the compression algorithm when data cannot be compressed. The original algorithm always imposed a small processing and data size overhead, even when not compressing. Second, because where the original algorithm would change compression levels frequently, AdOC changes it only per chunk. On the one hand, this makes AdOC less reactive to short term changes

in bandwidth, but keeping the same compression level for long runs of data also improves the compression ratio. For the typical environment AdOC was designed to be used, this is an optimization that increases performance compared to the original algorithm.

Each chunk is then fed to the compression algorithm, but with a limited output ("packet") buffer. This means that whenever the compression algorithm has produced a packet, the compression algorithm will wake up and return control to AdOC that can now put this packet on the output FIFO and then it resumes the compression of the chuck. This means that even though the compression function is processing a large chunk of data, the output queue will be continuously replenished with data, keeping it from going empty.

The algorithm of the compression thread is given in Figure 2. `fifo_size` is a global object that contains the number of non-empty packets in the FIFO. `packet_size` is a constant giving the maximum size of packets stored in the FIFO. When level = 0 (no compression is used), we read a packet from the file and store it directly into the FIFO. `buffer_size` is the (also constant) size of the chunk buffers to be compressed (level$\neq$0). As outlined above, a chunk is given to the compression library that produces the packets. The function `compress`, returns the next compressed packet and its actual size. We add delimiters in the FIFO to tell when a chunk buffer is completely compressed or when the file is completely read.

At the beginning of each step we save the size of the FIFO in `prev_fifo_size` . At the end of each step we update the level of compression using the difference between the current size of the FIFO and the size of the FIFO at the beginning of the chunk.

```
1  level=0;
2  while there is still data in the file
3     prev_fifo_size=fifo_size;
4     if (level=0)
5        packet=readfile(packet_size);
6        add_fifo(packet,packet_size,level);
7        add_fifo(NULL,0,0);
8     else
9        buffer=readfile(buffer_size);
10       while buffer is not fully compressed
11          (packet,size)=compress(buffer,packet_size,level);
12          add_fifo(packet,size,level);
13       add_fifo(NULL,0,0);
14    level=update_level(level,fifo_size-prev_fifo_size);
15 add_fifo(NULL,0,0);
```

Figure 2. *Compression Thread Algorithm*

3.4 Updating Level of Compression

As shown in Table 1, increasing the compression level decreases the size of the compressed file but increases the compression time. At the outset, we set the level to 0 (no compression) and we slowly increase this level if we believe that we have enough time to compress buffers.

AdOC was designed for large file transfers and to reduce the number of changes in compression level, and thus needs to use a different approach than the original algorithm for changing compression levels — it monitors changes in queue length instead of absolute queue length.

```
input : level : current compression level
        D : FIFO variation size
output : new compression level.
1  if(FIFO_size<10)
2     if(D≤0)
3        level=level/2;
4  else if(FIFO_size<20)
5     if(D>0)
6        level++;
7     else if (D<0)
8        level--;
9  else if(FIFO_size<30)
10    if(D>0)
11       level+=2;
12    else if(D<0)
13       level--;
14 else if(D>0)
15       level+=2;
16 if (level>9)
17    level=9;
18 else if (level<0)
19    level=0;
```

Figure 3. *Updating the compression level*

The first approach we tried was to modify the compression level in a slow–start fashion. We used D, the difference between the size of the FIFO at the end of a compression step and the size of the FIFO at the beginning of the step. This approach was conservative in the sense that when $D > 0$ the compression level was increased by one and when $D < 0$ we divided the compression level by 2. We gave up with this approach because it reduced compression too aggressively and required too much memory when it overestimated the network bandwidth. Indeed, the size of the FIFO often became very large, which implies that we could have used higher compression levels.

Figure 3 describes the new algorithm we settled on. It monitors the FIFO size and how the FIFO size changes.

When the FIFO is small (< 10), it will aggressively decrease the compression level if the FIFO is shrinking, and will not increase the compression level at all. For moderate FIFO sizes ($10--20$), it will slowly adjust the compression level up or down in response to FIFO size changes. When the FIFO size increases ($20--30$), it will begin to increase the compression level more aggressively in response to increases in FIFO size. For large FIFO sizes (> 30), the compression level will no longer be decreased in response to FIFO size decreases.

Since there is a limited range of valid compression levels, it cannot exceed compression level 9, and since compression level 0 means no compression, the level cannot decrease below this level.

3.5 Tuning AdOC

AdOC is designed to be tunable, to make it more flexible and allow users of it to fit it to different needs. Two factors that influence the way the algorithm works is the size of the chunk buffers and the size of the packets.

3.5.1 Determining the Chunk Buffer Size

Our algorithm splits the file into buffers and compresses them independently. Due to the compression algorithm, the obtained compression ratio depends on the size of the given buffer (the larger the buffer, the better the compression ratio). However, since the compression level cannot be changed while compressing a buffer, too large buffers have a bad impact on the reactivity of our algorithm. Hence, we need to find a trade-off for this value. Figure 4 shows the evolution of the compression ratio when increasing the buffer size for the level 6 compression of `oilpann.hb`. We see that the compression ratio rapidly increases at the beginning and converges on a limit which is the compression ratio when treating the entire file as one buffer. We obtain similar results for other compression levels as well as other files, e.g. `bin.tar`.

For our experiments, we have chosen a chunk buffer size of 204 800 bytes because in most of the cases, with this size we obtain a difference less than 5% between the compressed file and the optimal compressed file size. Moreover, our experiments show that compressing 200Kb on our system (Pentium III 500 MHz) takes about 0.1 seconds. Hence, the reactivity of the algorithm with such a chunk buffer size is acceptable.

3.5.2 Determining the Packet Size

When not compressing the file (e.g. at the beginning of file or when dealing with a fast network), the file is split into packets which are sent uncompressed. If we have too large packets, the reactivity of our algorithm at the beginning will be affected. However if we have small packets, we will have a lot of elements added in the FIFO which implies more overhead for managing the FIFO and sending the packet. When in compressing state, a buffer is read and compressed. The compression of the buffer stops each time a packet is generated. This packet is put in the FIFO and the compression resumes. Hence, dealing with too small packets implies too many overhead for managing the FIFO, whereas dealing with too large packets limits the compression/communication overlap. We need to find a trade-off. In order to have an idea of what is a good packet size we have performed the following computation which is inspired from optimizing the packet size for computation/communication overlap [5].

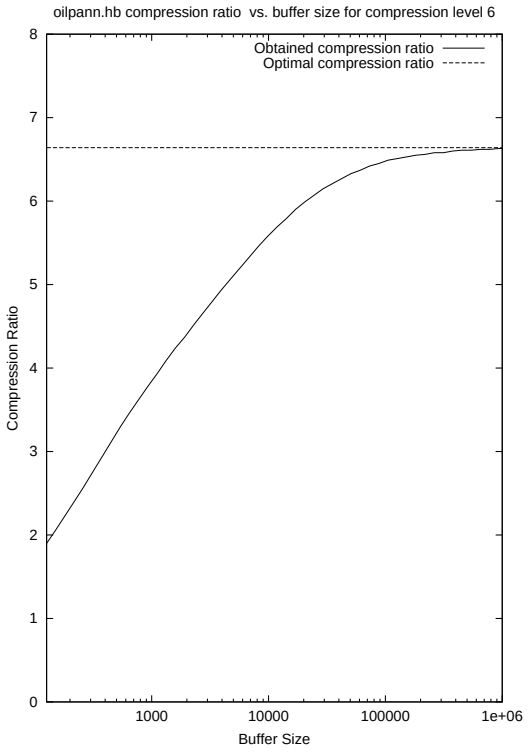

Figure 4. *oilpann.hb compression ratio vs. buffer size (logarithmic scale)*

Let ν be the packet size. Let B be the size of the buffer to be compressed. Let ρ be the compression ratio (hence we obtain a compressed buffer size $B' = B/\rho$). Let C be the time to compress one byte (the buffer is compressed in time CB), and C' the time to uncompress one byte (the buffer is uncompressed in time $C'B$). Let γ_c be the constant overhead to store a packet in the FIFO. Let $t = \frac{B}{\rho\nu}$ be the number of packets generated by the compression thread.

Hence the time to generate one packet is $T_a(B,\nu) = C\frac{B}{t} + \gamma_c = C\rho\nu + \gamma_c$, whereas the time to uncompress one packet is $T_c(B,\nu) = C'\rho\nu$.

For modelling the network we use a simplistic model that assumes a constant throughput and latency. Let γ_e be the constant overhead to extract a packet from the FIFO. Let τ be the time to transmit one byte and β the network latency. Hence the time to send one packet is $T_b(\nu) = \gamma_e + \beta + \tau\nu$. This is an old and simplistic model but, as shown in the following, it will give us an idea of the order of the packet size.

We have a pipeline scheme, hence, the time to transmit the whole buffer is :

$$\begin{aligned} T_t(B,\nu) &= T_a(B,\nu) + tT_b(\nu) + T_c(B,\nu) \\ &= C\rho\nu + \gamma_c + \frac{B}{\rho\nu}(\gamma_e + \beta + \nu\tau) + C'\rho\nu \\ &= (C+C')\rho\nu + \frac{B}{\rho\nu}(\gamma_e + \beta) + \gamma_c + \frac{\tau B}{\rho}. \end{aligned}$$

We want to minimize $T_t(B,\nu)$. We have:

$$\frac{\partial T_t}{\partial \nu} = (C+C')\rho - \frac{B}{\rho}(\gamma_e+\beta)\frac{1}{\nu^2}$$

and

$$\frac{\partial^2 T_t}{\partial \nu^2} = 2\frac{B}{\rho}(\gamma_e+\beta)\frac{1}{\nu^3}.$$

Since, $0 < \nu \leq \frac{B}{\rho}$ we have $\frac{\partial^2 T_t}{\partial \nu^2} > 0$. Hence $T_t(B,\nu)$ is concave and is minimum if

$$\frac{\partial T_t}{\partial \nu} = 0.$$

This is true when

$$(C+C')\rho = \frac{B}{\rho}(\gamma_e+\beta)\frac{1}{\nu^2}$$

or when

$$\nu = \sqrt{\frac{B(\gamma_e+\beta)}{\rho^2(C+C')}}.$$

Our experiments show that

$$1 \leq \rho \leq 7$$

$$100\,10^{-6}s \leq \beta \leq 50\,10^{-3}s$$

$$\gamma_e \approx 3\,10^{-6}s$$

$$1.9\,10^{-7}\text{s/byte} \leq C \leq 1.4\,10^{-6}\text{s/byte}$$

$$1.5\,10^{-7}\text{s/byte} \leq C' \leq 4.4\,10^{-7}\text{s/byte}$$

Since $B = 204800$ bytes we have $500 \leq \nu \leq 173\,550$. This interval is very large and just tells us that ν must be on the order of a few kilobytes. Moreover, we want our algorithm to be very reactive at the beginning when the compression level is zero. Hence we do not want a large value for ν. In order to optimize memory and disk accesses, the value of ν should be greater than the page size (4096 bytes in our case), so we finally settled on 8192 bytes — the `BUFSIZ` constant of `stdio.h`, also used by e.g. FTP.

4 Experimental Results

In this section, results on the behaviour of the AdOC algorithms are described. We have conducted our experiments on several UNIX versions (Linux, Solaris, FreeBSD). Performance depends only on the speed of the processor and the network, hence we only show the results gathered for Linux.

We have built a sender and a receiver that implements the AdOC algorithm. The current implementation uses POSIX threads, mutex and semaphore. Communication is done using TCP sockets.

4.1 Compression Level and FIFO Size Variation

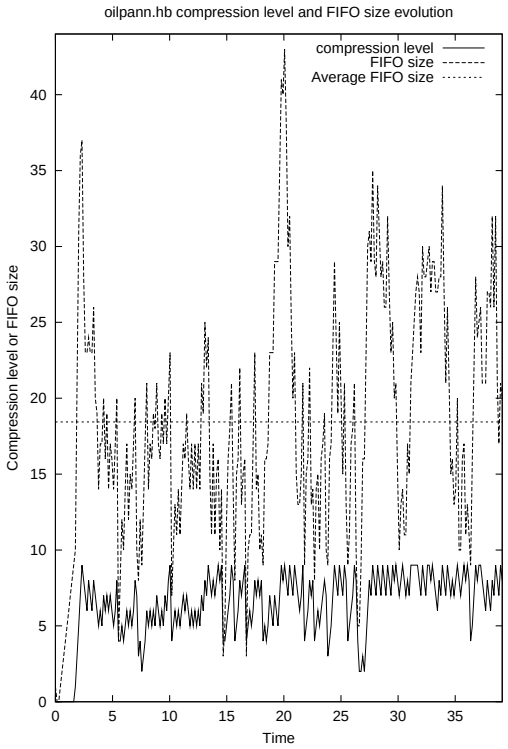

Figure 5. *Evolution of the FIFO size and the compression level vs. time*

In figure 5 we show the evolution of the FIFO size and

the compression level when sending `oilpann.hb` over the Internet between Nancy University and Rutgers University. Our goal is to maintain the compression level as high as possible and the FIFO size between 10 and 30.

Results show that most of the time the FIFO size is between the two values. The average size is 18.44. We see that when the FIFO size becomes small the compression level is reduced (e.g. at time 26) until the size regain an acceptable value.

4.2 File Transmission Timings

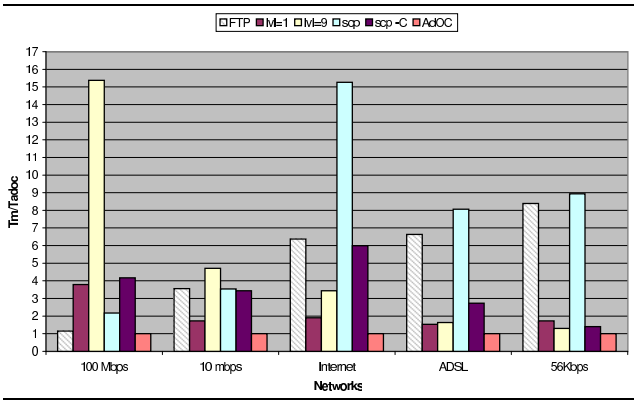

Figure 6. *Average Timings Ratio for oilpann.hb*

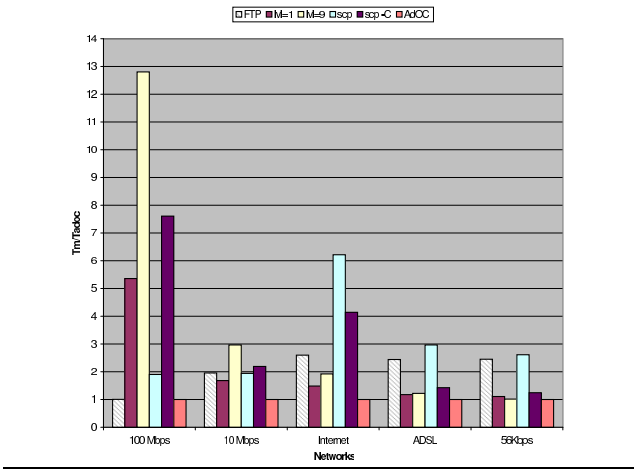

Figure 7. *Average Timings Ratio for bin.tar*

We have transmitted our two bench files on various networks. Results for `oilpann.hb` (resp `bin.tar`) are shown in Figure 6 (resp. Figure 7).

Results are the average of 10 timings of the transmission of the file. Results from the Internet measurements (between Nancy and Rutgers) are obviously not easily reproducible. However, repeated experiments show that regardless of when we run the experiment, AdOC outperforms any other method we have tried, since the ability to adapt is the core of the algorithm. This is the expected result.

We compare AdOC to a number of other ways of sending the data. An uncompressed FTP session will serve as the baseline for comparison. One obvious comparison would be to first compress the file and then send it. We have done this using gzip with compression levels 1 and 9. We have also compared to using the SSH network file copy command, `scp`, both with (`-C`) and without compression.

Figure 6 and 7 are bar graphs that show the relative performance of these methods against AdOC. For each kind of network we have divided the average timing of the method (Tm) by the average timing of AdOC (Tadoc). Hence if Tm/Tadoc is greater than 1, AdOC outperforms the considered method for the given network. For instance in Figure 7 one can see that for the Internet, FTP transfer is more than 6 times slower than AdOC.

These results show that our algorithm outperforms any other method for all kinds of networks we have tested.

The fact that AdOC outperfoms FTP transfers for 100Mbps network is possible only because when not compressing data, AdOC does not go through the zlib library. Rather, AdOC directly moves the uncompressed data to the FIFO. If we were to always go through the zlib library, as the original algorithm did, AdOC would take twice as long (11 seconds compared to 5.5 seconds) to send oilpann.hb over a 100Mbps network.

It should be noted that the utility of compressing increases with increasing CPU power, and with processing power doubling every 18 months (Moore's Law), this means that what may not be realistic today will be so in a few years.

The original algorithm was tested on 133 MHz Pentium processors that could just barely outperform sending uncompressed on a 10 Mbps Ethernet, and our results here show a noticeable improvement on 100 Mbps networks. Compression at gigabit speeds should only be a few years away with the existing algorithms, never mind future improvements.

5 Future Improvements

We could improve the performance of the receiver by decomposing it into two threads : one for reading the network and one for decompressing the packets. This would allow for an increase of network throughput in some cases, since network throughput will no longer depend on the speed of the receiver decompression.

Another improvement would be to have a faster compression method that would be between level 0 and level 1. This could allow compression to work even on networks so fast that they today would have to switch compression off, and would allow a smoother transition from the non-compressing state to the compressing state. The original paper tried using some special modes of `zlib` for this, but the results were not conclusive and were still relatively expensive.

Another extension would be to add better compression algorithms. Unless a leap in compression algorithms happens, this will typically mean one that either uses more memory, or more CPU, or both. One such candidate is the algorithm used in `bzip2` and the corresponding `libbzip2` library, which also have the advantage of having an API similar to that of `zlib`.

6 Concluding Remarks

This paper has revisited the adaptive algorithm proposed in earlier work by a subset of the authors of this paper. While the basics of the algorithm have not changed, the algorithm presented in this paper have explored new directions that allow the algorithm to be implemented without interaction with kernel-internal data and is significantly more portable. Other changes include measures that should make it more suitable for the transfers of large amounts of data often seen in distributed systems.

The paper also presents an analysis of compression, specifically in the domain of online or "on-the-fly" compression.

Future work is directed toward improving the performance of AdOC and integrating this algorithm in PSE environments such as NetSolve [3] or DIET [1].

We are currently developing a freely available AdOC library, more information is available at the web page `http://www.loria.fr/~ejeannot`.

References

[1] DIET (Distributed Interactive Engineering Toolbox). `http://www.ens-lyon.fr/~desprez/DIET/index.html`.

[2] E. Caron, S. Chaumette, S. Contassot-Vivier, F. Desprez, E. Fleury, C. Gomez, M. Goursat, E. Jeannot, D. Lazure, F. Lombard, J. Nicod, L. Philippe, M. Quinson, P. Ramet, J. Roman, F. Rubi, S. Steer, F. Suter, and G. Utard. Scilab to Scilab$_{//}$, the OURAGAN Project. *Parallel Computing*, 27(11), 2001.

[3] H. Casanova and J. Dongarra. Netsolve : A network server for solving computational science problems. In *Proceedings of Super-Computing'96*, Pittsburg, 1996.

[4] CCITT. *Recommendation V.42 bis*, F ITU 1990 edition.

[5] F. Desprez, P. Ramet, and J. Roman. Optimal Grain Size Computation for Pipelined Algorithms. In *Europar'96 Parallel Processing*, volume 1123 of *Lecture Notes in Computer Science*, pages 165–172. Springer Verlag, Aug. 1996.

[6] P. Deutsch. *GZIP file format specification version 4.3*. IETF, Network Working Group, 1996. RFC1952.

[7] P. Deutsch and J.-L. Gailly. *ZLIB Compressed Data Format Specification version 3.3*. IETF, Network Working Group, 1996. RFC1950.

[8] F. Douglis. The compression cache: Using online compression to extend physical memory. In *proceedings of the Winter Technical Conference, USENIX Association*, pages 519 – 529, Jan. 1993.

[9] D. Huffman. A method for the construction of minimum redundancy codes. In *Proceedings of the Institute of Radio Engineers*, volume 40, pages 1098–1101, 1952.

[10] B. Knutsson. *Architectures for Application Transparent Proxies: A Study of Network Enhancing Software*. PhD thesis, Uppsala University, May 2001.

[11] B. Knutsson and M. Björkman. Trading computation for communication by end-to-end compression. In *Third International Workshop on High Performance Protocol Architectures (HIPPARCH'97)*, 1997.

[12] B. Knutsson and M. Björkman. Adaptive end-to-end compression for variable-bandwidth communication. *Computer Networks*, (31):767–779, 1999.

[13] D. A. Lelewer and D. S. Hirschberg. Data compression. *Computing Surveys*, 19(3):261–297, 1987. Reprinted in Japanese BIT Special issue in Computer Science (1989) 165-195.

[14] A. Mallet, J. Chung, and J.M.Smith. Operating systems support for protocol boosters. In *Third International Workshop on High Performance Protocol Architectures (HIPPARCH'97)*, 1997.

[15] G. S. Pall. *Microsoft Point-To-Point Compression (MPPC) Protocol*. IETF, Network Working Group, 1997. RFC2118.

[16] D. Rand. *The PPP Compression Control Protocol (CCP)*. IETF, Network Working Group, 1996. RFC1962.

[17] M. Sato, H. Nakada, S. Sekiguchi, S. Matsuoka, U. Nagashima, and H.Takagi. Ninf: A Network based Information Library for a Global World-Wide Computing Infrastracture. In *HPCN'97 (LNCS-1225*, pages 491–502, 1997.

[18] V. Schryver. *PPP BSD Compression Protocol*. IETF, Network Working Group, 1996. RFC1977.

[19] T. A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, June 1984.

[20] J. Woods. *PPP Deflate Protocol*. IETF, Network Working Group, 1996. RFC1979.

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

[22] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.