ACE: A Resource-Aware *A*daptive *C*ompression *E*nvironment

Sezgin Sucu Chandra Krintz

Computer Science Department
University of California, Santa Barbara
{sucu,ckrintz}@cs.ucsb.edu

Abstract

We present an adaptive compression environment (ACE) that uses both underlying resource performance (network and CPU) predictions and the compressibility of the data stream to perform on-the-fly compression. ACE couples technologies from Computational Grid Computing and dynamic compilation research to predict the efficacy of using compression and to intercept and compress socket-based communication transparently. Our results show that ACE is able to accurately determine when to compress and not to compress. Over all file formats studied, ACE enables significant improvements over solely performing compression or no compression.

1 Introduction

The growth and evolution of the Internet has facilitated mass demand for always-available, high-performance, and world-wide access by Internet users. These users, as indicated by the popularity of email, web pages, mobile languages, and peer-to-peer systems, use the Internet primarily for data transfer. However, transfer performance is dictated by the underlying resource performance of the Internet and as such, is sensitive to the vast differences in network, end-point technology and the high variability in performance.

Compression is commonly used to reduce the number of bytes transferred and to increase the effective bandwidth available. However, there are three primary constraints that must be considered in the selection of a compression technique for *on-line* compression: (1) compression and decompression time depends on CPU performance and data transfer time depends on network performance; (2) the data characteristics impact compression performance (compression rate and compression ratio) and it is not possible to learn this effect without first compressing [2]; and (3) compression techniques vary in performance in terms of compression rate and compression ratio [14]. Each of these constraints make it increasingly difficult to identify the "best" compression technique in all circumstances.

To address challenges associated with on-line compression and to improve Internet transfer performance, we have developed ACE, an *Adaptive Compression Environment* which automatically and transparently selects between competitive compression techniques (and not compressing) based on underlying resource performances (CPU, bandwidth, and memory). ACE is novel in that it combines existing technologies from Computational Grid Computing [6] and dynamic compilation research [4, 12] to perform adaptive compression. ACE is able to accurately identify and implement compression when its use will reduce transfer time. In all other cases, ACE approximates to the default "no-compression" case by sending data uncompressed.

2 The Adaptive Compression Environment (ACE)

The ACE runtime executes programs on both local and remote machines and intercepts TCP/IP socket read and write calls made by programs for data transfer. ACE then estimates the cost of performing on-the-fly compression and makes a compression decision for each fixed size block of data to speed-up data transfer.

ACE computes the predicted cost of data transfer with compression using past socket behavior, data stream characteristics, properties of the available compression and decompression algorithms, network bandwidth, and local and remote CPU load. Bandwidth and CPU load are acquired by ACE from the Network Weather Service (NWS) [20], a resource performance monitoring and prediction system used for Computational Grid Environments [6]. All other cost components are measured and computed directly by ACE.

To implement ACE, we extended a Java Virtual Machine (JVM) called the Open Runtime Platform (ORP) [4]. ORP is a dynamic optimization system from Intel Corporation that is available as open source. Our decision to use Java as our language and ORP as our infrastructure for ACE implementation enables us to implement automatic socket interception for Java programs transparently. For our ACE prototype we extended ORP with the zlib compression library [21]. Since zlib requires the size of the compressed data block to decompress it, we include include a 4-byte header (indicating size) with every block.

2.1 Performance Models

ACE considers only send requests that are larger than 32KB and divides larger requests into 32KB blocks; smaller send requests are forwarded unimpeded. We empirically discovered that 32KB is the best value given many constraints. Smaller values for the block size cause zlib to attain smaller compression ratios but allow ACE to quickly adapt to changes within the data stream. Larger values impose a high performance cost when data is not compressible, without enabling significant improvement in compression ratio.

To determine when to compress each 32KB block, ACE uses two performance models which assume that the overall goal of the application is to speed-up end-to-end data communication. The two performance models are the *Sequential Model* and *Pipeline Model*. We consider only these two here for brevity. However, ACE is extensible and as such, performance models with different goals can be added easily.

The *sequential model* assumes that the rate at which data is produced at the local host is slow enough to preclude overlap of compression with data transfer. ACE monitors the *data generation rate*: the number of bytes per second that the local host sends through socket-write calls. If this rate is less than the bandwidth between the local and remote host, ACE uses the sequential model to make compression decisions.

Using the sequential model, ACE computes the compressed transfer time for each 32KB of data to be sent as $T_c(l,r) = 32KB/(CR * NWS\_bandwidth(l,r)) + C_l(CR) + D_r(CR)$, where CR, $C_l(CR)$ and $D_r(CR)$ are the predicted values for compression ratio, compression time, and decompression time, respectively, for each 32KB block sent. $NWS\_bandwidth(l,r)$ is the bandwidth between the local and remote hosts as predicted by the NWS. $32KB$ is the ACE block size (transfer unit). ACE computes uncompressed transfer time as $T_u(l,r) = 32KB/NWS\_bandwidth(l,r)$ and compares this value with compressed transfer time to make a decision. If the uncompressed transfer time is larger, the sequential model selects compression, otherwise the original data is sent.

ACE uses the *pipeline model* when the data generation rate is greater than the available bandwidth, i.e., compression can be overlapped with data transfer. ACE considers three pipeline stages: compression rate, effective data transfer rate, and decompression rate. If the slowest of these rates is larger than the available bandwidth (indicating that overlap is possible), compression is used; otherwise the original data is sent uncompressed. Compression rate is computed as $32KB/C_l(CR)$. Decompression rate is computed as $32KB/D_r(CR)$. The effective data transfer rate is computed as $CR * NWS\_bandwidth(l,r)$.

A fourth pipeline stage is also necessary: the *data consumption rate* of the remote host. The data consumption rate is the number of bytes per second the remote host consumes data via socket reads. When this consumption rate is slower than the available bandwidth, ACE discontinues the use of the pipeline model. ACE discovers the consumption rate indirectly: when the data consumption rate of the remote host is slower than the network transfer speed, after some time the kernel-level receive buffer at the remote host will fill and cause the local host to block. This, in turn, decreases the data generation rate at the local host which causes ACE to reconsider using the pipeline model.

2.2 ACE Prediction Infrastructure

The ACE prediction infrastructure makes forecasts of compression ratio, compression time, and decompression time for each 32KB block sent. We define *compression ratio* as S_{old}/S_{new} where S_{old} is the size of the original block (32KB in our case) and S_{new} is the size of the data block when compressed. For convenience, we also define *inverse compression ratio* as S_{new}/S_{old}.

ACE estimates compression ratio via sampling. We compress the first block of data regardless of whether doing so results in the best performance and use the resulting compression ratio for subsequent blocks. As long as ACE decides to compress, it uses the previous compression ratio to make a prediction for the next block. So CR is computed as the compression ratio of the last compressed block. If ACE decides not to compress, it discontinues compression until NWS values for CPU and bandwidth change in favor of compression.

One limitation of this implementation is that the entropy of the data may change, making compression feasible again, while ACE has suspended compression. Therefore, we periodically re-introduce compression. If ACE discovers that performing compression continues to be the wrong decision, the period is extended by the *initial period value* so that ACE waits longer next time before attempting re-introduction.

The *initial period* is a weighted block count. ACE initially waits until this number of blocks has been sent before attempting re-introduction. A user-defined block count is weighted by the ratio of the maximum of compression time and decompression time to uncompressed transfer time to calculate the *initial period*. We use an empirically determined, initial period of 10. We refer to this value as K. We compute the re-introduction initial period as: $initialP = K*(max(C_l(CR), D_r(CR)))/T_u(l,r))$. By weighting K, we effectively reduce initial period when compression (or decompression) time is short compared to uncompressed transfer time. Similarly, we increase initial

Table 1: Machine characteristics.To illustrate the differences in computational power across these machines, the third and fourth columns show the regression line parameters for compression; those for decompression are shown in columns five and six.

Machine Name	CPU	RAM	c^0	c^1	d^0	d^1
suns	1xPentium 4 (Xeon) 2.4GHz	2 GB	0.0005	0.0038	0.0003	0.0011
heat	1xPentium 4 (Xeon) 2.2GHz	512 MB	0.0005	0.0044	0.0003	0.0012
gibson	2xPentium 3 500Mhz	512 MB	0.0026	0.0166	0.0010	0.0028

Table 2: File statistics.

File	Size (MB)	ICR (ratio)	CS (byte/microsec)	DS (byte/micsec)	MICR (ratio)	MCS (byte/microsec)	MDS (byte/micsec)
hs_chrY.gbk	28.43	0.36	15.06	35.40	0.36	14.70	37.35
DG_1.mpg	46.77	0.33	21.84	75.36	0.34	27.49	94.45
partial.adl_catalog.txt	8.93	0.13	47.74	84.94	0.13	39.00	84.00
cc_xacts.MYD	10.58	0.73	9.45	31.99	0.73	8.71	31.74
HUH-Publ.xml	4.28	0.30	22.41	52.54	0.30	19.24	51.79
lion.mpg	9.72	0.92	8.45	36.30	0.92	8.15	38.58
paper	10.88	0.40	19.07	58.38	0.40	25.16	64.60
adobe.tar	11.71	0.80	10.17	66.03	0.80	10.99	172.53

period when the opposite is likely. Note that the initial period does not affect ACE when compression continues without interruption.

To predict compression time ($C_l(CR)$), we used the *Canterbury corpus* [3] to compute the correlation between inverse compression ratio and compression time at the block level. Across this data set, the correlation is 0.9889. The scatter plot of compression time vs. inverse compression ratio (not shown here due to space constraints) for each 32KB block confirms this linear relationship. We use this correlation via a linear regression to estimate compression time as follows: $C_l(CR) = (c_l^0 + c_l^1 * (1/CR))/NWS\_CPU_l$, where NWS_CPU_l is the available CPU percentage reported by NWS for the local machine. c_l^1 is the slope of the regression line and the c_l^0 is the y-intercept of the regression line. These values vary from host to host reflecting the speed of each.

The authors in [13] believe that it is not safe to relate compression rate with compression ratio across the data streams. However, we found that by using a regression line to compute compression time, we achieve accuracy levels that are acceptable for compressed transfer time predictions within ACE.

We compute decompression time ($D_r(CR)$) similarly using a regression line constructed from the Canterbury corpus. The regression line for decompression is represented with d_r^0 and d_r^1 parameters (slope and y-intercept of the line). Decompression time is computed as: $D_r(CR) = (d_r^0 + d_r^1 * (1/CR))/NWS\_CPU_r$, where NWS_CPU_r is the predicted available CPU percentage for the remote host.

We can obtain regression line parameters for compression time at the local host via measurement. However, we must retrieve the remote decompression parameters (d_r^0 and d_r^1) via alternate means. We obtain these values using the lightweight directory access protocol (LDAP). LDAP is commonly used in distributed and Grid computing environments to store system configurations and characteristics about distributed resources. For hosts without LDAP entries, these values can also be communicated via a handshake protocol between the local and remote ACE runtimes. For cases in which these values cannot be computed off-line, an estimated value can be used.

Table 3: Benchmark files.

File	Description
hs_chrY.gbk	Homo sapiens chromosome 1 [7]
DG_1.mpg	Concatenation of a series of computer generated movies [5]
partial.adl_catalog.txt	A text dump of a table from Alexandria Digital Library [19]
cc_xacts.MYD	A dbase3 table with 16777230 records from TPC-W [18]
HUH-Publ.xml	Botanical publications XML file [9]
lion.mpg	Higly compressed mpeg file [15]
paper	Artificially created file whose entropy dramatically changes from start to end [19]
adobe.tar	Uncompressed archive of class files for reading pdf files [19].

3 Experimental Methodology

We used two *non-dedicated* networks to empirically evaluate ACE. The first is a cross-campus link between the Computer Science department at the University of

Table 4: ACE fast-network performance results. *NetLd* is the network load level, *RemLd* is the CPU load level on the client (`heat`), and *LocLd* is the CPU load level on `suns`, the server. T_{Never} is the total time for the `heat` to receive all 7 benchmark files using Never-Compress method. T_{Always} is the total time using Always-Compress method and T_{ACE} is the total time using our adaptive ACE method.

Fast Network Results					
NetLd	LocLd	RemLd	T_{Never}	T_{Always}	T_{ACE}
No Load	No Load	No Load	15.86	8.71	10.12
No Load	No Load	Medium	18.62	23.91	18.44
No Load	No Load	Heavy	30.31	42.97	32.95
No Load	Medium	No Load	15.90	102.57	15.87
No Load	Medium	Medium	18.55	92.26	18.64
No Load	Medium	Heavy	32.24	44.71	33.81
No Load	Heavy	No Load	15.98	232.21	15.96
No Load	Heavy	Medium	18.50	237.55	18.81
No Load	Heavy	Heavy	32.03	51.75	32.03
Medium	No Load	No Load	26.01	14.97	20.14
Medium	No Load	Medium	30.10	26.82	26.01
Medium	No Load	Heavy	33.76	51.60	33.39
Medium	Medium	No Load	29.42	131.62	25.94
Medium	Medium	Medium	29.64	95.59	27.75
Medium	Medium	Heavy	33.26	49.16	34.82
Medium	Heavy	No Load	28.00	305.22	28.39
Medium	Heavy	Medium	27.56	424.14	27.17
Medium	Heavy	Heavy	32.89	49.64	37.41
Heavy	No Load	No Load	34.95	20.10	22.94
Heavy	No Load	Medium	37.27	34.19	32.01
Heavy	No Load	Heavy	41.24	58.99	57.93
Heavy	Medium	No Load	37.14	59.33	40.48
Heavy	Medium	Medium	35.62	41.56	32.37
Heavy	Medium	Heavy	39.96	57.76	47.92
Heavy	Heavy	No Load	31.88	131.32	32.55
Heavy	Heavy	Medium	36.30	58.61	36.65
Heavy	Heavy	Heavy	39.42	55.12	41.82

California, Santa Barbara (UCSB) and our research lab (the RACELAB [19]). We achieve an average bandwidth over this 100Mb/s link of 71Mb/s; we refer to this link as the *Fast Network* in our results. The second is an Internet link that uses Abilene technology [1] between the UCSB Computer Science Department and the University of Tennessee, Knoxville (UTK). We experience an average bandwidth of 1.7Mb/s on this link; we refer to this link as the *Slow Network* in our results.

To generate the results that follow, we implemented Java client and server programs that execute within ACE (or any JVM). The client requests files over a network from the server which responds with the requested files. We measured file transfer time at the client host one after another for each experimental setup. This time includes the time for the client host to request the file, any overhead introduced ACE (at either end), compression time, transfer time, and decompression time. The endpoint machines in our experiments are `suns` (in the UCSB CS department), `heat` (in the UCSB RACELAB), and `gibson` at UTK. For all experiments, `suns` is the server host and `heat` and `gibson` are the client hosts. Table 1 shows the various performance characteristics of these machines.

We selected seven files (described in Table 3) of various types with different compression characteristics for ACE experimentation. Table 2 shows the general statistics of these files. The *Size* column is the total size of the file (in MB). *ICR* is the inverse compression ratio achieved when the whole file is compressed in memory. *CS* and *DS* are compression and decompression rates, respectively, for in-memory compression. The last three columns show block-level compression statistics: *MICR* is the average inverse compression ratio per 32KB block, and *MCS* and *MDS* show the average compression and decompression rates (bytes per microsecond).

Notice that the use of a 32KB block compression does not significantly impact compression ratio. In fact, for files with a high standard deviation for compression ratio (indicating that some blocks are significantly more compressible than others), we discovered that using 32KB blocks commonly yields higher compression and decompression rates over compression over entire files. We believe that this occurs since, for block-based compression, we force zlib to flush all pending output and process all the input, i.e., zlib processes the input data and outputs all compressed data at once.

4 Results

To empirically evaluate the effectiveness of ACE, we considered three scenarios: *Never-Compress*, *Always-Compress*, and *ACE compression*. With Never-Compress, ACE transfered data without compression. For Always-Compress, ACE compressed and transfered the data in 32KB blocks and then decompressed the data at the client host. For both of these cases, ACE introduces no overhead for prediction or NWS access. The final scenario, ACE compression, is the system that we describe herein.

For each of our experiments, we considered unloaded and loaded resources (CPU and network) to evaluate how effectively ACE adapts to changing resource performance conditions. We considered four levels of CPU load: *No Load* in which there are no other processes running; *Medium Load* in which there are 4 dummy processes running on the client hosts, and 8 running on the server host; *Heavy Load* in which there are 8 dummy processes on the client hosts and 20 on the server host; and *Very Heavy Load* in which there are 180 dummy processes on the server host. The dummy processes are programs that repeatedly execute floating point operations.

For our fast network experiments, we introduced network traffic using custom traffic generators: client-server pairs repeatedly exchange 4KB of data. We introduced three levels of network load: *No Load* in which there is no traffic generated, *Medium Load* in which there are 12 traffic generator pairs, and *Heavy Load* in which there are 20 traffic generator pairs. For the slow network, we did not introduce traffic generators since

Table 5: ACE performance results for slow network (suns-gibson).

Slow (Internet) Network Results					
NetLd	LocLd	RemLd	T_{Never}	T_{Always}	T_{ACE}
No Load	No Load	No Load	285.08	138.23	126.72
No Load	No Load	Medium	270.92	136.01	139.02
No Load	No Load	Heavy	269.00	137.58	134.87
No Load	Medium	No Load	252.30	121.69	135.11
No Load	Medium	Medium	256.25	129.92	152.59
No Load	Medium	Heavy	246.25	128.42	134.55
No Load	Heavy	No Load	237.16	114.50	119.64
No Load	Heavy	Medium	194.01	115.89	114.54
No Load	Heavy	Heavy	236.49	117.43	122.17
No Load	Very Heavy	No Load	103.75	1993.23	131.99
No Load	Very Heavy	Medium	198.11	2032.10	181.72
No Load	Very Heavy	Heavy	484.15	2232.10	580.75

the bandwidth is very low without them.

In Tables 4 and 5, we show the average transfer times (secs) across all benchmark files using various CPU and network loads for the fast and slow network, respectively. The data indicates that for a given experiment, ACE (T_ACE) successfully approximates to either Always-Compress (T_{Always}) or Never-Compress (T_{Never}) depending on which enables better performance given the underlying resource performance.

For our fast network experiments using unloaded end-points and variable network load, ACE enables significant performance improvement over Never-Compress. ACE enables similar improvements over Always-Compress as CPU load increases. Our results indicate that CPU load has a very dramatic effect on the compression decision. As such, Always-Compress does poorly when CPU load increases.

ACE can achieve the *best* performance across benchmarks since it considers individual data characteristics. This is exemplified in Figure 1 which shows the fast network transfer time (secs) for the individual benchmark files using the Medium-Medium-Medium load scenario (the first Medium refers to the network load, the second Medium refers to the server CPU load (`suns`), and the last Medium refers to the client CPU load (`heat`)). For some files (*paper, partial.adl_catalog.txt*), ACE correctly determines that compression should be used; for others, ACE avoids compression.

Since bandwidth is a significant bottleneck for the slow network, only under very heavy CPU load should compression be avoided as indicated by the ACE results that approximate to Never-Compress for these cases. We have left out the results for Very Heavy CPU load for the fast network for brevity since the effect of CPU load and hence the efficacy of ACE can be observed under Medium and Heavy load.

The results also indicate that our use of adaptive re-introduction of compression is effective and that the overhead introduced by ACE is very small. The former is shown by the results in which ACE approximates to

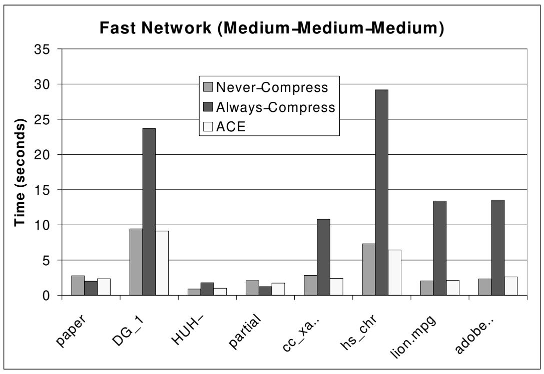

Figure 1: Per-file fast network results for the Medium-Medium-Medium load scenario (medium network load, medium server CPU load, and medium client CPU load).

Never-Compress. This occurs when the underlying resource performance changes in a way that makes on-the-fly compression infeasible, e.g., when CPU load increases. The latter (minimal ACE overhead) is exhibited in the fast network results (in Table 4) for the NoLoad-Heavy-Heavy and the Heavy-NoLoad-NoLoad scenarios in which ACE compresses few blocks and almost all blocks, respectively. By comparing Never-Compress with ACE in the first case and Always-Compress with ACE in the second case, it is seen that the difference is very small. On average over all files, this difference is 0.0s and 0.4s, respectively.

To see the effect of compression level we have conducted several experiments using different zlib compression levels (1, 3, 6 and 9). For the previous results, we used a compression level of 1 only since it is very fast. Figure 2 shows the effect of varying compression level for the fast network in which the client, server, and network is unloaded. Others [17] have shown that using higher compression levels commonly results in lower compression rates without any significant gain in compression ratio. Our experimental results confirm this. In addition, as shown by the results for the hs_chrY.gbk file, increasing the compression level can severely degrade performance. In our slow-network experiments (not shown here), the performance difference between compression levels is not noticeable since bandwidth is the main bottleneck and compression ratios do not differ much by compression level.

5 Related Work

The research that is most closely related to ACE uses resource performance characteristics to make compression decisions. We previously considered dynamic network performance to select between Java programs that were pre-compressed using different compression formats [12]. ACE is significantly more general in that it can be used for any type of file and performs on-the-fly

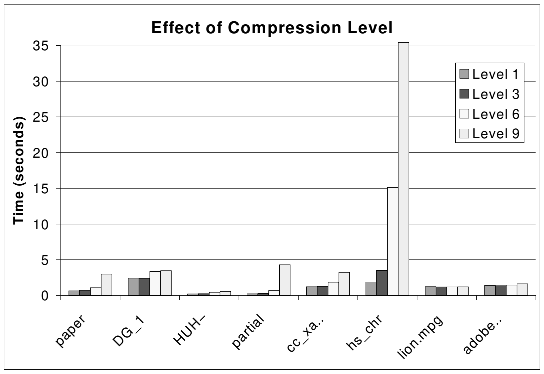

Figure 2: The effect of compression level on transfer performance. The graph shows the total time (for compression, compressed-transfer, and decompression) using the fast network in which the client, server, and network is unloaded.

compression. In addition, ACE considers CPU load and intercepts TCP/IP socket calls transparently.

In [16], the authors describe a dynamic compression algorithm for text files which considers network performance and server load. Their work is restricted to text files and intended for web servers. Our work dynamically adapts to all types of files as well as to the entropy within a single file. In addition, ACE considers CPU load which our results indicate is vital for improved transfer performance in an Internet setting. In contrast, this prior work only considers the number of clients the server processes. The authors in [8] present a similar system but again only consider network performance.

A form of adaptive compression that is similar to ACE is described in [11] and then extended in [10]. The goal of this work is to vary the level of on-the-fly compression used so that the network is never under-utilized. This work differs from ACE in that it does not consider the CPU load of the remote host. In addition, this prior work assumes that higher compression levels will result in better compression ratio. We find that this is not always true, e.g., when data is not compressible. ACE is also different in that, it uses forecasts of future resource performance which effectively prevents oscillations in the algorithm presented in this prior work.

6 Conclusions

We present ACE, an adaptive compression execution environment that improves Internet transfer performance by dynamically selecting between competitive compression algorithms for on-the-fly compression. ACE makes its decisions by predicting and comparing transfer performance for both uncompressed and compressed transfer. ACE is able to adapt to both the changes in resource performance and the compressibility of the data. Our empirical evaluation of ACE for both local area and Internet links, indicate that the overhead introduced by ACE is minimal and that the accuracy with which ACE makes its estimates enables significant Internet transfer performance improvement.

References

[1] Abilene Network Technology. `http://abilene.internet2.edu/`.

[2] Ross Arnold and Timothy C. Bell. A corpus for the evaluation of lossless compression algorithms. In *Designs, Codes and Cryptography*, pages 201–210, 1997.

[3] Canterbury corpus. `http://corpus.canterbury.ac.nz/`.

[4] M. Cierniak, G. Lueh, and J. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 13–26, June 2000.

[5] A series of computer generated movies used in our experiments. `http://www5.in.tum.de/forschung/visualisierung/duenne_gitter.html/`.

[6] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., 1998.

[7] Homo sapiens genomic sequence for chromosome 1. `ftp://ftp.ncbi.nih.gov/genomes/H_sapiens/CHR_Y/`.

[8] Ningning Hu. Network aware data transmission with compression. Technical Report CMU-CS-01-164, Carnegie Mellon, 2001.

[9] Botanical publications xml file. `http://www.huh.harvard.edu/databases/cms/download.html`.

[10] E. Jeannot, B. Knutsson, and Mats Bjorkman. Adaptive online data compression. In *HPDC'02*, July 2002.

[11] B. Knutsson and M. Bjorkman. Adaptive end-to-end compression for variable-bandwidth communication. *Computer Networks*, 31(7):767–779, April 1999.

[12] C. Krintz and B. Calder. Reducing Transfer Delay with Dyanamic Selection of Wire-Transfer Formats. In *Tenth IEEE International Symposium on High Performance Distributed Computing*, August 2001.

[13] J. Lee, M. Winslett, X. Ma, and S. Yu. Enhancing data migration performance via parallel data compression. In *International Parallel and Distributed Processing Symposium*, April 2002.

[14] D. A. Lelewer and D. S. Hirschberg. Data compression. *ACM Computing, Springer Verlag (Heidelberg, FRG and NewYork NY, USA)-Verlag Surveys, ; ACM CR 8902-0069*, 19(3), 1987.

[15] Highly compressed mpeg file used in our experiments. `http://home.in.tum.de/~paula/mpeg/lion.mpg`.

[16] N. Motgi and A. Mukherjee. Network conscious text compression system (nctcsys). In *International Conference on Information Technology: Coding and Computing*, April 2001.

[17] O. Pentakalos and Y. Yesha. Online data compression for mass storage file systems. Technical Report TR-CS-95-05, University of Maryland Baltimore County, July 1995.

[18] Tpc-w benchmark suit. `http://www.tpc.org/tpcw/default.asp`.

[19] UCSB RACELAB: The laboratory for Research on Adaptive Compilation Environments. `http://www.cs.ucsb.edu/~racelab`.

[20] R. Wolski. Dynamically Forecasting Network Performance Using the Network Weather Service. *Journal of Cluster Computing*, 1:119–132, January 1998.

[21] ZLib compression library. `http://www.gzip.org/zlib/`.