Adaptive On-the-Fly Compression

Chandra Krintz and Sezgin Sucu

Abstract—We present a system called the Adaptive Compression Environment (ACE) that automatically and transparently applies compression (on-the-fly) to a communication stream to improve network transfer performance. ACE uses a series of estimation techniques to make short-term forecasts of compressed and uncompressed transfer time at the 32KB block level. ACE considers underlying networking technology, available resource performance, and data characteristics as part of its estimations to determine which compression algorithm to apply (if any). Our empirical evaluation shows that, on average, ACE improves transfer performance given changing network types and performance characteristics by 8 to 93 percent over using the popular compression techniques that we studied (Bzip, Zlib, LZO, and no compression) alone.

Index Terms—Adaptive compression, dynamic, performance prediction, mobile systems.

1 Introduction

Due to recent advances in Internet technology, the demand for network bandwidth has grown rapidly. New distributed computing technologies, such as mobile computing, P2P systems [10], Grid computing [11], Web Services [7], and multimedia applications (that transmit video, music, data, graphics files), cause bandwidth demand to double every year [21]. Not only does the volume of data transmitted by these applications increase, but so does the frequency of transmission. As a result, novel techniques are needed that increase the network bandwidth available to Internet applications.

Compression is one such technique that increases the amount of bandwidth available to an application. Compression can be used offline (before compressed transfer commences) or on-the-fly (as the data is generated). Compression techniques reduce the amount of data transmitted by eliminating much of the redundancy that is characteristic in most data sets.

Unfortunately, there are several limitations inherent in the use of compression for efficient data communication. First, compression techniques vary in their performance characteristics: compressed size, compression time, and decompression time. Techniques are either optimized to be fast or to enable significant compaction. This trade-off is fundamental to performing compression due to the algorithmic complexity required to remove a significant portion of the redundancy in the data. Thus, no single compression technique enables the best performance for all data formats.

Second, compression performance is dependent upon the performance and availability of the underlying resources, e.g., network bandwidth and latency, CPU loads. This performance varies significantly across technologies as well as over time for the same technology. The latter impacts mobile devices like laptops for which the underlying networking technology available changes regularly, e.g., a user connects her laptop to an ISDN link at home, takes her laptop to work and connects via a 100Mb/s Ethernet link, and attends a conference or visits a coffee shop where she uses the wireless communication infrastructure that is available. The compression technique that performs best for this user will depend on the network she is connected to. Moreover, as the load changes at communication end-points or on the network itself, the best-performing compression technique for the same network can also change.

Each of these limitations makes it increasingly difficult for users to identify the best compression technique in all circumstances. In this paper, we present a novel compression system that identifies the best compression technique automatically and transparently. Our system, called the Adaptive Compression Environment (ACE), intercepts program communication and applies compression on-the-fly. On-the-fly compression is useful to or required by many applications due to storage capacity (limited or abundant), or to the nature of the data-generation process, e.g., dynamic Web and data streaming applications.

ACE *adaptively* selects a compression technique that best suites the underlying networking technology, performance available, and data characteristics, to improve communication performance. To enable this, ACE predicts whether applying compression will be profitable, and when it is, which compression algorithm to apply. ACE selects between a number of well-known, competitive, compression techniques including Bzip [3], Zlib [26], and LZO [18]. To make predictions of underlying resource performance, ACE employs the Network Weather Service (NWS) [24], [23], an efficient and accurate forecasting toolkit used in Computational Grid [9] systems. ACE couples NWS predictions with those from its own internal models that estimate compression performance and changes in data compressibility, i.e., entropy.

We empirically compare ACE to commonly used compression techniques: no compression, Bzip, LZO, and Zlib. Given different network and end-point performance, each of these techniques outperforms the others for different transfer scenarios. ACE is able to perform

• *The authors are with the Computer Science Department, University of California, Santa Barbara, CA 93106-5110.*
E-mail: {ckrintz, sucu}@cs.ucsb.edu.

Manuscript received 28 Jan. 2004; revised 16 Nov. 2004; accepted 30 Mar. 2005; published online 28 Nov. 2005.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0027-0104.

similarly to the best performing technique in all scenarios. The key contribution made by ACE is that it can adapt to *changing* performance levels (CPU and network performance). We investigate three such scenarios in which the underlying network technology changes and when CPU and network load changes. We show that ACE is able to improve transfer performance by 8 to 93 percent on average across the experiments and scenarios that we investigated.

In the following section, we overview the ACE system and the extant technologies that it employs and extends. We overview the ACE design and implementation in Section 3 and its prediction system in Section 4. We then present our empirical evaluation of ACE in Section 5, the related work in Section 6, and conclude in Section 7.

2 The Adaptive Compression Environment (ACE)

To evaluate the efficacy with which we can apply compression automatically and transparently based on predictions of future resource performance and data entropy, we developed the Adaptive Compression Environment (ACE). ACE couples two different extant technologies: The Open Runtime Platform (ORP) [6], [1] from the Intel Microprocessor Research Lab (MRL), and the Network Weather Service (NWS) [25], [23] from the University of California, Santa Barbara.

ORP is an open-source, dual compiler, adaptive compilation system for Java. We selected ORP because of its efficient implementation and cutting-edge technologies (dynamic compilation and adaptive optimization). ACE, however, is not Java-dependent. That is, we can implement the ACE modules in an operating system, e.g., Linux. However, we chose to prototype our techniques in a Java Virtual Machine due to the reduced complexity of the system implementation (over Linux) and to our extensive prior experience with the language and execution environment.

We extended ORP with a module that transparently intercepts TCP/IP socket communications and automatically decides when to apply compression and which compression technique to use. Since compression performance (compressed or uncompressed transfer time, compression speed, and decompression speed) depends on the availability and performance of the underlying resources (network, CPU), we also extended ORP with an interface to the Network Weather Service (NWS).

The NWS is an open-source, resource performance measurement and prediction toolkit, developed for Computational Grid Environments [9], [2]. The NWS monitors and makes periodic measurements of a wide range of resources including CPU, memory, network latency and bandwidth, and disk latency. Lightweight processes, called NWS sensors, execute on the devices of interest and sample the performance. The sensors communicate their data to a distributed storage system. When users of the toolkit require a prediction of future resource performance, the NWS prediction system treats the stored data values as a time series, applies a set of very fast, adaptive, statistical forecasting techniques, and returns an accurate, short-term performance estimate [24]. The NWS exports a well-defined interface through which ACE can efficiently acquire the predictions and actual measurements; ACE uses the latter to evaluate the efficacy of its decisions.

ACE intercepts (blocking) TCP/IP socket communications made by Java programs. When a connection between two hosts is established, ACE registers the host IP addresses with the (possibly remote) NWS system. Each host executes an NWS CPU and network bandwidth sensor that are initiated by ACE if they are not already running.

When ACE determines that a transmission of sufficient length is being made, it queries the NWS system for forecasts of future network bandwidth and latency between the hosts and the future CPU availability of each. In addition to predictions of underlying resource performance, ACE uses past socket behavior, predicted compression ratio, feedback on the accuracy of its previous compression decisions, and the characteristics of the available compression algorithms, to compute the *predicted transfer performance*. Using this information, ACE decides when to apply compression and which compression technique to use (if any).

ACE identifies when an *incoming* stream is compressed and decompresses it. ACE forwards the decompressed data to the application. We next describe the primary components of the ACE.

3 ACE Implementation

To enable transparent compression at the TCP socket level, ACE intercepts calls within the native socket implementation. That is, ACE does not modify the code of the programs that are executing, in any way. ACE intercepts socket connect, accept, read, write, send, and recv calls.

ACE determines when to compress by monitoring socket write[1] behavior. ACE considers data in 32KB blocks. We selected a 32KB block size by collecting compression performance metrics for a large number of diverse file types and three different compression algorithms (LZO, Bzip, and Zlib). We empirically evaluated block sizes ranging from 4KB to 128KB. The 32KB block size exhibited the best trade-off across file types and metrics.

Once ACE identifies a potential opportunity for on-the-fly compression (i.e., at least 32KB is being transmitted), it employs its prediction system to determine whether compression will improve transfer performance. If compression will be beneficial, then ACE compresses and transmits the block; otherwise, it transmits the block uncompressed.

ACE appends a 4-byte header to each block to indicate the block size (which is required by some decompression algorithms). This header also indicates the compression technique used, if any. To indicate that a block is not compressed, we use a negative value for the size. At the destination, ACE extracts the size and, if it is nonnegative, extracts the identifier of the compression technique used to compress the block. ACE then decompresses the block using this algorithm.

1. We use the term write to mean a socket write or send call.

3.1 Compression Algorithms

We considered three popular compression algorithms for use in ACE. They are LZO [18], Zlib [26], and Bzip [3]. In addition to their wide-spread use, we selected these algorithms due to the relative performance differences of each.

We evaluated the performance of compression techniques using three metrics: *Compression ratio* which is $\frac{compressed\_size}{uncompressed\_size}$, *compression rate* which is the number of bytes compressed per microsecond, and *decompression rate* which is the number of bytes decompressed per microsecond. LZO has a fast compression and decompression rate but a large compression ratio, i.e., it reduces the redundancy in a file to a lesser degree than either Bzip or Zlib. Bzip enables the best compression ratio at the cost of compression and decompression time. The performance characteristics of Zlib falls between that of LZO and Bzip.

We analyzed the performance metrics for each algorithm for a number of different file types which we describe in our evaluation section. The performance difference between the different algorithms remains relatively constant across each file. LZO performance is more variable than Zlib and Bzip. The files exhibit different block-level behavior (which was in some cases highly variable); however, the relative difference between algorithms is consistent.

3.2 Pipeline Model versus Sequential Model

Since ACE intercepts communication at the socket write level, it does not have complete knowledge of the transmission behavior of the program. ACE does not know the total size of the data, the frequency with which it will arrive, or its characteristics, e.g., whether it is a file or if it is data that is being generated dynamically. To estimate these details, ACE uses past socket behavior. We developed two different performance models to predict future socket behavior using this information.

The first performance model is the *sequential model*. Using this model, ACE assumes that successive calls to the socket write method are temporally disjoint such that compression cannot be overlapped with data transfer or decompression. ACE uses this model for applications that send data infrequently or at periods which preclude overlap. Using this model, ACE computes the transfer time of a *compressed* block, $T_{C(l,r)}$, between host l and host r as:

$$T_c(l,r) = \frac{32KB}{CR * NWS\_BW(l,r)} + C_l(CR) + D_r(CR),$$

where CR, $C_l(CR)$, and $D_r(CR)$ are the predicted values for compression ratio, compression time, and decompression time, respectively. $NWS\_BW(l,r)$ is the bandwidth between the local and remote hosts as predicted by the NWS.

ACE computes uncompressed transfer time as $T_u(l,r) = \frac{32KB}{NWS\_BW(l,r)}$ and compares this value with the 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.

Alternately, it may be possible to overlap compression with data transfer or decompression. We refer to the number of bytes the sender is sending per second as the *data generation rate*. When the data generation rate is high, ACE can overlap compression and transmission with decompression. For such cases, ACE employs a second performance model, called the *pipeline model*.

The pipeline model divides the transfer process into five components. The *data generation rate* is the number of bytes sent by the sender per second. The *compression rate* is the number of bytes compressed per second at the sender. The *effective transfer rate* is the number of bytes per second required for compressed transmission. The *decompression rate* is the number of bytes decompressed per second at the receiver. The *data consumption rate* is the number of bytes read per second by the application at the receiver.

The speed of the pipeline is determined by the slowest component in the pipeline. When this speed is greater than the available bandwidth between the sender and the receiver, ACE compresses the data; otherwise, ACE sends the data uncompressed.

ACE initially assumes infrequent or intermittent transmission and uses the sequential model. As the sender transmits data, ACE observes the frequency of transfer as well as transmission size. When ACE detects that the data generation speed exceeds the available bandwidth, ACE switches to pipeline model.

ACE computes the data consumption rate of the receiver for the pipeline model indirectly. For each TCP/IP socket connection, the operating systems at each end point will allocate buffers in their respective kernel spaces. When the receiver does not consume data fast enough, the buffer at the receiver will fill up and the send calls from the sender will block until there is space available for the send. This process will automatically slow the rate at which ACE sends data and, thus, reduce the data generation rate of the sender. This indirect method precludes the need to change the semantics of the read/write socket calls and allows ACE to adjust to the data consumption rate at the receiver without requiring ACE to measure and communicate this information (which would impose additional overhead).

Regardless of the performance model, ACE computes each component of the model for *every available compression algorithm*. ACE then selects the compression algorithm which will lead to the best performance (if any). Before sending each 32KB block of data, ACE invokes the appropriate performance model parameterized by forecasted values for available bandwidth and CPU (local and remote), as well as compression ratio and compression and decompression time for the next block. As mentioned previously, the NWS provides predictions to ACE of network bandwidth, latency, and CPU availability. In the next section, we describe how ACE predicts the remaining parameters.

4 ACE Prediction System

To forecast when to apply compression, ACE must predict compression ratio, compression time, and decompression time for each implemented compression algorithm. ACE uses these values within its sequential or pipeline model to estimate compressed and uncompressed transfer time.

4.1 Predicting Compression Ratio

To predict compression ratio for a block using a specific compression algorithm, ACE uses the compression ratio of the previous block. ACE can use this methodology as long

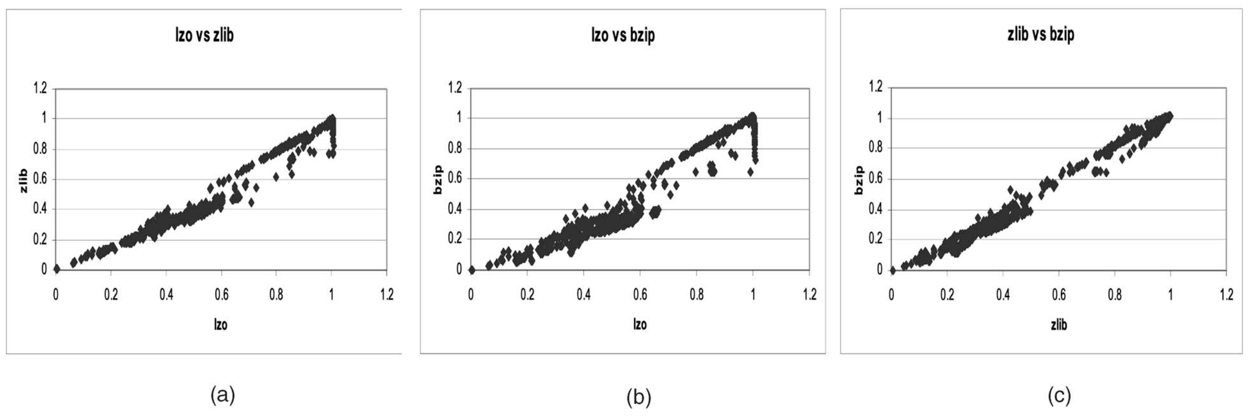


Fig. 1. Comparison of compression ratio of two algorithms. Graph (a) compares LZO and Zlib, graph (b) compares LZO and Bzip, and graph (c) compares Zlib and Bzip. Each point represents one block of data from a number of different files; we placed each point according to the compression ratio achieved by each respective algorithm. The data indicates that there is a linear relationship between the compression ratios of each techniques. ACE uses the regression line parameters from this data to predict the compression ratio from one technique using another.

as it continues to compress. However, there may come a point at which ACE decides not to compress. For example, if the previous block is not compressible or if the CPU loads at either end of the transmission are high enough to negate the benefit from the use of compression. In such cases, ACE does not have a recent compression ratio to use as an estimate.

One solution to this problem is to monitor changes in the CPU and network performance. When changes occur that make applying compression profitable, ACE can start using compression again. However, even when bandwidth and CPU loads remain constant, changes in the entropy of the data stream may make compression desirable again. For example, a program which sends a compressed JPEG image through a socket may later send a large number of text messages that are highly compressible. To capture both cases, ACE monitors changes in resource performance *and* periodically reintroduces compression to evaluate the compressibility of the data stream.

In the worst-case (in which the data stream is already compressed), the periodic introduction of compression by ACE will increase the total data transfer time rather than decreasing it. To limit degradation in transfer performance, ACE adapts the length of time it waits before reintroducing compression. Specifically, ACE identifies an initial period using forecasts of CPU load and network bandwidth. ACE multiplies this value by 10 (determined empirically) to compute the amount of time it should wait before trying to use compression again. At the end of each period, ACE applies compression and determines whether doing so improved transfer performance (by using the actual resource and compression performance of the block once it transfers). If compression does not improve performance, ACE increases the weight value by 1; ACE then repeats this process until compression pays off (if ever). If compression ever turns out to be a good decision ACE resets the weight to 10.

To predict compression ratio, we must consider one additional detail. ACE selects between different compression algorithms depending upon the performance of each. As such, for each block, ACE must predict the compression ratio of all algorithms. Since the algorithm used to compress the last block will be different from the other compression algorithms in the system, we need a way to predict the compression ratio of one technique using that of another.

Since compression algorithms enable different compression ratios, we cannot assume that the ratio achieved by one technique will be the same for all other techniques. We therefore empirically analyzed the relationship between the compression ratios achieved by the compression algorithms in our system: LZO, Zlib, and Bzip. Fig. 1 shows the relationship between the compression ratios for each pair of algorithms. The x-axis and y-axis show the compression ratio for each of two different algorithms. A point in the graph represents one block of data in each of the files we studied. We place each point according to the compression ratio achieved by each algorithm.

We refer to the set of files that we used to generate this data as our training set. We employ the training set to profile the performance characteristics of compression techniques offline. We use this profile information to parameterize the ACE prediction functions. We describe this set of files in our empirical evaluation section. They include files in a number of different, commonly used, formats. These files are different from those that we use as part of our performance evaluation of ACE.

In all three graphs, there is an approximately linear relationship between the compression ratio of the different algorithms despite their significant implementation differences. From these point data sets, we compute a regression line that we use within ACE to approximate the compression ratio of one algorithm given that of another. This technique enables ACE to predict compression ratio with very few operations. There is some error in performing such an approximation. However, our empirical evaluation shows that such error does not prevent ACE from achieving significant gains in transfer performance.

4.2 Predicting Compression and Decompression Time

To compute compression time, we profiled compression performance on each machine on which ACE executes. We implemented this process as part of ACE installation; it can be redone manually if ever the hardware changes. We must ensure that the machine is unloaded during profiling since ACE will combine this profiled rate with estimates of the

TABLE 1
Training and Benchmark Files

File Name	Contents	Size (KB)
Training Files		
aaa_long.txt	Letter 'a' repeated 400000 times	390.63
adobe.tar	Java Class files for Adobe Reader	11990.00
book1	Text file	750.75
chemical-ms-080-s.asf	Highly compressed WAW file	206.87
chemical-wav.wav	Uncompressed sound file	3546.68
E.coli	Complete genome (text) of E. Coli virus	4529.97
kennedy.xls	XSL text file	1005.61
obj2	Apple Macintosh object code	241.03
pic	Black/white fax picture	501.19
Total across files		23162.73
Benchmark Files		
HUH-Publ.xml	XML text file	9954.17
jedit.jar	Java Class files (uncompressed) for Jedit	2450.00
lion.mpg	MPEG-encoded file	4382.54
mysql.MYD	Binary database file	10837.75
partial.adl_catalog.txt	Text file	9143.77
Total across files		36768.23

We used the training set to collect profile information from the compression techniques that we considered; we generated regression lines from this data which ACE uses to make fast predictions of compression and decompression time for arbitrary transfers given a predicted compression ratio. We evaluate ACE using a different set of files (to reflect arbitrary transfers), which we call benchmark files. The table data shows the file types, the size of the files in kilobytes, and the total size of all of the files in each set.

resource performance that is available at the time of transmission (program execution and communication), to accurately predict whether compression will improve transfer performance.

We compressed each block 100 times and averaged the compression time for each across our training files. The results indicate that compression ratio and compression time as well as compression ratio and decompression time exhibit a near-linear relationship for each of the compression algorithms we investigated. Thus, we generate a linear regression for compression time and ratio and decompression time and ratio, similarly to our regression lines for predicting compression ratios across compression algorithms. ACE uses this function to efficiently approximate compression and decompression time from compression and decompression ratio at runtime.

Adding a new compression algorithm to ACE requires that these same experiments be run for the new algorithm, i.e., to compute the relationship between compression ratio and compression/decompression time as well as the relationship between compression ratio of the new algorithm and that of the existing algorithms. However, the number of comparisons that must be performed for each 32KB block increases linearly with the number of available algorithms within ACE. As such, algorithms should be chosen carefully. The overhead of finding the best compression algorithm may reduce the savings enabled by adaptive compression if the number of available algorithms is very large.

5 Evaluation

To evaluate the efficacy of ACE, we performed a number of experiments. We first describe our empirical methodology and then present our experimental results.

5.1 Experimental Methodology

For each experiment that we conducted, we used a simple server and a client program written in Java. The server acts as an FTP server and the client requests files from this server. For each file, the client measures the time required to receive and possibly decompress the file. The client requests the same file several times and computes the average of the measured timings.

We acquired the files that we studied from well-known compression corpora, including the Canterbury Corpus [5] and Calgary Corpus [4]. The Canterbury Corpus is a new corpus introduced to replace the old Calgary Corpus. The Calgary Corpus, despite its age, remains a well-respected corpus that is frequently used for the comparison of compression algorithms. Both corpora include both English text (bibliography, book, paper, etc.) and nontext sources (picture, object code, geophysical data, etc.). In addition to these files, we collected several other files from various Internet sources. ACE only considers applying compression to transmissions larger than 32KB. ACE, therefore, has no impact on the performance of transmissions smaller than 32KB. We thus omit corpus files that are smaller than 32KB in size.

We profiled the compression performance of the ACE compression algorithms using a subset of these data files. As described in Section 4.2, we compute regression lines from this data that ACE uses to make fast estimates of compression and decompression speed for a given compression ratio. To ensure that our experimental results are fair and realistic, we evaluate ACE using a completely different set of files called *benchmark* files. We list both of these subsets, their types, their contents, and their sizes in Table 1. We divided the files into training and benchmark files arbitrarily, but in such a way that different file formats were included in both. The total size of the training files is 23MB. The total size of the benchmark files is 37MB. All of

TABLE 2
Fast Network Results

Network	Suns	Gibson	Never	Zlib	Bzip	LZO	ACE	Pct. Degrd. Over Best	Pct. Imprv. Over Worst	Pct. Imprv. Over Worst w/o Bzip
Unloaded	Unloaded	Unloaded	9.970	**5.613**	50.493	5.980	6.990	25 %	86 %	30 %
Unloaded	Unloaded	Heavy	20.451	16.064	151.453	**9.007**	10.671	18 %	93 %	48 %
Medium	Unloaded	Medium	14.872	12.959	92.365	9.900	**9.869**	0 %	89 %	34 %
Medium	Unloaded	Heavy	19.737	24.294	193.577	**11.789**	15.054	28 %	92 %	38 %
Heavy	Unloaded	Unloaded	**26.727**	55.902	226.737	60.385	45.004	68 %	80 %	25 %
Heavy	Medium	Unloaded	30.646	**29.367**	277.160	63.812	30.270	3 %	89 %	53 %
Heavy	Medium	Medium	49.179	44.414	388.405	43.316	**28.740**	0 %	93 %	42 %
Heavy	Medium	Heavy	44.051	64.519	301.924	36.414	**34.898**	0 %	88 %	46 %
Heavy	Heavy	Medium	63.582	**34.314**	380.225	43.645	42.517	24 %	89 %	33 %

the results that we present in this section are for the *benchmark* file set.

We used two campus networks to evaluate our system; we refer to them as Fast and Slow. *Fast* exhibits an average bandwidth of 71Mb/s, consists of 100Mb Ethernet, and links two cross-campus computers at the University of California, Santa Barbara (UCSB). *Slow* is a cross-country link between UCSB an the University of Tennessee, Knoxville (UTK); it provides an average bandwidth of 1.7Mb/s.

We used three machines at the end points of these network links: `suns` (a Pentium 4 2.4GHz Xeon with 2GB RAM) and `heat` (a Pentium 4 2.4GHz Xeon with 512MB RAM) at UCSB, and `gibson` (a dual 500MHz Pentium 3 with 512MB RAM) at UTK. Suns and heat are connected via the fast network and suns and gibson are connected via the slow network. The server is executed on suns in both cases.

For each experiment, we considered unloaded and loaded resources (CPU and network) to evaluate the performance of ACE and other compression techniques given a wide range of resource performance conditions. The levels of CPU load that we considered include: *No Load* in which there are no other processes running, *Medium Load* in which there are four dummy processes running (client hosts) and eight running (if the host is the server), *Heavy Load* in which there are eight dummy processes (client hosts) and 20 (if the host is the server), and *Very Heavy Load* in which there are 180 dummy (server only). The dummy processes are programs that repeatedly execute floating point operations.

To observe the effect of network performance variation on ACE, we also introduced artificial network load for the fast network. We introduced three levels of network load. Each traffic generator pair periodically exchange 4KB of data repeatedly. For the unloaded level, there is no traffic generator running, for medium load there are 12 traffic generators and for heavy traffic load there are 20 traffic generators running. On average, we experience an average bandwidth of 20Mb/s with medium load and 3Mb/s with heavy load.

5.2 Results

We empirically compared the performance of ACE to four compression scenarios: *Never*, *Zlib*, *Bzip*, and *LZO*. With *Never*, ACE transfers data without compression. For Zlib, ACE compresses and transfers data in 32KB blocks and then decompresses the compressed data at the client using Zlib. Similarly for Bzip and LZO, ACE uses only the respective algorithms. The data we present for ACE includes all of the overheads introduced by ACE, including NWS access and forecasting.

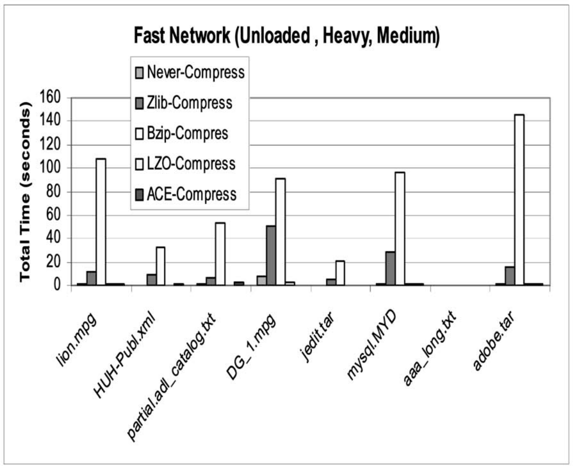
We first present a set of results for different underlying resource performance conditions (different load levels) for each network type. These load levels do not change dynamically during the experiments.

5.2.1 Fast Network Results

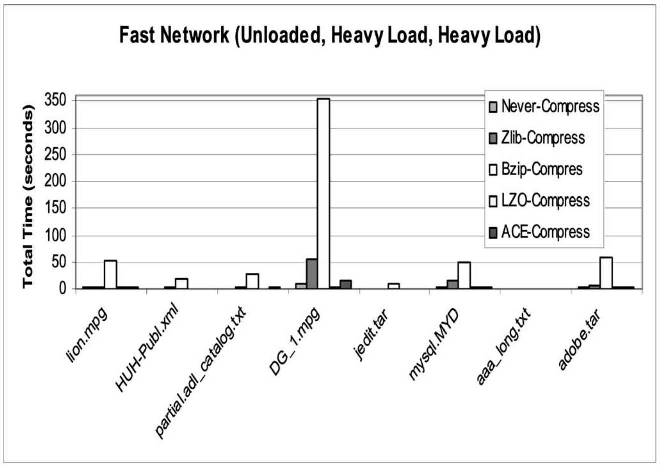
The results for the Fast network are shown in Table 2 as total transfer time in seconds. The first column in each table is the network load, the second column is the server CPU load (suns), and the third column is the client CPU load (heat). Column 4 shows the results when compression is never used (Never), columns 5, 6, and 7 show the cases in which Zlib, Bzip, and LZO, respectively, is always used. Column 8 shows the performance results of ACE. We show only a subset of all possible experiment combinations for brevity. The results are representative of the omitted data however. Each compression technique, including no compression, enables the best performance under different conditions.

The bold values distinguish the best-performing case. For the fast network, the most common, best-performing algorithm is LZO. We expect this since the network performance is fast enough (even under heavy load) not to warrant a high degree of compression. In a few cases, ACE is the best-performing configuration. This occurs since ACE is able to use a combination of compression techniques for a single stream leading to a better combination of compression ratio, compression time, and decompression time.

In every case, ACE enables performance that is similar to that of the best-performing algorithm. The differences between ACE and the best-performing algorithm for these results as well for the slow network results in the next section are due to the three types of overhead imposed by ACE. First, ACE applies compression initially to determine whether the stream is compressible. If it is not, or if compression will not improve performance, this test is pure overhead. Moreover, this overhead can be significant when ACE attempts to apply Bzip—due to the large compression time cost that Bzip imposes.



(a)



(b)

Fig. 2. A detailed look at example Fast network results. The x-axis shows the individual files used in the experiment. The y-axis is time in seconds.

The other two sources of overhead are due to prediction error and computation overhead. Both the NWS and the internal ACE model estimations introduce some error. A large enough error can cause ACE to make the wrong decision. A wrong decision causes ACE to either apply compression when it should not or to avoid compression when it is necessary. The most common reason for misprediction by ACE is at the start of file transfer. ACE requires some history to make accurate decisions. Until this information is available, ACE is unable to make educated estimations. The final source of overhead is due to the computation required for ACE to estimate the compression performance for each of the individual compression algorithms (as well as no compression).

Of these three overheads, when the best performing technique is no-compression, the first source of overhead (introduction of compression) is the primary cause for the difference between ACE and no-compression. When the best-performing technique is another compression technique, the primary ACE overhead is the introduction of compression.

In the final three columns, we present the percent degradation over the best-performing case (column 9), the percent improvement over the worst-performing case (column 10), and the percent improvement over the worst-performing case when Bzip is excluded (column 11), due to ACE. We include the latter case (without Bzip) since Bzip is never the best-performing case due to its high compression and decompression costs. The column 9 data shows the overhead that ACE imposes over the ideal case: A user transmits files, performing compression on-the-fly, and uses an oracle to identify the best-performing algorithm. The average degradation over the best-performing algorithm due to ACE across all experiments (including those omitted from the table) is 89 percent due to the short transfer times. This percentage equates to 14.2 seconds.

The percent improvement over the worst-case shows the benefit that is available due to ACE making the compression decisions for a user that, in every case, selects the poorest-performing compression technique. These results are biased to some degree since Bzip is never selected for this network; however, it exemplifies the point that ACE can save a user that arbitrarily applies a random compression technique without insight into its efficacy. Due to the bias from including Bzip in these results (percent improvement over the worst-case), we also show the results when we omit Bzip from this comparison (column 11 in the table). On average, ACE improves performance by 90 percent over Bzip and by 52 percent over the best-performing case when Bzip is not included.

We next present a set of individual results that provide more detail about the performance of ACE. Fig. 2 shows two graphs for two different load configurations: Unloaded network, Heavy server, Medium client (Fig. 2a) and Unloaded network, Heavy server, Heavy client (Fig. 2b). For each file in graph (Fig. 2a), ACE performs similarly to the best case, LZO. For some of the files in graph (Fig. 2b), ACE is unable to achieve the performance of the best case. This is because ACE periodically tries more computationally costly algorithms, such as Zlib or Bzip; this causes ACE to spend more time on compression unnecessarily. However, the overhead introduced by ACE is small for each file.

5.2.2 *Slow Network Results*

Table 3 shows the transfer times (in seconds) using the Slow network for a representative subset of our experimental results. The bold values identify the best-performing case which changes for different load configurations. We again provide the percent degradation over the best case (column 9), percent improvement over the worst case (column 10), and the percent improvement over the worst-performing case when Bzip is excluded (column 11), due to ACE. Overall, ACE again performs similarly to the best-performing case for different underlying resource load conditions.

We further break down the results into individual configurations in Fig. 3. For all results for the Slow network, the network is unloaded. Fig. 3a shows the experimental results for the Slow network and an unloaded server and client load configuration. Fig. 3b shows the individual

TABLE 3
Slow Network Results

Network	Suns	Gibson	Never	Zlib	Bzip	LZO	ACE	Pct. Degrd. Over Best	Pct. Imprv. Over Worst	Pct. Imprv. Over Worst w/o Bzip
Unloaded	Unloaded	Unloaded	87.321	**45.026**	70.971	51.373	49.999	11 %	43 %	43 %
Unloaded	Medium	Unloaded	90.984	**45.617**	265.922	51.332	50.155	10 %	81 %	45 %
Unloaded	Heavy	Unloaded	94.029	**42.710**	324.258	55.796	51.232	20 %	84 %	46 %
Unloaded	VeryHeavy	Unloaded	**89.321**	598.879	1201.230	119.373	175.373	96 %	85 %	71 %

results for the very-heavy-loaded-server and unloaded-client configuration. ACE enables transfer performance that is similar to the best compression algorithm for all files. For the Slow network, attempting to use Bzip does not have any significant effect on the overall performance.

5.2.3 Adaptivity to Changing Conditions

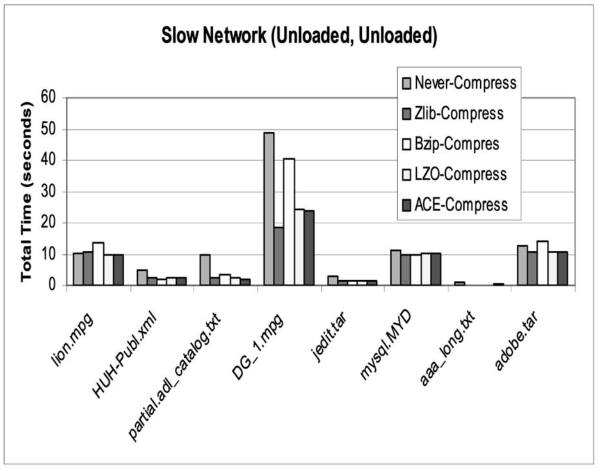
In the prior sections, we showed how ACE compares to individual compression techniques for different load scenarios. The performance of the underlying resources in these experiments does not change over time since we employed artificial load and performed our experiments at night. Our results show how effectively ACE enables transfer performance similar to that of the best-performing technique.

A more realistic setting is one in which the load changes over time. It is in such a setting that ACE has the advantage over any single technique. For example, if a user transfers files back and forth between her laptop and her file server using different networking technologies, e.g., the wireless network at the conference, the 100Mb/s Ethernet in her office, the cable modem at her house, ACE can significantly improve her overall transfer performance. Moreover, as the performance varies for a single network connection, ACE can adapt to these changes to outperform any single compression technique potentially.

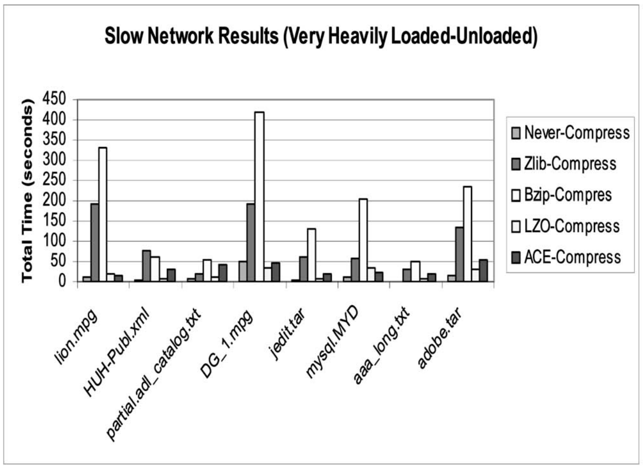
To empirically evaluate the use of ACE in such settings, we constructed three scenarios in which performance changed over time. In *Scenario1*, the user uses both the Fast network and the Slow network. This scenario is similar to that experienced by a student using her laptop on the University network in a research lab from early morning (no contention) to midday (increased contention and shared CPU use) and then moving to a wireless connection at a local coffee shop in the afternoon (light contention) to evening (heavy contention). Using the format Network:NetworkLoad-ServerCPULoad-ClientCPULoad, we specify this scenario as Fast:Unloaded-Unloaded-Unloaded → Fast:Heavy-Medium-Medium → Slow:Unloaded-Unloaded-Unloaded → Slow:Unloaded-Very Heavy-Heavy.

The other two scenarios exemplify other performance transitions using the same network. *Scenario2* uses only the Fast network. The transition for this scenario is from heavy to light to heavy (as in a student lab before, during, and after lunch time): Fast:Unloaded-Heavy-Heavy → Fast:Medium-Unloaded-Unloaded → Fast:Heavy-Heavy-Medium. *Scenario3* uses only the Slow network. The transition for this scenario is: Slow:Unloaded-Unloaded-Heavy → Slow:Unloaded-Medium-Heavy → Slow:Unloaded-VeryHeavy-Heavy. This scenario represents behavior similar to that of a popular Web server, the network load to which increases from afternoon to evening (as an increasing number of users finish their workday).

For each scenario, we transfered all of the benchmark files during each load level. We then computed the percent improvement in transfer performance enabled by ACE over using any single compression algorithm. Fig. 4 shows the results for each scenario. The x-axis shows each of the



(a)



(b)

Fig. 3. A detailed look at example Slow network results. The x-axis shows the individual files used in the experiment. The y-axis is time in seconds.

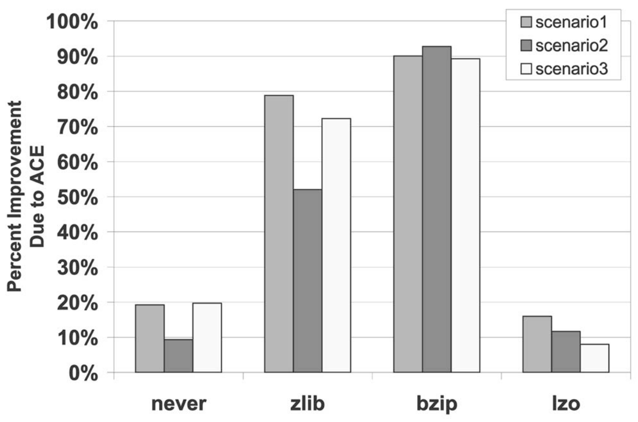


Fig. 4. The performance of ACE under various performance scenarios. The bars show the percent improvement in transfer performance (y-axis) due to ACE over each of the individual compression techniques alone (x-axis). The transition for scenario1 is Fast:Unloaded-Unloaded-Unloaded → Fast:Heavy-Medium-Medium → Slow:Unloaded-Unloaded-Unloaded → Slow:Unloaded-Very Heavy-Heavy. The transition for scenario2 is Fast:Unloaded-Heavy-Heavy → Fast:Medium-Unloaded-Unloaded → Fast:Heavy-Heavy-Medium. The transition for scenario3 is Slow:Unloaded-Unloaded-Heavy → Slow:Unloaded-Medium-Heavy → Slow:Unloaded-VeryHeavy-Heavy.

compression techniques to which we compare ACE. The bars for each compression technique represent the percent improvement due to ACE over using the technique alone across the scenario. The results indicate that ACE significantly outperforms Bzip (by 89-93 percent on average), which is a very popular and widely used compression utility. ACE outperforms always using Zlib by 52-79 percent on average and LZO by 8-16 percent. ACE improves performance over never using compression by 9-20 percent, on average.

6 Related Work

Our work focuses on increasing the transfer speed using compression. Related work includes systems that apply compression on-the-fly.

We developed one such system in prior work, called DCFS: The Dynamic Compression Format Selection System [17]. DCFS uses NWS forecasts of dynamic network performance to guide its selection of a precompressed Java bytecode program file. Each program file is compressed with a number of different compression algorithms. All versions are stored on disk. DCFS predicts which file should transmitted (and decompressed) on-the-fly. ACE is significantly more general than DCFS in that it can be used for any type of file (not just Java bytecode) and performs on-the-fly compression. With DCFS, we focused on domain-specific compression algorithms, such as PACK [20] and JAR [14], as well as gzip [12] to increase the performance of Java programs that are executed remotely. Another significant difference between DCFS and ACE is that DCFS does not consider remote CPU availability.

We extended DCFS for on-the-fly compression in an initial version of ACE described in [22]. In this system, we considered only a single compression technique, Zlib, and did not detail the ACE implementation. The prior system determined adaptively only whether to apply Zlib compression or not. Herein, we extend this system to enable ACE selection across multiple compression techniques. In addition, with this work, we perform a much more extensive empirical evaluation of the ACE system.

In work other than our own, the authors of [19], present a dynamic compression system for text files called *Network conscious text compression system* (*NCTCSys*). NCTCSys considers only network performance and server load. This prior work is restricted to text files and primarily intended for Web servers. Our system, in contrast, dynamically adapts to all types of files as well as to the entropy within a single file. In addition, we consider CPU load which our results indicate, is vital for improved transfer performance in an Internet setting. Moreover, this prior work only considers the number of clients the server processes and uses heuristics to make compression decisions, rather than using strict mathematical models. NCTCSys uses its own modules to detect and measure the network bandwidth. It does not make performance predictions. NCTCSys selects the compression technique based on various factors, e.g., client line speed, the number of clients, and server load.

In [13], the author uses Remos [8], a network performance prediction system much like the NWS, to guide compression decisions. Remos provides statistical information about its data to applications, e.g., confidence level, error rate, etc. Unlike the NWS though, Remos makes its measurements at the network level. It uses the low-level SNMP protocol to query routers and switches to estimate the performance characteristics of the underlying network. NWS takes the end-to-end approach and it is TCP/IP-based. As a result, the NWS reports more accurately the performance that TCP/IP-based applications experience.

This prior work also differs from ACE in that it does not consider remote CPU load in the computation of decompression time. In addition, both compression rate and compression ratio are computed based on the type of the data which must be known prior to transfer. ACE requires no such information a priori. In addition, many file types are too general to make an accurate estimate. For example, binary files exhibit a wide range of compression characteristics which cannot be estimated using a limited number of samples. We overcome this limitation in our system by sampling the compressibility of blocks online and by using regression lines to predict compression time for a wide range of file formats.

In addition, this prior work considers only the impact of compression and transmission. Our work shows that decompression is an important part of the overall transmission performance. This prior work considers the possible overlap between compression and transmission of data as we do in our system. The system predicts compression time and compression ratio based on the type of file being transmitted. The study claims that the standard deviation within each type of file category for compression ratio is very small; thus, it is reasonable to use the type of the file as a guide for a prediction of its compression ratio. However, we believe that this claim is not true, since our empirical evaluation of different file types shows that files can exhibit large variations in compressibility.

A form of adaptive compression that is similar to the system we describe herein is the system described in [16] and is extended in [15]. The goal of this prior work is to vary the compression level (of a single compression technique) on-the-fly so that the network is never under-utilized. This 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. In addition, using a higher level usually results in longer compression times. Our system also differs in that it uses forecasts of future resource performance which effectively prevents oscillations reported in this prior work. Finally, this prior work does not use the remote CPU availability information to make compression decisions, thus it is subject to higher error than that produced by our system.

7 Conclusions

We present ACE, an adaptive compression environment that improves Internet transfer performance by dynamically selecting between competitive compression algorithms and applying on-the-fly compression transparently. ACE makes its decisions by predicting and comparing transfer performance for both uncompressed and compressed transfer. ACE is able to adapt to the changes in resource performance and network technology (as occurs for mobile devices), as well as to the compressibility of the data.

We evaluated ACE using a wide range of performance scenarios, i.e., network loads, CPU loads at each end point, and changes in the underlying network technology. We found that when the communication performance is steady, ACE performs similarly to the best-performing compression technique that it implements. Perhaps more importantly though, ACE does considerably better than the worst-performing (yet popular) compression algorithm. On average, we found that ACE improves performance over the worst case by 50-90 percent for all files that we studied. The benefits from using ACE, in addition to the provision of protection from selecting the "wrong" compression technique, become apparent when the underlying communication performance varies or the network technology changes (as for a mobile laptop). Our results indicates that ACE can improve transfer performance by 8-93 percent over commonly used compression algorithms in such cases.

Acknowledgments

The authors thank the reviewers for their insightful suggestions that helped them to significantly improve the final version of this paper. This work was funded in part by US National Science Foundation grants No. NGS-0204019 and No. EHS-0209195, and Intel Corporation.

References

[1] A. Adl-Tabatabai, M. Cierniak, G. Lueh, V. Parikh, and J. Stichnoth, "Fast, Effective Code Generation in a Just-in-Time Java Compiler," *Proc. ACM SIGPLAN '98 Conf. Programming Language Design and Implementation,* Oct. 2000.
[2] F. Berman, G. Fox, and T. Hey, *Grid Computing: Making the Global Infrastructure a Reality.* Wiley and Sons, 2003.
[3] BZIP Compression, http://sources.redhat com/bzip2/, 2005.
[4] Calgary corpus, http://links.uwaterloo.ca/calgary.corpus.html+, 2005.
[5] Canterbury corpus, http://corpus.canterbury.ac.nz/+, 2005.
[6] M. Cierniak, G. Lueh, and J. Stichnoth, "Practicing JUDO: Java under Dynamic Optimizations," *Proc. ACM SIGPLAN 2000 Conf. Programming Language Design and Implementation,* Oct. 2000.
[7] Microsoft Corp., Microsoft.Net, http://www.microsoft.com/net/+, 2005.
[8] A. DeWitt, T. Gross, B. Lowekamp, N. Miller, P. Steenkiste, J. Subhlok, and D. Sutherland, "ReMoS: A Resource Monitoring System for Network-Aware Applications," Technical Report CMU-CS-97-194, Dept. of Computer Science, Carnegie-Mellon Univ., Dec. 1998.
[9] I. Foster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers, Inc., 1998.
[10] Gnutella, http://www.gnutella.com/+, 2005.
[11] Grid Portal Collaboration, http://www.globus.org/retreat00/presentations/ngridproxynovotny/, 2005.
[12] Gzip homepage, http://www.gzip.org/+, 2005.
[13] N. Hu, "Network Aware Data Transmission with Compression," Technical Report CMU-CS-01-164, Dept. of Computer Science, Carnegie-Mellon Univ., 2001.
[14] Sun Microsystems Inc., The Java ARchive Utility, http://java.sun.com/products/jdk/1.1/docs/tooldocs/solaris/jar.html, 2005.
[15] E. Jeannot, B. Knutsson, and M. Björkman, "Adaptive Online Data Compression," *Proc. IEEE Int'l Symp. High Performance Distributed Computing '02,* July 2002.
[16] B. Knutsson and M. Bjorkman, "Adaptive End-To-End Compression for Variable-Bandwidth Communication," *Computer Networks,* vol. 31, no. 7, pp. 767-779, Apr. 1999.
[17] C. Krintz and B. Calder, "Reducing Transfer Delay with Dyanamic Selection of Wire-Transfer Formats," *Proc. IEEE Int'l Symp. High Performance Distributed Computing (HPDC),* Aug. 2001.
[18] Lempel-Ziv-Oberhumer (LZO) Compression, 2005, http://www.oberhumer.com/opensource/lzop/.
[19] N. Motgi and A. Mukherjee, "Network Conscious Text Compression System (NCTCSys)," *Proc. Int'l Conf. Information Technology: Coding and Computing,* Apr. 2001.
[20] W. Pugh, "Compressing Java Class Files," *Proc. SIGPLAN '99 Conf. Programming Language Design and Implementation,* May 1999.
[21] P. Sevcik, "Internet Bandwidth: It's Time for Accountability," *Business Comm. Rev.,* vol. 31, no. 1, pp. 1-3, Jan. 2001.
[22] S. Sucu and C. Krintz, "ACE: A Resource-Aware Adaptive Compression Environment," *Proc. Int'l Conf. Information Technology: Coding and Computing (ITCC '03),* Apr. 2003.
[23] R. Wolski, "Dynamically Forecasting Network Performance Using the Network Weather Service," *Cluster Computing,* 1998.
[24] R. Wolski, "Experiences with Predicting Resource Performance On-Line in Computational Grid Settings," *ACM SIGMETRICS Performance Evaluation Rev.,* vol. 30, no. 4, pp. 41-49, Mar. 2003.
[25] R. Wolski, N. Spring, and J. Hayes, "The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing," *Future Generation Computer Systems,* 1999.
[26] ZLib compression library, http://www.gzip.org/zlib/+, 2005.

Chandra Krintz is an assistant professor at the University of California, Santa Barbara (UCSB). She joined the UCSB faculty in 2001 after receiving the MS and PhD degrees in computer science from the University of California, San Diego (UCSD) under the advisement of Dr. Brad Calder. Chandra's research interests include automatic and adaptive compiler, virtual runtime, and operating system techniques that improve performance (for high-end systems) and increase battery life (for mobile, resource-constrained devices). In particular, her work focuses on exploiting repeating patterns in the time-varying behavior of underlying resources, applications, and workloads to guide dynamic optimization and specialization of program and system components.

Sezgin Sucu is a PhD student in the Computer Science Department at the University of California, Santa Barbara (UCSB). Sezgin received his the MS degree from UCSB in June of 2003, under the advisement of Dr. Chandra Krintz. His MS thesis is entitled "Resource-Aware Lossless Compression." He is currently studying under the advisement of Dr. Tao Yang in the area cluster-based network services.