In Proceedings of the International Symposium on High Performance Distributed Computing (HPDC), August 2001.

Reducing Delay With Dynamic Selection of Compression Formats

Chandra Krintz Brad Calder

Department of Computer Science and Engineering
University of California, San Diego
{ckrintz,calder}@cs.ucsd.edu

Abstract

Internet computing is facilitated by the remote execution methodology in which programs transfer to a destination for execution. Since transfer time can substantially degrade performance of remotely executed (mobile) programs, file compression is used to reduce the amount that transfers. Compression techniques however, must trade off compression ratio for decompression time due to the algorithmic complexity of the former since the latter is performed at runtime in this environment.

With this work, we define ***Total Delay*** *as the time for both transfer and decompression of a compressed file. To minimize total delay, a mobile program should be compressed in a format that minimizes total delay. Since both the transfer and decompression time are dependent upon the current, underlying resource performance, selection of the "best" format varies and no one compression format minimizes total delay for all resource performance characteristics. We present a system called* ***Dynamic Compression Format Selection*** *(DCFS) for automatic and dynamic selection of competitive, compression formats based on predicted values of future resource performance. Our results show that DCFS reduces 52% of total delay imposed by compressed transfer of Java archives (jar files) on average, for the networks, compression techniques, and benchmarks studied.*

1. Introduction

Compression is used to reduce network transfer time of files by decreasing the number of bytes transferred through compact file encoding. However, compression techniques must trade off space for time. That is, the compression ratio achieved is dependent upon the complexity of the encoding algorithm. Since similar complexity is required to decompress an encoded file, techniques with a high compression ratio are time consuming to decompress. Alternately, techniques with fast decompression rates are unable to achieve aggressive compression ratios (resulting in longer transfer times).

Compression is commonly used to improve the performance of applications that transfer over the Internet for remote execution, i.e., *mobile* programs. The overhead imposed by compression-based remote execution includes the time for mobile code requests (program invocation and dynamic loading) and transfer. In addition, decompression time must also be included in this metric, since it occurs on-line while the program is executing. We refer to the combined overhead due to program request, transfer, and decompression as **Total Delay**.

Due to the inherent space-time trade-off made by compression algorithms, total delay is minimized only when selection of a compression technique is dependent upon underlying resource performance (network, CPU, etc). Moreover, since such performance is highly variable [1, 22], selection of the "best" compression algorithm should be able to change dynamically for additional program files transferred on the *same* link as the program executes. Such adaptive ability is important since the selection of a non-optimal format may result in substantial total delay (8-10 seconds in the programs studied) at startup or intermittently throughout mobile program execution. Much prior research has shown that even a few seconds of interruption substantially effects the user's perception of program performance [2].

To address this selection problem, we introduce **Dynamic Compression Format Selection** (DCFS), a methodology for automatic and dynamic selection of competitive compression formats. Using DCFS, mobile programs are stored at the server in multiple compression formats. DCFS is used to predict the compression format that will result in the least delay given the bandwidth predicted to be available when transfer occurs. We use the Java execution environment for our DCFS implementation since it is the most common language used for remote execution.

One key contribution this work makes is the design and implementation of a system for the exploitation of the benefits available from existing compression techniques without the associated costs. Empirically, DCFS reduces 36% (5.5

seconds) of total delay of compressed transfer on average, for the range of network characteristics and compression techniques studied. DCFS achieves 52% (10 seconds) reduction in total delay on average over the commonly used Java archive (jar) format.

Another contribution of this work is the implementation of two optimizations that extend DCFS to further reduce total delay: *Selective Compression* and *Compression-On-Demand*. The former is an optimization in which only the Java class files used during execution are included in the compressed archive. In the second approach, we extend DCFS to use compression-on-demand in which an application is compressed when the program is requested by the client at runtime. Our results show that on average selective compression reduces delay an additional 7% to 11% over DCFS alone. We show that DCFS using compression-on-demand can reduce 20% to 52% of the delay imposed by pre-compressed jar files (for which on-the-fly compression is *not* used).

In the following two sections we describe the design and implementation of DCFS. In Section 3.1 we detail our experimental methodology and empirically evaluate the performance of DCFS given cross-country Internet performance trace data. We then discuss the utility of DCFS in a non-simulated, Internet environment. In the remainder of the paper, we present the two DCFS extensions and conclude.

2. Compression Techniques

Code compression can significantly reduce the transfer delay by decreasing the number of bits that transfer [3, 21, 6, 5, 13]. In this paper, we consider three commonly used compression formats for mobile Java programs: JAR, PACK, TGZ [8, 17, 18]. We evaluated others as part of this study but include only these three for brevity. The first is the Java archive (jar) format (referred to from here forward as JAR). JAR is the most common tool for collecting (archiving) and compressing Java application files and is based on the standardized PKWare zip format [15].

PACK [17] is a tool from the University of Maryland for the compression of JAR files. This utility substantially reduces redundancy by exploiting the Java class file representation and by sharing information between class files. The compression ratios achieved by this (Java-specific) tool are far greater than any other similar utility. However, PACK has very high decompression times since the class files must be reconstituted from this complex format.

Gzip is a standard compression utility commonly used on UNIX operating system platforms. Gzip does not consider domain specific information and uses a simple, bitwise algorithm to compress files. As such, applications in this format can be decompressed very quickly but do not achieve the same compression ratios as more complex compression algorithms. The TGZ format refers to files that are first combined (archived) using the UNIX tape archive (tar), then compressed with gzip. Tar combines a collection of class files, uncompressed and separated by headers, into a single file in a standardized format.

Characteristics of each format are shown in Table 1 for the benchmarks used in the empirical evaluation of the techniques presented in this paper. The benchmarks are from the SpecJvm95 [20] benchmarks and other Java studies [16, 11]. The first two columns are the static class count and the total size (in kilobytes) of each application, respectively. For each compression format (PACK, JAR, and TGZ), we present three columns of data: the compressed size (in kilobytes) and the compression and decompression time (in seconds). The average across all benchmarks is shown in the bottom row in the table.

The inherent trade-off made by compression techniques (compression ratio for decompression time) is exhibited by this data. For example, the PACK format requires over 2.3 seconds to decompress the applications on average (JAR and GZIP require 89% and 98% less time, respectively), yet it enables a compressed file size that is 81% and 74% smaller than JAR and GZIP archives, respectively. That is, for slow networks PACK should be used due to its compression ratio, and for fast links TGZ should be used since it is inexpensive to decompress. No single utility enables the least total delay (request, transfer, decompression time) for all network performance characteristics and applications. The choice of compression format should therefore be made dynamically, depending upon such circumstances to enable the best performance of mobile programs. To do this, we introduce **Dynamic Compression Format Selection** (DCFS), a technique that automatically and dynamically selects the format that results in the least total delay.

3. Dynamic Selection

Figure 1 exemplifies our DCFS model. The client-side Java Virtual Machine (JVM) incorporates a DCFS class loader. When an executing program accesses a class file for the first time (or the program itself is invoked), the request made to the JVM is forwarded to the DCFS class loader. Concurrently, a network performance measurement and prediction tool (resource monitor in the figure) monitors the network connection between the client and the server at which the application is stored. We examine two network prediction systems and discuss the implications of their use in section 4.1.

The DCFS class loader at the client acquires the network performance value from the network monitor and forwards the value(s) with the request to the server. A list of the available decompression utilities and their average decompres-

Compression Format & Decompression Characteristics Sizes are in kilobytes and times are in seconds												
			PACK			JAR			TGZ			
	Class		Compression		Decomp.	Compression		Decomp.	Compression		Decomp.	
Program	Count	Size	Size	Time	Time	Size	Time	Time	Size	Time	Time	
Antlr	118	418	58	16.53	3.66	222	2.19	0.30	172.30	0.31	0.03	
Bit	53	152	18	6.40	1.25	85	1.07	0.14	57.00	1.07	0.03	
Jasmine	207	404	34	8.68	2.69	219	2.93	0.32	127.70	2.93	0.03	
Javac	76	548	49	14.99	3.32	276	2.93	0.34	179.20	2.93	0.03	
Jess	151	387	23	4.32	1.83	185	2.37	0.34	164.80	2.37	0.03	
Jlex	20	85	14	5.95	1.04	48	0.56	0.20	37.80	0.56	0.04	
Average	104	332	33	9.48	2.30	172	2.01	0.27	123.13	1.70	0.03	

Table 1. Compression characteristics of the benchmarks using PACK, JAR, and TGZ. The first two columns of data show the class count and the uncompressed size of each application. Data in the following columns are divided into three sections for each of the different compression formats, PACK, JAR, and TGZ. For each format, we show the compressed size of the application in kilobytes and the compression and decompression time in seconds. These final three sections exhibit the space-time trade-off made by the compression techniques. JAR and GZIP are 89% and 98% faster to decompress, respectively, on average than PACK. However, PACK is able to achieve an average compressed size that is 81% and 74% smaller on average than JAR and GZIP, respectively.

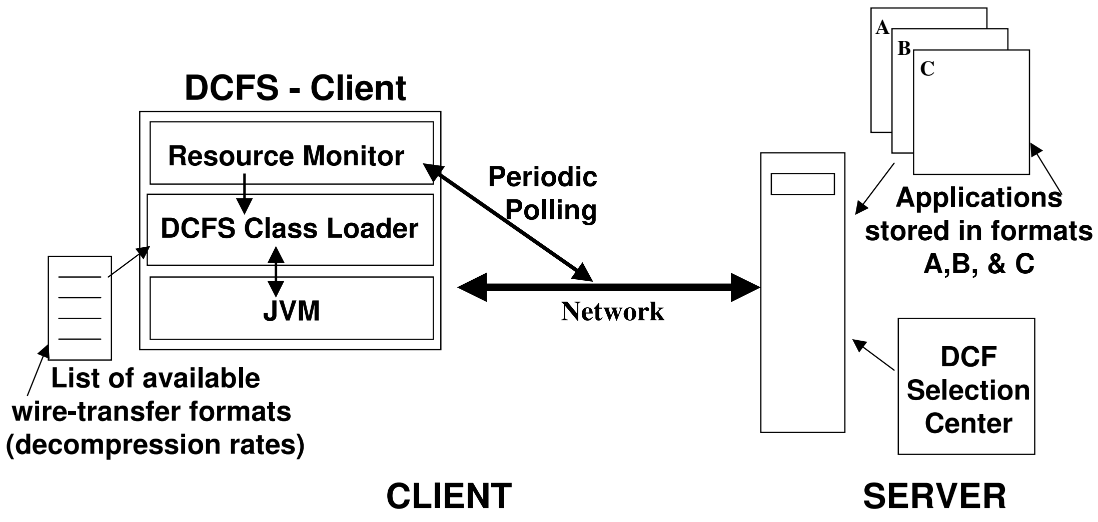

Figure 1. The Dynamic Compression Format Selection (DCFS) Model. At the client, an application or class file request is forwarded from the JVM to the DCFS class loader. Concurrently, a resource monitor queries the network between the client and the server to determine the network performance (and in our implementation predict the performance that will be available when the transfer is performed). The DCFS class loader requests the file from the server with which it includes the network performance measure and list of available decompression rates. The server using the sizes of the application in various compressed formats, the predicted resource performance, and decompression rates, computes the format that will result in the least total delay and transmits the application or file in that format back to the client.

Name:	From:	To:	BW	RTT
MODEM	Residence (East)	UTK (East)	0.03	95.0
ISDN	Residence (East)	UTK (East)	0.11	48.0
INET	UCSD (West)	UTK (East) (Internet)	0.53	79.0
VBNS	UCSD (West)	UTK (East) (vBNS)	0.71	93.0
LAN	UCSD (West)	LAN (10Mb/s Ethernet)	4.01	8.0

Table 2. Description of the networks used in this study. Each network is represented by the average bandwidth value (BW) in megabits per second and round-trip time (RTT) in milliseconds shown in the final two columns. These values were obtained from actual trace data collected using the connection.

sion rates are also included in this request by the client to the server. It may also be beneficial to provide non-network resources such as CPU and memory so that more realistic decompression times can be computed. The current implementation of the DCFS uses network performance prediction only and estimates decompression time using average decompression rates supplied by the client.

At the server, applications are stored in multiple compression formats. When a server receives a request for an application or file, it uses the information sent (predicted resource performance value(s) and available compression formats) to calculate the potential total delay for each format. The server selects the format that results in the least total delay for the file(s) requested.

3.1. Experimental Methodology and Results

To evaluate DCFS, we implemented both of the DCFS modules (DCFS client and server). However, to enable repeatability of results, we execute the modules on the same machine and simulate different networks between them. In this section, we examine the potential of DCFS assuming that the bandwidth and round-trip times are constant during transfer. We collect empirical measurements for a variety of bandwidths and round-trip times taken from actual network traces. This enables the presentation of the upper-bound potential of DCFS. We provide a discussion of the impact of incorporating real-time performance prediction in the next section.

Table 2 shows the bandwidth and round-trip time values used in this study and the corresponding networks. The networks include a 28.8 baud modem (MODEM) and an integrated services digital network link (ISDN) between a residence and university, a cross-country common-carrier Internet connection (INET), a cross-country vBNS connection (VBNS), and a 10Mb/s local area Ethernet connection (LAN). ISDN is a system of phone connections that allows data to be transmitted simultaneously using end-to-end digital connectivity. The vBNS is an experimental transcontinental ATM-OC3 network sponsored by NSF that can be used for large-scale and wide-area network studies. The table includes the average bandwidth and round-trip time measurements in the final two columns from a 24-hour JavaNws [12] trace data for each network.

To compute total delay using this experimental execution environment, the server computes the sum of the average round-trip time (for the request), the transfer time (the average bandwidth value multiplied by the size of the compressed application), and the decompression time for a given compression technique. Actual decompression times for the three compression techniques for the application being executed are supplied by the client as part of the initial request. Real decompression times are used as opposed to using decompression rates (multiplied by the compressed application size) since this is an upper-bound computation of total delay. The total delay is computed by the server for each of the compression formats, TGZ, JAR, and PACK, and the minimum is selected.

To illustrate the performance potential of dynamic format selection, we present the total delay (in seconds) in Figure 2 for the six benchmark applications. Total delay again consists of the number of seconds required to request, transfer and decompress the application. A logarithmic scale is shown (for the y-axis) since the substantial total delay for the slow networks obscures fast-link results when a normal scale is used. The first three bars in each graph show the total delay for PACK, JAR, and TGZ compression, respectively, without DCFS. The fourth (far right, striped) bar of each set shows the DCFS total delay. The DCFS bar is always equivalent to the minimum of the prior three bars since it is the "best" performing format. Averaged across all networks, DCFS reduces total delay from 0.3 to 1.6 seconds over PACK, from 2.1 to 16.1 seconds over JAR, and from 1.4 to 9.7 seconds over TGZ. The overall benefit from DCFS does not simply result from using a different compression utility, it results from selecting the **best** compression utility given the underlying network performance. The benefits achieved using DCFS are quite substantial for every benchmark and network performance rate.

The average percent reduction in total delay (across all benchmarks) is shown in Figure 3. The percent reduction is defined as $(TD\_Base - TD\_DCFS)/TD\_Base$ for each network, where TD_Base is the total delay for the base case and TD_DCFS is the total delay using DCFS. The bars in this figure represent each base case (a compression format of PACK, JAR, or TGZ without DCFS) for each network bandwidth. Each bar indicates the performance improvement as a percentage a user would experience if DCFS were used instead of the base case. Notice that the percent reduction can be zero when DCFS selects the base case and the

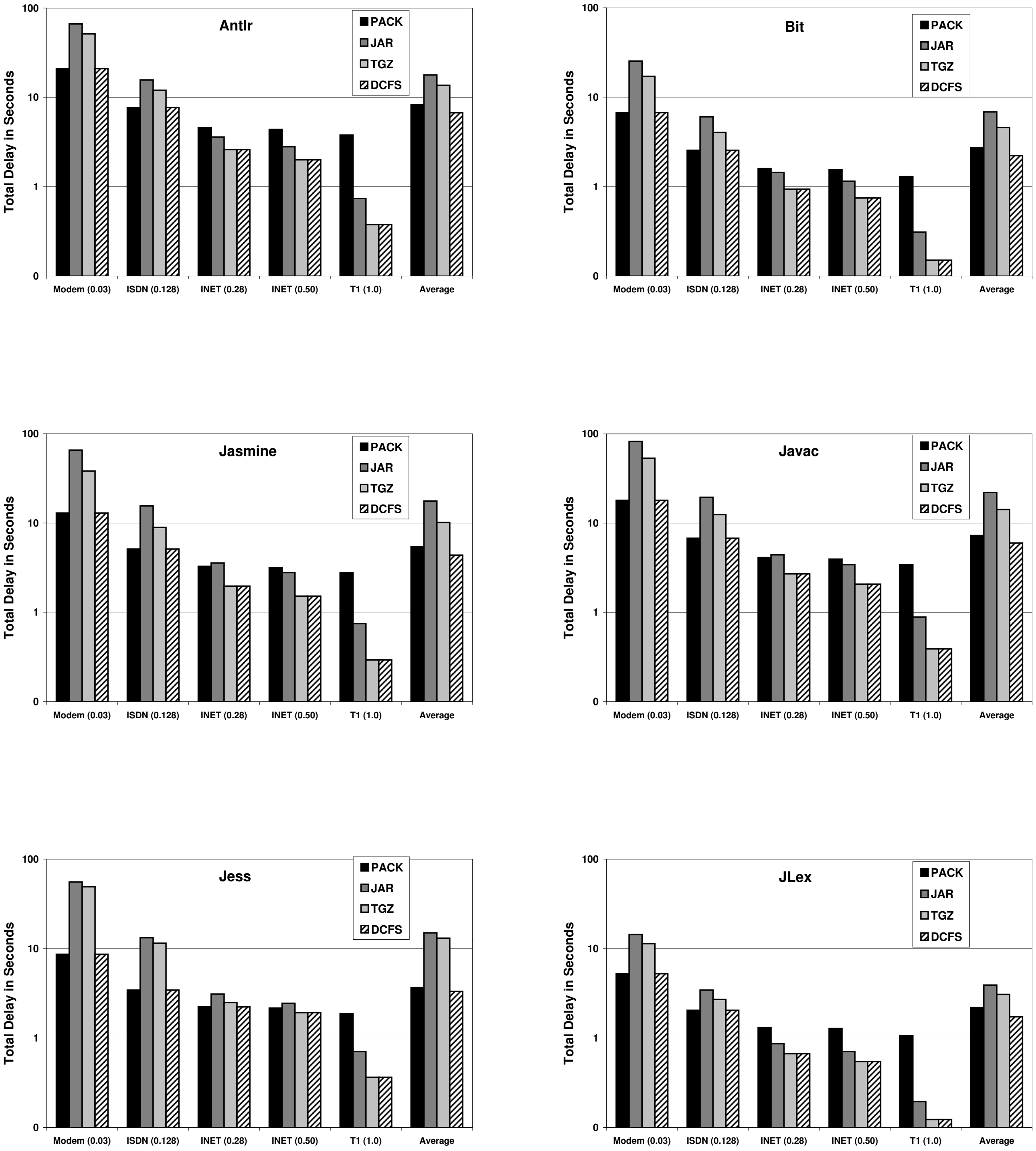

Figure 2. Total Delay in seconds (on a logarithmic scale) without and with (striped bar) DCFS. This set of graphs shows (for each benchmark) the total number seconds required for request, transfer, and decompression using each compression technique (bar), PACK, JAR, TGZ. The fourth (striped) bar of each set is the total delay when using DCFS. DCFS selects the format that results in the minimum total delay.

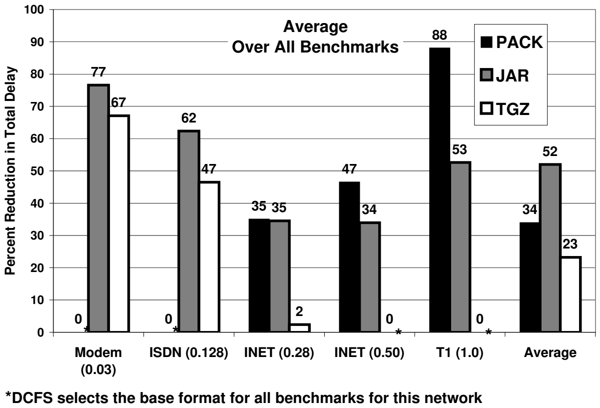

Figure 3. Average reduction in transfer delay over all benchmarks enabled by DCFS. Each bar shows the average percent reduction in total delay across all benchmarks for each of the three compression formats, PACK, JAR, and TGZ. Bars with zero values indicate that the base case was the format selected by DCFS, i.e., the base case was optimal and no additional benefits are possible. In every case, DCFS correctly determines and uses the format that requires the minimum total delay and significantly reduces it.

base case results in the minimum total delay for that network. That is, when the base-case format is the optimal one, DCFS selects it and no additional improvement can be gained. If PACK is used for non-MODEM bandwidths, e.g., LAN, then DCFS reduces total delay over PACK by almost 90% (2 seconds) on average across all benchmarks. For the LAN bandwidth, the optimal format is TGZ. However for MODEM and INET bandwidths, DCFS provides 67% and 46% average reduction (24 and 4 seconds), respectively, over TGZ. On average across all networks, 33% (1 second), 52% (10 seconds), and 23% (6 seconds) of the delay can be eliminated over PACK, JAR, and TGZ, respectively, using DCFS.

Interestingly, this summary graph shows that JAR compression is never selected by DCFS, i.e., there are no zero-valued JAR bars in Figure 2, using the bandwidths examined. This implies that using DCFS with only two compression formats can improve (substantially, in many cases) the performance of programs compressed using the Java archive for any network technology. This also implies that only two formats need to be stored at the server for application download, given current compression technology, to achieve the substantial reductions in total delay presented here. As compression utilities change, however, providing additional DCFS choices enables additional opportunity for improved performance. On average, across all benchmarks and bandwidths, DCFS reduces total delay imposed by jar compression, the most commonly used Java application compression technique, by more than half.

4. DCFS Use and Implementation

In the previous section, we articulated the DCFS design and reported results to indicate the potential of dynamic selection of compression formats to improve mobile program performance. Our results use average performance values from real network traces. In this section, we discuss practical implementations of DCFS and the implications of incorporating predictions of future network bandwidth.

The results in the previous section show that DCFS is able to select the appropriate compression format that results in the minimum delay assuming a constant bandwidth throughout execution for the different links. For example, for modem links, DCFS commonly chooses PACK; for LAN, DCFS chooses TGZ. However, the affect of variance is not represented by these results since we use a single network bandwidth value (the trace average). Network variance can cause DCFS to change the selection for a single link. For example, Figure 4 shows the bandwidth for two Internet connections. The first row of graphs is the data trace from which the INET bandwidth average was obtained. The second row provides data for a different Internet connection between the University of Tennessee and the University of California, San Diego. The left graph of each pair is the raw bandwidth measurement taken; the right is the cumulative distribution function (CDF) over all bandwidth values. Measurements were taken at just under one minute intervals over a 24 hour period that began at approximately 8PM.

In the right, CDF graphs, we have incorporated a vertical and horizontal line. The vertical line indicates the average bandwidth value 0.32 Mb/s at which the DCFS selection changes from PACK to TGZ over all of the benchmarks studied. For less than 42% of the values, PACK is chosen by DCFS for the link represented by the top pair of graphs; the remainder of time TGZ is chosen. For the link represented by the bottom pair, over 50% of the values on average, causes DCFS to select PACK. These results show that it is unclear as to which compression technique to use for this network. Hence, dynamic selection should be used to achieve the least total delay.

4.1. Prediction of Network Characteristics

The goal of our DCFS work is to determine what the total delay *will be* when the application or file transfers from the server to the client. To enable this, DCFS must *predict* the total delay value. Since the variable factor in the

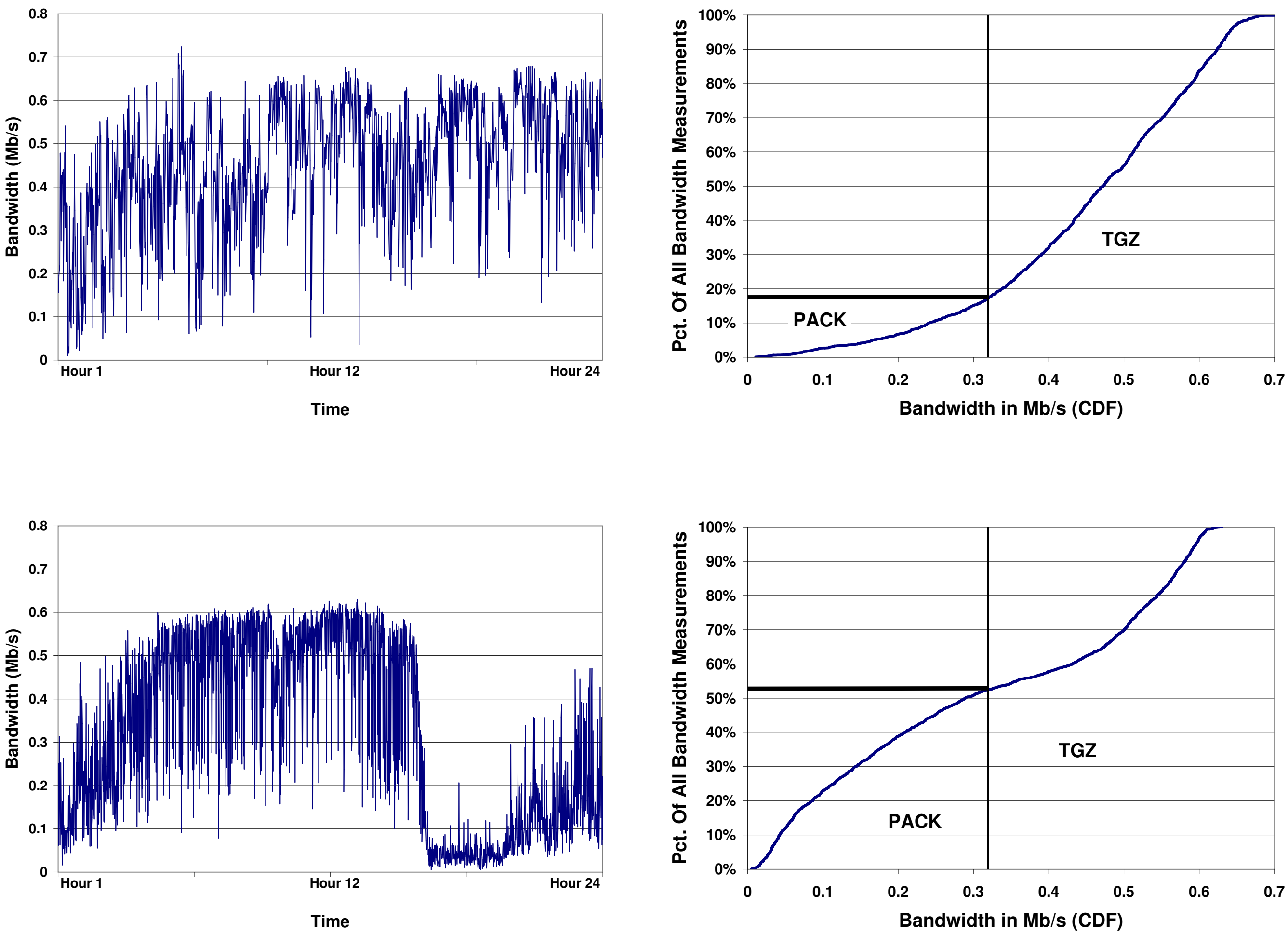

Figure 4. Raw data (left graphs) and cumulative distribution functions (CDF) (right graphs) for two data sets. The top pair is data from the INET trace used throughout this study. The bottom pair in also Internet data, however, between two different hosts. In the left graphs, the y-axis is bandwidth, and the x-axis is time. Both traces are of a single 24-hour period starting approximately at 8PM at night. The data indicates that these connections are highly variable and hence different DCFS choices can be made for a single link. The right graphs indicate (given the average bandwidth value, 0.32Mb/s indicated by the vertical line, at which DCFS chose a different format over all benchmarks studied) that in the top pair, PACK is chosen less than 19% of the time. In the bottom pair, the number of times PACK is chosen by DCFS is about the same number as that for TGZ.

total delay calculation is resource performance, we incorporate existing resource performance prediction techniques into the DCFS resource monitor depicted in Figure 1. Performance prediction is a well studied area, and we refer the reader to [14, 22, 7, 1] for more information on this vast research area. The contribution of DCFS is not the forecasting techniques themselves but instead is their use to exploit the space-time trade-off made by compression techniques to reduce the total delay imposed on mobile programs.

In this section, we describe the network performance techniques that we considered for bandwidth prediction and articulate the effect of using prediction on the upper-bound result performance presented in Section 3.1. Inaccuracy in predicted values can cause non-optimal DCFS performance. To measure accuracy, it is common to examine the *error* of a predicted value value: the difference between the predicted value and the actual value when it occurs. We next describe two techniques for bandwidth prediction and the affect inaccuracy of each has on optimal DCFS performance. The first is last bandwidth prediction via probes and the second is a Java implementation of the network weather service (NWS).

4.1.1 Last Bandwidth Prediction Via Probes

One way to predict the bandwidth when a transfer occurs, is to probe the bandwidth immediately prior to transfer. Using this approach, we predict that the bandwidth at a transfer time in the near future, will be equal to the current bandwidth. This is called last bandwidth prediction. Last bandwidth prediction can be incorporated into the DCFS resource monitor using any network probe utility available, e.g., ping, netperf [9], TTCP [19], JavaNws [12], etc. In addition, simple probing socket routines can easily be written from scratch.

As mentioned above, the accuracy of last bandwidth prediction is demonstrated by its error values. In the Internet connection data in Figure 4 in the top pair of graphs, the average error using last bandwidth prediction is 11.3KB/s. This value is obtained by taking the bandwidth values and subtracting them from the previous value in the trace, and taking the average of this difference over all measurements. On average, the difference between the last bandwidth value and the bandwidth when the application transfers is 11.3KB/s. However, since DCFS is making a binary decision (1 for PACK and 0 for TGZ in this case) based on whether the prediction is above or below a given threshold, this error will only effect predictions that are $+-$ 11.3KB/s of this threshold. To determine the extent to which this error limits the overall improvement by DCFS, we selected 100 random bandwidth values from both sets of INET trace data presented in the figure and computed the total delay required by each of the wire-transfer formats, as well as by DCFS. The total delay reduction from DCFS using last bandwidth prediction to be within 4% of that from DCFS using the actual trace values, i.e. perfect information.

4.1.2 NWS Prediction

The second performance prediction technique we considered is the JavaNws resource monitoring and prediction service. The JavaNws [12] is a Java implementation of an extension to the Network Weather Service (NWS) [23]. The NWS is a toolkit designed for use on the Computational Grid [4], a new architecture proposed to frame the software infrastructure required to implement high-performance (parallel and distributed) applications using widely dispersed computational resources. The NWS prediction facility (in the JavaNws version) is used by DCFS to predict network performance and hence total delay. The forecasting techniques are further described in [22].

The DCFS resource monitor is an extension of the JavaNws which takes periodic measurements of the network performance between the client and the server. A set of very fast[1], adaptive, statistical forecasting techniques are applied to the measurements (treated as a time series) to produce accurate, short-term predictions of available network performance [22].

The average JavaNws prediction error from the top Internet connection data in Figure 4 is 7.5KB/s (for an average bandwidth value of 0.5 (INET), this is 111ms). Smaller average error improves the potential for correct selection (and improved performance) by the DCFS. Since one of the forecasters used by JavaNws is a last bandwidth predictor, JavaNws will always enables equal or better accuracy than a last bandwidth predictor alone. Using 100 randomly selected bandwidth values from both sets of INET trace data presented in the figure, we achieve total delay reduction from DCFS using JavaNws prediction within 2% of that from DCFS using the actual trace values. For these links, DCFS with JavaNws prediction enables an additional 2% reduction in total delay than last bandwidth prediction alone.

5. DCFS Extensions

We next extend DCFS in two ways to further reduce total delay. The first technique is *Selective Compression*, in which only those class files used during execution are included in the compressed archive. The second is *Compression-on-demand* in which the application is compressed at the server when it is requested by a client.

[1]The JavaNws forecasting techniques require approximately 0.25ms to produce a single prediction, on the processor used in this study.

Dynamic Characteristics of Programs (Ref) (Train in parenthesis)			
Program	Classes	Executed Methods	Pct. of Total Size
Antlr	67 (69)	538 (549)	64 (64)
Bit	40 (37)	158 (153)	90 (88)
Jasmine	165 (159)	714 (669)	81 (78)
Javac	139 (132)	740 (713)	86 (41)
Jess	133 (135)	412 (412)	91 (92)
Jlex	18 (18)	99 (97)	95 (95)
Avg	68 (67)	313 (308)	57 (51)

Table 3. Dynamic benchmark characteristics. The three columns of data contain the number of classes used, methods invoked, and the size of the used classes as a percentage of total application size, respectively. These columns contain data for the Ref input followed by that for a second, Train, input in parenthesis.

5.1. Selective Compression

To further reduce transfer delay we combine and compress only those class files that are used by the application. At the server, applications are currently stored in various compression formats. In addition to this, we store Java applications in archives which contain only those class files that will be used during execution of the program. DCFS is extended to also consider these *selectively-compressed* archives. However, selective compression does not depend on DCFS for its implementation. We implement the optimization within the framework of DCFS to further reduce total delay and to demonstrate the extensibility of DCFS.

When an application is initially invoked and requested from a server, the compressed file of classes predicted to be used is sent for execution. When a class is accessed that is not contained in the selectively-compressed archive it is requested by the DCFS class loader at the client and transferred (possibly compressed if doing so results in minimized total delay). If there is very little difference in the size of the selectively compressed archive and the archive of the entire application, DCFS selects the latter to avoid unnecessary overhead introduced when a class file is used that is not contained in the archive. This heuristic is detailed in [10] and omitted here for brevity.

Table 3 shows the dynamic characteristics of the benchmarks given two inputs: Ref and Train. The first column of data is the number of classes that are used during execution for the Ref input (results for the Train input are shown in parentheses). The second column is the number of methods used and the final column is the percentage of total application size that used classes require. On average 33% of an archive is needlessly transferred.

Since we are unable to know precisely the set of class files that will be required by every execution of an application (this information is input-dependent), we use execution profiles generated off-line to estimate this set and to construct a compressed archive. As in the table above, we present data from two different program inputs indicate the effect of using profile data. All results presented were generated using the Ref input (as was done for the prior DCFS results). We denote selective compression results that use a profile generated using the Ref input to guide optimization by Ref-Ref (the first Ref indicates the input used for profile generation, the second indicates that used for result generation). Using the same data set for both profile and result generation provides ideal performance since we have perfect information about the execution characteristics of the programs. Results denoted Train-Ref are those that use the profile generated using the Train input to guide optimization. Train-Ref, or cross-input, results indicate realistic performance since the characteristics used to perform the optimization can differ across inputs and the input that will be used is not commonly known ahead of time.

Figure 5 shows the average improvement over all benchmarks when selective compression is combined with DCFS. The graphs present the results as percent reduction in total delay over PACK, JAR, and TGZ compression for each network bandwidth. The first bar in the set are the results due to DCFS alone, presented previously. The second and third bars indicate the performance benefit from selective compression alone using different profile inputs (Train-Ref and Ref-Ref). The final two bars show the combined effect of selective compression and DCFS using the different inputs (for selective compression). The cross-input (Train-Ref) results show that on average across all benchmarks, selective compression alone reduces transfer delay 8% for the modem link (1.0 seconds) and 8% for the T1 link (0.2 seconds) over always using the PACK utility. When selective compression is combined with DCFS, delay is reduced 10% for the modem (1.2 seconds) and 90% for the T1 link (2.2 seconds). Average improvements over always using JAR compression are 10% for the modem (5.2 seconds) and 16% for T1 (0.1 seconds) using selective compression alone and 80% for the modem (41.3 seconds) and 61% for T1 (0.4 seconds) when combined with DCFS. Improvements over TGZ are, on average, 18% for the modem (6.6 seconds) and 18% for T1 (0.1 seconds) using selective compression alone and 71% for the modem (26.1 seconds) and 19% for T1 (0.1 seconds) with DCFS.

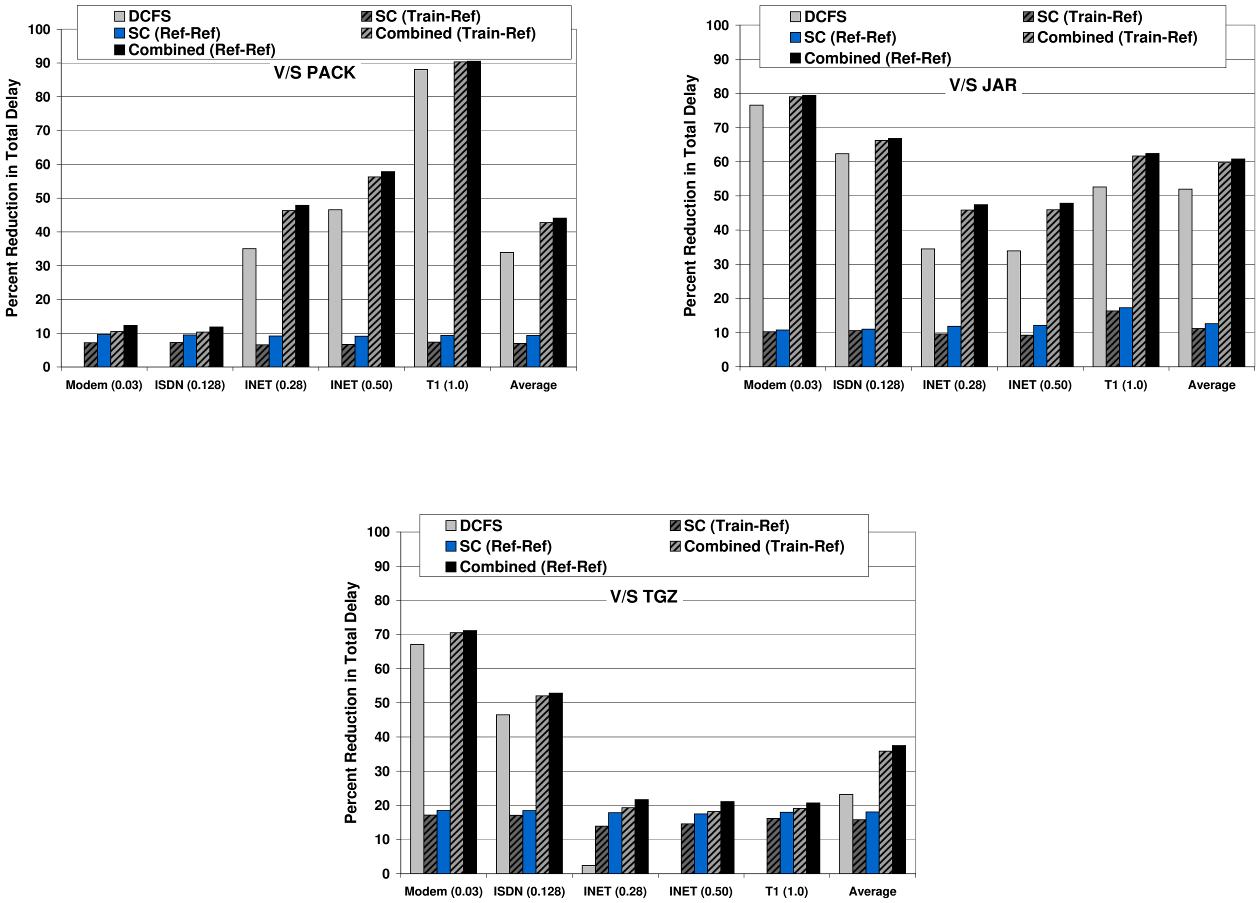

Figure 5. Percent reduction in total delay due to selective compression and DCFS on average across benchmarks. The graphs are a summary of results using PACK (top), JAR (middle), and TGZ (bottom) compression as base case, respectively. A series of bars is shown for each network bandwidth. From left to right, the five bars represent the percent reduction in total delay due to DCFS alone, selective compression alone (Train-Ref and Ref-Ref), and DCFS and selective compression combined (Train-Ref and Ref-Ref). The average across the range of network bandwidth is given by the final set of five bars.

Total Delay In Seconds			
	JAR	DCFS	
	T + D	C + T + D	T + D
Program	Time	Time	Time
MODEM (0.03)	51.6	21.8	12.1
ISDN (0.128)	12.2	7.6	4.6
INET (0.28)	2.8	1.8	1.9
INET (0.50)	2.2	1.5	1.5
T1 (1.00)	0.6	0.5	0.3
Average	13.9	6.6	4.1

Table 4. Compression-on-demand with DCFS on average across benchmarks. The first column of data is the total delay (without compression) using JAR files. The second column is the total delay *including* compression using DCFS with compression-on-demand. The final column is the total delay without compression using DCFS.

5.2. Compression-On-Demand

An alternative to requiring that the server store applications in a number of different wire-transfer formats, we consider simply storing class files. Then, when a request is made to a server for application download, the server is instructed to compress the application prior to transfer. The format is chosen by DCFS and is sent to the server upon application request. In this section, we articulate some preliminary results of the second DCFS extension we are implementing as part of future work.

In addition to decompression rates (Kb/s) and compression ratios, DCFS at the server also incorporates compression rates. Despite not being optimized for compression time, existing compression techniques can still be incorporated into DCFS to give insight into feasibility of including compression with dynamic format selection at class load time. As utilities change, improve, or are optimized for compression rates, the DCFS can incorporate and select them; those for which compression times prove impractical will not be selected. Total delay, when compression-on-demand is performed, consists of time for archival and compression as well as for the request, transfer, and decompression of the files. Average results across benchmarks are shown in Table 4. The first column of data is the time to request, transfer, and decompress using JAR compression (T+D). The second column of data shows the total delay using DCFS with compression-on-demand, denoted C+T+D. This is the time to request, compress, transfer, and decompress using DCFS (with compression formats PACK, JAR, and TGZ) in second. The final column of data is the average DCFS results from the prior section: DCFS without compression-on-demand. These values, included as a reference, consist of the request, transfer, and decompression time (T+D) *only* like that for the JAR data in the first column of data. Using DCFS with compression-on-demand reduces total delay by 20% to 50% over using jar files *without* compression (using the wire-transfer formats and networks from this study). That is, it is faster to compress, transfer, and decompress applications using dynamically selected wire-transfer formats than it is to simply transfer and decompress jar files.

6. Conclusion

Despite reductions enabled by the use of file compression, transfer time continues to impede the performance of mobile programs. With this work we exploit the trade-off that must be made by compression techniques (compression ratio for decompression time) to reduce the delay that remains from compressed transfer. Since no single transfer format is best (in terms of transfer and decompression time) for every network link at all times, we introduce *Dynamic Compression Format Selection* (DCFS).

DCFS is a utility that dynamically selects the compression technique based on the underlying, available resource performance predicted to be available when the transfer occurs. We incorporate the forecasting utility of a Java implementation of the Network Weather Service [22] (NWS) for performance prediction. DCFS computes the *total delay* that will result for various compression techniques using the predicted request, transfer, and decompression time of the application or file. The application in the format resulting in the least total delay is then transferred to the client. We show that DCFS reduces total delay substantially: 52% (10 seconds) over the Java archive (jar), the most commonly used compression format for mobile Java programs. DCFS reduces total delay for fast links (T1) 90% (2 seconds) over PACK compression on average for the benchmarks studied. For slow links (modem) it can reduce total delay on average 59% (25 seconds) over TGZ (tar and gzip) compression.

We also present two DCFS extensions to further reduce total delay called *Selective Compression* and *Compression-on-Demand*. The former optimization ensures that only the Java class files used during execution are included in the compressed archive that is transferred to the client. We use off-line profiles to guide archive exclusion. If a class file is used that is not included in the selectively-compressed archive it is transferred (possibly compressed) via existing dynamic class file loading mechanisms. Selective compression reduces total delay an additional 7% to 11% over DCFS alone.

The second DCFS extension, compression-on-demand, obviates the need to store applications in multiple formats if it is infeasible to do so. We extend DCFS to use compression-on-demand. Using this approach, an archive is constructed and compressed at runtime by the server when a program or class file is requested by the client. The compression format used by the server is the one that is predicted to result in the least total delay (the calculation of which now includes compression time). We show that DCFS using compression-on-demand can reduce 20% to 52% of the delay imposed by pre-compressed jar files (for which on-the-fly compression is *not* used). That is, it is faster to compress, transfer, and decompress applications using DCFS than it is to simply transfer and decompress jar files over the range of network performance values studied.

To enable these performance improvements, this DCFS implementation requires that applications be stored in various formats at the server or that the server perform additional work (compression-on-demand). Both of these requirements enable the server to choose between competitive compression formats to reduce the total delay imposed by always using using a single format regardless of the underlying resource performance. Currently, servers supply users with mirror sites to improve download times. Companies that manage servers are motivated by competition and continuously improve sites to ensure the satisfaction of customers/users. Our results motivate the need for compression format selection, and the storage of applications in additional formats on a server is a reasonable trade-off with the users reduction in load delay achieved from using DCFS.

Acknowledgments

We would like to thank the anonymous reviewers for providing useful comments on this paper. This work was funded in part by NSF CAREER grant No. CCR-9733278, a grant from Intel Corporation, and a grant from Compaq Computer Corporation.

References

[1] M. Crovella and A. Bestavros. Self-similarity in world wide web traffic: Evidence and possible causes. In *Proceedings of the 1996 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, 1996.

[2] W. Doherty and R. Kelisky. Managing VM/CMS systems for user effectiveness. *IBM Systems Journal*, pages 143–163, 1979.

[3] J. Ernst, W. Evans, C. Fraser, S. Lucco, and T. Proebsting. Code compression. In *Proceedings of the SIGPLAN'97 Conference on Programming Language Design and Implementation*, pages 358–365, Las Vegas, NV, June 1997.

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

[5] C. Fraser. Automatic inference of models for statistical code compression. In *Proceedings of the SIGPLAN'99 Conference on Programming Language Design and Implementation*, May 1999.

[6] C. Fraser and T. Proebsting. Custom instruction sets for code compression. http://www.cs.arizona.edu/people/todd/papers/pldi2.ps, Oct. 1995.

[7] N. Groschwitz and G. Polyzos. A time series model of long-term traffic on the nsfnet backbone. In *Proceedings of the IEEE International Conference on Communications (ICC'94)*, May 1994.

[8] S. M. Inc. The Java ARchive utility. `http://java.sun.com/products/jdk/1.1/docs/tooldocs/solaris/jar.html`.

[9] R. Jones. `http://www.cup.hp.com/netperf/`. Netperf: a network performance monitoring tool.

[10] C. Krintz. Reducing Load Delay to Improve Performance of Internet-Computing Programs. Technical Report UCSD CS2001-0672, University of California, San Diego, May 2001. PhD Dissertation.

[11] C. Krintz, B. Calder, and U. Hölzle. Reducing Transfer Delay Using Java Class File Splitting and Prefetching. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Nov. 1999.

[12] C. Krintz and R. Wolski. JavaNws: The network weather service for the desktop. In *ACM JavaGrande 2000*, June 2000.

[13] C. Lefurgy, P. Bird, I. Chen, and T. Mudge. Improving code density using compression techniques. In *30th International Symposium on Microarchitecture*, Research Triangle Park, NC, Dec. 1997.

[14] W. Leland, M. Taqqu, W. Willinger, and D. Wilson. On the self-similar nature of ethernet traffic. *IEEE/ACM Transactions on Networking*, February 1994.

[15] Pkzip format discription. `ftp://ftp.pkware.com/appnote.zip`.

[16] T. Proebsting, G. Townsend, P. Bridges, J. Hartman, T. Newsham, and S. Watterson. Toba: Java for applications a way ahead of time (wat) compiler. In *Proceedings of the Third Conference on Object–Oriented Technologies and Systems*, 1997.

[17] W. Pugh. Compressing Java class files. In *Proceedings of the SIGPLAN'99 Conference on Programming Language Design and Implementation*, May 1999.

[18] B. Quetzalcoatl, R. Horspool, and J. Vitek. Jazz: An efficient compressed format for java archive files. In *CASCON*, Nov. 1998.

[19] C. N. Solutions. Test TCP (TTCP). `http://www.ccci.com/product/network_mon/tnm31/ttcp.htm`.

[20] Specjvm'98 benchmarks. `http://www.spec.org/osg/jvm98`.

[21] A. Wolfe and A. Chanin. Executing compressed programs on an embedded RISC architecture. In *25th International Symposium on Microarchitecture*, pages 81–91, 1992.

[22] R. Wolski. Dynamically forecasting network performance using the network weather service. *Cluster Computing*, 1998.

[23] R. Wolski, N. Spring, and J. Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 1999.