COMPRESSING THE X GRAPHICS PROTOCOL

John Moffatt Danskin

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

November 1994

Abstract

This dissertation is about compressing networked graphics for transmission across low bandwidth networks like phone lines and wireless networks. It concentrates on the X Window System Version 11 (X) because X is the most widely available networked window system today. The dissertation is divided into three main sections:

The first section describes a system for tracing the X protocol, the traces gathered, and an analysis of the results. The traces were analyzed by breaking the protocol down into discrete instructions or messages, and analyzing the frequency and network resource requirements of these messages. This analysis has resulted in preliminary results in 4 areas: the relative importance of X messages, how X usage changes during different application stages, differing application idioms for accomplishing similar tasks, and the size distributions of X messages and groups of X messages. Some of these results have architectural implications for network protocols and implementations.

The second section examines the performance of the Xremote protocol, which was designed as part of a product by Network Communications Devices Inc. (NCD). The Xremote protocol is a compressed transformation of the X Window System protocol, designed to efficiently implement X connections across relatively slow serial lines. Using an Xremote simulator and 11 traces of X sessions, I found that Xremote's overall compression ratio is 2.4:1.

Text based applications commonly see about 2.5:1 compression, geometric applications achieve about 1.7:1 compression and image based document previewers achieve about 2.5:1.

The third section introduces a new technique for compressing structured data.

Embodied in the Higher Bandwidth X (HBX) protocol, this technique results in 7.5:1 compression across the trace suite, for about 3 times the performance of Xremote. HBX works by parsing X messages into individual fields and applying statistical modeling and arithmetic coding to individual fields of X messages. HBX typically achieves 5:1 compression on geometric data, between 10:1 and 20:1 compression on image based document previewing, and over 5:1 compression on text based applications.

It seems likely that the techniques used in HBX will prove applicable to compressing network headers, object code, RPC traffic and other structured data.

Acknowledgements

Digital Equipment supported me for the first 18 months under their Graduate Engineering Education Program. AT&T supported me for the last two years as an AT&T PhD fellow. Apple computer helped out with a grant.

I could not have even begun to do this work without the help, advice, and gentle prodding of my advisor, Pat Hanrahan, who has taught me all that I know about doing research.

Andrew Appel has had a series of neat ideas, which I hope I've stolen and not merely borrowed. I hope I can remember to try to think like Andrew when I leave.

My office mates over the years have helped me with useful and entertaining conversations, especially (in alphabetical order) Larry Aupperle, Michael Cox, S. V. Krishnan, and Jenny Zhao. My graduate school experience was enriched by their company, and I feel privileged to have spent time with these people.

My wife Allison Paschke has been a nurturing, encouraging, stabilizing, and entertaining companion for over ten years, and especially through the last four difficult years of graduate school. My son Arthur has kept me in touch with the basic joys of building gadgets, and my daughter Phoebe has always been game for a wrestling match with Thumby.

John Danskin
Princeton, NJ
1994

dedicated with love to Allison, Arthur and Phoebe

Contents

List of Tables

List of Figures

Chapter 1

Introduction

What we have here is a failure to communicate. Cool Hand Luke

Most of our time is spent transmitting from one end of the wire to the other Danny Hillis 1985

This thesis is about compressing graphics protocols. Immediately, two questions arise: what are graphics protocols, and why should I care about compressing them?

A graphics protocol is a language allowing an application program running on one computer to communicate with a graphics server running on another computer. Graphics protocols range from the quite limited abilities of some ASCII terminals, to protocols which allow the application to dynamically modify the protocol itself by downloading arbitrary programs which can draw arbitrary pictures. There are quite a few applications of this basic idea:

- Compute intensive scientific applications run on a supercomputer or a network of workstations while displaying visual results on and checking for user input from a graphics workstation.

- Students in computer science sit in front of inexpensive graphics terminals while their programs run on one or more centralized compute servers. This is especially efficient since the students can share cycles on the compute server, but they cannot share the keyboard, mouse and display.

- A financial analyst working at home runs complex forecasts on huge proprietary datasets on a computer at work, displaying results and changing parameters locally.

- A researcher on a train reads her mail on a remote computer using a laptop and a cellular modem. After reading her mail, she brings up a drawing program and updates some slides for a talk she is giving. The drawing program and the slides all reside on the remote computer with the rest of her files.

- Engineers in Massachusetts meet remotely with engineers in California, and share a virtual whiteboard via a graphics protocol.

- Surgeons at the medical school at Dartmouth work together with a doctor in Maine on surgical planning for an immovable patient, using a shared virtual surgery application and remote graphics.

All of these applications require some kind of graphics protocol and communication across some kind of network. These applications are also interactive. This means that if performance falls below some threshold, users will find them unusable, as illustrated in Figure 2. On the other hand, there is also a performance threshold above which users will not be able to detect improvements. I will be interested in cases where network performance is a problem, and where we can do something about it by compressing the graphics protocol.

Sometimes the network is fast enough. For instance, the network may be able to transmit graphics commands faster than the application can generate them, or faster than the graphics server can accept and act on them. In these cases, the network is not a bottleneck and compression is not appropriate. Another possibility is that the application runs fast enough with current resources. If the application can always provide feedback to the user in 100 milliseconds or less, performance improvements will probably be imperceptible, as illustrated in Figure 2. Again, compression would be inappropriate. However, it is common for the network to be a bottleneck. Whenever the network is a bottleneck, the compute server and the graphics server have spare cycles, which should probably be spent on graphics protocol compression.

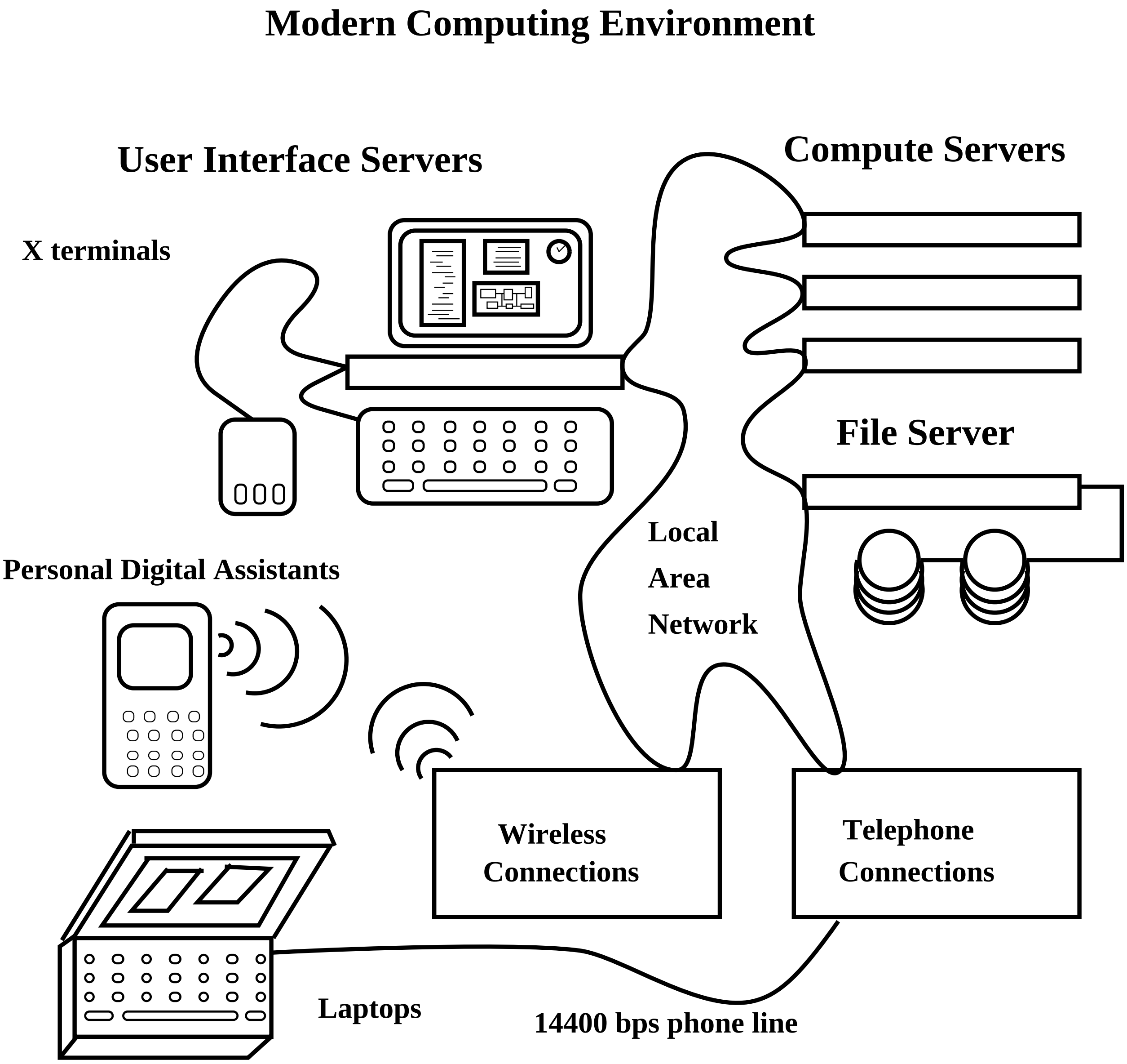


Figure 1: A modern computing environment. Some user interface servers are connected via high bandwidth local area networks, others are connected via low bandwidth telephone or wireless networks.

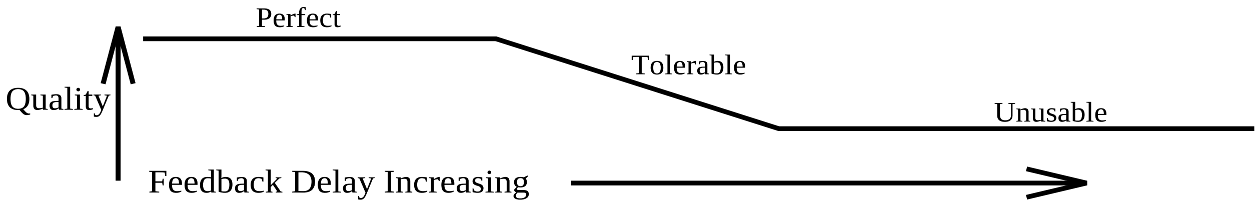


Figure 2: If an interactive application has feedback delay which is less than the interval of simultaneity for humans, then from a performance point of view, the application will be fast enough. Above that point, feedback delays will be perceptible, and lead to decreased performance, until, finally, the user gives up. The interval of human simultaneity is around 100ms, but varies from person to person and according to stimulus intensity [7]. The point at which an application becomes unusable is completely application specific, but some tasks, such as handwriting, become impossible with visual feedback delays as low as 150ms [24].

Network speed is relative. A one MIPS computer on a ten megabit per second Ethernet has a fast network: if the network is to be fully driven, then there are only 4 instruction times per 32 bit word, including application execution time and operating system overhead. Since most operating systems and applications are structured so that data has to be copied at least twice before reaching the network, there is little hope of saturating such a fast network with such a slow machine. Compression is clearly inappropriate here.

On the other hand, computers in the one and two hundred MIPS range are becoming common now, while the ten megabit network has not yet disappeared. These machines have four to eight hundred instruction times per 32 bit word, if the network is to be fully loaded. In this case, it is relatively easy to saturate the network, and there may be an opportunity for some inexpensive compression, depending on the application's need for cycles.

I have been most interested in the situation where a fast computer is connected to an extremely slow network, such as a wireless infrared connection, or a telephone line, or even a cellular phone. In these cases there may easily be hundreds of thousands of instruction times available per 32 bit word, and even compute intensive applications can find themselves spending most of their time waiting for the network. In this situation, it makes sense to devote considerable resources to extremely effective (and possibly resource intensive) compression.

It seems that a really well designed graphics protocol should already be fat free and fully compressed. Unfortunately, this is not the case. For example, if the network is extremely fast, then the protocol should be designed so that applications can generate messages as fast as possible. Fields should be aligned on word boundaries, with padding between them if necessary (increasing the number of bytes transferred!), all in the name of performance. On the other hand, if the network is extremely slow, it will be worthwhile to spend cycles keeping track of correlations between fields so that a compression algorithm can notice that 'u' nearly always follows 'q' and turn the 'u' into a very short message (usually a very small fraction of a single bit, in fact). Keeping track of these correlations, and carefully encoding messages would be too much of a burden for the application writer, especially when this level of compression is not always appropriate. The best way to design the graphics protocol is to assume the fastest possible network, so that the protocol can be fast when it needs to be, but to try to keep it simple so that it will be easy to write a compression filter in case the network is slow. The burden of compression can be assumed by a single filter (or by a set of performance appropriate filters) unloading the application writer, whose job is already hard enough.

To compress data means to remove all of the predictable bits, leaving only surprises. A fully compressed data stream should be completely indistinguishable from white noise. To do this, we can think of all of the checks we might apply to a data stream to distinguish it from noise. For instance, some bytes ('e' for instance) might be more popular than other bytes ('j'). We can take advantage of this type of regularity by assigning a short code to 'e' and a long code to 'j', saving some space in the total message. Looking at pairs of symbols, we might again notice some regularities: in English text, 'u' follows 'q' roughly 99% of the time. Surely we can save some space here too. Text can be effectively compressed working byte by byte, using only statistics about the previous few bytes. Compressing graphics protocols is a little bit harder.

Examples of regularity that we can take advantage of are easy to find in graphics protocols: when a user is typing, the terminal emulator sends text commands to draw characters. The x coordinate in these commands usually increments regularly from command to command, while the y coordinate remains fixed. In order to take full advantage of this fact, a compression algorithm has to understand the graphics protocol, and maintain statistics on correlations between elements in the datastream which are not necessarily adjacent or even in the same message. This is the fundamental idea of this thesis.

1.1 Graphics Protocols

Graphics protocols have existed in one form or another since the first graphics terminals. Somehow the computer had to tell the terminal what to display, and the terminal had to forward user input to the computer. I have chosen just four protocols for a quick discussion. These protocols were chosen either because they were seminal (Blit), because they epitomized a certain design philosophy (NeWS and PEX), or in the case of X, because X is the most widespread networked window system today, and part of the topic of this thesis.

1.1.1 Blit

The Blit [37] was one of the earliest bitmapped windowed terminal. The terminal consisted of a Motorola MC68000 microprocessor with 256KB of RAM, 24KB of ROM, an 800x1024x1 bitmap display, and a serial line capable of 19,200bps. (At home, Blits were used with 1200bps modems.) Standard graphics commands were implemented in ROM, but applications were encouraged to load code into the Blit and run as a pair of communicating processes, in order to reduce round trips and bandwidth requirements, and thereby improve performance. A multitasking operating system on the Blit allowed multiple applications to run and update their displays simultaneously.

The text editor, *jim*, is a good example of how applications were broken into communicating pairs. The part of *jim* residing on the Blit was responsible for maintaining the displayed text. The host side of *jim* maintained complete text of the edited files. The pair communicated mostly with requests to insert or delete a string. Operations which only updated text on the display (including most keystrokes) did not require any communications at all.

Users loved the Blit, and its applications performed well, but they were tied to a specific hardware architecture, and a specific CPU to network performance ratio. This meant that they were difficult to port to new systems, so most Blit applications never made the jump to the next architecture. There are still a few Blits in action now (after 12 years), and they still outperform X terminals when running on slow lines [38], but for whatever reasons, the Blit has been overrun by X (Section 1.1.4).

1.1.2 NeWS

NeWS [3] (the Networked extensible Window System) was a spiritual descendent of the Blit, with some of the problems fixed. Blit programs were tied to a certain implementation of the Blit terminal, involving a certain microprocessor. NeWS programs communicated with the graphics server, which could be a dedicated terminal or (more commonly) a workstation, using a slightly modified version of the POSTSCRIPT [1] page description language. POSTSCRIPT is a complete programming language, so applications could split themselves in half, as they did for the Blit, to reduce bandwidth requirements.

In practice, although there were some split applications, most applications used POSTSCRIPT strictly as a primitive graphics language and downloaded very few routines. This can be attributed to several causes:

- POSTSCRIPT is hard for C programmers to program in.
- The POSTSCRIPT interpreter was not very fast compared to compiled C programs, so there was very little performance advantage to downloading POSTSCRIPT programs.
- Code written in terms of graphics primitives can be ported from one window system to another more easily than code which has been rewritten in a special language specific to a single window system.

- A common example of an obvious opportunity for local processing is the input loop: cursor tracking and keyboard input. However, input processing is often complex, after all, input drives the application, and for the above reasons, it is inappropriate to rewrite complex application code in POSTSCRIPT.

It is likely that the above reasons had little to do with the failure of NeWS to become widespread, because most programmers seemed to like the POSTSCRIPT imaging model, even if they did not decide to implement complex functions in POSTSCRIPT. They do, however, point to the practical difficulties distributing applications across networks in order to reduce bandwidth.

1.1.3 PEX

PEX [44] (Phigs Extension for X) is a 3D extension for the X Window System (discussed below). It allows application programs to define "structures" which are like parameterized macros containing lists of 3D drawing commands. Structures are also known as display lists. These structures are transmitted across the network once when they are defined. Afterwards they can be redrawn by name. Since structures can contain references to other structures, objects of great complexity can be built while transmitting only a small number of bits across the network connection.

This architecture has proven successful for networked 3D graphics because specialized 3D drawing hardware is often faster than available networks. For instance, the (admittedly high-end) SGI Reality Engine can consume vertex-normal pairs at over 192Mbps, while the Ethernet delivers less than 10Mbps. Explicit compression has been successful for networked 3D graphics, because the bandwidth requirements are so high, and possibly because while the objects stored in display lists are often moved around, they are not often modified because they usually represent real world objects like nuts and bolts which are usually positioned but not warped.

Because 3D graphics is so demanding, it is not usually considered worthwhile to try to use 3D graphics across slow interconnects like phone lines. Just downloading models once across a phone line would be intolerably slow. The sort of networks that 3D graphics is usually used on today are too fast for compression to be useful; any reasonable compression algorithm would not be able to keep up with the network. In this arena, explicit compression is still appropriate.

1.1.4 X11

X version 10, designed at MIT as part of project Athena in 1984 and 1985, was released and widely distributed in 1985. As X10 suffered from many designed in limitations and barriers to portability, work started almost immediately on the specification for X11, with a sample implementation becoming available in the spring of 1987. X11 has proved to be a portable and versatile application platform, eventually displacing all other window systems as the primary interface for workstation graphics.

Probably the most complained about aspect of X is its low level interface. It can take days to implement a window with simple text input functions. The philosophy has been to keep the interface at a very low level so that different applications can implement very different user interfaces without interference from the window system. In a sense, X has tried to be a universal graphics machine language, and it has succeeded in that it has captured most workstation applications which use 2D graphics.

The part of X that runs on the machine in front of the user is known as the X server. The X server draws on the display and handles input devices such as the mouse and the keyboard. The user programs that connect to the X server are called clients. Probably the most commonly run X client is xterm, the X terminal emulator. Other popular clients include clocks, text editors (WYSIWYG and otherwise), and drawing programs.

The X server notifies X clients whenever an event that the client has expressed interest in occurs. These events include keystrokes, mouse motion, and button presses. X clients respond to events from the the X server by sending state changes and drawing commands. All communication between client and server is asynchronous so as to minimize round trip delays. There is no way for X clients to download intelligence into the X server.

Most X connections are either local (no network involved), or over an Ethernet connection using TCP/IP [40, 39] and Berkeley sockets. Both of these environments usually provide adequate bandwidth for X [14]. However, X is also used in environments where bandwidth is precious, such as over the internet and across public telephone lines using fast modems and a serial line network protocol [11].

1.1.5 Strategies for Managing Bandwidth

The Blit, NeWS and PEX all involve the programmer, and in the case of PEX, sometimes even the end user, in the task of bandwidth management. In principle these techniques are extremely powerful. Since NeWS and Blit applications can redefine the protocol, in principle, they can incorporate any compression algorithm.

In practice, explicit bandwidth management can be painful and difficult, turning an easy client-server programming task into a hard general message passing problem.

The alternative is implicit bandwidth management, as exemplified by X, where the application developer has no direct control over encoding. If compression is necessary, it must be done by the system or the application will run poorly. Another example of a graphics protocol with implicit bandwidth management is TV. Television directors and cinematographers do not worry about the possibility that their work might be shown over a low bandwidth network; they trust that the system will compress their data for them.

Implicit support eases application development by releasing programmers from the burden of managing bandwidth explicitly. The challenge for an implicit system like the one described in this thesis is to provide the power of an explicit system like NeWS or PEX with none of the pain.

1.2 Networks

Networks are everywhere. The phone system is a network. We all send and receive electronic mail across the global internet. Recently I received mail from a colleague using a laptop computer connected to a corporate network via packet radio. Soon the distinction between low bandwidth voice grade telephone lines and high bandwidth data grade networks will blur as electronic services emerge. Any day now we will all be grocery shopping, reading books, and watching movies on demand over computer networks.

However, networks, like roads and other infrastructure, are rarely upgraded once they are in place. The network I have seen the most of is the 10Mbps Ethernet. This network was supposedly about to become obsolete in 1986 when I first worked with it, and I am still using it eight years later. It is still about to become obsolete. While this network has remained the same, the CPU I use everyday has become 74 times faster.

We are currently imagining that the 10Mbps Ethernet, 14.4Kbps phone lines, and everything else slow is about to become suddenly obsolete because of fiber optic networks. I imagine that this will take time, and that we will have slow networks, and networks that are slow compared to our computers for a long time.

Here are some examples of networks which are available today:

Network	K bits per second
Atomic	496,000.0
Ethernet	10,000.0
Local Area Wireless (infra red)	4,000.0
ISDN	64.0
Telephone with modem	14.4
Wide Area Wireless (Cellular modem)	4.8

People working at a distance from their main computer installation are usually forced to use one of the slower network connections, a standard telephone or a cellular modem for instance. These networks have insufficient bandwidth to support networked graphics as is, but happily they are slow enough so that a modern computer has plenty of time to spend on compression while waiting for the network.

It is possible that improvements in wired networks could outstrip improvements in CPU power over a long period, rendering network compression unusable. This could happen in limited domains where improvements in network performance may outstrip improvements in computer performance for some time. However, as long as there are shared networks, and networks where users are charged for use, compression will increase efficiency and lower costs.

In the world of wireless networks, the outlook is not so rosy. There is a limited total amount of usable bandwidth, and the number of users is increasing. As the available bandwidth per user decreases, we will need more efficient transport mechanisms. These mechanisms will continue to include compression.

1.3 Compression

In order to compress data, you have to know something about it. This is because most bit strings are incompressible by any fixed scheme. You can show this by a counting argument: suppose you want to compress all strings of length 1024 to strings of length 1023 by some scheme S. After compressing half of the strings of length 1024, you have used up all of the strings of length 1023: the rest of the strings are incompressible by scheme S. If you want greater compression, the restrictions are more severe. We can usually compress data because it usually exhibits some kind of regularity which we can detect and take advantage of. In fact, perfectly random data streams are very rare and are prized for their lack of regularity.

It makes sense to ask what is the smallest possible representation for a given string or data file. In program size complexity, also known as Kolmogorov complexity [51], the smallest representation of a string ζ is shortest member of (τ, σ) where τ is the index of a Turing machine, σ is a string on τ's input tape, and (τ, σ) leaves ζ on the output tape when it halts. This measure of the minimum length of a string depends only on the existence of a computer program, chosen from the space of all possible computer programs, which given the right input, will eventually produce the string.

Kolmogorov complexity is not computable, because in order to compute Kolmogorov complexity, we would have to test Turing machines to see what their output is, and the Turing machines might not halt. Therefore we must use weaker methods for compression. Each of these weaker methods will be tuned to compress certain types of data, detecting certain types of redundancy, ignoring others. Below, I have listed some of the types of compression which are relevant to this thesis, how they work and how well they work.

1.3.1 Text Compression

There are several possibilities for attack on the problem of text compression. The first is character frequencies: if we give frequent characters shorter codes, and infrequent characters longer codes, we can compress text. The length of the code for a character c should ideally be about $-\log_2 p_c$ where p_c is the probability, or fractional frequency of c. For instance, if $p_c = .25$ (on average, every fourth character is c), then the code for c should be 2 bits long. Huffman coding is a technique for arriving at an optimal set of codes, given a set of characters and probabilities, which is exact when the probabilities are inverse powers of two (implying that characters are coded in whole bits), and approximate otherwise. Arithmetic coding allows the expression of characters in fractional bits, and is exact whether the probabilities are inverse powers of two or not. The arithmetic coding technique is outlined in Section 4.1. Using character frequencies to code English results in about 5 bits of output per 8 bit input character.

We will refer to this coder as a zero order coder, and its table of 256 probabilities as a zero order model. The model used to compress the text can be derived in one pass over the data, and the data encoded in a second pass, in which case the model must precede the coded data to make decoding possible. Another, more popular alternative is to start the model off predicting each character with equal probability, and to adaptively adjust probabilities as each character is read.

Another line of attack is to realize that in English, strings are often repeated. For instance, words usually appear many times, and even pairs of words are often repeated. There are two popular tacks for taking advantage of this kind of redundancy: dictionary coding and statistical coding.

The most well known and efficient example of dictionary coding is the variant of LZ coding known as LZ78, developed by Ziv and Lempel in 1978 [54], and popularized and named LZW by Welch in 1984 [53]. This is the coder used in the Unix *compress* program, and the popular CCITT V.42bis compression standard for modems [20]. LZ78 maintains a dictionary of strings, a prefix string and a new character. When a new character is read from the input, if the catenation of the prefix string and the new character is in the dictionary, the new character is appended onto the prefix string, and there is no output. When the catenation is not in the dictionary, the index of the dictionary entry corresponding to the prefix string is transmitted, and the prefix string is set to the new character. If the dictionary is intialized with codes for all possible single input characters, then progress is ensured. There is an example of an LZ78 coder in action in Table 5.

Typical performance on English text for an LZ78 coder is about 3.5 bits of output per character, although 2 bits per character is common for computer programs. LZ78 coders are extremely fast.

Statistical coding subsumes LZ78, in that LZ78 can be expressed as a special case of statistical coding. In statistical coding, characters are assigned probabilities according to their frequency in some context. This probability is then used to generate a code as in the zero order coder above. The key is the context. In the zero order coder, we had no context (hence, "zero order"). A first order coder remembers one character of context (the previous character) and makes predictions based on the previous character. For instance, a zero order model primed on the text of this introduction would assign a relatively long code to 'Z' ($\approx$ 11 bits), because 'Z' is relatively infrequent, but a first order model, given the context 'L' would assign a short code to 'Z' ($\approx$ 1 bit) because 'Z' often follows 'L' in this introduction.

A straightforward implementation of a zero order model for characters is a table of 256 frequencies, and a total character count. The probability of a given character is estimated as its frequency divided by the total character count. The straightforward implementation of a first order model for 8 bit characters would involve 256 zero order models, one for each possible previous character. The zero order model for 'a' keeps the frequencies of each of the 256 possible characters in the context 'a', together with the total count of characters found following 'a'.

After the first order model, straightforward implementations are no longer practical, because they would require too much storage, for instance, a second order model would require storage of 2^{24} frequencies, and a third order model would require storage of 2^{32} frequencies. The key idea in the practical implementation of high order models is that infrequent events are not very important, and so storage can be saved by keeping poor statistics for infrequent events.

Another problem with high order models is that they take a long time to prime, and while they are priming, the statistics in the tables are not very good. Because of this, it is common to blend statistics from lower order models with statistics from higher order models when higher order models have not yet been filled up. For instance, suppose that a 3rd order model has the context 'LZ7' from this thesis, for the very first time. It has no statistics at all for this context yet, so it cannot make a prediction. If the third order model has not yet seen 'LZ7', then the second order model has not seen 'Z7' yet either, so the second order model cannot make a prediction either. Finally, the first order model has seen '7', but only followed by a '4', so it has to code an escape character (this is the first (context, model) pair where a decoder might expect to get the character, because this context has been seen before), and finally the zero order model can make a prediction because '8' has been seen before in the input stream. If the zero order model had never seen an '8' before, then it too could have coded an escape symbol, and transmitted the ASCII bits for '8'.

This type of blended third order model is referred to as PPMC in the literature. It is discussed further in Section 4.3. PPMC usually reduces English text to about 2.5 bits per character.

Other types of predictive models can be built with statistical modeling techniques. There are state machine based models. It is possible to build a variable order statistical model with a dictionary scheme like LZ78. The possibilities are endless. At the moment, PPMC seems to give the best compression.

1.3.2 Bi-level Image Compression

There has been a lot of work done on compressing all kinds of images, but I am going to concentrate on just bi-level or bitmap images because they are the kind most commonly used for document previewing with the X Window System. Probably most of the work that has been done on bi-level image compression has been for FAX machines which send facsimiles of documents across normal phone lines.

Essentially all modern FAX machines use an International Telegraph and Telephone Consultative Committee (CCITT) group 3 FAX protocol [28]. This protocol presently supports two compression options.

- The first group 3 FAX compression technique is one dimensional, running across scanlines. Each scanline is broken into runs of black or white pixels. The lengths of the runs are coded using a static Huffman table, mapping common run lengths onto short codes and rare run lengths onto long codes. This scheme achieves an average of about 8:1 compression across the 8 CCITT test documents [32], with a compression ratio as high as 16.7:1 for a nearly all white document (document 2), and as low as 4.9:1 for a document densely covered with text (document 4).

- The second group 3 FAX compression technique is two dimensional, running between scanlines. The first scanline is sent using the one dimensional technique described above. Afterwards, the locations of beginnings and endings of runs are tracked from one scanline to the next, with short codes for small displacements. There are special codes to deal with new runs, disappearing runs, and runs that move too far. For more detail, see [32] or [28]. This scheme achieves an average of about 12.9:1 compression, with a high compression ratio of 32.4:1 for the nearly all white document 2, and a low compression ratio of 6.5:1 for the densely formatted document 4. Document 4 has about 5000 characters, taking 175704 bits to send, for a rate of about 35 bits per character.

1.3.3 Geometry Compression

Geometry compression is not as well developed as text compression or image compression, partly because geometry is not as common and partly because the problem is underconstrained. What kind of geometry? 2D or 3D? Floating point or integer? For the purposes of this thesis, I am only interested in 2D integer geometry, because that is what X does.

The Tektronix 4006-1 vector graphics terminal (see [15]) had a 10 bit coordinate space. Commands were specified in 7 bits each. Specification of a full 20 bit XY pair required 4 commands, with 2 bits of each command specifying which part of which value was being updated, and the low 5 bits carrying the data. After the first coordinate in a vector, remaining coordinates were interpreted as offsets from the previous position. Since these offsets were often small, Tektronix designed the terminal so that the 5 most significant bits defaulted to zero if their commands were skipped. This feature provided compression approaching 2:1 very inexpensively.

Mark Manasse at DEC SRC developed a low bandwidth 2D graphics protocol called SL [26]. SL has two versions of every geometry-carrying message, the usual one with 12 bit coordinates, and the fully general one with 32 bit coordinates. (This is hidden from the applications which think they always 32 bit coordinates.) By almost always avoiding full 32 bit coordinates, SL achieves almost 3:1 compression with respect to its full coordinate space.

X uses a 16 bit coordinate space. By limiting itself to 16 bits, X makes life difficult for applications with larger internal coordinate spaces, such as some VLSI applications. On the other hand, if X had used 32 bit coordinates, geometry bandwidth requirements would have doubled. However, if X had used 32 bit coordinates, but most applications had limited themselves to the 10 or 11 bits of onscreen coordinates, a compression filter would have been able to remove most of these extra bits, while still preserving functionality for applications which needed large coordinate systems. 16 bit coordinates were a premature optimization for X.

The system described in this thesis will be able to take advantage of small coordinate values, as SL does, and will even coerce strings of absolute coordinates into relative coordinates, which will further tend to reduce coordinate magnitudes. This is not the key idea however. The key idea is to recognize repeated strings of geometry, to recognize the structures or display lists that a skilled X programmer would have created if X supported display lists and he or she had the time. For floating point coordinates, and under rotations and scaling, recognizing repeated geometry is difficult (because of noise in the low order bits of the coordinates), and out of the scope of this thesis. With relative integer coordinates, text compression algorithms such as LZ78 or PPMC can exploit repeated geometry, even under translations, which are common in 2D graphics.

1.4 Software Network Compression

The opportunity for software network compression is proportional to the mismatch in performance between the computer and the network. When the network transmits data as fast as the computer can produce it, spending cycles on compression is counterproductive. When the computer must wait for the network, cycles spent on compression are well-spent.

Many modems now implement a version of the LZ78 [54] compression algorithm. This hardware compression algorithm speeds ASCII file transfers, usually by over 2:1, but has little or no effect on binary file transfers, and actually increases the latency of keystroke echoing. As you will recall, when the catenation of the prefix string and the new character are found in the dictionary, the LZ78 algorithm sends no output. A modem implementing a byte stream does not know whether the current input character is the last for a little while or not, so it has to wait a little bit to see if there will be another character before sending the code for the prefix string and new character.

The same problem occurs (incidentally) with error correction in the modem. Error correction is done on blocks of data, when a block is not full, the modem has to wait a bit to see if there is another character to fill the block or not before flushing.

These problems mean that hardware error correction and compression actually result in degraded interactive performance for modem users.

Computers are so fast compared to serial lines, that there is no reason not to do compression in software, especially since packet sizes are often available. If compression was done in software, then there would be more room for model or dictionary space, modems would be less expensive, different compression algorithms could be applied to different types of data. If error correction was done in software, software could use appropriate levels of error checking to accommodate different requirements, for instance, some users or applications might be willing to tolerate a lower bit rate for better confidence, while others might be willing to run quickly and dangerously.

1.5 Compressing X

The problem of compressing a graphics protocol such as X differs from classical compression problems such as image, text, audio, and video compression in several ways:

1. Graphics protocols support many different types of data including images, geometric objects, and text. Furthermore, switching between datatypes is fine-grained: most X messages are less than 100 bytes long, and contain several different types of data. An effective compression process cannot waste bits shifting gears between compression techniques.

2. Graphics protocols support interactive applications. Humans can detect, and are affected by feedback delays exceeding 100ms [7]. Delays can arise either from the compression architecture, or from insufficient network bandwidth. For instance, MPEG video compression is not suitable for interactive applications because of the intense computation required for high quality transmission.

3. Graphics protocols support traffic in two directions, it could be that the two streams are best served by different compression strategies, or that there is a way to use messages traveling in one direction (stimulus) to make predictions about messages traveling in the other direction (response).

4. Graphics applications often generate a lot of messages when they start up. This traffic is usually a small fraction of the total traffic generated by an application over the minutes or hours the application is in use, but it can make a big impression on the user if it contributes to sluggish application startup. Other messages, which may seem more important because of total bulk, may not need compression because the interactions which they support already appear instantaneous to the user.

5. Like text compression, but unlike image, audio, and video compression, most graphics protocols must be compressed losslessly. Bit-rate goals cannot be met by gradually throwing more and more of the user's bits away.

Effective compression solutions for the X protocol will have repercussions beyond just the current X community. Ideas that work for X will work for other networked graphics protocols and for the graphics component of multimedia protocols. Reduced bandwidth graphics is a key component of mobile computing systems which use remote computing and networked graphics, and also a key component of ubiquitous computing where there may be very many small computers supporting only a graphical user interface. There are many applications waiting to be enabled by effective efficient graphics protocol compression.

1.6 Thesis Overview

Chapter 2 tells how I profiled X, describes the traces I gathered and which I used throughout this work, introduces a new taxonomy for X commands, and presents data on how X is used, and how much bandwidth is consumed by the different types of X messages.

In Chapter 3, I introduce the reader to the Xremote compression protocol. Xremote [11] is a transformation of the X protocol designed (by Network Computing Devices, Inc.) as a way to efficiently implement X connections over serial lines. This protocol is widely used to support X sessions across telephone lines for home use. The anecdotal evidence is that user satisfaction is low. There is a closely related follow-on to Xremote in the works called Low Bandwidth X [16](LBX). A prototype of LBX (without protocol specification), apparently based on Xremote, is now shipping with X11 release 6. Xremote is implemented as a pipeline with two major compression stages. I studied the performance of these two stages, separately and together, as a function of message type and for entire traces. I discovered that Xremote achieves about 2.4:1 compression overall, but performs poorly on geometry and images. Xremote's two compression stages: Delta compression and LZW compression each provide some compression, but they do are not complementary, they compete: they recognize and remove the same type of redundancy from input streams.

Chapter 4 introduces Structured Data Compression (SDC), a technique for using custom predictive models to predict each field in a structured data stream. An arithmetic coder uses these predictions to send messages using the minimum number of bits. This technique allows for significantly better compression of structured data than what is possible with byte-stream coders such as LZ78.

SDC is the underlying technology for Higher Bandwidth X (HBX), a compression system for X, which is discussed in Chapter 5. Custom predictive models were developed for fields in popular X messages. The result provides compression which is over three times that of Xremote (7.5:1 over my trace suite), while using fixed storage and linear algorithms. Chapter 5 details the architecture, implementation, compression performance, and resource requirements of HBX.

The main contributions of this thesis are:

- The technique of profiling a graphics protocol as if it were an instruction set, finding the most resource intensive graphics messages and classes of messages, for each member of a trace suite.

- A taxonomy for messages in the X protocol, allowing us to analyze the behavior of groups of X messages, such as geometry and text messages.

- The results of profiling X: average message size, average sizes of trains of messages, most important messages, and the distribution of popularity of messages. X obeys the 90-10 rule, meaning that over 90 percent of the network traffic is generated by fewer than 10 percent of the different types of messages.

- The measurement of the compression performance of the Xremote compressed X protocol, using the profiling techniques introduced above. By analyzing the performance of the components of Xremote separately, I was able to show that Xremote's two compression components are not complementary. By analyzing Xremote's performance by message type, I was able to show that Xremote has very poor compression on images and geometry, important types for a compressed graphics protocol.

- The idea of preconditioning for compression. In HBX, first the input stream is preconditioned. Frequency based predictive models cannot predict fields with arithmetic relationships, for instance, incrementing fields like timestamps.

Timestamps increase from message to message by small integer amounts. These integers can be successfully predicted by a frequency based predictive model. HBX allows protocol transformations which transform fields into a form where they can be predicted by frequency based predictive models.

- The idea of associating an arbitrary predictive context with a data field, rather than just using the previous few bytes as is usual for text compression. For instance, in an X_PutImage message, the message length field is a reasonable, although imperfect predictor of the width field, and the length and width fields together are an almost perfect predictor of the height field. So where, for example, a text compressor you search for the predictive model indexed by the context 'th' which predicts 'e' with probability .67 (0.58 bits), HBX searches for a predictive model indexed by the message type (X_PutImage), the offset of field being predicted (the height field in this example), the message length (8 words), and the image width (8 pixels). The model indexed by this context is constrained by X semantics to predict heights of 5, 6, 7, or 8 only, so the 16 bit height field will be encoded in 2 bits at most.

- The combination of the two ideas above constitutes a new paradigm for compressing structured data with possible applications in RPC, serial line network protocols like CSLIP, and other areas where there is a possibility of compressing structured data.

- The HBX implementation is a proof of concept for the new compression ideas, achieving 7.5:1 compression, or more than three times the compression of the commercial Xremote protocol across a widely based trace suite. The details of the HBX implementation also highlight the effectiveness of certain specific models for compressing specific types of data, providing guidance for practitioners attempting to duplicate these results on another graphics protocol.

Chapter 2

Profiling the X Protocol

2.1 Introduction

This chapter presents a method for capturing and analyzing the X protocol, as well as the results. Figure 4 is an example of a display of a trace which I gathered, showing the time, size, and type of X messages processed during a three minute period. I will present a taxonomy of X messages. This taxonomy will allow a discussion of the relative importance of groups of X messages such as drawing, imaging, and windowing operators. I continue with a discussion of how X applications go through different stages, and how this is more important for interactive applications than for batch applications. After adding the dimension of time, I zoom in to the level of individual X messages, examining X application idioms for implementing keystroke echoing. Finally, I will examine the size distributions of X messages and groups of X messages. To conclude, I will point out some of the implications of these results.

2.2 About X

X applications (or clients) connect to the X server using a reliable byte stream connection such as TCP/IP and Berkeley sockets, as shown in Figure 3. Typical X clients include terminal emulators, text editors, 2D drawing programs and POSTSCRIPT previewers. The X server selectively forwards user input to the clients, and updates

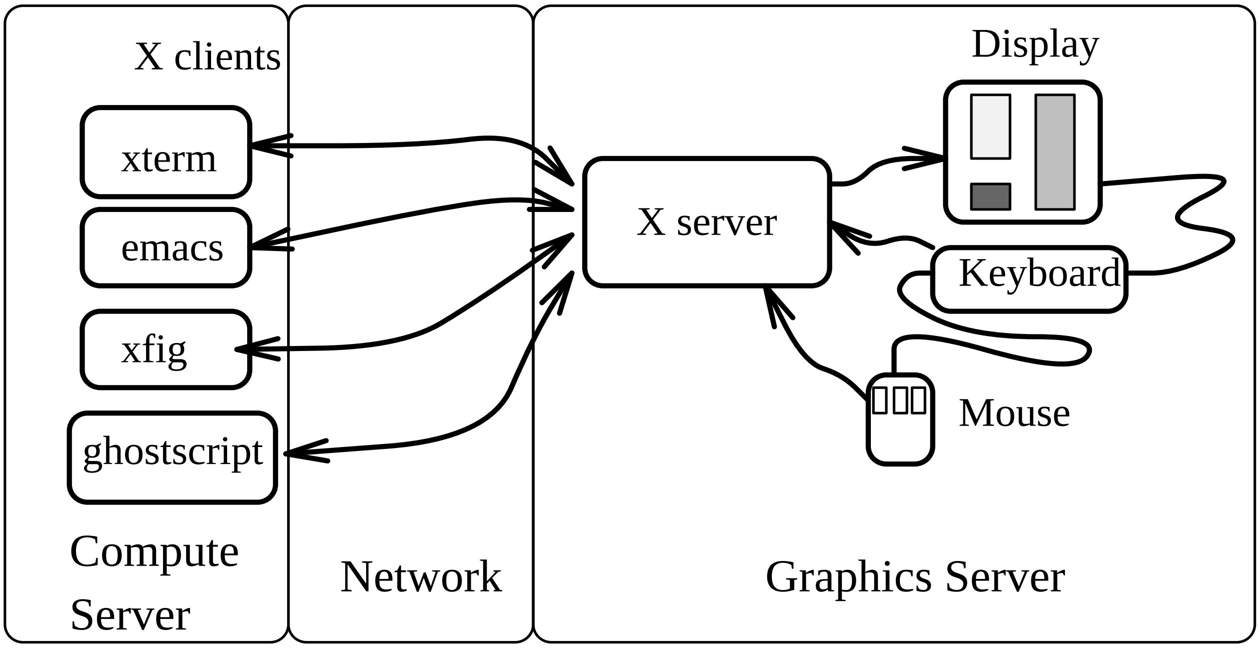


Figure 3:

the display according to client requests.

There are 172 different types of X messages, making X a fairly complex protocol. Messages from the client to the server are called *requests*. There are 120 *requests* including CreateWindow, PutImage, GetImage (generates a reply), Polyline, and ImageText8. Messages from the server to the client fall into three classes: *errors* (17), *events* (35) and *replies* (counted with reply generating requests). Some important events are: ButtonPress, MotionNotify (mouse movement), and Expose (window needs redrawing).

X clients communicate with the X server using a full duplex reliable byte stream protocol, usually TCP/IP and Berkeley sockets. Each client establishes a single connection which is used for all of the client's windows. Communication in X is asynchronous; that is, the client is not suspended while waiting for notification of request completion. The designers of the X protocol went to great lengths to remove unnecessary round trip delays for performance reasons [17]. Even with commands that do require replies, it is possible to send several requests, and then receive several replies, with only one round trip delay. This requires a little bookkeeping on the part of the client, and on the part of any filter that attempts to match requests to replies as this one does.

2.3 Goals for Gathering Data

I wanted to gather traces of network traffic, to actually capture an entire protocol stream. Once I had captured the stream, I had to save it for later analysis. For the sake of accuracy, the logging process had to interfere as little as possible with the content and timing of the data processed.

A unique aspect of tracing a window system running an interactive application is that there is a human in the loop. This has ramifications for the instrumentation: unlike most computer programs, humans behave in a time dependent way. If the tracing system slows down the system too much, the user will act differently. Visual feedback delays can adversely affect hand to eye coordination just as audio feedback delays can make it difficult to talk. For instance, handwriting performance falls off dramatically with visual feedback delays of 150ms [24]. If the instrumentation adds a latency which is a large fraction of 100ms, then many interactive operations (such as keystroke echo) which previously appeared instantaneous will take a noticeable amount of time [7]. This would distract the user, affecting typing performance and the content of the trace. The effect of excessive latency would be even more severe for users using drawing programs where mouse tracking has to appear instantaneous. Throughput is also important because reducing throughput can result in increased perceived latency, adversely affecting the performance of the subject.

Another goal was to avoid changing the X server and the X clients. Changing the clients would be a big problem because I wanted to get traces of users in their day to day environment. Some people run dozens of X clients on many machines, including machines on which only they have privileges. It would have been hard to install custom versions of all of the X clients on all of those machines. It would be easier to change the X server, but then the subjects would have had to use a special machine for a few days, since I didn't have source for the X server in all the X terminals and computers which the subjects were using. To avoid modifying the X

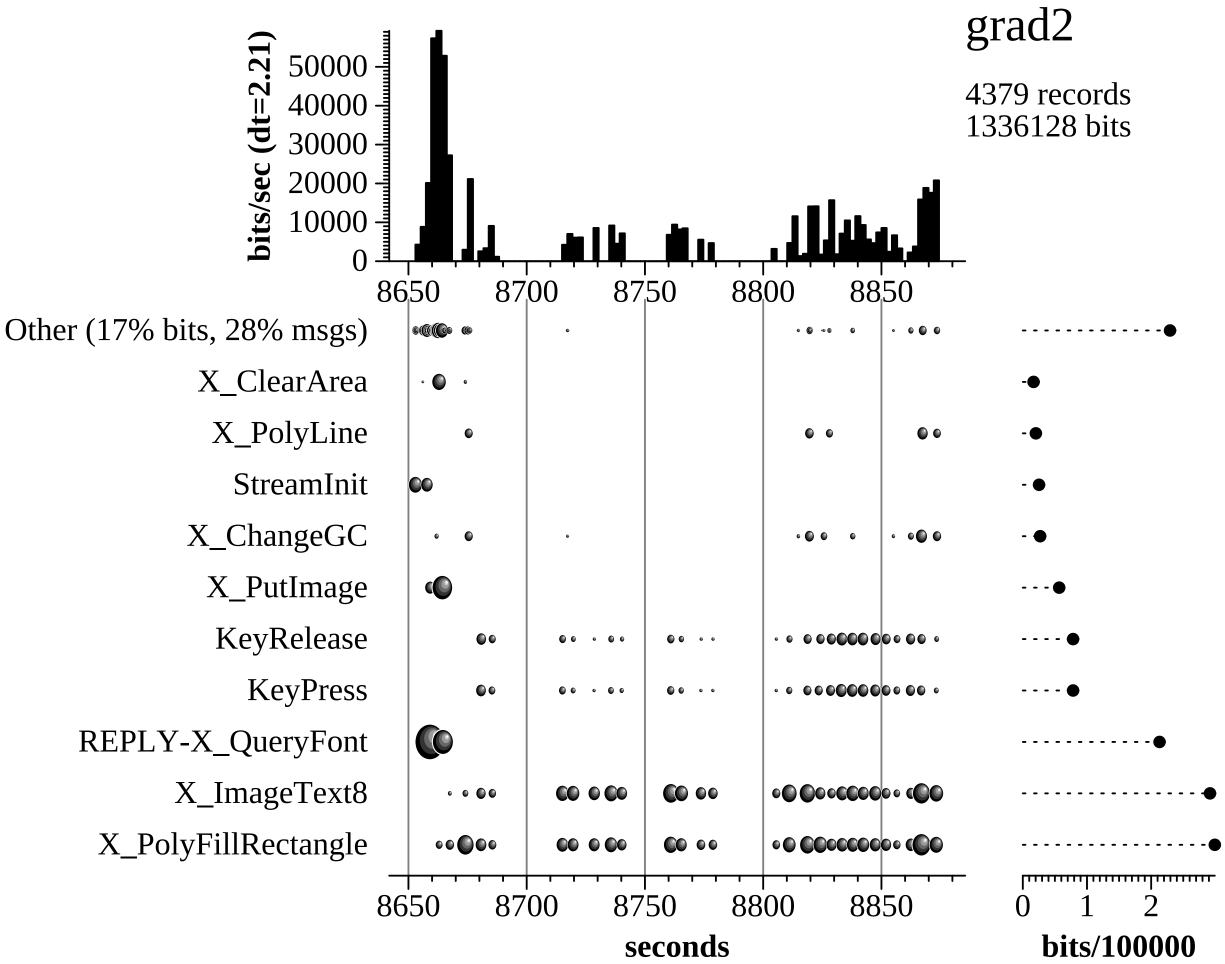


Figure 4: This graph is a view of almost 4 minutes excerpted from the grad2 trace. The X axis along the bottom of the central area in each graph is seconds since session startup. The Y axis in the main area is marked off by message type. The volume of spheres in the data area is proportional to the size of the messages which they represent. Spheres which are vertically stretched represent multiple nearly simultaneous messages, other spheres represent single messages. The dash-dot graph to the right shows the total number of bits for each type of packet. The bar chart at the top shows the number of bits per second as a function of time. The integration interval is 2.21 seconds, too long to check on requirements for interactive use, but short enough to see expensive events like application startups.

At the left we see an application starting up. It creates windows, allocates colors, and queries fonts. The instantaneous network load is comparatively high. After about a minute, we settle down to simple typing (keypress, polyfill rectangle, image text, keyrelease, etc.).

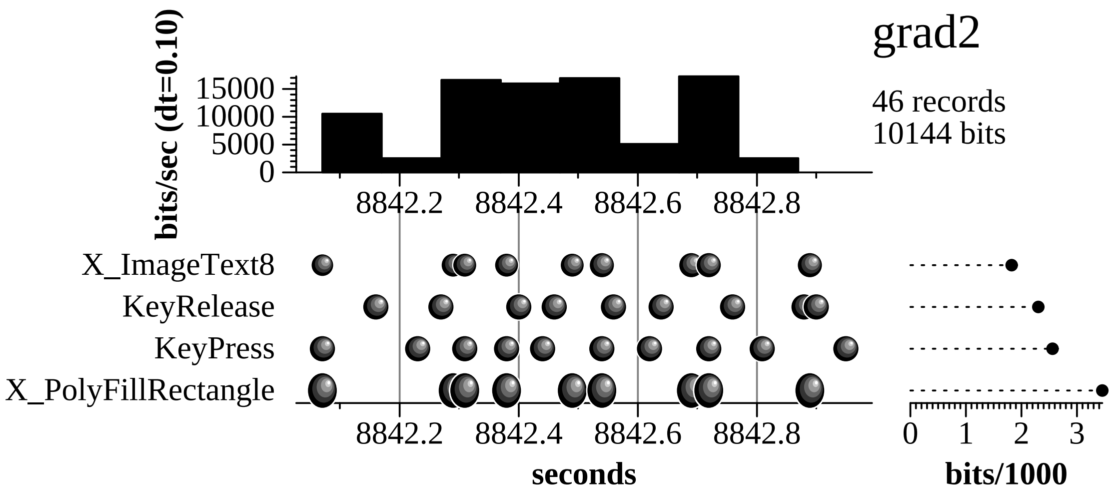


Figure 5: This graph shows a one second excerpt from the grad2 trace, essentially a detail from Figure 4. For an explanation of the graph, please see the legend for Figure 4.
The application here is a terminal emulator, and the user is typing at roughly 10 characters per second, which is extremely fast. A key echo loop consists of a KeyPress event from the server, which the application responds to with X_PolyFillRectangles to update the cursor position and clear a space for the new character, and X_ImageText8 to draw the new character. Finally, the user releases the key and the server sends a KeyRelease event. The delay between a KeyPress event and its graphic feedback varies from just a few milliseconds to about 100ms, which would be a just detectable delay.

server and inconveniencing the subjects, I decided to extract the data directly from the communication stream between client and server.

The operating system and file system communities have a lot of experience gathering and analyzing traces, so I decided to look there for ideas. One non-intrusive way to capture a trace would be to eavesdrop on the Ethernet capturing all relevant packets and reassembling the protocol stream from the captured data. As long as the data could be logged locally (as opposed to sending it over the network), tracing would have no practical effect on the measured stream. Blaze [6] implemented an eavesdropping system like this for NFS, which uses the UDP datagram protocol [41] for transport. I chose not to implement a passive system because I had a requirement of zero packet loss: X is a byte-stream protocol. Losing a single packet could cause us to lose track of X message boundaries. As it turned out, with the hardware configuration I used to gather most of the traces, a passive logging system would not have been able to keep up with peak loads. Passive logging was therefore not a viable alternative for us.

An approach which would have helped control the size of the traces and the impact of the instrumentation was taken by Ousterhout et al [33] and Baker et al [4] who gathered abstracted traces of the UNIX filesystem by recording just open, close, and file pointer repositioning operations, omitting reads and writes. Abstracting their traces allowed much longer traces and reduced the impact of instrumentation tremendously. I could not abstract the traces because I was interested in measuring the performance of various compression algorithms on these traces, and any abstraction would have made this impossible. This points out a general problem with abstracting traces. Traces are difficult to gather, and valuable tool once gathered. Abstracting traces to make them appropriate for one measurement task could be shortsighted in that you might remove information that someone else needs later.

When gathering complete X traces, privacy is an issue. The X protocol stream includes all of the characters typed by the user and all the characters seen by the user, including passwords, love letters, grades and so on. Eliminating this data from X traces would make it possible to redistribute them and would eliminate the risk of inadvertent or malicious disclosure of confidential information. For some applications

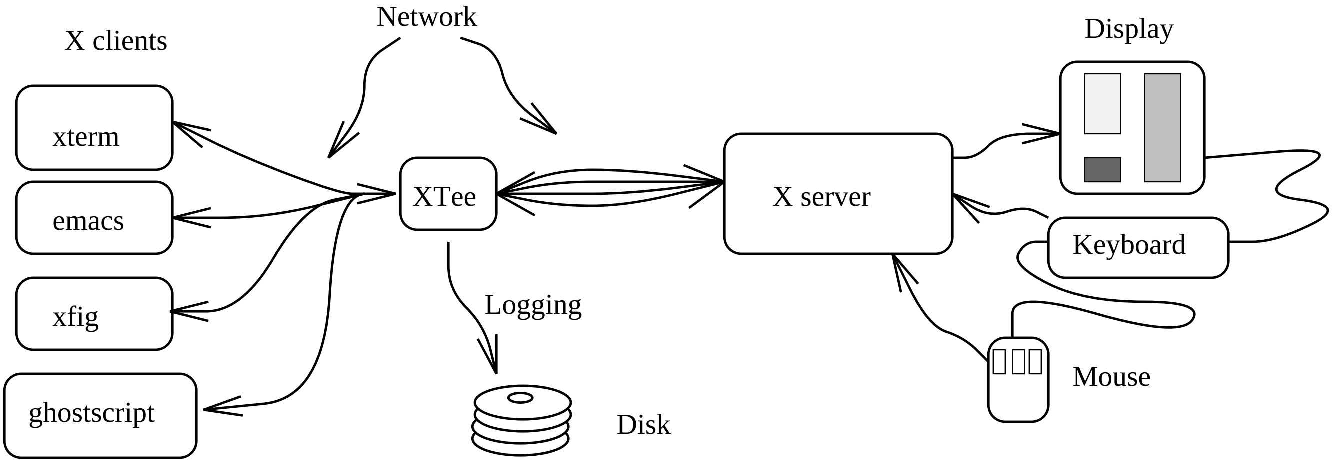


Figure 6: X clients attach to X server through logging process *XTee*

it might be possible to replace all of the text with random characters which would provide a much greater degree of privacy. Unfortunately, this technique would perturb the data unacceptably for my ultimate application: network compression. The subjects were warned that their sessions were not strictly private, but that I would do my best to keep their contents secure.

The tracing technique which I adopted was derived from work by Droms and Dyksen [14]. They realized that if X clients and the X server converse using Berkeley sockets and TCP/IP, it would be easy to interpose a program between client and server which forwarded traffic in both directions while keeping a complete log. Complete data integrity is guaranteed at the price of a little performance. Droms and Dyksen used their traces to characterize the load on an Ethernet due to an X session. Another program which uses the interposition technique for snooping on X traffic is *xscope* by Peterson [36]. I used Peterson's table lookup technique for matching X queries with X replies.

2.4 Gathering Data

On UNIX systems X clients determine the address and method for connecting to the X server from an environment variable. The most common type of address is a machine name followed by a colon followed by a display number, for example "foo:0". To the X library, this address means to open a TCP socket to a machine named foo using port ($6000 + displayNumber$). Usually, $displayNumber$ is 0. Most X servers only have one display and hence are not listening on ports 6001, 6002 etc.. It is easy to write a third program *XTee* which waits for new connections on one of these unused ports. When *XTee* detects a new connection A, *XTee* opens a new connection B to the X server. Input on A is forwarded to B, and input on B is forwarded to A. Input from both connections is logged (See Figure 6). To an X client, *XTee* appears to be the X server, while to the X server, *XTee* appears to be the X client.

Internally *XTee* parses each client-to-server or server-to-client stream into separate messages, and then multiplexes the packetized half duplex streams into one half duplex stream going in each direction. Rather than annotate each message in the merged stream with a stream identifier, I pass a "ChangeStream" token down the merged stream whenever the current stream changes. New streams are identified with a "NewStream" token, and streams whose sockets have closed are shut down with "CloseStream" tokens. These three extra tokens provide enough information for demultiplexing and stream management. This multiplexing system is similar to the system used by Xremote which is discussed in Chapter 3. Messages in these two half duplex streams are logged into two files (client-to-server and server-to-client) with their length, a global sequence number, a unique timestamp, and a magic number which was used to increase programmer confidence in correct parsing of the logged data. The three stream management tokens are included in the logged data so that the streams can be isolated in later processing.

I ran *XTee* on a DECpc 450ST (50Mhz 486) running CMU Mach version 2.6, except for one trace (*xbattle1*) where I ran *XTee* on a 100Mhz DEC Alpha running OSF1. The X servers used were NCD X terminals, except for the grad2 and grad4 traces, where the X servers were SGI Personal Irises. Most application programs were run on DecStation 5000 model 240s.

To test the impact on performance on tracing, I ran *xbattle* (a game which continuously updates it's display) in two different scenarios:

- competing for cycles with *XTee* on a 50 Mhz 486 PC (xbattle trace)

Configuration	latency	increase
No Tracing	3.6 ms.	0%
XTee on DEC Alpha	4.8 ms.	33%
XTee on 486	5.3 ms.	47%

Table 1: *XTee* latency
Impact on X round trip latency due to logging with *XTee*

- on the 486 PC, with *XTee* executing on a 100 Mhz DEC Alpha (xbattle1 trace).

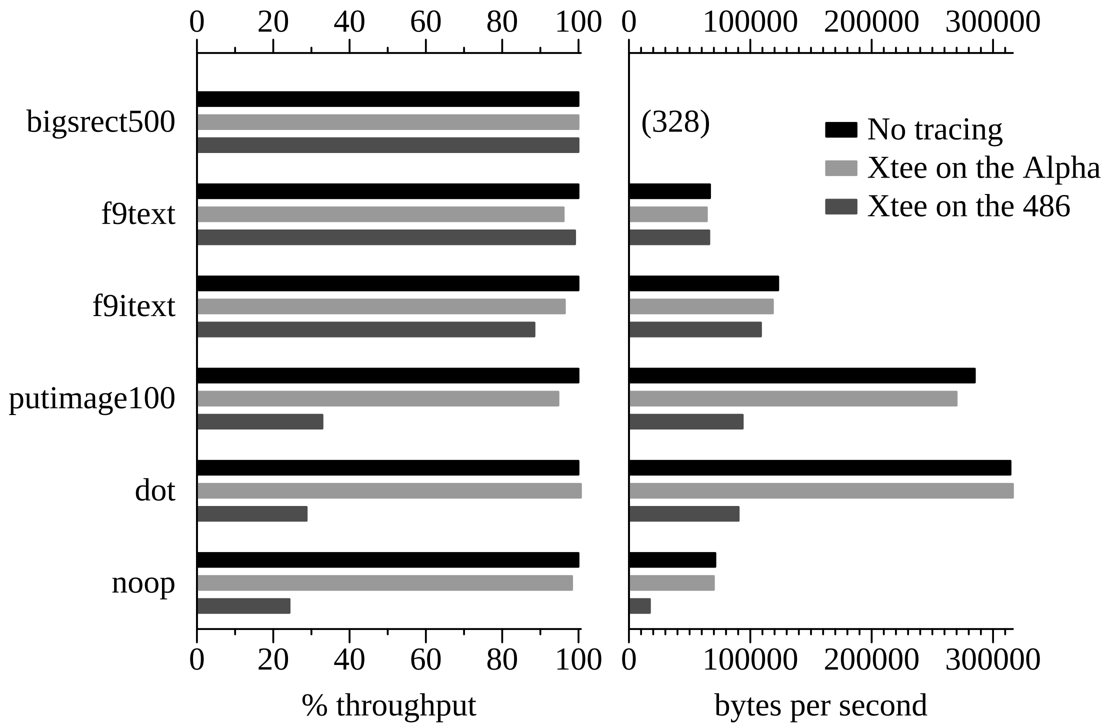
Although the games seemed to play the same in both cases, and I played for the same length of time in both cases, the xbattle1 trace is twice as long, indicating twice the update bandwidth.

I was concerned about my tracing system's effect on performance because poor performance will affect my users and thus my traces. In order to determine the effect of the tracing system on performance, I ran *x11perf* (an X benchmark program) on a DEC DS5000 model 240 with display on an NCD X terminal in three configurations:

1. Connecting directly (over the Ethernet) to the remote X terminal.
2. Connecting through *XTee* running on a 100 Mhz DEC Alpha to the X terminal.
3. Connecting through *XTee* running on a 50Mhz 486PC to the X terminal.

Figure 7 shows the X performance achieved in the three different configurations. Large rectangles show no degradation, even with logging running on the slower 486, which is what we expect because the X server has to update many pixels for each rectangle (250,000), and because rectangles can be concisely specified in X: about 8 bytes each. Large rectangles are server bound, not network bound, even when we degrade the network by introducing a filter.

Examining the figure, we see that the highest network bandwidth achieved is 316K bytes per second (Bps), except that *XTee* actually doubles network traffic, which means that we actually achieved 630KBps over the Ethernet. This implies that we are always server limited in the case where there is no tracing. For these tests, the DEC Alpha can always keep up with the X server. The 486 seems to keep up with the



command	explanation
bigsrect500	500x500 stippled rectangle, 161x145 stipple pattern.
	A stipple pattern is like a silk screen.
	Pixels are only updated where the stipple pattern is set.
f9text	Characters in 60-char line (9x15). Background is untouched.
f9itext	Characters in 60-char line (9x15). Background is cleared
	before character is drawn.
putimage100	PutImage 100x100 square.
dot	Dot.
noop	X protocol NoOperation.

Figure 7: Impact on X throughput due to logging with *XTee*

The bar graph shows X performance as measured by the *x11perf* benchmark program for three test configurations: without tracing, with the trace process running on a DEC Alpha, and with the trace process running on a 486 PC. I show the relative throughputs on the left (normalized so that untraced throughput is always 100), and the number of bytes per second achieved on the right.

The bottom table is a legend explaining the command names. The names are the same as the *x11perf* commands used to generate the data.

X server until the data rate reaches 90-110KBps, which is as fast as the 486 can go. The exception to this rule is the *noop* test, where the 486 only reaches about 17KBps. I believe that this is because *XTee* has some per message overhead, and *noop*s come singly. All of the other primitives tested come in large messages (*PolyRectangle, PolyText*, etc.) with many primitives per message, allowing amortization of overhead over many bytes. X does not support a *PolyNoop* command.

When X11 was designed in 1986, UNIX workstations connected via Ethernet typically achieved data rates of from 62.5 K bytes per second to 125 K bytes per second, and peak character display rates were from 5K to 30K characters per second [48]. The slowest (and major) tracing configuration has lower bandwidth than the untraced setup, but would possibly have been one of the fastest machines available in 1986. It is important to point out, however, that our traces show that most applications spend very little time sending data at rates approaching 100KB/s. In most cases, round trip latency is a more important measure of interactive performance.

Table 1 shows the X round-trip latency in the three configurations (as measured by the *x11perf* benchmark program). In both cases, introducing the filter significantly increases the latency, but the latency never quite doubles as I expected. I conjecture that this means that the 486 and the Alpha handle interrupts more quickly than the X terminal, although I have not investigated further. Note that the new latencies are well below the 100 millisecond human threshold and that it would take many round trips in either configuration before a human could see a difference attributable purely to latency (most user-X interactions are handled in a single round trip).

The X round-trip latency was measured in 1986 in [48]. There the X round trip latency was reported as 15 to 30 milliseconds. We now measure X round trip latency for a traced connection at just over 5 milliseconds. My current system with tracing installed is substantially faster than the systems available to the X designers in 1986.

2.5 Traces

I took a number of traces of applications that I thought would be representative: two 2D drawing programs, a POSTSCRIPT previewing program, an X benchmark, some games, and five graduate students doing their daily work. These traces are summarized in Table 2

I chose 2D drawing programs because they are based more on geometry than text and I wanted to characterize geometry based applications. I used two programs hoping that if their traces were very different, I might learn something. I was the user in both cases, drawing relatively simple pictures consisting of lines and polygons, and then recursively duplicating these pictures several times to build up a large amount of geometry.

The POSTSCRIPT previewer was chosen as a locally popular imaging application. I traced the previewer generating a page of text, which it does by sending character sized images rather than by using the text commands.

An obvious trace for a trace suite is a benchmarking program, if only to demonstrate an atypical load. The benchmark I used, *x11perf*, is included in the X distribution. While *x11perf* can provide a comprehensive performance report after running for hours, I didn't have enough disk space to log a comprehensive trace, so I generated a relatively short trace, only checking the performance of dots.

A few years ago, it might have been considered frivolous to include game traces in a suite like this, but very large companies are now seriously investigating the possibility of charging people to play games over the telephone and cable networks [2]. Probably X will not be the major graphics protocol for graphics games, but X traces may still provide a useful characterization of the traffic. Xtetris animates a simple arrangement of falling rectangles while listening for keyboard and mouse button events. Xbattle animates a map with armies on it, spending most of its bandwidth redrawing the map grid with lines, and army pieces with rectangles.

Finally, I included traces of graduate students working. I gathered real life traces because they seemed to be the most representative of an actual network load. I used graduate students because they were available and because they generate most of the network traffic around here. It turns out that graduate students spend most of their X bandwidth on text. This is why I did not specifically include a text trace.

There are a number of important categories of traces which I left out: 3D, electrical computer aided design (ECAD), vlsi design, and color imaging, to name a few. 3D

Trace	Term	Size	Duration	Description
idraw	b&w	4.5 Mb	11 min	A 2D drawing program, making a very large model by duplicating a small one repeatedly.
xfig	b&w	5.6 Mb	17 min	A 2D drawing program, making a very large model by duplicating a small one repeatedly.
ghostscript	b&w	30 Kb	10 sec	A POSTSCRIPT previewer, previewing a single page
x11perf	b&w	1.9 Mb	33 sec	An X benchmark program configured to test point drawing performance.
xbattle	b&w	7 Mb	14 min	A multiplayer interactive wargame.
xbattle1	b&w	15 Mb	14 min	The same game with logging executed on a DEC Alpha at 100 Mhz instead of on a 50 Mhz 486
xtetris	b&w	350 Kb	11 min	Online 2D bin-packing (a game)
grad1	b&w	13 Mb	6 hrs	A graduate student, preparing a paper and previewing graphs.
grad2	color	14 Mb	45 hrs	A fast typing graduate student.
grad3	b&w	2.4 Mb	54 min	A graduate student doing software development.
grad4	color	21 Mb	11 hrs	A graduate student, making significant use of a dvi file previewer
grad5	b&w	10 Mb	110 hrs	A graduate student who reads mail and writes papers.

Table 2:

graphics isn't directly supported by X, but it is supported by the PEX extension to X. ECAD is an important native X application. Color imaging is another important X application which is partly supported by the core X protocol, and enhanced by imaging extensions. I expect that ECAD applications are similar to 2D drawing programs in their use of X, but that 3D and color imaging applications are completely different, particularly since they use extensions to the core X protocol.

2.6 X Protocol Activity

In the following sections I will talk about categories of X messages; however, before I can talk about these categories, I need to develop a classification scheme or taxonomy defining them. The X message taxonomy was built as a two level hierarchy, using function as the criteria for making distinctions between messages. The complete taxonomy is reproduced in Appendix A. The 5 top level categories of X messages are summarized below:

1. Geometry includes line, polygon, arc and point drawing primitives.
2. Image includes cursor manipulation and image manipulation primitives (X_PutImage X_GetImage and X_CopyArea, for instance).
3. Input includes events generated directly by the user, such as KeyPress, as well as primitives often used in the inner loop of mouse tracking operations such as X_QueryPointer and X_TranslateCoords.
4. Misc includes access control, the bell, errors, X extensions, and the X interprocess communication facilities such as cut and paste.
5. Text includes font primitives and drawing primitives. Note that when text is actually drawn, most applications use a rectangle to clear the area behind the new character, so that text intensive applications will usually see about 50% text and 50% geometry.
6. Window includes colormaps, input focus manipulation, keyboard maps, graphics state changes, X's database facility, and window movement and resizing.

Analysis of the traces lead to preliminary results in four areas:

- The relative importance of X messages.
- How X usage changes during application stages.
- Short sequences of messages representing higher-level application idioms.
- Size distribution of X messages and groups (trains) of X messages.

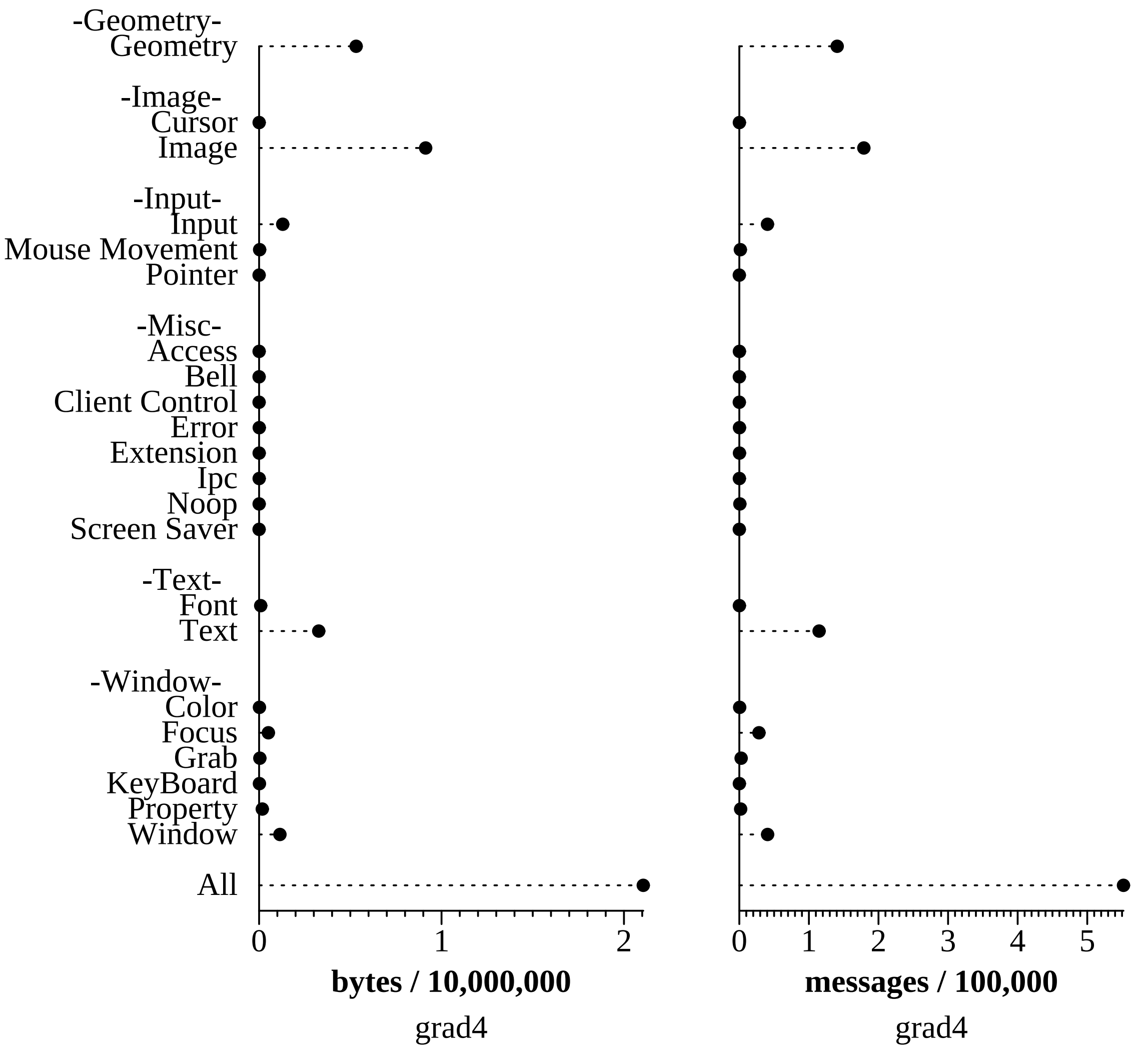
2.6.1 Relative Importance of X Messages

I began with two questions: which individual messages are important, and which groups of messages are important. I defined importance in two ways: message frequency, and total bytes for each group. Since message frequency is not directly linked to resource consumption, total bytes is a more general measure. Analogously, in a complex computer instruction set, some string instructions may occur infrequently, and yet account for a large percentage of total cycles. I included graphs of both measures for most of the studies.

In Figure 8, taken from the grad4 trace, we see that image operations use more bytes than any other category. This is because the subject was using a dvi file previewer (to view TEX output), and X's support for high quality text is such that document previewers usually use the X image operators rather than X's text primitives. Geometry is the next most important category in this trace, although it is likely that about half of the geometry was rectangles clearing letter spaces for text. We also see activity in the window category, which encompasses window manipulation, graphics state changes, input, and focus. Focus events are usually seen when the mouse enters or leaves a window. None of the other 17 categories see any appreciable use.

Comparing bytes per message type with the number of messages of each type in Figure 8, we see that in this trace, image operations were larger on average than the other packet types, since image operations are weighted more heavily in the byte counting graph than in the message counting graph. We can compute from the figures that the average message was about 38 bytes, while the average image operation was about 53 bytes (size distributions can be derived from Figure 11, discussed below). These image operations are actually surprisingly small until we realize that many document previewers (*ghostscript* for example) operate in monochrome and send images to X a single glyph at a time, sending only the rectangular image enclosing each character.

Comparing the detailed breakdown of message types in Figure 8 to the high level breakdown also available in the same graph, little information is lost because most of the categories have so little traffic, and active low level categories are always grouped



Detailed Application Signatures (bytes and messages), trace=grad4

Figure 8: Messages are categorized according to Appendix A.

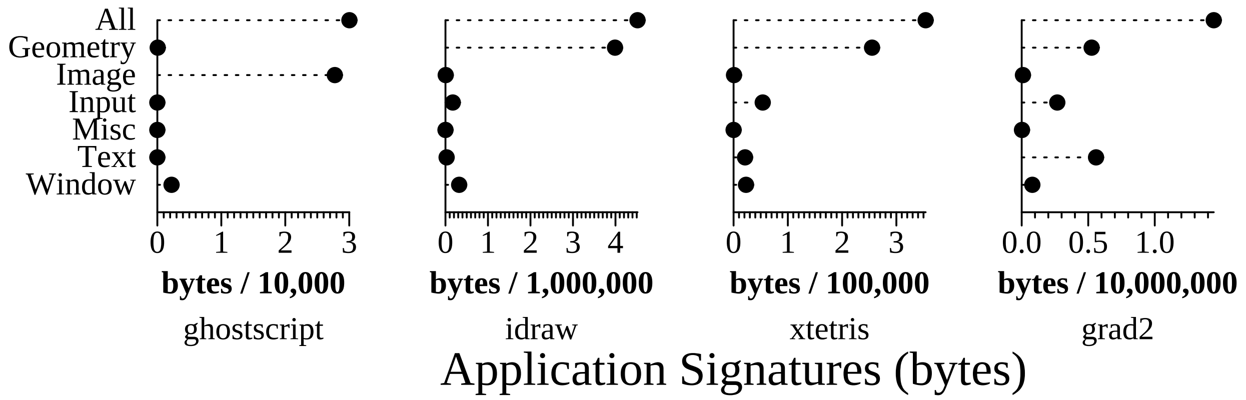


Figure 9: Messages are categorized according to the high level categories in Appendix A. Magnitudes correspond to total bytes in each category.
Note that graphs in this series may have different horizontal scales

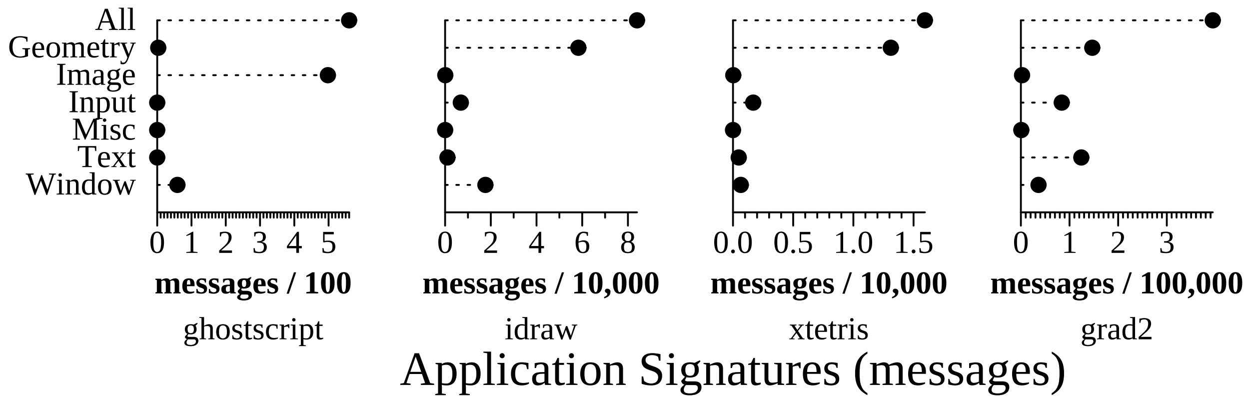


Figure 10: Messages are categorized according to the high level categories in Appendix A. Magnitudes correspond to total message count in each category.
Note that graphs in this series may have different horizontal scales

trace		N				
		4	8	16	32	64
ghostscript	messages	94%	97%	99%	100%	100%
	bytes	96%	98%	99%	100%	100%
idraw	messages	65%	79%	94%	99%	100%
	bytes	86%	92%	97%	99%	100%
xtetris	messages	91%	98%	99%	99%	100%
	bytes	85%	96%	98%	99%	100%
grad2	messages	82%	91%	94%	98%	99%
	bytes	86%	91%	95%	98%	99%

Table 3: Percentage of messages or bytes accounted for by the top N message types in each trace. Messages are sorted according to message popularity for the "messages" row, and aggregate bytes for the "bytes" row.

in different high level categories. This is part of a pattern of X command use: many commands are used, but only a few are used frequently. Table 3 quantifies this effect for several of the traces.

2.6.2 Application Stages

Application profiles lose time dependent data, masking temporal changes in running applications. If optimization is applied only according to the static profile, then total run-time may be decreased, but the interactive user may not perceive a positive effect. Frequent operations which already appeared instantaneous may have been improved, leaving irritatingly slow but infrequent operations untouched.

All of the application signatures in Figure 9 have one thing in common; most of the data is spent on directly useful operations: geometry, text, or imaging, rather than IPC, window manipulations, database management or other overhead. This is encouraging, but it doesn't show the whole picture. Figure 4 shows the first four minutes or so in the life of an application (xterm). Figure 5 graph shows a detail from this same trace about 1 second. In Figure 5, the user is typing, and only 4 primitives are in use. Anyone wishing to optimize this steady state mode of xterm would know exactly where to start working.

A nice feature of this type of graph is that you can isolate an interaction visually, add up the bits involved, and find the bit rate necessary to support the interaction at the desired speed. Usually, interactions should complete in less than 100ms. Longer delays will be perceptible and distracting.

Looking back to Figure 4, almost 20 seconds are spent in application startup. In this mode, otherwise infrequently used commands are used exclusively, and network use is much higher than during normal operation. Perceptually, this startup time is very important. A long startup time can contribute to a perception of sluggishness, even if the startup time is a very small percentage of the total time spent with the application. This sort of problem highlights a fundamental difference between interactive programs and batch mode programs like benchmarks. When optimizing batch mode programs, the object of optimization is to minimize total run time. When optimizing interactive programs, the goal should be to minimize perceived run time: operations should be weighted by their importance to the user before optimization. Infrequent events such as application startup can be very important to the user; this tends to spread out the area of optimization and makes substantial improvements in perceived performance more difficult to achieve.

Graphs like those of Figure 4 and Figure 5 can be very useful for analyzing interactive applications for optimization. Instead of summarizing the whole run-time, an interval of interest is displayed. Once a single interaction is isolated, the interaction can be optimized as if it were a batch mode program, with this graph supplying much of the necessary information.

2.6.3 Application Idioms

Figure 9 shows differences among the ways applications and groups of applications use X. The *ghostscript* document previewer uses mostly image operations. The drawing programs *idraw* and *xfig* use mostly geometry. The user session traces were mixed between geometry, imaging, and text.

Something I didn't expect, is that on a microscopic level, X applications do things very differently. For example, I examined three applications echoing keystrokes at the end of a line:

1. **xwsh** (a terminal emulator) clears from the end of the text on the current line to the right edge of the window using X_PolyFillRectangle, draws all of the text for the current line, including the new character using X_ImageText8, and draws a cursor using X_PolyFillRectangle.

2. **xterm** (the standard X terminal emulator) clears the old cursor with X_ImageText8, clears a slightly larger area around the new character with X_PolyFillRectangle, draws the new character with X_ImageText8, and draws the new cursor using X_ImageText8.

3. **emacs** (a native X11 text editor) uses X_ClearArea to remove the old cursor, draws the new character with X_ImageText8, and draws a new cursor with X_ImageText8.

These instruction sequences all produce the same visual result with different methods. X applications have not settled into stable idioms for accomplishing identical trivial tasks. Anyone wishing to optimize some trivial task has to find all of the applications of interest, see how they accomplish the task, and then optimize for each one. Graphics architects are going to have a hard time really optimizing X or developing really good protocol transformations until toolkits develop to the point that most programs use them and idiom becomes more consistent.

2.6.4 Size Distribution of X Messages and Trains of Messages

I measured the size distribution of X messages. Results of these measurements are graphed in Figure 11, which plots the cumulative number of messages against message size for four traces. The result is that although most messages are small, large messages appear occasionally. Client messages in the x11perf trace (not graphed) average almost 1600 bytes (as opposed to less than 100 bytes per message for the traces shown) because *x11perf* is able to send large groups of primitives in each packet. Non-benchmark programs seem to have a harder time generating these large packets.

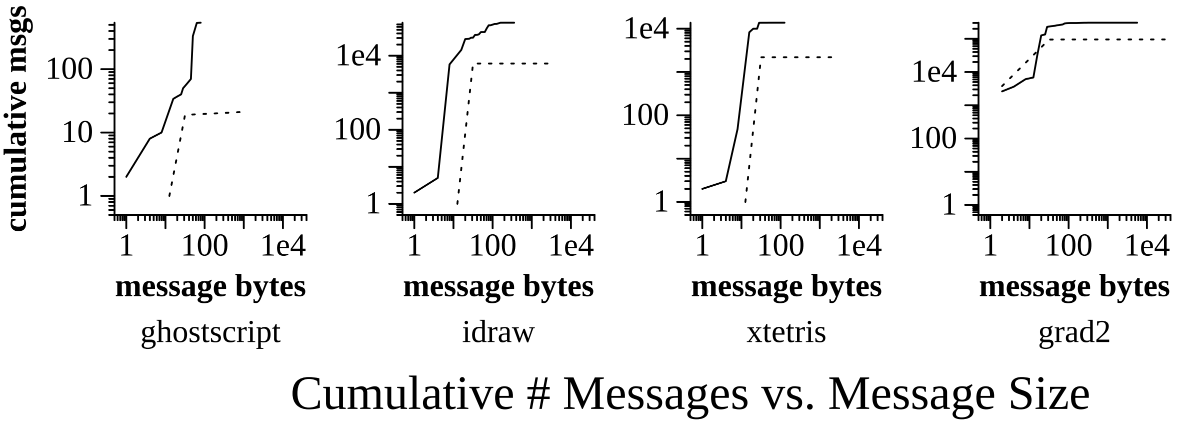


Figure 11: The cumulative number of messages is graphed against message size. The solid line is for client traffic, while the dotted line indicates server traffic. We can see that for these traces, the average message size is always less than 100 bytes. Most server messages are 32 bytes, although the X server occasionally sends large messages, usually replies to database queries or requests for font information. The client sends more messages than the server, and is more likely to send large numbers of large messages.
Note that these graphs have varying vertical scales.

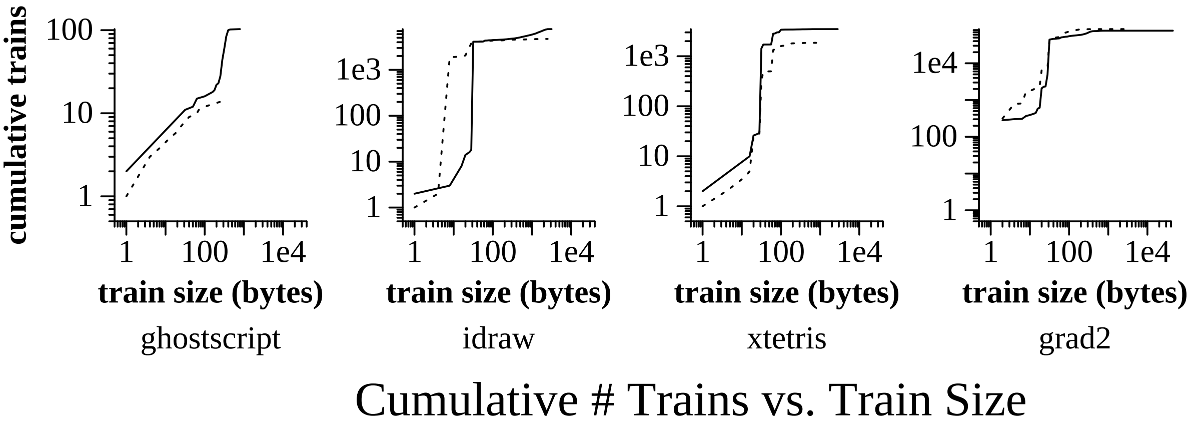


Figure 12: A train is an uninterrupted group of messages arriving from a single source, separated by no more than 1ms. The significance of trains is that some transport layer could collect them and process them as units. In this case, cumulative numbers of trains are graphed against train size. The solid line is for client traffic, while the dotted line indicates server traffic. Clients and servers usually end up with nearly the same total number of trains because client commands are provoked by server events, and server replies are provoked by client queries.
Note that these graphs have varying vertical scales.

I think that a more interesting measure than message size is train size. The intent is for a train to be all of the messages which arrived in a single network packet. For our purposes, a train of X messages is an uninterrupted stream of X messages all traveling in the same direction, all on the same socket with no inter-message gaps of more than a millisecond. An interruption would be a message arriving from a different source, or traveling in a different direction.

This is roughly the unit of conversation between an X application and the X server. Figure 12 shows that clients and servers usually generate similar numbers of trains, even though they have different size distributions. This confirms our expectation of X behavior, that a client usually responds to a server event with a single train. I believe that the client side of the ghostscript trace has more trains because the application was not fast enough to completely update its screen in a single train.

Train information allows us to make some inferences as to good and bad system decisions. We know that the client issues at least one system call per train, and we can estimate TCP/IP [40, 39] header overhead from the train size as was done in [14]. In three out of the four traces shown, most trains are less than 100 bytes long, implying that most TCP packets will also be under 100 bytes. The TCP/IP protocol will add 48 bytes of header information to each packet, implying an overhead of at least 50%. Although this might be tolerable for a fast network, it is a disaster for a slow one. CSLIP/PPP (Compressed Serial line IP [22] layered on the Point to Point Protocol [50]) would be a very efficient network protocol for an X session traversing a slow point to point connection since it only adds about 7 bytes of header information to each packet. A modem that reallocated asymmetric bandwidth after 50 bytes or so would thrash for most interactive use, but would be efficient at application startup when larger transfers dominate, as shown in Figure 4.

2.7 Conclusions

This X tracing methodology has allowed me to gather complete traces of X sessions with a tolerable impact on performance. A new taxonomy of X messages allowed me to present profiles of X traffic, revealing that X applications spend most of their network bandwidth on useful operations like updating the screen. As a function of the total number of bytes transmitted, X applications spend very little on database operations, window manipulation, or IPC.

Time dependent snapshots of X activity allow us to see how an application is progressing through it's various stages, and make it easy to zoom in on areas of interest. In particular, we can see that X database operations are clustered at application startup, contributing to a common perception that X applications are sluggish. These operations are important to user perceptions, even though they are not a large contributor to total bandwidth.

My interest in sending the X protocol over low bandwidth networks led me to to investigate the size distribution of X messages and groups of X messages that could be combined into single network packets (trains). I found that even trains of X messages were usually smaller than 100 bytes, less than a tenth of the maximum Ethernet packet size and only twice the TCP/IP header size. I conclude that TCP/IP is not a very efficient way to transport X traffic.

In this chapter I have tried to apply the careful measurement techniques now common in the computer architecture literature to a graphics protocol. The techniques presented here will apply to other graphics protocols. I expect to see further work in this area, resulting in a quantitative understanding of graphics protocols and the systems necessary to support them.

Chapter 3

Performance of Xremote

Whenever network bandwidth is a limiting factor, there is an opportunity for compression (assuming redundancy in the bitstream). This is the case for X connections using normal telephone service, ISDN, and even sometimes Ethernet. Xremote [11] is a transformation of the X protocol designed (by NCD) to efficiently implement X connections over serial lines. Right now, Xremote seems to be the most commonly used such system. There is a follow-on to Xremote called LBX [16], but it is still in the design stage. In this chapter, we will study the performance of the Xremote protocol with the aim of finding opportunities for further compression.

I implemented and instrumented the compression portion of the Xremote protocol. The instrumentation outputs, for each message, the name of the message, the original arrival time of the message, the number of bytes in the message before compression, and the number of bytes in the message after compression. The implementation allows for different features of the Xremote protocol to be enabled or disabled, so that the effects of the features could be assessed.

I processed all 11 of the traces introduced in the previous chapter through a batch mode version of my Xremote implementation to produce the results presented in this chapter.

Across my trace suite, Xremote's overall compression ratio is 2.4:1. This figure varies widely depending on the trace. A study of compression ratio as a function of message type shows text based messages commonly achieving 3:1 compression, while

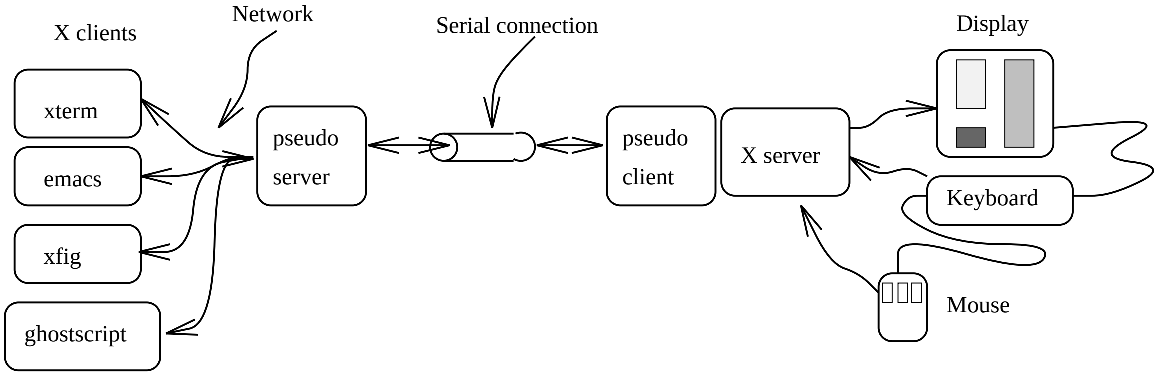


Figure 13: Xremote architecture: X clients connect to pseudo server which multiplexes connections and forwards coded messages to the pseudo client. The pseudo client decodes and demultiplexes messages before forwarding to the X server. Messages from the X server to the X client move in the opposite direction in exactly the same manner. The pseudo server appears to be the X server to the X clients, and the pseudo client appears to be the X clients to the X server.

geometric messages usually achieve only 1.6:1 compression. By examining bandwidth requirements and compression performance as a function of time, we see that Xremote performs adequately for some applications which are text based or which use small geometric datasets, except at application startup, where more bandwidth is required. More aggressive compression is required to adequately support the initialization stage of X applications and medium to large geometric databases. Following chapters will address this issue.

3.1 X and Xremote

The Xremote protocol includes a transport mechanism for compressing and multiplexing multiple X connections onto a single byte-stream, and file access mechanisms for starting up a diskless X terminal. Both of these protocols are layered on a reliable datagram protocol which is also part of Xremote. We will only be discussing the compression and multiplexing mechanisms, and we will only discuss the multiplexing mechanism briefly.

On Unix machines, X clients use an environment variable to determine the network address of the X server to connect to. In order to use Xremote, this environment variable should be set to the address of an Xremote pseudo server instead. Clients connect to the pseudo server as if it were the real X server. Figure 13 shows how clients connect to the pseudo server, the pseudo server connects to the pseudo client, and the pseudo client connects to the X server. The pseudo server multiplexes messages on multiple connections by using three additional messages:

1. NEWSTREAM(id)
2. CLOSESTREAM(id)
3. CHANGESTREAM(id)

These three extra messages are sufficient to manage the multiplexing and demultiplexing tasks. I reused this scheme in the trace gathering code discussed in Chapter 2, and in the new compression system discussed in Chapter 5.

Figure 14 shows an overview of Xremote's compression component. The delta encoder remembers the last 16 messages of length less than or equal to 64 bytes. Most X messages are in this category (see Chapter 2). Incoming messages are compared to cached messages of the same length. If there are fewer than 8 differing bytes, then a delta message is output instead of the original message, as long as the difference message is not longer than the original message. The delta message consists of a one byte opcode, 4 bits selecting one of the 16 cached messages, a 4 bit count (ranging from 0-8) and a list of two byte update fields, where the first 8 bit update field is the offset of a differing byte, and the second 8 bit update field is a new value for the differing byte.

As an example of the delta compressor in use, consider the example delta compression of a mouse motion event in Table 4. In the expected case, two consecutive mouse motion events will have 5 different bytes, leading to a 12 byte delta packet and 2.6:1 compression.

Table 5 has an example of an LZW[1] coder in action, together with an explanation

[1] The compression community calls this coder LZ78, referring to Ziv and Lempel who developed the basic algorithm [54], and dropping the reference to Welsh, who popularized it. For consistency I will continue to use the LZW terminology used in the Xremote documentation.

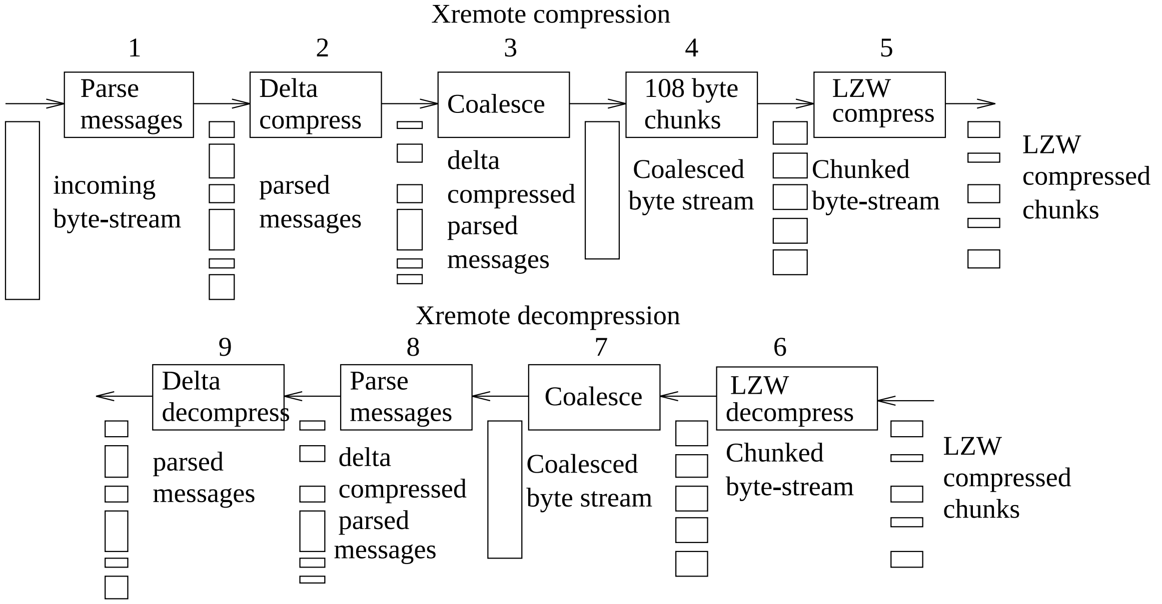


Figure 14: Xremote compression architecture: bytes available from the network are parsed into X messages (1) and delta compressed (2). Output from the delta compressor is gathered together (coalescing) (3), and broken into chunks no larger than 108 bytes (chunking) (4). These chunks are run though an LZW compressor (5) and sent out across a serial line. Compressed chunks incoming from the serial line are run through the LZW decompressor (6), gathered together into a byte stream (7), parsed into X messages (8), run through the delta decompressor (9), and sent out to the network.
This figure is simplified by the omission of stages irrelevant to compression.

Field	bytes	likely difference	delta output
			2 bytes (opcode and mask)
opcode	1	none	
detail	1	none	
seq no	2	none	
timestamp	4	low order byte	2 bytes
root window	4	none	
event window	4	none	
child window	4	none	
root x	2	low order byte	2 bytes
root y	2	low order byte	2 bytes
event x	2	low order byte	2 bytes
event y	2	low order byte	2 bytes
key state	2	none	
same screen	1	none	
padding	1	none	
Total bytes	32		12 bytes

Table 4: Here is an example of how consecutive motion notify events (mouse motion messages) might be coded by the delta coder. The delta packet consists of an opcode, a mask, and up to eight update fields. The opcode is one byte long. 4 bits in the mask indicate which of the 16 cached packets this one is most similar to. The other 4 bits in the mask indicate how many bytes differ between the two packets. Each update field is 2 bytes long. The first 8 bit field is the offset of a changed byte. The second 8 bit field is a new value for the changed field. In this example, 5 differing bytes result in a 12 byte delta packet, replacing the original 32 byte mouse motion event, for a 2.6:1 compression ratio.

Dictionary	prefix	input	output	bits
[a-z,_]	[]	p	-	
[a-z,_]	[p]	a	#[p]	5
[a-z,_,pa]	[a]	t	#[a]	5
[a-z,_,pa,at]	[t]	_	#[t]	5
[a-z,_,pa,at,t_]	[_]	b	#[_]	5
[a-z,_,pa,at,t_,_b]	[b]	a	#[b]	6
[a-z,_,pa,at,t_,_b,ba]	[a]	t	-	
[a-z,_,pa,at,t_,_b,ba]	[at]	_	#[at]	6
[a-z,_,pa,at,t_,_b,ba,at_]	[_]	h	#[_]	6
[a-z,_,pa,at,t_,_b,ba,at_,_h]	[h]	a	#[h]	6
[a-z,_,pa,at,t_,_b,ba,at_,_h,ha]	[a]	t	-	
[a-z,_,pa,at,t_,_b,ba,at_,_h,ha]	[at]	_	-	
[a-z,_,pa,at,t_,_b,ba,at_,_h,ha]	[at_]	s	#[at_]	6
[a-z,_,pa,at,t_,_b,ba,at_,_h,ha,at_s]	[s]	a	#[s]	6
[a-z,_,pa,at,t_,_b,ba,at_,_h,ha,at_s,sa]	[a]	t	-	
[a-z,_,pa,at,t_,_b,ba,at_,_h,ha,at_s,sa]	[at]	EOF	#[at]	6

Table 5: LZW example: the initial dictionary contains the lower case alphabet and space (_). The input string is "pat bat hat sat." There is a prefix string which is initially empty. Whenever a new character is read, if the prefix string catenated with the new character is in the dictionary, the new character is appended to the prefix string. Otherwise, the index of the prefix string in the dictionary is output, the catenation of the prefix string and the new character is appended to the end of the dictionary, and the prefix string is set to the new character. In this example, #[a] is 0, and #[at] is 29. The number of bits occupied by output symbols corresponds to the number of bits necessary to represent the index of the last string in the dictionary.

of the algorithm. The LZW coder in Xremote outputs varying size indices or pointers, as in the example. Its initial, or root dictionary contains the 256 possible byte values, and a CLEAR symbol at index 257. Whenever an entry is about to be made in the dictionary which would require a 13 bit index, CLEAR is sent, and the encoder and decoder both reset their dictionaries. This scheme is very similar to the LZC scheme described in [52]. Additionally, whenever the LZW coder outputs a chunk of more than 120 bytes (input was broken into chunked of 108 bytes or less) the uncompressed chunk is sent instead and both the encoder and the decoder's dictionaries are cleared.

3.2 Xremote Simulator

My simulator duplicates the Xremote protocol down to but not including Xremote's physical transport layer. The simulator can run on traces or on a live X session. If coalescing and chunking are enabled (stages 3 and 4 in Figure 14), then message boundaries are lost after LZW compression; we cannot see how effective LZW compression is for each message type. My simulator optionally disables these stages so that we can measure LZW compression as a function of message type. Later, we will show that exercising this option has little effect on overall compression.

3.3 Results

Chunking and Coalescing: I wanted to study compression as a function of message type, but with Xremote's message coalescing enabled, this is impossible, because adjacent messages are scrambled together by the LZW coder. I was curious to see how much effect chunking and coalescing had on compression. I expected that coalescing alone might help, by providing longer input strings for the LZW coder. I expected that chunking alone would hurt, by shortening the coder's input strings. Finally, I expected that chunking and coalescing together would hurt because the LZW coder would no longer start off aligned with the beginning of each message. Figure 15 shows the result of the experiment with chunking and coalescing. Chunking and coalescing almost never have a significant impact on compression, probably because most messages, and even most groups of messages eligible for coalescing are smaller than the chunking threshold of 108 bytes (see previous chapter). This negative result allowed me to proceed with the rest of my experiments.

Compression by Xremote component: Figure 16 shows how the components of Xremote performed separately and together across each trace in the trace suite. Figure 17 shows, for four selected traces, how the components of Xremote performed on each of the six message categories introduced in Section 2.6. Below, I discuss the performance of the two components, separately and together.

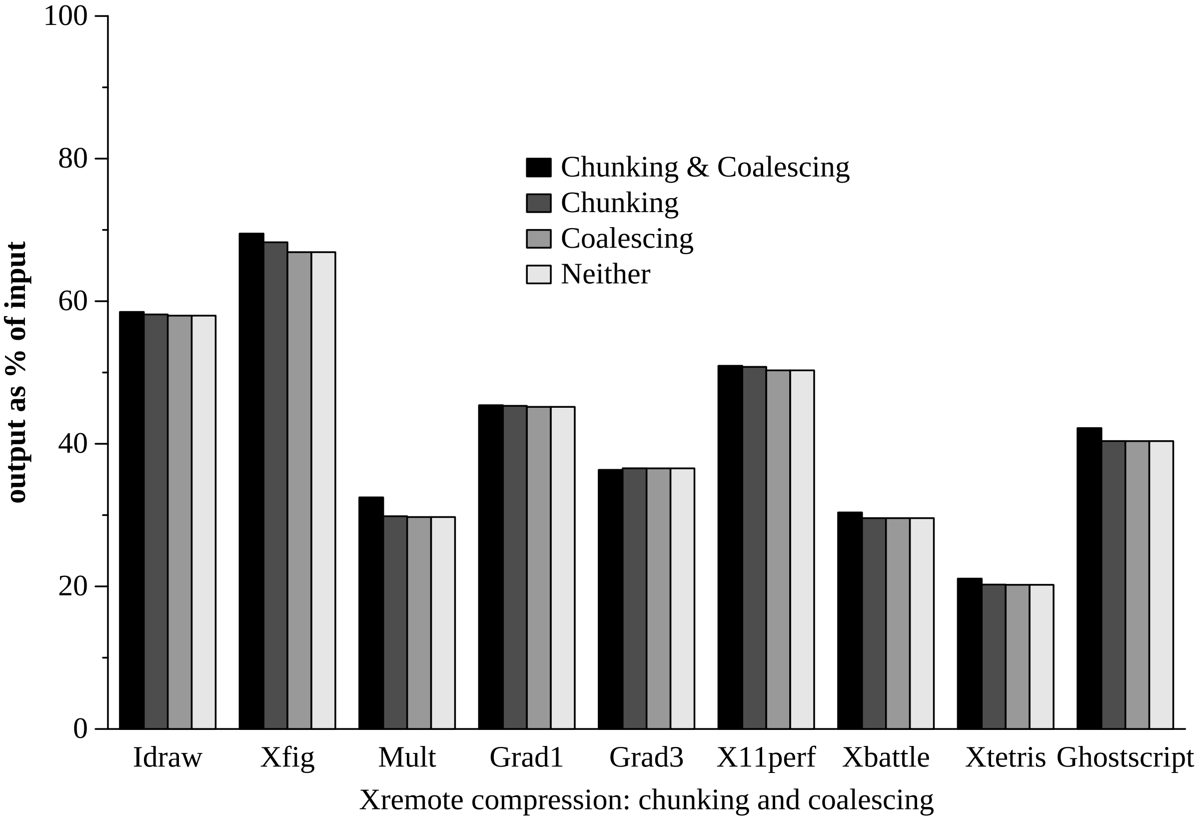


Figure 15: This bar-chart illustrates both the range of overall compression one is likely to get from Xremote, and that the difference in compression performance resulting from coalescing messages after delta compaction, or from breaking messages into maximum sized chunks, or both is not significant.

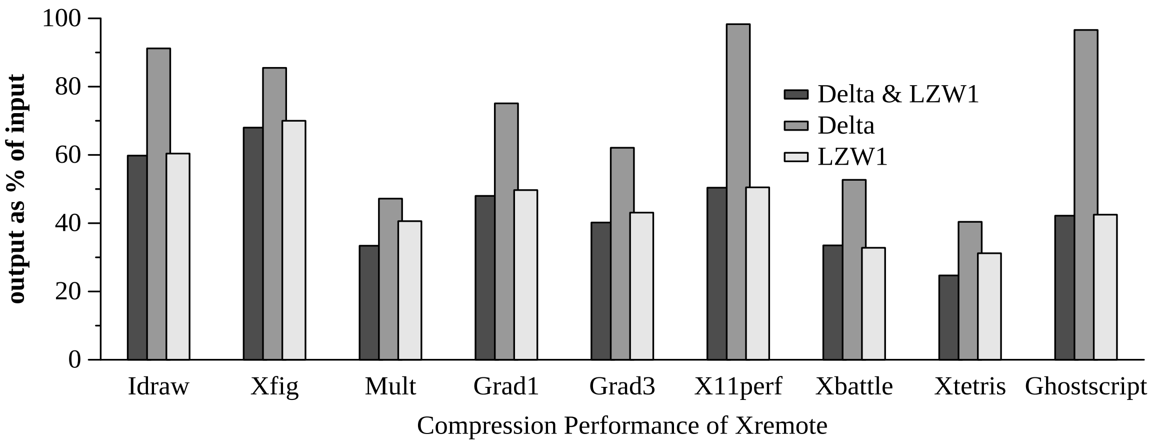


Figure 16: This graph shows output size as a function of input size for each of the traces when compressed by Xremote's two compression stages, separately and together. Note how little improvement the complete Xremote protocol (Delta&LZW1) is over the LZW1 protocol by itself. Also note Xremote's poor compression of the geometry in the Idraw, Xfig and X11perf traces.

Delta compression: The Delta compression stage should work well on small messages which are repeated almost verbatim. Input messages (keystrokes, mouse movements) fall into this category, as does xtetris' geometry (a few rectangles moving around). The images in the ghostscript trace, the complicated geometry in the idraw trace, and the text in the grad2 trace were not compressed by the delta compressor.

The text in grad2 trace would probably have been more amenable to compression if the user had used a terminal emulator other than *xwsh*. *Xwsh* redraws the whole line at each keypress instead of just drawing the new character.

The reader will notice that text and geometry are roughly of equal importance in the grad2 trace, even though the user spent most of his time typing. This is because the terminal emulator uses rectangles to clear the space behind each character and to implement a cursor.

LZW compression: The LZW compression stage achieves good compression on every important datatype in this trace. The best compression (better than 3:1) was on the repetitive xtetris trace. The worst compression was

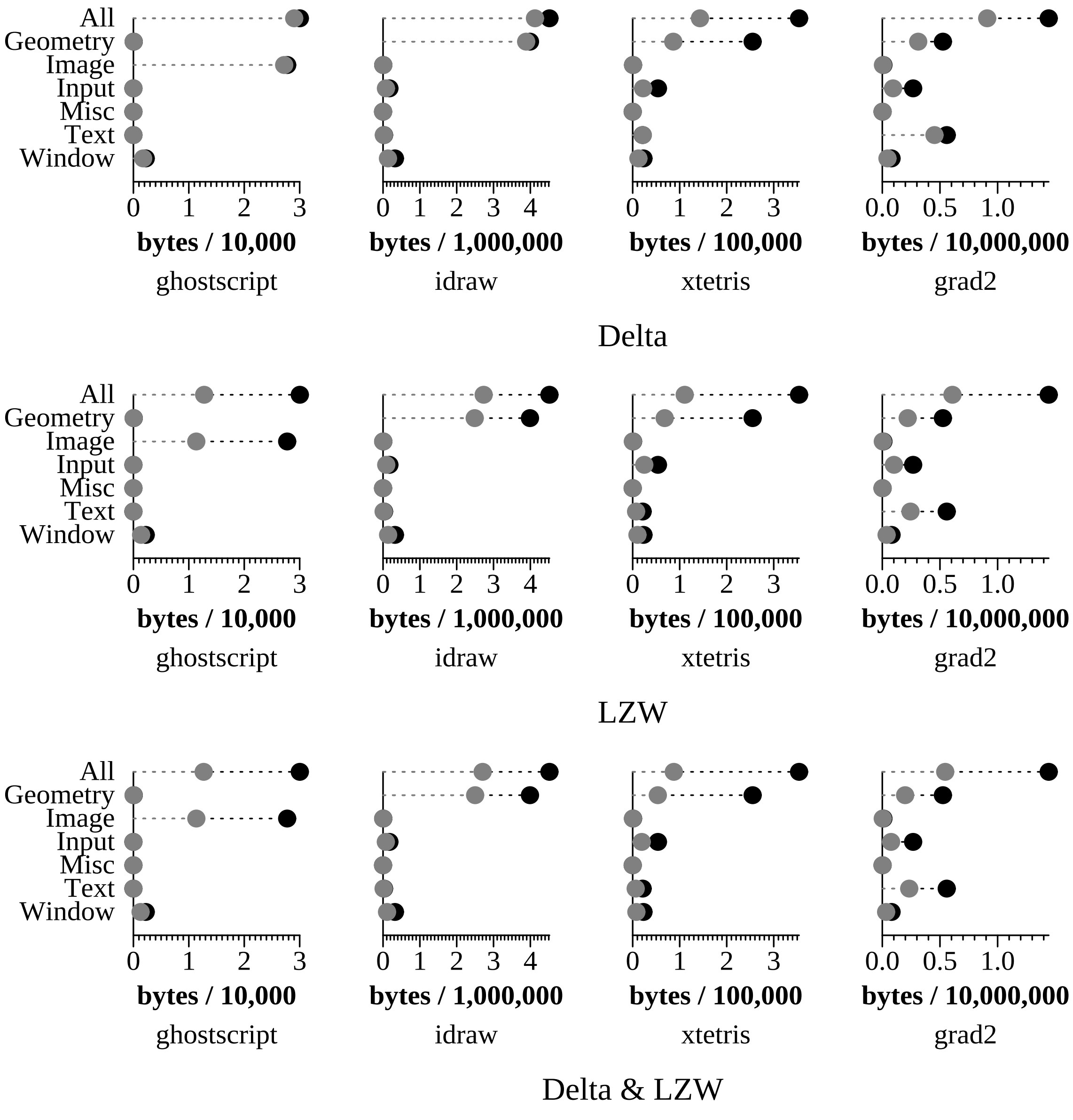


Figure 17: Magnitudes correspond to total bytes in each category. Black dots show the size of the input data while grey dots show the size of the compressed data.
The top graph shows how well the delta compactor works by itself, the middle graph shows how well the LZW encoder works by itself, and the bottom graph shows how well they work together.
Note that each trace has the same horizontal scale in all three rows, but that horizontal scales vary between traces, since each graph is scaled to the size of its trace.

on the idraw trace which is dominated by complicated geometry. This geometry is very regular because it was created by making many copies of a simple geometric dataset. Unfortunately, the copies are translated and the coordinates are absolute, so the LZW algorithm cannot recognize the similarities. I conjecture that LZW was able to compress this geometry somewhat because the coordinates are represented as pairs of 16 bit numbers, but the visible screen was small enough so that fewer than 10 bits were used.

I would not normally expect LZW compression to achieve the kind of compression on images that we see in the ghostscript trace. This image was a black and white page from a document, which is presumably an easier target for lossless compression than a landsat image, for example.

Delta and LZW compression: Across all of the traces, the lack of synergy between the delta compressor and the LZW compressor is disappointing. Although the delta compressor does well on some of the traces, it never improves overall compression by more than about 7% of the input size.

Delta compression together with LZW compression is much better than delta compression alone, and slightly better than LZW compression alone. The improvement over LZW compression alone is especially noticeable for the insignificant looking but all important input category. I conjecture that delta compression was included in the Xremote protocol entirely for the sake of input.

Compression over time: How does Xremote respond to application requirements over time? Figure 18 shows the first 8.3 minutes of the 11.5 minute idraw trace. At the left, we see *idraw* starting up. There is a flurry of activity at startup time, with an almost 5 second period where the bit-rate averages 75Kbps, but the compressed bit-rate would be only about 25Kbps. If we supposed a serial line connection with a 14400 bps modem and a low overhead network protocol, Xremote would speed application startup by a factor of three over the vanilla X protocol, but startup would still take nearly twice as long as it would if there

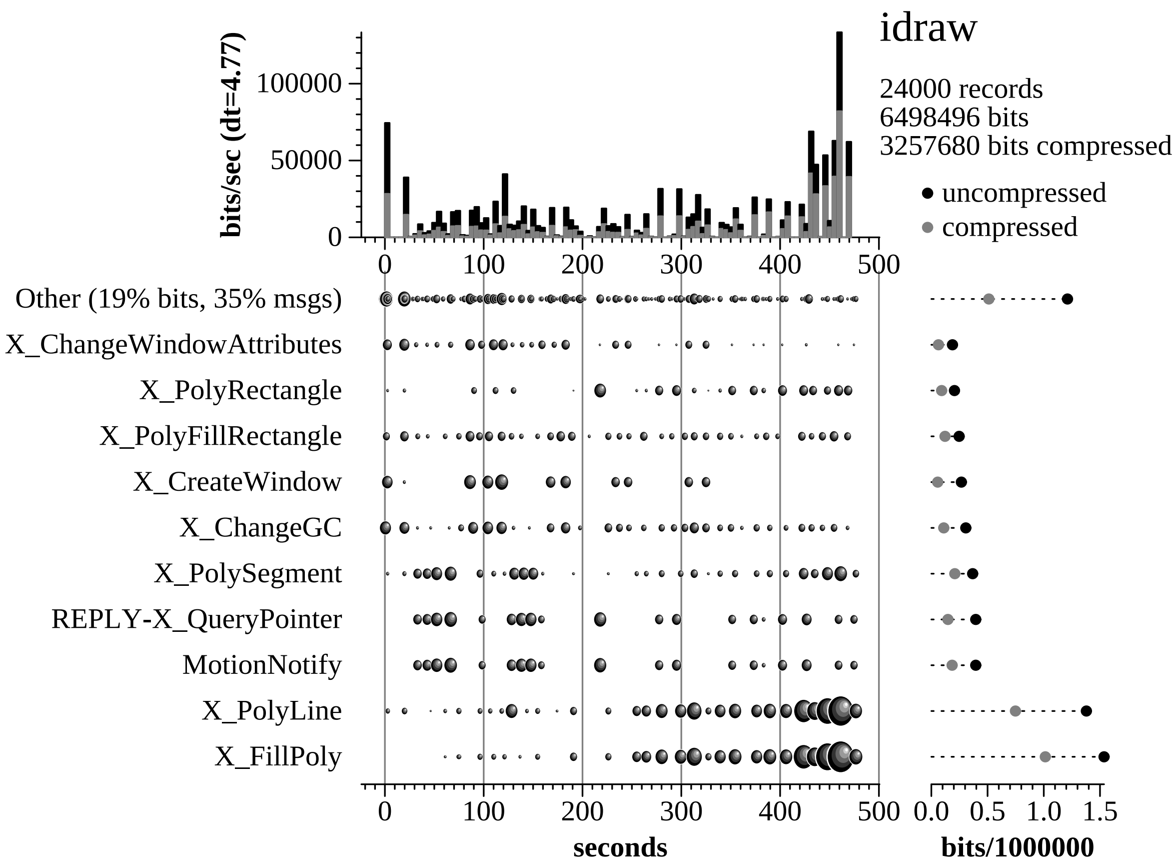


Figure 18: This graph covers the first 8.3 minutes (and 29% of the messages) from the 11.5 minute idraw trace. The X axis along the bottom of the central area is seconds since session startup. The Y axis in the main area is marked off by message type. The volume of spheres in the data area is proportional to the size of the messages which they represent (before compression). Spheres which are vertically stretched represent multiple nearly simultaneous messages, other spheres represent single messages. The dash-dot graph to the right shows the total number of bits for each type of packet before and after compression. The bar graph at the top shows the number of bits per second (before and after compression) as a function of time. The integration interval is 4.77 seconds.

were a fast network connection. Through the next 400 seconds, the compressed bit-rate stays under the 14400 mark, while the raw bit-rate drives well past this mark. Eventually, as the number of polygons and lines in the idraw database increase geometrically, geometry dominates the network and the raw bit rate passes 100,000 bps. At this stage, a slow serial line would clearly be the limiting factor even with Xremote's compression. Looking to the graph of compression ratios as a function of message type at the far left of the figure, we see that the polylines and polygons flooding our hypothetical serial line are also the most poorly compressed of all of our messages.

Overall compression: Xremote's performance on geometry and images was disappointing. On the Idraw and Xfig traces, which are both geometry bound, compression is well under 2:1. The X11perf benchmark is extremely repetitive, yet Xremote achieves only 2:1 compression. Fax protocols regularly achieve 10:1 compression on documents, yet the ghostscript trace sees less than 3:1 compression. Xremote performs poorly on geometry and images: important components of a graphics protocol.

Xremote's overall compression rate was about 2.4:1.

3.4 Discussion

Just as file compression makes disks seem bigger, network compression enhances network bandwidth. Compression algorithms tuned specifically for graphics protocols will perform far better than generic network compression because they can apply image compression algorithms to images, geometry compression algorithms to geometry, and intelligent caching algorithms to queries. Unfortunately, by treating the X protocol almost as an unstructured byte-stream, Xremote just scratches the surface of what is possible in graphics protocol compression.

Will LBX [16], the logical successor of Xremote, fix Xremote's problems? LBX promises to fill in some gaping holes by skipping over padding in the protocol, and caching server responses to standard queries. Image performance may be improved, as the status update promises special image compression code. Otherwise, LBX performance should remain close to Xremote's, as LBX is essentially a preprocessing layer for Xremote. It is not clear how well LBX's proposed image compression code will work on the tiny (single character) images usually used in document previewing where, as we will see later, image destination coordinates, window IDs and other fields make up the bulk of the data transmitted. The part of the LBX proposal dealing with caching seems very complex, and may be part of the reason why LBX is still incomplete and without a specification.

In the next chapter, I will introduce a technique for taking advantage of a structured protocol to achieve higher compression rates.

Chapter 4

Compression of Structured Data

Your line of work requires a specialized vocabulary.
Etta in *Butch Cassidy and the Sundance Kid*

In this chapter, I will introduce a technique for compressing structured data, which will be applied to the X protocol in the next chapter.

There are text compressors and image compressors, but for most types of structured data such as graphics protocols, object code, and databases, the standard mode of compression seems to be a standard text compression program like the Unix *compress* utility. Predictably, this does not work very well, because text compression programs only work if a byte is well predicted by the last few bytes, while fields in structured data are not necessarily well predicted by adjacent fields, and often are not even the same data type. Even zero order compression techniques do not usually work terribly well because probabilities are often spread evenly across all byte values.

The key to compressing structured data is to find a way to predict each field. Prediction in compression is like assigning odds for a bookmaker. If the bookmaker assigns even odds to all events, she loses money. If she assigns low probability to events that turn out to be common, she loses money faster. If she assigns probabilities perfectly, she cannot lose money in the long run. A good bookmaker finds out all that can be known about her area, and assigns probabilities fairly and judiciously. We will treat the compression of each data field as a separate event, and predict as well as we can.

This kind of compression is more work than text compression, or image compression, partly because text and image compression are already done, and partly because we have to think about each field in each new type of structured data all over again. I propose that structured data compression should only be used when the benefit justifies the cost. The cost depends on the programmer, and the benefit depends on the application, but if the performance of the X protocol across serial lines could be tripled for everybody for a one-time cost of one programmer year, this would probably seem like a good investment. If the overhead of compressed serial line IP could be further reduced by 50% for a few months of programmer time, we would have spent our money wisely. There are plenty of applications where it is worth some effort to get better compression of structured data.

In the rest of this chapter, I will discuss the implementation of structured data compression (SDC) which makes it possible to predict values for arbitrary data fields in a single unified framework. SDC is currently implemented as part of the HBX system discussed in the next chapter.

Section 4.1 introduces the ideas and techniques of arithmetic coding and statistical modeling which underlie this work. Section 4.2 discusses preconditioning data so that it can be predicted by simple frequency-based statistical models. Section 4.3 discusses implementation of statistical models. Section 4.4 tells how models are assigned to data fields. Finally, Section 4.5 summarizes the contributions of this chapter.

4.1 Arithmetic Coding and Statistical Modeling

SDC uses arithmetic coding and statistical modeling to achieve high compression rates. The beauty of this technique is the separation of modeling and coding. In this method, the sender and receiver each have a predictive model which generates a table of probabilities for the next input symbol. An arithmetic coder on the sender side accepts the table and the symbol s_i which actually occurred in the input, and communicates the symbol to the decoder in just $-\log_2(p_i)$ bits, where p_i was the estimated probability of symbol s_i appearing at this position in the input. The arithmetic decoder in the receiver decodes the coded bits using its identical table of probabilities as an inverse map. Bits are shared between adjacent messages, which means that individual messages can be expressed in fractional bits. For example, when $p_i = .5$ then 1 bit is necessary as expected, but when $p_i = .75$ then only $\approx .41$ bits are necessary to update the receiver.

Compare this to the LZ78 compression technique. With arithmetic coding, the predictive model can be built in any arbitrary way. LZ78 can only hope for repeated substrings. The data could be completely predictable and yet lack any more common substrings than a random string. LZ78's predictive model is hardwired into the algorithm. Compression of structured data requires more flexibility.

4.1.1 From Probabilities to Bits

In 1948 Shannon [49] established a lower bound on message size of

$$(-\sum_{i=1}^{n} \log_2(p_i))$$

Where p_i is the probability of each event. How to achieve this lower bound was not at all obvious. In fact, it was 29 years before an efficient technique was discovered [46, 19, 42].

Figure 19 illustrates the fundamental idea of arithmetic coding. Given an ordered set of symbols S_i with probabilities p_i summing to 1, the symbols occupy disjoint intervals covering $[0-1)$. The interval $I(S_{1\cdots n})$ of this probability space occupied by the string $S_{1\cdots n}$ is the interval of S_n $(= I(S_n))$ applied to the interval of $S_{1\cdots(n-1)}$ $(=I(S_{1\cdots(n-1)}))$. For example, if

$$I(S_{1\cdots(n-1)}) = [0, \frac{1}{2})$$

and

$$I(S_n) = [\frac{1}{3}, \frac{2}{3})$$

then

$$I(S_{1\cdots n}) = [\frac{1}{6}, \frac{1}{3}).$$

That is, $I(S_{1\cdots n})$ is the middle third of $I(S_{1\cdots(n-1)})$ because $I(S_n)$ occupies the middle third of $[0, 1)$.

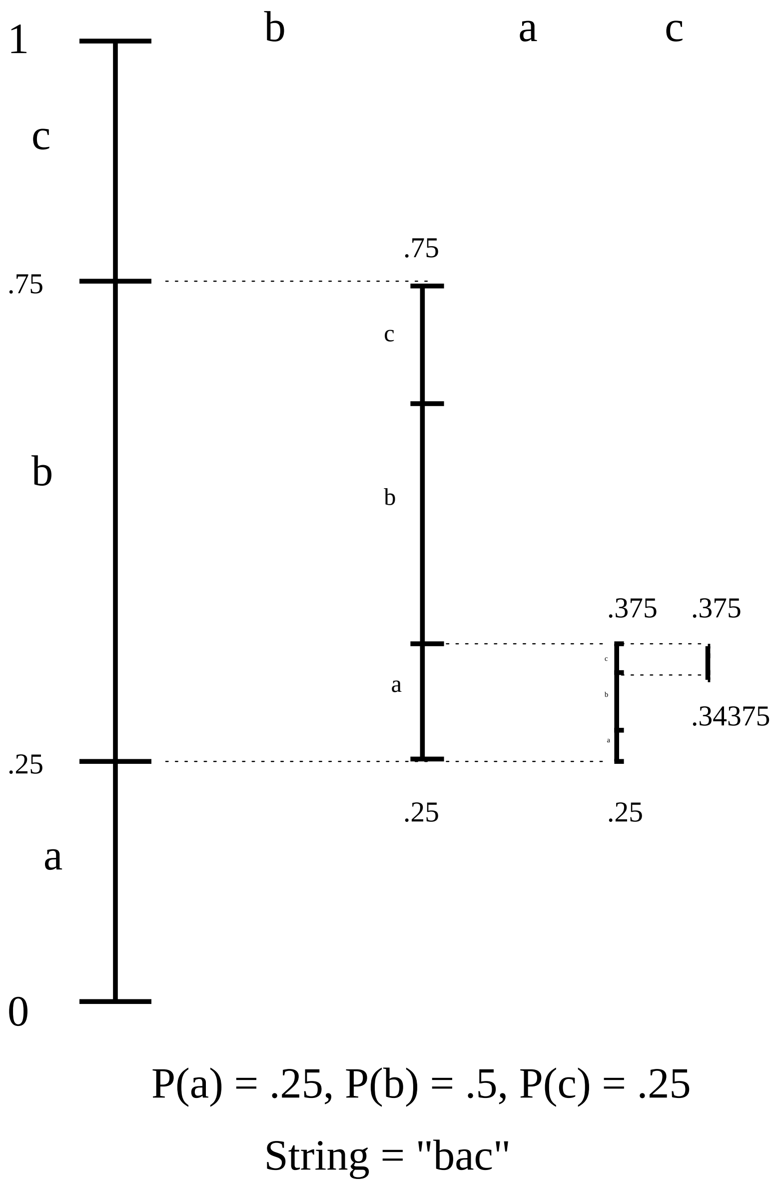


Figure 19: Example of the arithmetic coding process for the string "bac" with the assigned frequencies. The final interval is $[.34375 - .375)$, or $[.01011 - .011)$ in binary. This interval could be sent unambiguously to a decoder as any bit string beginning with 01011.

In general, if

$$I(S_{1\cdots(n-1)}) = [L_{1\cdots(n-1)}, H_{1\cdots(n-1)})$$

and

$$I(S_n) = [L_n, H_n)$$

then

$$I(S_{1\cdots n}) = [L_{1\cdots(n-1)} + L_n R, L_{1\cdots(n-1)} + (H_n - L_n)R)$$

where

$$R = H_{1\cdots(n-1)}) - L_{1\cdots(n-1)}.$$

The encoder sends a string $S_{1\cdots n}$ to the decoder by sending any binary number (or code) c in the interval $I(S_{1\cdots n})$ to the decoder. For instance, the interval of $S_{1\cdots n}$ above was $[\frac{1}{6}, \frac{1}{3})$, so any binary number c in the range $[.00\bar{1}\bar{0}, .\bar{0}\bar{1})$ such as .001 or .0100 would decode to $S_{1\cdots n}$. This is not exactly correct. Any code c in $I(S_{1\cdots n})$ will decode to an infinite length string with a prefix of $S_{1\cdots n}$. Because every code c is a member of an infinite number of nested intervals, the encoder and the decoder have to agree on a convention for finding the end of the string, such as an EOF character or by prepending a symbol count to the message.

Because of the way intervals are built, $I(S_{1\cdots n}) \subset I(S_1)$, so a code c specifying $S_{1\cdots n}$ immediately specifies S_1 because $c \in I(S_{1\cdots n})$. Furthermore, if T_i is the linear transformation mapping $I(S_i)$ onto $[0, 1)$, then $T_1(c) \in I(S_2)$ and $T_2(T_1(c)) \in I(S_3)$ and so on. The decoder works by repeatedly looking up a symbol S_i, and rescaling the input code with T_i.

The magnitude of the interval of a string is the product of the magnitudes of the intervals of the symbols in the string. The number of digits which must be sent to uniquely identify the interval of a string, is the ceiling of minus the log of the magnitude of the interval of the string. Ignoring the ceiling, which adds less than one bit to the code for a string of arbitrary length:

$$-\log_2 \prod_{i=1}^{n}(|I(S_i)|) = -\sum_{i=1}^{n} \log_2(p_i))$$

so this scheme achieves Shannon's lower bound on message size.

Now we need to know how to compute the codes using only finite precision arithmetic. Note that the only result of the encoding computation is the transmission of a number in the interval $[L, H)$. As long as we keep L and H far enough apart so that they are distinct numbers in our finite precision representation, we will be able to find a number to transmit.

Suppose the low and high ends of an interval both fall below $\frac{1}{2}$. Then, the first bit of the output must be 0. We can send the 0, and rescale our interval by doubling L and H without prejudicing the rest of the calculation. Conversely, if the low and high ends of the interval fall above $\frac{1}{2}$, the first bit of the number must be 1: we can output the 1, subtract $\frac{1}{2}$ from the interval, and double L and H, again without prejudicing the calculation.

Now we only have to worry about the case where L and H approach each other, straddling the midpoint. Notice that if $L > \frac{1}{4}$ and $H < \frac{3}{4}$, then whatever the high order bit turns out to be, the next bit will have the opposite value. A *bits_to_follow* counter is incremented, and the interval is doubled. When a bit is finally output, *bits_to_follow* bits of the opposite value are sent as well.

Table 6 shows pseudo code for encoding. Decoding is analogous to encoding. This section should have given you the flavor of the arithmetic coding process. For a full description of how to encode and decode, see [5].

4.1.2 Predictive Models

To the extent that the model's probability estimates are accurate and fully express all available knowledge, this arithmetic coding procedure is optimal. In practice, unless the source is controlled by radioactive decay or some other truly random source, we accept that the predictive models which we implement are approximations to what is known about the source. As in the bookmaker example above, generating powerful predictive models is still hard.

As an introduction to predictive models, some examples from text compression are illustrated in Table 7. These models are adaptive, which means that probability estimates are revised after each input character. In particular, probability estimates

```
int bits_to_follow, low, high;
bit_plus_follow(int bit)
{
  output(bit);
  while (bits_to_follow - -) output(!bit);
  bits_to_follow = 0;
}

encode(freq_less, sym_freq, tot_freq)
{
  long range;
  range = (high - low) + 1;
  high = low + (range * (freq_less + sym_freq)) / tot_freq - 1;
  low = low + (range * freq_less) / tot_freq;
  for (;;) {
      if (high < HALF) {
        bit_plus_follow(0);
      }
      else if (low ≥ HALF) {
        bit_plus_follow(1);
        low -= HALF;
        high -= HALF;
      }
      else if (low ≥ FIRST_QTR && high < THIRD_QTR) {
        bits_to_follow += 1;
        low -= FIRST_QTR;
        high -= FIRST_QTR;
      }
      else break;
      low = 2 * low;
      high = 2 * high + 1;
  }
}
```

Table 6: This code maps from a symbol's probability interval to output bits. The symbol's interval begins at $freq\_less/total\_freq$ and ends at $(freq\_less + sym\_freq)/total\_freq$.

Model	Example	Size
0 order context	$p(\text{'e'}) \gg p(\text{'j'})$	256 frequencies
1st order context	$p(\text{'u'})$ after 'q'$> .99$	256 0 order contexts
2nd order context	$p(\text{' '})$ after 'xy' $> .5$	256 1st order contexts

Table 7: Examples of predictive models for text compression

are based on relative frequencies. For instance (using frequencies from the introduction to this thesis), if the input stream so far has seen 4315 'e's and 21 'j's out of 42526 total characters, then the probability of 'e' would be estimated by a zero order model as $\frac{4315}{42526} \approx .101 \Rightarrow 3.3$ bits, while the probability of 'j' is estimated as $\frac{21}{42526} \approx 0.0005 \Rightarrow 11$ bits. For exact arithmetic, frequencies are usually stored as integers, rather than as floating point numbers.

A first order model conditions predictions based on the previous character. For instance, in the context 'T', 'h' was the next character in 99 out of 118 instances, leading to an estimated probability for 'h' in the context 'T' of about $0.84 \Rightarrow 0.25$ bits.

A second order model conditions predictions based on the previous two characters. For instance, in the context 'Th', 'e' was the next character in 79 out of 99 instances, leading to an estimated probability for 'e' in the context 'Th' of about $0.80 \Rightarrow 0.33$ bits.

The context is just a set of values used to index a particular predictive model from a set of predictive models. The order of the context is just the number of values used to index the model.

In text compression, the most useful contexts seem to be the immediately preceding characters, but this is not a requirement. The values used to index a predictive model could be any values known to both the encoder and the decoder at the current point in the input.

The type of context based model illustrated in Table 7 compresses English text most effectively at 3rd or 4th order [5]. A complete 4th order model would incorporate 2^{40} frequencies, so successful implementations make provisions for blending estimates from lower order models with estimates from higher order models when higher order statistics are incomplete. A discussion of blending techniques appears in Section 4.3.2.

4.2 Preconditioning Data for Compression

The models that we have discussed so far make predictions based only on symbol frequencies. They count on fields repeating themselves. Suppose that a field has a unique value each time. How can it be predicted by such a model? The short and uninteresting answer is that non-repeating fields cannot be predicted by such a model. The useful answer is that if these fields are predictable, there is usually some easy transformation of the data which makes them accessible to frequency predicting models.

For instance, fields that increase by small integers like time stamps and sequence numbers are not well predicted by frequency based models. However, if they are transformed from absolute values into relative values by subtracting their previous value, the small integer increases are usually relatively predictable.

Absolute mouse coordinates look almost like random numbers to a frequency based predictive model. Changing to relative mouse coordinates by subtracting off the previous mouse location reduces the magnitude of the numbers involved, which helps quite a bit, but does not take advantage of information that may be available about mouse speed and direction. Predicting mouse coordinates using linear extrapolation on the last two mouse locations, and encoding the difference helps even more. Quadratic extrapolation would include information on mouse acceleration, but did not perform quite as well as linear extrapolation on our traces.

Geometric shapes could be remembered by predicting each coordinate with a third order context based on the previous three coordinates so that if an object is redrawn it will be well predicted. Suppose however, that object was translated before being redrawn. All of the coordinates would be changed and the old models would be useless. A simple transformation to relative coordinates fixes this problem. Other geometric transformations are more difficult to deal with, but for 2D graphics, translation is probably the most interesting geometric transformation.

A possible problem with preconditioning is that it can change the size of data fields. For instance, the difference of two values is, in general, one bit bigger than either. If preconditioning was implemented as an independent pass over a protocol, it would require a whole new protocol to hold the new oversized values. This would be painful at best. The SDC implementation preconditions at the point of encoding so that oversized preconditioned data fields are only held in temporary variables and never put together into a protocol stream.

4.3 Model Implementation

A predictive model can be viewed as an abstract datatype implementing a one-to-one and onto mapping from symbols to disjoint intervals in $[0-1)$ where the lengths of the intervals correspond to estimated symbol probabilities. For concreteness, symbol probabilities are stored as integer frequencies, where the probability of a symbol is it's frequency divided by the sum of the frequencies of the symbols in the model.

Operations on *PredictiveModel* include the following:

IntervalOf(Symbol) returns the interval corresponding to a symbol, or else $\perp$ if the symbol has no interval (is predicted with zero probability).

SymbolOf(num) returns the symbol corresponding to the interval which the number is a member of.

Update(Symbol) increments the symbol's frequency.

Rescale() divides all of the symbols' frequencies by 2. Since frequencies are stored as integers, frequencies can fall to zero. There is no benefit to storing a symbol which is predicted with probability zero, so *Rescale* can be used to reclaim storage being held by infrequently used symbols. *Rescale* must be invoked whenever the total frequencies exceed a threshold defined by the arithmetic precision available on the coding and decoding machines (typically, frequencies must fit in 15 bits so that they can be multiplied on 32 bit machines). Sometimes there are symbols whose frequencies are not allowed to fall to zero, as discussed below. In this case, *Rescale* sets the new frequency to the greater of half of the old frequency and one.

4.3.1 Zero Frequency Symbols

An arithmetic coder cannot send a symbol which is predicted with zero probability $p(s) = 0$, since it would take $-\log 0 = \infty$ bits to do so. Therefore, every symbol must be predicted with some non-zero probability. Unfortunately, the laws of probability give us no guidance as to the probability which should be assigned to symbols which have not yet appeared in the input stream. We have to try to use some other knowledge about the input to assign a reasonable probability to unseen symbols. For a discussion of the zero-frequency problem, see [5].

One reasonable approach might be to assign unit frequency to each unseen symbol, increasing this frequency as the symbols occur. This would work for small alphabets such as acsii characters or bytes, but for large alphabets such as 32 bit timestamps, where the number of symbols in the alphabet is larger than the maximum implementable frequency count, this is impossible. We need a way to represent probabilities smaller than $1/MAXFREQ$. The answer is escape symbols. An escape symbol is a way for the encoder to tell the decoder to switch to another model, in this case, a special model for new symbols. In SDC, every model but one has an escape symbol. The minimum frequency of the escape symbol is 1, which means that it can always be sent in a finite number of bits. The only model which does not have an escape symbol is the flat byte model which is used to send new symbols. This model contains all 256 byte values, all maintained with minimum frequencies of 1. If, for instance, a 32 bit TimeStamp is not found in the model predicting it, first the escape symbol for that model, and then the four byte symbols are coded and transmitted.

In SDC, whenever a new symbol is encountered in some context, first the escape symbol is sent, then the new symbol is sent using the escape model, and finally, the frequencies of the escape symbol and the new symbol are incremented in all of the models involved (the new symbol had an implicit frequency of zero, and now has a frequency of one). In essence, SDC estimates the probability of a new symbol using the frequency of unknown symbols in the past.

Note that the escape symbol is the only symbol in an empty model, and is thus predicted with probability one, and thus takes no bits to transmit. In fact, suppose the escape symbol has probability p_e in the top-level model and the new symbol has p_n in the escape model. Then, the number of bits used to transmit the escape symbol and the new symbol is $-\log p_e - \log p_n = -\log(p_e p_n)$. The probability p_e is just a scaling factor for the probabilities in the escape model, essentially allowing probabilities to be represented with extended precision without incurring any coding cost.

There are often hierarchies of escape models, as will be discussed in Sections 4.3.2 and 4.3.3. Other possibilities for updating the frequency of the escape symbol are to maintain it at the minimum value (1) or to increment the frequencies of the escape symbol and the new symbol each by 1/2 instead of incrementing by 1, so that the new symbol event doesn't increase the total frequency by more than 1. The literature suggests that the technique used by SDC is noticeably better than maintaining the escape symbol at a minimum, and insignificantly worse than incrementing by $\frac{1}{2}$ instead of by 1 [5]. In practice, results will depend completely on the input.

Escape symbols are also useful for implementing high order contexts, where new symbols are more common, and for implementing blended models, where multiple models are used simultaneously.

4.3.2 High Order Contexts

As we have described them, predictive models implement zero order contexts, where the relative frequencies of the members of some alphabet are maintained without reference to any sort of context. In SDC, high order predictive models are implemented by (implicitly) storing one model for each possible context in a table which maps from contexts onto predictive models. Models which are not explicitly stored contain no symbols except the escape symbol, which has probability one, and therefore doesn't need to be coded or transmitted.

For instance, suppose we wish to code text using a third order model, and the input stream is 'ababab....' We cannot code the first three characters using a third order model because we do not have enough context yet, so we code them using some other model.

To code the fourth character, 'b', we look up the model 'aba' in the model table. It isn't there, so we (implicitly) send an escape, use the escape model to send the 'b', and create a new model 'aba' with 'b' and the escape character in it, each with a frequency of 1.

Coding the fifth character proceeds analogously except that the character is 'a' and the context is 'bab.'

Finally, when we code the seventh character, 'b', in the context 'aba,' we find the 3rd order context 'aba' in the context table. It predicts 'b' with probability .5, so we code 'b' in 1 bit, and increment its frequency. If the next character we saw in this context was 'b', we would code it in $-\log(2/3) \approx .58$ bits. In a coder implemented with 15 bit frequencies, this particular input stream would eventually be compressed at a rate of about .00004 bits per character, while a 0 order predictive model would only compress this input stream to about 1 bit per character.

Unfortunately, it can take a while to build up enough statistics for effective compression using only high order models. Blended models address this problem.

4.3.3 Blended Models

Zero order statistics accumulate more quickly than first order statistics, which accumulate more quickly than second order statistics, and so on. So, if you are using a third order model, but you don't have any third order statistics for an event in the input stream, you would like to be able to use a second order model, and if you don't have statistics for the second order model, you would like to use a first order model, and so on. A predictive model which combines predictions from several contexts like this is called a blended model. Blended models can be implemented directly, but it is more convenient to use a hierarchy of models, signaling transitions with the escape symbol.

For example, suppose the previous three characters were 'xyz,' and the next character in the input is 'd.' The models are shown in Table 8. The actions associated with the transmission of 'd' using this hierarchical model are shown in Table 9. In this hierarchical model, the probability of 'd' was $\frac{1}{96}$, costing ≈ 6.58 bits to communicate.

Of course, there was some inefficiency in this process: after the decoder sees an escape in the context 'xyz' it knows that the new symbol is not 'a', 'b', or 'c', since those symbols are all in model 'xyz'. A more efficient coder would temporarily exclude

model for context 'xyz'			model for context 'yz'		
char	frequency	probability	char	frequency	probability
a	1	$\frac{1}{12}$	a	6	$\frac{1}{2}$
b	2	$\frac{1}{6}$	b	1	$\frac{1}{12}$
c	3	$\frac{1}{4}$	c	2	$\frac{1}{6}$
escape	6	$\frac{1}{2}$	escape	3	$\frac{1}{4}$
model for context 'z'			**model for zero order context**		
char	frequency	probability	char	frequency	probability
a	3	$\frac{1}{4}$	a	2	$\frac{1}{6}$
b	6	$\frac{1}{2}$	b	3	$\frac{1}{4}$
c	1	$\frac{1}{12}$	d	6	$\frac{1}{2}$
escape	2	$\frac{1}{6}$	escape	1	$\frac{1}{12}$

Flat model: maps non-zero frequencies onto all byte values

Table 8: An ordered set of predictive models forms a single hierarchical predictive model. The probabilities in the third order model 'xyz' are used as is. The probabilities in the second order model 'yz' are effectively scaled by the probability of the escape symbol in the third order model 'xyz', the probabilities in the first order model are effectively scaled by the (scaled) probability of the escape symbol in the second order model, and so on. The flat model can be used to send any symbol in byte sized chunks if the symbol isn't found in one of the parent contexts.

these symbols from consideration in models 'yz', 'z', and zero, uniformly rescaling the remaining probabilities so that they scale to 1. The results of applying exclusions in this case are shown in Table 10

The first technique (no exclusions) is known as “lazy exclusions.” Typically, exclusions give an improvement of about 5%, in compression when compared to lazy exclusions, while doubling the computational cost [5]. SDC implements blended contexts using the lazy exclusion method, accepting slightly poorer compression performance for greatly reduced computational overhead.

4.3.4 Update Policy

Whenever a symbol (including the escape symbol) is encoded or decoded using a given model, its frequency in that model is incremented. Additionally, the symbol is added to the last model in which it was searched for unsuccessfully, with a frequency of 1. The models of Table 8 are shown updated after encoding or decoding 'd' in Table 11.

Sending 'd'

Model	Found?	Sent	Probability	Bits
'xyz'	no	escape	$\frac{1}{2}$	1
'yz'	no	escape	$\frac{1}{4}$	2
'z'	no	escape	$\frac{1}{6}$	≈ 2.58
zero order	yes	'd'	$\frac{1}{2}$	1
Totals			$\frac{1}{96}$	≈ 6.58

Table 9: Sending 'd' using the hierarchical model of Table 8. The (sub) models are searched in order until one containing ' d' is found. Whenever 'd' is not found, the escape symbol for the concerned model is sent.

Sending 'd' with exclusions

Model	Found 'd'?	Sent	Probability	Bits
'xyz'	no	escape	$\frac{1}{2}$	1
'yz'	no	escape	1	0
'z'	no	escape	1	0
zero order	yes	'd'	$\frac{6}{7}$	≈ 0.22
Totals			$\frac{3}{7}$	≈ 1.22

Table 10: Sending 'd' with exclusions, using the hierarchical model of Table 8. As the sub-models are searched in order, a list of symbols which the decoder will know are impossible is built. Before a symbol (in this case, "escape" or 'd') is sent, the impossible symbols are temporarily deleted from the sub-model in question, increasing the probabilities of the remaining symbols. In practice, exclusions yield an improvement of about 5% in compression for twice the computational cost [5].
SDC does not implement exclusions for performance reasons.

model for context 'xyz'

char	frequency	probability
a	1	$\frac{1}{13}$
b	2	$\frac{2}{13}$
c	3	$\frac{3}{13}$
escape	7	$\frac{7}{13}$

model for context 'yz'

char	frequency	probability
a	6	$\frac{6}{13}$
b	1	$\frac{1}{13}$
c	2	$\frac{2}{13}$
escape	4	$\frac{4}{13}$

model for context 'z'

char	frequency	probability
a	3	$\frac{3}{14}$
b	6	$\frac{3}{7}$
c	1	$\frac{1}{14}$
d	1	$\frac{1}{14}$
escape	3	$\frac{3}{14}$

model for zero order context

char	frequency	probability
a	2	$\frac{2}{13}$
b	3	$\frac{3}{13}$
d	7	$\frac{7}{13}$
escape	1	$\frac{1}{13}$

Flat model: (unchanged)

Table 11: Updated hierarchical model. The model of Table 8, updated after encoding or decoding 'd'. The escape symbol's frequency was incremented in the first three models, 'd''s frequency was incremented in the fourth model, and 'd' was added to the third model with frequency 1.

Finally, the last model searched for but not found is created. The new model is created containing only an escape symbol (with frequency and probability 1). For instance, if, in the last example, there was no model for the contexts 'xyz' and 'yz', but there was a model for the context 'z', HBX would have created a new empty model for 'yz', but would not have created a model for 'xyz'.

4.3.5 Deletion Policy

As mentioned above, the frequencies of symbols in models are periodically rescaled (halved), and symbols whose (integer) frequencies reach zero are deleted, except for the escape symbol, whose frequency is set to one whenever it is rescaled to zero. Any model which is found to contain only the escape symbol after rescaling is also deleted.

Rescaling occurs under the following circumstances:

- When the sum of frequencies in a model reaches a maximum value. This maximum value can be a function of arithmetic precision, in which case 14 or 15 bits would be appropriate for a machine with 32 bit arithmetic, or it can be a function of storage space, in which case 8 bits might be appropriate, or it can be set small to encourage more rapid rescaling of models, for a compression system which reacts more quickly to statistical fluctuations in the input stream. HBX sets the maximum value to 256 for quicker adaptation.

- SDC maintains an ordered list of models and an index into this list. Whenever and while the total number of symbols exceeds 5000, the indexed model is rescaled, and the index is incremented (wrapping at the end of the list). This process serves to keep memory use constant, rather than proportional to the size of the input.

4.3.6 Data Structures

Some arithmetic coders, optimized for small alphabets [9], maintain symbols and frequencies in 4 linear arrays. The first array maps from symbol indices to indices in a frequency table (the second array). The second array maintains the frequencies corresponding to the symbols in non-ascending order. The third array maintains, for each index in the second array, the sum of all the frequencies with greater indices. The fourth array maps from frequency indices to symbol indices.

Given this data structure, the *IntervalOf(Symbol)* operator executes in $O(1)$ time. The lower bound of the interval is the (scaled) sum of all frequencies with greater indices. The upper bound of the interval is the lower bound plus the frequency of the symbol, and the scaling factor (total of all frequencies) is available from the first element of the array.

The *SymbolOf(num)* operator executes in $O(\log n)$ time using binary search to find the enclosing interval in the summed frequency array, and then using the fourth array to map from a frequency index to a character.

Unfortunately, *Update(Symbol)* executes in $O(n)$ time (where n is the size of the alphabet) because up to n elements in the summed frequency array must be updated each time, although this is mitigated somewhat by keeping the frequency and summed frequency arrays sorted in non-ascending order, so that popular symbols are near the bottom of the array and require fewer updates.

Another problem with this data-structure is that the array size must be the same as the alphabet size, which could be very expensive for higher order coders where most predictive models will have seen only a few characters.

Clearly a data structure supporting faster updates is required for large alphabets. Moffatt developed an implicit tree structure for the frequency array which trivially supports $O(\log n)$ updates, and which is, in fact, linear because a heap-like move to top policy for popular symbols allows update time to be bounded by a constant [29]. Unfortunately, this structure, as described, like the previous array based structures, presupposes a maximum alphabet size, and is thus not practical for large alphabets, or for high order contexts where many models containing just a few symbols may exist.

Moffatt used a data structure for word-based text compression which was based on balanced binary trees [31], after the (not necessarily balanced) binary trees used by Cleary and Witten [9] to implement higher order character-based text compression. Storing each model in a balanced tree structure allows *IntervalOf(Symbol), SymbolOf(num), and Update(Symbol)* to complete in $O(\log n)$ time, while using space proportional to the number of symbols actually instantiated in the model. In both of these implementations, higher order models are pointed to by the records for lower order models. For instance, if there is a third order model for characters which might follow 'LZ7', then the record for '7' in the second order model for 'LZ' has the pointer to this model. In this way, higher order models can be found in $O(o \log n)$ time, where o is the order of the model. These trees of models are called forward trees.

Bell, Cleary and Witten [5] improve on this scheme with an extra pointer per node (called a vine pointer), pointing to a context with one fewer character, so that in case there was no pointer to a third model 'LZ7' at the record for '7' in the model 'LZ', a pointer to the model for 'Z7' is available, so that the model for 'Z7' is available with a single indexing operation. In this way, symbols can be encoded in $O(\log n)$ time (dropping the dependency on the order of compression). There is an example of a forward tree with vine pointers in Figure 20.

Bell et al's vine pointer optimization depends on an assumption which is completely reasonable for text compression: the symbols being used to index a predictive

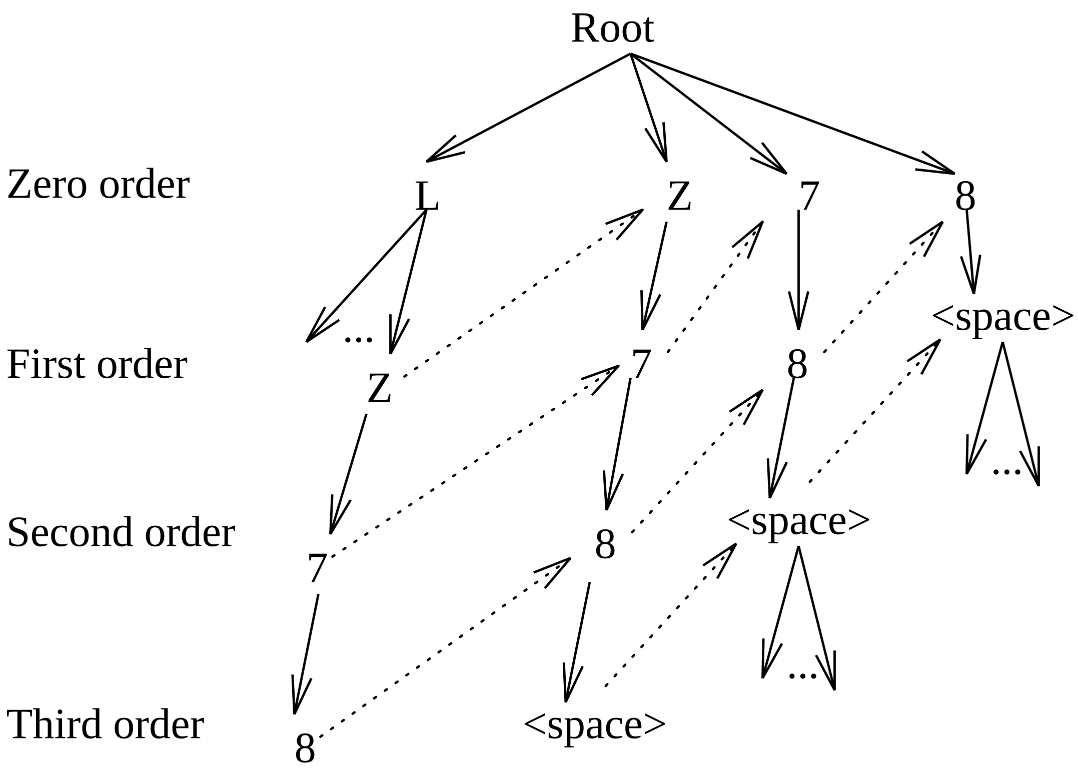


Figure 20: A forward tree with vine pointer. The root points to the entries in the zero order model. Forward pointers (solid arrows) in each entry in the zero order model point to entries in the first order model, and so on. Moving down in the tree requires an indexing operation. When the coder discovers that there is no fourth order model indexed by 'LZ78', it can find the third order model indexed by 'Z78' immediately by following a vine pointer (dotted arrow) from the entry for '8' in the third order model indexed by 'LZ7', to the entry for '8' in the second order model indexed by 'Z7'.

model immediately precede the current symbol. This allows incremental movement through their data structure. In HBX, the context used to predict the next symbol may be completely independent of the previous symbol and the context used to predict it. Furthermore, SDC deletes infrequently used models, which makes it inconvenient to maintain pointers between models.

SDC maps from contexts to models using a balanced tree (a deterministic skip list [34], a variant of the red-black tree). SDC models are stored in deterministic skip lists, enhanced so that each internal node maintains the sum of the weights of its left children so that *IntervalOf(Symbol), Update(Symbol)* and *SymbolOf(num)* all complete in $O(\log n)$ time, where n is the number of symbols in the model. Model retrieval completes in $O(\log m)$ time where m is the number of models. Since the symbols are stored in the models, and the number of models is bound by the total number of symbols $s \leq 5000$, the cost of retrieving and querying o models implementing an o order hierarchical model is $O(\log s)$.

Note that since the *Rescale* operation, halves frequency counts, and can be implemented in linear time in the number of Symbols with any of the data structures, its cost can be amortized under the cost of the *Update* operation without affecting the order of the cost of the *Update* operation.

4.4 Context Specification

Some mechanism must be provided to map from fields in data structures to contexts and preconditioning transformations. SDC provides a virtual machine language for parsing and simple context specification. Each instruction specifies a field length, an action, and a context specifier. The language includes arithmetic operations and branching, enabling parsing of complex messages.

In HBX, C code maps from X message types to fragments of SDC code. Each fragment of SDC code is responsible for parsing a single X message. Most message types do not generate a lot of bytes, and so they are parsed with a generic piece of code which assigns each 2 byte piece of the message a context consisting of the actual type of the message, and the offset of the 2 byte chunk into the message. Important messages are parsed much more carefully.

The best feature of the little language is that the same code executes during encoding and decoding, removing a potential source of error. The worst feature of the little language is that it enforces front to back parsing of messages, which is not always the best order. For instance, in an X_PutImage message, given an image width and height pair, there is only one possible length. A predictive model for the length field, with width and height as context, would very quickly learn to predict the length in a very small fraction of a bit. Unfortunately, the length field precedes the width and height fields in the X_PutImage message, and because of padding considerations, there are usually several different heights that could correspond to a length-width pair, resulting in the transmission of at least a couple of bits.

If I had it to do over again, I might dispense with the little language and specify contexts in C.

4.5 Summary

SDC was developed to compress the X protocol, but may be applied to any type of structured data such as object code, network headers, RPC traffic, or databases.

SDC exploits the separation of predictive model and coder in arithmetic compression to apply custom compression techniques to fields in structured data. To the extent that the predictive models used accurately encapsulate all of the available knowledge about the fields in question, this technique is optimal. Although this technique follows directly from the theory of arithmetic coding, it appears to be new.

Chapter 5

Higher Bandwidth X

Higher Bandwidth X (HBX) is a compressed X protocol intended to provided improved interactive performance across low bandwidth interconnects such as serial lines.

HBX uses the technique and implementation for compressing structured data discussed in the previous chapter. This technique requires good predictive models, so this chapter will focus on the details of some of the most important models used in HBX. These models result in a compression performance of about 7.5:1 across my trace suite, which is about three times the compression performance of Xremote.

5.1 HBX Overview

Figure 21 shows the gross architecture of HBX. HBX is transparent to both client and server, just as Xremote and XTee were, and using the same mechanisms for multiplexing multiple streams. Inside HBX, traffic from multiple streams is parsed into messages, tagged by message type, and multiplexed together. The encode module compresses messages, and precedes them with a length field, so that the decode module can be sure that it has a whole message before it starts decoding it. (If the decode module had been written so that it could stop and wait for input in the middle of a message without blocking the encode module, the length field would not be necessary. This would have been easy to implement using a threads package, but

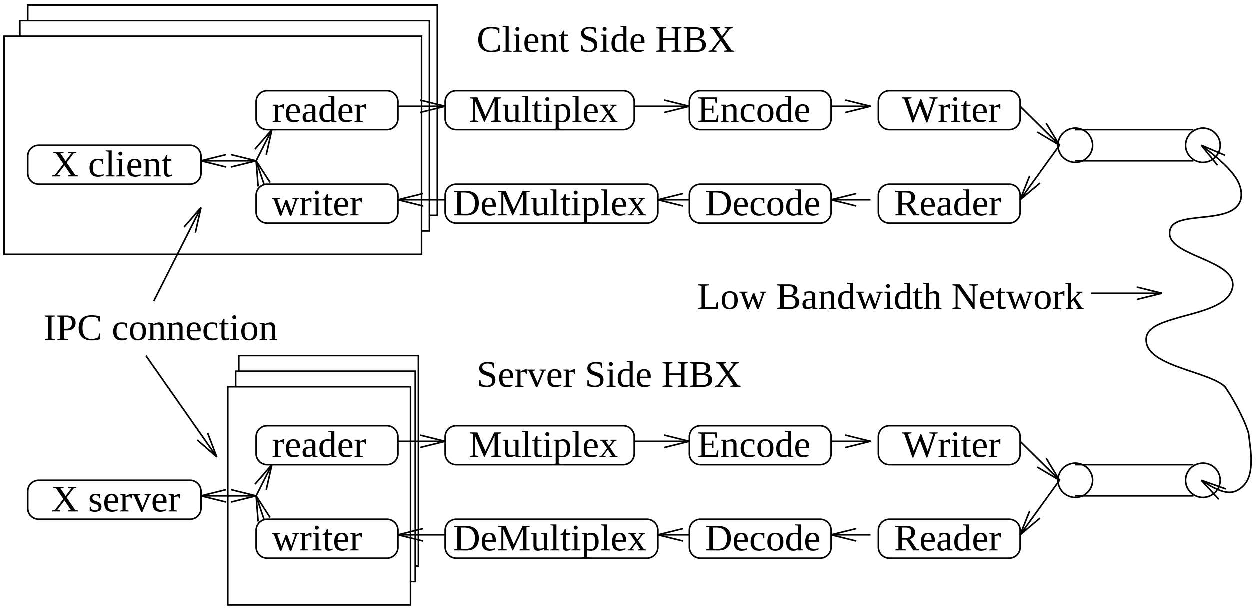


Figure 21: This is the HBX architecture, the multiple boxes on the client side show that there can be many clients, and reader/writer pairs. The multiple boxes on the server side show that although there is only one X server, there can be many connections and reader/writer pairs.

Compressed message length	Size of length field
1-7 bits	1 bit
13-29 bits	3 bits
30 bits - 16 bytes	1 byte
17 bytes - 2048 bytes	2 bytes
2049 bytes - 2^{19} bytes	3 bytes

Table 12: Size of the length field for compressed HBX messages.

was judged to be too difficult using standard C.)

Message length fields are statically coded to make it easy for the decoder to determine whether it has received the entire length field before it starts decoding it. Table 12 shows the sizes of HBX length fields as a function of compressed message size. These length code sizes very roughly approximate a static Huffman code for compressed message lengths which I generated from one of the multi-application traces.

Compressed messages are padded to whole bytes and forwarded individually to the writer module, which buffers up all of the messages immediately available from the network and writes them out all at once to minimize network protocol overhead.

HBX would have achieved somewhat higher compression ratios if the encoder had compressed all of the messages immediately available from the network into a single buffer, before prepending the length field and forwarding the message to the writer module. About half of the savings would have come from sharing length fields, and half from avoiding the wasted bits that come from padding compressed messages out to whole bytes. I did not do this because it would have been harder to measure the performance of HBX as a function of message type.

Note that the compression software operates on the multiplexed stream, not on individual streams. It would be easy to operate on individual streams instead, but memory requirements would be vastly increased, and individual streams would forfeit the possibility of sharing useful models from other streams.

5.2 Some Predictive Models in HBX

To recapitulate the previous chapter: Since the values or symbols appearing in each field in each X message type may have different distributions, HBX associates a context specifier indexing a predictive model with each field in each X message type. When different message fields generate identical contexts, they share models. Some message fields include their message type and byte offset in their context so that they will never share a model, while other message fields use other types of context which encourage sharing. Since there are many possible contexts, predictive models are instantiated lazily. So that the group of instantiated predictive models does not grow in proportion to the size of the input, the frequencies of the symbols in the models are periodically rescaled (actually halved, implementing exponential decay). Symbols whose (integer) frequencies reach zero are deleted, and empty models are deleted. There is an escape mechanism for gracefully dealing with symbols whose estimated probability is zero.

The key to optimizing the compression of structured data is to find a good predictor for each field. X is a graphics protocol, supporting text, geometry and images. Other types of data are also important, such as window-ids, timestamps, and sequence numbers.

Here are the predictive models generally used by HBX for each of these datatypes (the models have to be specified field by field, and so there are many fields in infrequently occurring messages which still use the default model of Section 5.2.5, although they might logically be encoded using the models of Sections 5.2.1-‘refsec:incrementingFields):

5.2.1 Text

Text compression has been widely studied. HBX uses one of the best text compression algorithms available: Moffatt's PPMC' [30, 5]. PPMC' uses a hierarchical predictive model which uses the escape technique to blend probabilities from models of orders three through zero. The contexts indexing the predictive models are the previous three, two, one and zero characters, respectively. See Section 4.3.3 for an example of this type of hierarchical predictive model in action.

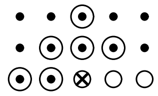


Figure 22: Model for predicting bits in bi-valued (bitmap) images: Pixels with known values have black dots. The values of the six pixels with circled dots identify a predictive model for the next pixel to be encoded, which is identified with an 'X'.

5.2.2 Geometry

As I mentioned in the introduction, geometry compression has not been studied as much as text or image compression. The most popular classical technique for integer geometry has probably been to use relative coordinates, and to use varying size coordinates for small, medium and large relative values.

HBX transforms geometry into relative coordinates, and then predicts the relative coordinate values using a third order models similar to the text prediction model: The context used to build the hierarchical predictive model for each X coordinate is the previous three X coordinates. Y coordinates are treated equivalently.

Using relative coordinates increases the likelihood that many of the relative values will be small, and that values will be repeated, which makes them easier for a zero order model to compress.

Relative coordinates are also invariant under translation, so that if objects are drawn, translated, and redrawn, the high order models will be able to take advantage of the repeated strings and provide good compression.

5.2.3 Bi-Level Images

There are two image models: one for large images, and one for small ones. Across my trace suite, small images account for most of the bytes because the document previewers transmit pages one glyph at a time. (See the size distribution of messages in the ghostscript trace shown in Figure 11 for example. The largest message in the trace is about 80 bytes, implying a maximum image size of about 21 pixels square.)

Small Images (no dimension exceeding 64): treated as atomic entities and cached in the same symbol store as characters and coordinates. A third order hierarchical model is used to predict small images: the third level model is indexed by a hash of hashes of the values of the previous three small images, and a hash of the width and height of the current image. The second level model is indexed by a hash of hashes of the values of the previous two small images, and a hash of the width and height of the current image, and so on until the zeroth level model is indexed only by a hash of the current image's width and height. When compressing these small images, it is critical to do a good job of predicting X and Y information as well as image width and height, because these fields can easily dominate the image, especially after compression. For a further discussion of the compression of the X_PutImage message, see Section 5.4.2.

Large Images (some dimension exceeds 64): compressed pixel by pixel using a set of previous pixels to provide context for predictive models, allowing for prediction of repeated bit patterns. Figure 22 shows the pattern of pixels used to index the predictive model for the next bit in the input image. This 6 bit pattern (similar patterns appear throughout the literature: see for example [23]), picks up both horizontal and vertical patterns in the input image, providing a kind of cheap 2D compression. I experimented with larger and smaller patterns, but this one seemed to give the best compression on my trace suite. I assign a third value to out-of-image pixels to account for image edges. This gives us 163 possible different predictive models, or contexts in which to predict a bit-value, each storing the relative frequencies (probabilities) of 0 and 1. This is a first order model with no hierarchy. Whenever a model is not found, or a bit value is not found in the model, the bit is to be sent as a byte using the flat escape model.

My trace suite did not include color or grayscale image transmission.

5.2.4 Incrementing fields

Important examples of incrementing fields in X are timestamps and sequence numbers. Timestamps usually appear in events sent from the server to the client. They are 32 bits long, and represent milliseconds since some arbitrary event, usually server initialization. Sequence numbers are 16 bits long and usually appear in messages from the X server to the X client, they refer to the last message from the client received by the X server.

As mentioned in Section 4.2, frequency based predictive models cannot effectively predict fields with incrementing behavior because individual values are not repeated very often. However, the change in these fields from message to message is usually a small integer, so HBX predicts differences instead of absolute values for these fields.

5.2.5 Defaults

There are many X messages with many many fields. The generic message routines index a model for these fields with the type of the actual message, and the offset of the field, hoping that the field can be effectively predicted with a 0 order frequency based predictive model. This is actually extremely effective for window IDs graphics contexts, and length fields, among others.

5.2.6 Models for all the Messages

The complete set of models used for each X message is documented in Appendix B.

5.3 Optimizing Compression of an X Message

There are many X messages. Specifying predictive models for each field in each message would be a dreary task. Fortunately, most X traffic is generated by just a few messages as we saw in Table 3. This means that we can reap most of the available benefits by optimizing just a few important messages. By default, HBX assigns a 0 order context model to each field in each message. During the optimization task, I assign different models to fields in important messages which did not compress well

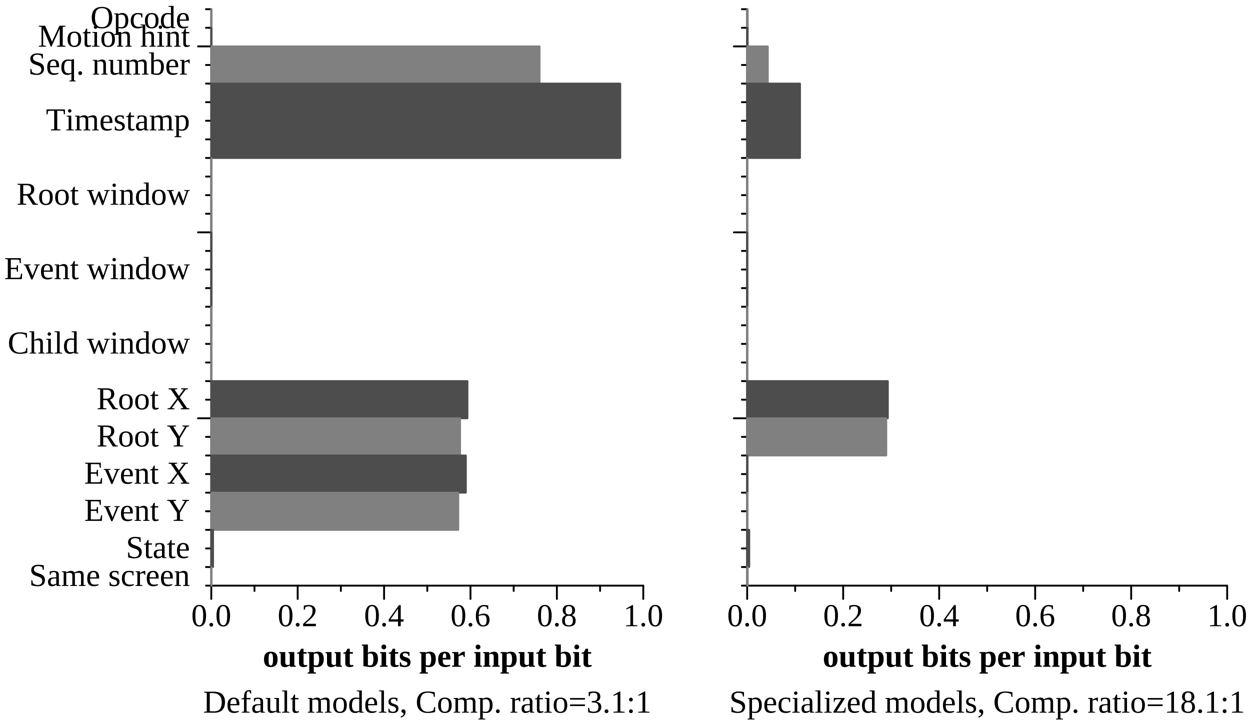


Figure 23: Output bits per input bit for each field in a mouse movement event. The area of each bar is proportional to the compressed size of the indicated field. To the left, all of the fields were compressed using default 0 order contexts. To the right, the *Seq. number* and *Timestamp* were compressed using relative contexts, which predict differences from previous values rather than absolute values. The first coordinate field was predicted by using linear extrapolation on the previous two mouse coordinates, and the last coordinate field was predicted as an offset from the first.

with the default models. An X message profile isolates important messages. Once an important message has been isolated, I generate profiles like those in Figure 23, which show which fields in the MotionNotify event are compressing and which are not. The profile on the left shows the MotionNotify event before optimization. Many fields compressed well, but the incrementing fields: *Seq. number,* and *Timestamp*, and the coordinate fields *RootX, RootY, EventX,* and *EventY* are not compressing very well at all. Specializing the predictive models used by these fields resulted in the profile on the right. The incrementing *Seq. number,* and *Timestamp* were predicted with a relative model which predicts differences from previous values. The first coordinate field was predicted with linear extrapolation, and the second coordinate field was predicted to be at a constant offset from the first. Optimization resulted in an *improvement* of 6:1 for an overall compression ratio of 18:1 for the fields in this message. (The delivered compression rate, which is what the other figures use, will be somewhat worse than the weighted average of the fields, because HBX adds overhead in the form of length and tag bits to each message.)

In it's current state, HBX has 557 lines of code mapping predictive models onto message fields. Of these lines, 136 were necessary to parse the messages so that default models could be applied to each field. The other 421 lines of code constitute my optimization efforts to date. This code is mostly written in a special low level language which enforces interfaces, and makes it easy to use the same mapping code for encoding and decoding, which removes a potential source of error from the system. This mapping code could potentially be written by a specification compiler or an RPC stub generator. (I wrote the image compression code in C. My little language was not expressive enough for the manipulations I needed.)

5.4 Compression Performance

The relative compression performances of Xremote and HBX are shown for each trace in Figure 24. HBX's average performance was over 3 times Xremote's and HBX outperformed Xremote on every trace. The relative difference was most apparent for geometry and images, as in the Idraw, Xfig, and Ghostscript traces, and especially

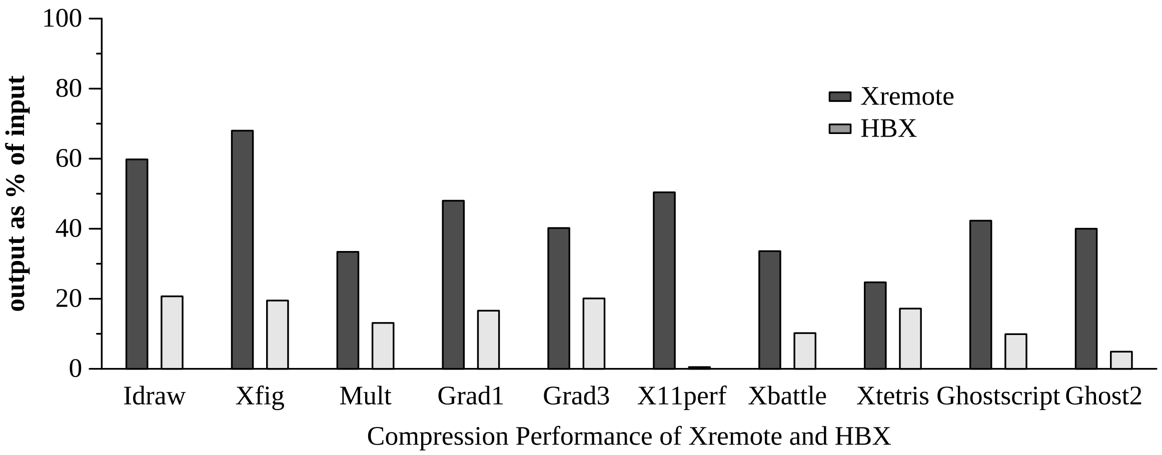


Figure 24: The height of the bars indicates output size as a fraction of input size. HBX outperforms Xremote on every trace.

the X11perf trace. Xremote's performance on the X11perf trace was disappointing since the trace was very repetitive; HBX was able to recognize this regularity and achieve 216:1 compression on the this trace. Certain types of 3D applications generate similarly repetitive traffic since it is common to iterate on sending a transformation matrix and some geometry, with only the transformation matrix changing from iteration to iteration.

So far, we have examined only the average behavior of HBX. Figure 25 shows the instantaneous behavior of HBX across a ten second excerpt from the Mult trace. As you can see, the most important message type in this interval is MotionNotify, with about 30K bits, compressing to around 3K bits for a 10:1 ratio overall. The bits/sec bar chart at the top of Figure 25 shows uncompressed X traffic attaining a peak bandwidth of almost 80K bits/sec, while HBX compressed X traffic attains only about 10K bits/sec. Based on this plot, I believe that a mouse based X applications using HBX compression would be usable at the bit-rates available with cellular modems.

5.4.1 Application Startup

I gathered a trace (called startup) of a single user starting up an X session, doing a little bit of work, and repeating (with slightly different work each time). Each time the user started the X session, a terminal, a clock, a mailbox, and a CPU load monitor

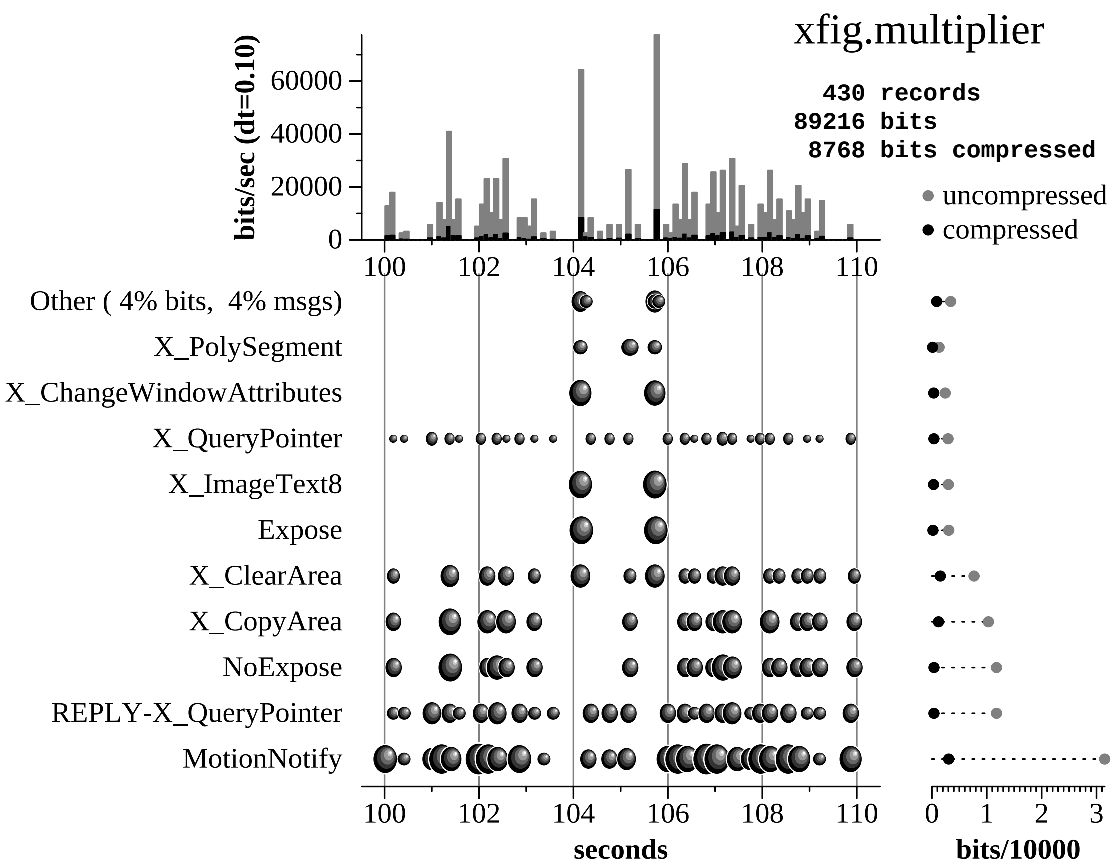


Figure 25: This graph is a ten second excerpt from the Mult trace, mostly showing mouse tracking. In the central area: The X axis indicates seconds since session startup, while the Y axis shows the types of the messages being sent. Each sphere represents a single message unless it is elongated vertically, in which case it represents multiple messages arriving nearly simultaneously. The volume of each sphere is proportional to the number of bits in the message(s) represented. The dash-dot plot to the right of the spheres shows the total number of bits for each type of packet, before and after compression. The bar chart above the spheres shows the number of bits per second as a function of time (before and after compression). The integration interval for time-wise plots is 100ms, the human interval of simultaneity.

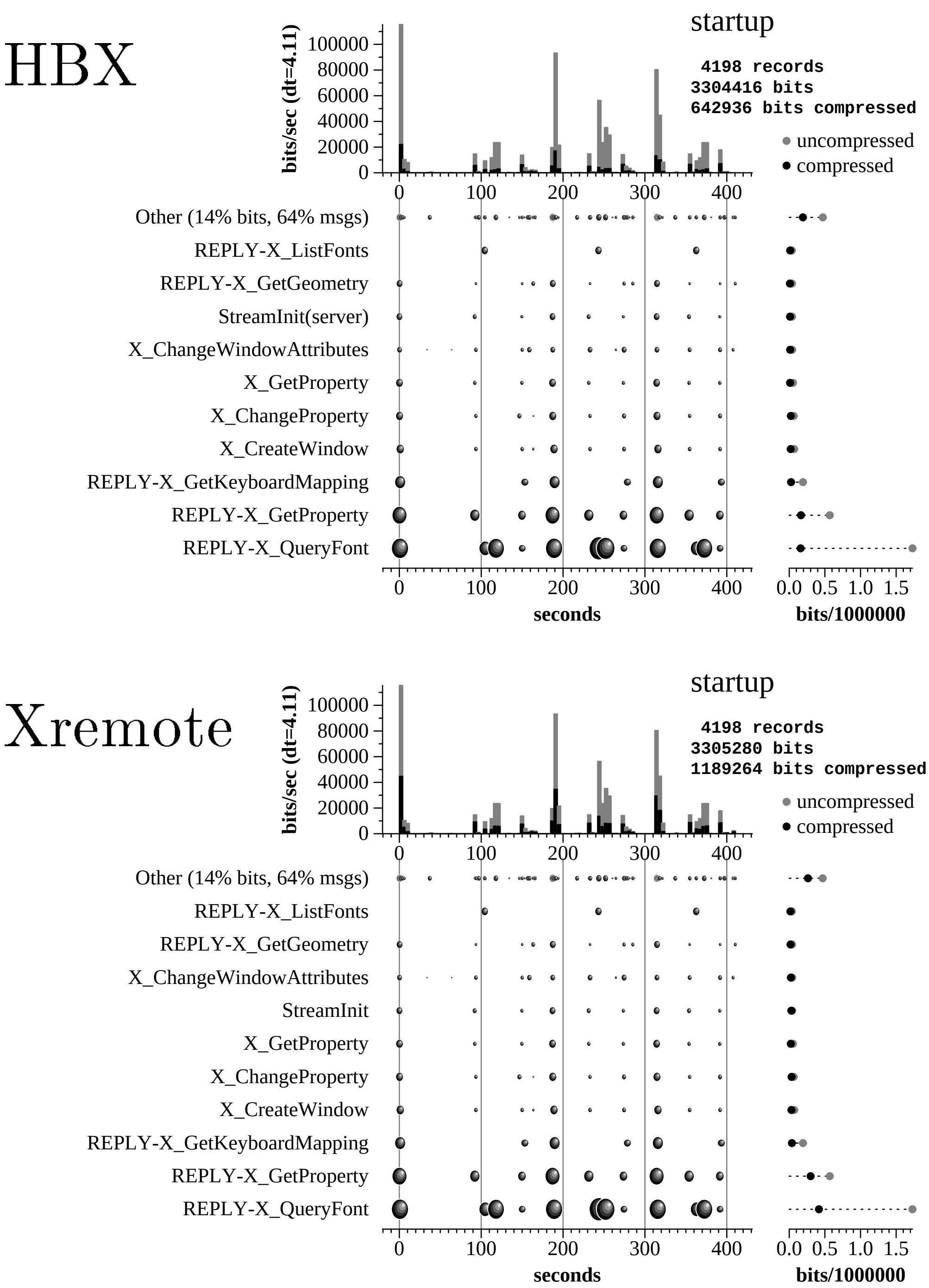


Figure 26: These graphs show a user starting up an X session, doing a little bit of work, and logging off, repeatedly. The compression algorithms were run on complete traces, but only packets connected with application startup are shown here. HBX achieves about 5:1 compression on these packets while Xremote achieves 2.8:1 compression.

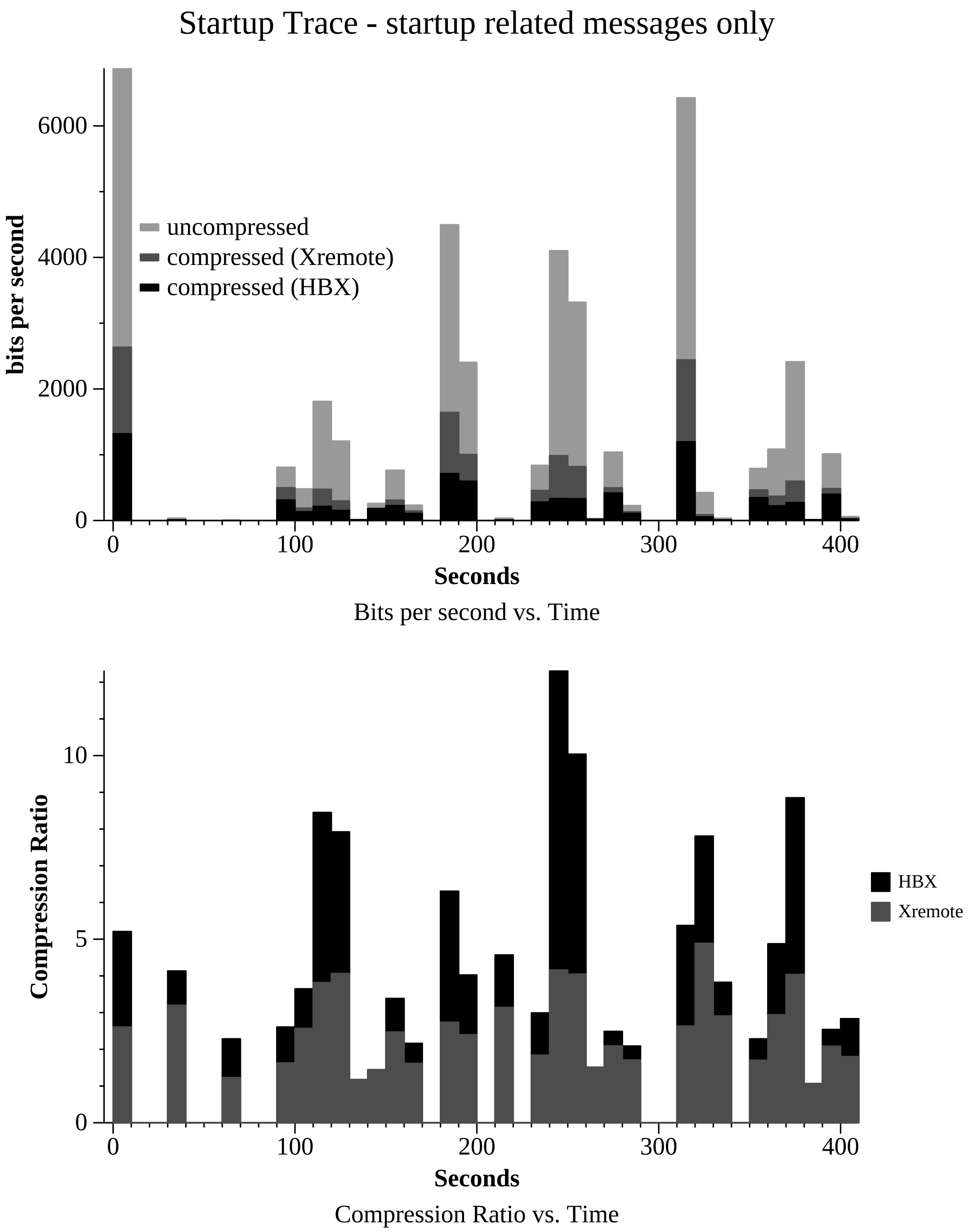


Figure 27: These graphs show a user starting up an X session, doing a little bit of work, and logging off, repeatedly. The compression algorithms were run on complete traces, but only packets connected with application startup were used to make this graph.

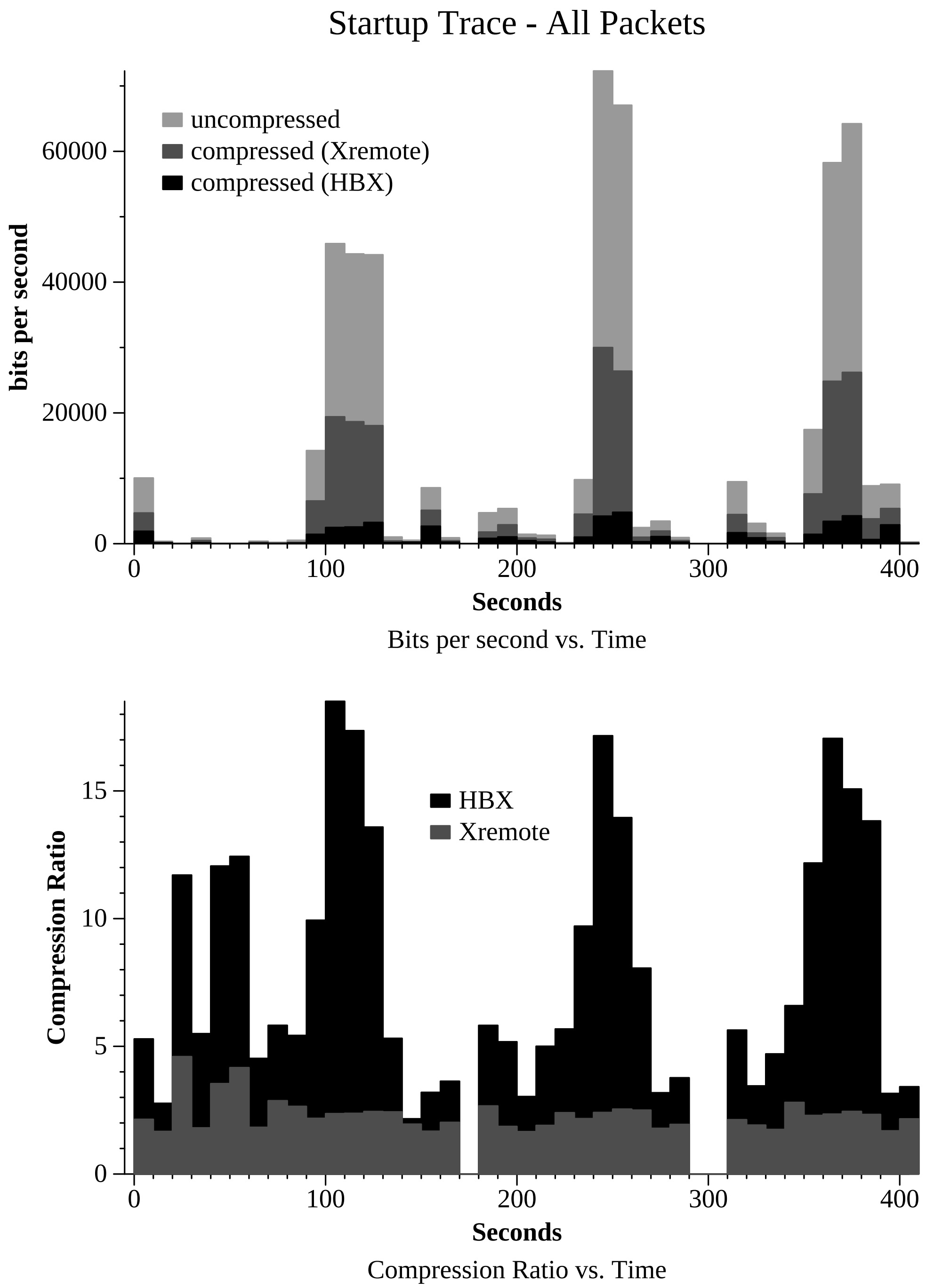


Figure 28: These graphs show a user starting up an X session, doing a little bit of work, and logging off, repeatedly. The effects of all of the messages are shown here.

were automatically started up (the user's default environment). This was to compare HBX's startup efficiency to Xremote's and to see how useful it might be to retain predictive models between sessions. The work in each case was either scanning some ASCII mail files, or previewing a document with ghostscript.

Figure 26 shows just the messages associated with application initialization (all geometry, imaging, and text messages deleted). There seem to be about 6 major startup events. HBX achieves about 5:1 compression across all of these events, while Xremote achieves about 2.8:1 compression across the same events. Both systems seem to compress a little better after the first event, but at this scale, the effect of learning seems minimal. Neither system has a large enough store to effectively remember much from one startup event to the next. The effect of retaining predictive models from one session to the next seems positive, but small.

Figures 27 and 28 show bandwidth and compression ratios for Xremote and HBX, for just startup packets and for the whole application, respectively. Overall, it appears that neither HBX nor Xremote are especially good or bad at application startup.

5.4.2 The Limits of Compression with HBX

Since HBX is amenable to optimization, it makes sense to ask what the limits of HBX compression are, and to measure HBX on this yardstick. Finding a reasonable upper bound for compression is easy for applications which only transfer text. For instance, *ghostscript* is a previewer for POSTSCRIPT files, which are usually derived from ASCII documents with embedded markup commands. It is reasonable to expect that the bitstream which ghostscript generates will not be compressed so that it is smaller than a compressed version of the original document with markup commands removed, as long as a good text compression algorithm is used on the original text.

The ghost2 trace is derived from a technical paper written using LaTeX. After the formatting commands have been removed with *detex*, the paper is 38625 bytes long. A simulated third order context based text compressor with lazy exclusions (STC) reduces this file to 11935 bytes or 2.47 bits per character, for a compression ratio of 3.23:1. *Ghostscript* generated 30976 X_PutImage messages in 1484196 bytes, essentially one for each printing glyph. There were 657 different glyphs from about

X_PutImage

field	bits	comment
opcode	0.01	
format	0.01	
length	1.89	proportional to the size of the glyph
drawable	0.01	
gcontext	0.01	
width	1.81	partially predicted by the length
height	0.08	predicted by length and width
x	2.52	predicted by prev x and prev width
y	1.50	predicted by prev y and (prev height - curr height)
pad	0.01	
depth	0.01	
image	3.00	predicted by prev 3 glyphs and width and height
total	10.86	

Table 13: Average number of bits generated for each field of the X_PutImage message by HBX in the ghost2 trace. HBX will average about 3 bits of message length information for messages this size, and messages are expressed in multiples of 8 bits, so the total number of bits sent for each message was 16.94. This is larger than the number of bits sent per character in the original source file because many of the characters in the original source file did not generate glyphs (space, newline, etc.).

X_ImageText8

field	bits	comment
opcode	0.01	
strlen	6.23	n chars in string
length	0.00	derived from strlen
drawable	0.01	
gcontext	0.21	
x	1.61	predicted by prev x
y	1.39	predicted by prev y
string	(23.53×2.16)	predicted by 3rd order context model
total	60.28	bits for 23.53 characters = 2.56 bits each

Table 14: The average number of bits generated for each field of the X_ImageText8 message by HBX in the text trace. The average string length was 23.53 characters. Characters were encoded in an average of 2.16 bits each. HBX adds about 8 bits for length to a message this size, and an average of 3.5 bits to pad to a byte boundary, leading to burdened cost of about 3.05 bits per transmitted character.

79 different printing characters in the source document. (The increase in the size of the character set presumably results from font changes.) HBX compressed these messages to 65630 bytes, or 13.59 bits per source character. From the point of view of X, this is 22.6:1 compression. From the point of view of the compressed source file, we have a 5.5:1 expansion. The limits of compression for this trace will be somewhere between these points.

Table 13 shows where the bits went in the X_putimage commands used to send glyphs in the ghost2 trace. Just under 11 bits were spent on the message, and just over 6 bits were spent on overhead: the length field for the packet, and padding packets to an integral number of bytes. If HBX grouped messages on bit boundaries, and associated a single length field with a large number of messages, it might achieve about 8.7 bits per source character on this trace.

The next place to look for further compression on this primitive might be carriage return prediction (which is probably the major source of uncertainty in x) and glyph-with-descender detection, (probably the major source of uncertainty in y). Other fields like *width, height* and *length* could be determined from the glyph, but since these fields are used to help predict the glyph, it is not clear how much rearranging or grouping these predictions would help.

Incidentally, there are systems which extract fonts from scanned images, and match marks in scanned text to characters in the scanned font. This is a much harder problem than the one I have addressed here because of the noise and registration issues associated with scanning. A recent system, which achieved 37.8:1 lossless compression on a library catalog scanned at 400dpi was described in [23]. (The high compression ratio is because of all of the white space in a scanned document. HBX's compression ratio is lower because its input has denser content.)

Another example of an application where it is easy to establish an upper bound for compression is simple text transfer. The text trace, which I am introducing for the first time here, was generated by *cat*-ing the source for HBX in a terminal window. The source has 270073 bytes. A Simulated Third order Compressor (STC) reduces this file to 58489 bytes, or 1.73 bits per character. *Xterm* generated 10823 X_ImageText8 commands, containing 254703 characters, using 446520 bytes, which

HBX compressed into 97612 bytes, for 2.89 bits per source character, for an expansion of 1.67:1 over a reasonable optimum.

Table 14 shows where the bits went in the X_ImageText8 messages used to draw the text in this trace. The string length field consumes over 6 bits per message but $-\log_2(23.53)^{-1} \approx 4.56$ so replacing the length field with an end of string character probably would not help much. The HBX length field and byte-boundary padding consume 11.5 bits per message, and could either be eliminated or amortized across multiple messages. HBX isn't compressing individual characters as well as STC by about 25%, possibly because STC does not have any model size or precision restrictions, and possibly because STC has a slightly different update policy.

In the ghostscript document previewer example, HBX has achieved a 5.5:1 expansion with respect to the compressed source text. We can see how to get to an expansion of only 3.5:1 over the compressed source with buffering, but further progress would probably require a deeper understanding of how characters are positioned. In the text example we are much closer to the compressed source text size with an expansion factor of only 1.67:1. Buffering would bring us to 1.39:1, but further improvements would require improvements in HBX's text compression, which might either require more storage, or non-obvious changes in the way predictive models are updated.

5.4.3 Further Opportunities for Optimization

Performance profiles help direct optimization by directing attention to the important parts of a large problem. To improve HBX's compression on the trace suite, I need to know which messages are important across the whole trace suite.

Table 15 shows the top twenty messages across the HBX trace suite, sorted according to the average percentage of compressed bytes in each trace. For instance, the X_PolyPoint message is responsible for about 69% of the compressed bytes in the x11perf trace, but is barely present in the 9 other traces, so it's overall contribution is 6.9%. Compression ratios were inverted, weighted according to their importance to the trace, and averaged. Compression ratios are averaged inverted, so that 100:1 compression and 1:1 compression average to 1.01:1 compression, not 50.5:1 compression.

Message Type	percentage of uncompressed bytes	percentage of compressed bytes	compression ratio
ALL	100.0%	100.0%	7.5:1
X_PutImage	20.6%	15.5%	16.6:1
X_PolyLine	10.9%	10.5%	5.1:1
X_PolySegment	8.9%	10.4%	6.4:1
X_PolyPoint	9.9%	6.9%	152.3:1
X_PolyFillRectangle	6.7%	6.9%	5.8:1
X_ImageText8	5.9%	5.9%	5.4:1
X_ClearArea	5.1%	5.2%	5.9:1
X_FillPoly	4.2%	5.0%	4.1:1
X_CopyArea	3.6%	3.7%	6.9:1
X_PolyText8	2.4%	3.4%	3.6:1
REPLY-X_GetProperty	0.4%	3.2%	1.6:1
MotionNotify	3.0%	1.9%	10.5:1
ButtonPress	0.9%	1.6%	3.2:1
ButtonRelease	0.9%	1.5%	3.2:1
NoExpose	4.1%	1.4%	19.0:1
X_PolyRectangle	1.2%	1.4%	4.7:1
KeyPress	0.9%	1.3%	3.6:1
Expose	1.6%	1.3%	4.4:1
X_ChangeGC	1.1%	1.2%	6.8:1
REPLY-X_GetImage	0.1%	0.8%	33.8:1

Table 15: These are the twenty messages generating the largest percentage of bytes across the HBX trace suite. Percentages were averaged across traces so that larger traces were not weighted more heavily than smaller traces. The compression ratios for each message were weighted according to the importance of the message in each trace.

The full formula for the average compression ratio for a message follows:

$$r_m = \frac{\sum_i cpct_{i,m}}{\sum_i cpct_{i,m} \cdot \frac{c_{i,m}}{u_{i,m}}}$$

where $cpct_{i,m}$ is the fraction of compressed bytes in trace i which were generated by message m (importance after compression), $c_{i,m}$ is the number of compressed bytes, and u_i is the number of uncompressed bytes.

The profile shows that time spent optimizing text, mouse motion, and especially image compression was well spent, but that for substantial improvements, the next target for further improvements across this trace suite should be geometry compression.

5.5 Resource Requirements

5.5.1 Compute Cycles

I measured how long it took HBX and my Xremote simulator to process each trace, so that I could see which program provides higher effective communications bandwidth in a given environment. The experiment was conducted on a DEC 3000 AXP 400 with a SPECint92 rating of 74.7. I chose this machine because it was readily available and seemed representative of the machines available in my department. Averaged across the trace suite, CPU limited Xremote accepts uncompressed bits at about 1Mbps, while CPU limited HBX accepts uncompressed bits at about 100Kbps. On average, Xremote can process input ten times as fast as HBX, but HBX's compression is over 3 times Xremote's.

The difference in speed between HBX and Xremote, and its magnitude are as expected because Xremote spends most of it's time in the LZW compression algorithm which uses a hash table to compress each input character in $O(1)$ time, while HBX searches for predictive models based on context, and implements predictive models using balanced binary trees, costing $O(\log n)$ time for each input symbol, where n is the maximum number of symbols stored (5000). I expect that HBX could be made quite a bit faster by using a hash table to locate predictive models. And, as mentioned above, it would be worthwhile to investigate new linear time data structures using dynamic tables to represent predictive models.

The CPU limited number of (compressed) output bits per second ($obps$) times the compression ratio (r) for a given compression system is the CPU limited maximum communication rate (C) through that system. If we include a maximum network rate n, we have

$$C = min(n, obps)r.$$

To find the point of equal performance, we set Xremote's communication rate equal to HBX's communication rate ($XC = HX$) (Note that this can only happen when the network is bottleneck for Xremote, and that the network cannot be a bottleneck for HBX at the crossover point, or HBX would outperform Xremote)

$$min(n, Xobps)Xr = min(n, Hobps)Hr$$

or

$$n \cdot Xr = Hobps \cdot Hr$$

or

$$n = Hobps\frac{Hr}{Xr}$$

where $Xobps$ is Xremote's output rate, $Hobps$ is HBX's output rate, Xr is Xremote's compression ratio, Hr is HBX's compression ratio, and n is the limiting network bandwidth at which both systems perform equally well.

Averaged across the trace suite, the tradeoff point on the test machine is reached when $n \approx 44Kbps$. However, this point increases linearly with CPU performance, and CPU performance is increasing by 54% per year [35]. Figure 29 illustrates this point by plotting the HBX-Xremote crossover point as a function of expected CPU performance for the next few years.

Given readily available 14.4Kbps modems, and a 74.7 SPECint CPU, HBX outperforms Xremote (and thus LZ78 based compression) across every trace. The breakeven point between HBX and Xremote is almost 100Kbps for document previewing. Using projected CPU performance improvements, HBX will outperform Xremote for every trace across 64Kbps ISDN connections before 1997. By 2000, HBX will outperform Xremote for every trace at rates up to 200Kbp.

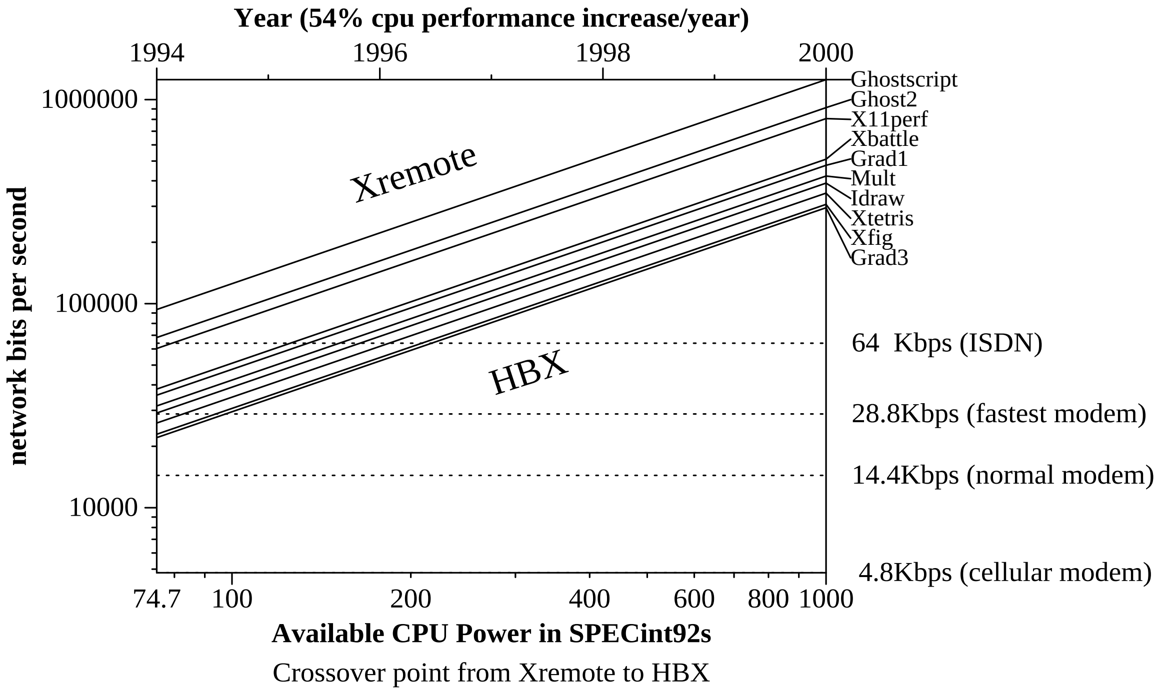


Figure 29: The X axis shows cpu performance extrapolated for the next 6 years according to the 54% increase per year of [35]. The machine used as the basis point for this extrapolation was the DEC 3000 AXP 400 with a SPECint92 rating of 74.7. The Y axis indicates available network bandwidth in bits per second. The data lines indicate the point at which HBX and Xremote would work equally well, for each application. In the region below the data lines, HBX provides a higher effective bit rate because it has better compression. In the region above the data lines, Xremote provides a higher effective bit rate because it is faster.

These figures do not include improvements in HBX, which are expected in both CPU performance and compression efficiency.

5.5.2 Memory

On a DEC Alpha with 8 byte pointers, it takes about 88 bytes to represent a model with an escape symbol and one other symbol in it. Symbols are mostly about 6 bytes long including tag and length byte, so the maximum storage that HBX could be expected to consume per symbol is about 100 bytes per symbol. The encoder and decoder each have 5000 symbols, so the maximum storage allocation for one end of

HBX (client side, or server side) should be about 1Mbyte on this 64 bit machine, and just over half that on a 32 bit machine.

In contrast, Xremote's LZ78 coder requires a hash table supporting upto 4096 entries, each of which require 8 bytes. A simple double hashing implementation with table of roughly 8192 entries requires 64K bytes. Therefore, the memory requirement for one end of Xremote, including both an encoder and a decoder, is about 128K bytes, or a quarter to an eighth of HBX's requirements.

LZ78 is currently amenable to hardware implementation. HBX, as currently implemented, requires far too much memory for a hardware implementation, but is within reasonable bounds for software implementations.

5.6 Remaining Opportunities for Compression

HBX currently underuses information available from the client side stream in compressing the server side stream. A significant remaining opportunity for compression in HBX is to use context from queries in predicting replies. Server replies do not include the types of their generating queries, so information from the client side is used to tag replies with the types of their queries. This more specific tag helps in compression when the type tag is used together with an offset to predict a word in a reply. However, more is possible: For some queries, such as X_ListFonts and X_GetProperty, there are parameters in the query which would completely specify the reply. If these parameters were used in the context used to predict words in the reply to a specific query, then after one reply to the query had been seen, future replies to that query could be transmitted in just a few bits.

5.7 Summary

HBX exploits the separation of predictive model and coder in arithmetic compression to apply a custom compression technique to each field in each message in a graphics protocol, within a unified framework, and without wasting any bits in the transition from field to field. To the extent that the predictive models used accurately encapsulate all of the available knowledge about the fields in question, this technique is optimal. In practice, HBX achieved 7.5:1 compression over this suite of representative traces, over 3 times the 2.4:1 compression achieved by the LZW-based Xremote protocol.

While HBX is more effective than Xremote given commonly available (fast) workstations and a common slow network connection such as a phone line, HBX is still CPU intensive. I believe that HBX could be optimized to run 2-4 times as fast as it does now, but it will probably never run as fast as the LZW based Xremote system. Aggressive compression systems like HBX will become even more appropriate as computers become faster.

Chapter 6

Discussion

6.1 Summary

In Chapter 2 I presented a method for capturing and analyzing the X protocol. We saw that only a small number of message types account for nearly all of the traffic. We discovered that most X messages are small, less than 32 bytes. Even the groups of X messages that could appear together in a network packet are relatively small, usually less than 100 bytes. The profiling and analysis tools developed in this chapter proved useful throughout the rest of the thesis.

In Chapter 3 we studied the performance of the widely used Xremote protocol. Xremote uses a delta compressor and a variant of the LZ78 text compression algorithm hooked together sequentially to compress the X protocol stream. My experiments demonstrated that the redundancy removed from the input stream by Xremote's delta compressor is a subset of what the LZ78 algorithm recognizes, and that the two compression stages are not complementary. I also demonstrated that Xremote compresses geometry and images poorly, which is expected behavior for a text compression protocol. Overall, Xremote achieves a compression of about 2.4:1 across my trace suite.

Chapter 4 introduces a new technique for compression of structured data, and its implementation. In this technique, each field in a data structure is preconditioned so that it can be predicted by a frequency based predictive model, and then encoded using a predictive model selected using a custom set of contexts. This technique follows directly from information theory, but seems not to have been used before.

Finally, in Chapter 5 I describe the Higher Bandwidth X (HBX) system, which uses the techniques of Chapter 4 to compress the X protocol. By applying default models to the fields of the three types of X messages, and specialized models to the fields of just 10 X messages, HBX achieved about 7.5:1 compression across my trace suite, more than 3 times the performance of Xremote.

6.2 Discussion

Structured Data Compression (SDC) introduces the concept of optimizing compression performance. Whenever a program using SDC performs poorly because of insufficient compression, the application of profiling techniques and a little time and thought will usually generate better predictive models resulting in a better coding. This is exactly analogous to the process of improving the performance of a computer program: when a program performs poorly, a profile is used to isolate a set of resource intensive functions. The programmer examines these functions, and tries to eliminate waste, or introduces better algorithms to improve performance. This process is repeated until the expected performance improvement resulting from another profile and re-code cycle is not worth the cost of the cycle. (It is extremely rare to find a sequence of more than a few computer instructions that cannot be improved on. See [27].) As a relatively new program compressing a complex and important protocol, HBX's compression performance will probably continue to improve for some years.

The possibility of optimization afforded by SDC is a marked improvement over monolithic single model compression algorithms which have a fixed level of performance. With SDC, if you need more compression, you can usually spend more money and get it. With LZ78, if you need more compression, you are stuck.

The performance difference between SDC-based HBX and LZ78-based Xremote will be striking to real users. For instance, a user waiting for X to transmit a single page of the (densely formatted) document previewed in the ghost2 trace through a 14.4Kbps connection would wait about 117 seconds without compression. With Xremote, the same user would wait about 47 seconds per page. With HBX, the user would wait less than 6 seconds per page. Document previewing has been one of the most painful operations for remote users; with HBX's near eightfold speedup, previewing across a phone line will be fun.

CPU performance remains a concern, but it seems likely that over time SDC and thus HBX performance could be improved by a factor of 2-4 by switching over to faster data structures. With this algorithmic performance increase, taken together with the expected continued exponential improvement in CPU speed, performance should cease to be an issue even for small portable machines in the near future. Even now there are laptop computers with faster CPUs than the one I used for development.

The main contribution of HBX has been the introduction of a conceptual framework for compressing structured data. Some work of a similar flavor has been done on compressing Pascal programs [8, 25], but the leap from the special case of Pascal to the general case of structured data was not made at that time.

6.3 Future Work

The SDC technique should be applicable to a wide variety of compression and network bandwidth problems, for instance RPC across slow networks. There is no reason why an RPC stub generator could not generate code to compress arguments as they are being marshalled. In fact, the RPC stub generator usually has type information available, which should make selection of predictive models easier. In an important application, it might be worthwhile for a programmer to map predictive models or transformations onto selected arguments or types of arguments by hand, as was done with Xremote.

The CSLIP (compressed serial line IP) protocol [22], was generated by experimentally determining which IP fields change often, and which IP fields change only occasionally. Applying the adaptive statistical techniques used in HBX to the IP protocol should result in a considerably leaner protocol than the static CSLIP scheme.

Machine code is difficult for most coders to compress even though compilers generate very stylized code because they cannot see the regularity. A machine code compressor using SDC would be able to take advantage of common instruction sequences, even though the operands might be different, locality of register usage, for instance, the fact that a recently written register is likely to be read soon, and for some architectures, the increased likelihood of a NOOP in the shadow of a branch or load instruction. It isn't clear now what kind of compression is available, but these techniques should be able to get whatever is available to a greedy algorithm.

Automatic generation of SDC models would be very interesting, because at present, the task is difficult and time consuming. Although the general problem is intractable, it should be possible to make a list of likely models and preconditioning filters, and statically determine which models and preconditioning filter are appropriate for each field in a data set.

From a research point of view, the most useful next step would be do develop a compression testbed, so that researchers in all different fields could apply SDC to their data. Goals would be ease of use, speed, and reduction of memory requirements while preserving optimal coding and improving overheads.

A parallel goal is to distribute HBX itself to see how it performs in practice, and to see what problems appear. HBX works as two communicating processes connected by sockets right now. There is no reason why the TCP connection could not be a CSLIP/PPP connection across a serial telephone or even cellular connection.

Appendix A

Taxonomy of X Messages

X messages are categorized functionally. We have a two level hierarchy. Top level categories are boxed. Second level categories begin with a $\cdot$ and are on the same line with a message. Messages beginning with "X_" are client requests or queries. Other messages are server events (such as "KeyPress"), "StreamInit" or "Error". "StreamInit" is the connection startup sequence. Server replies to queries are categorized with the queries, and are therefore not listed.

Class	Message
GEOMETRY	
·Geometry	X_ClearArea
	X_FillPoly
	X_PolyArc
	X_PolyFillArc
	X_PolyFillRectangle
	X_PolyLine
	X_PolyPoint
	X_PolyRectangle
	X_PolySegment
IMAGE	
·Cursor	X_CreateCursor
	X_CreateGlyphCursor
	X_FreeCursor
	X_QueryBestSize
	X_RecolorCursor
·Image	X_CopyArea
	X_CopyPlane
	X_GetImage
	X_PutImage

INPUT

·Input	ButtonPress
	ButtonRelease
	EnterNotify
	FocusIn
	FocusOut
	KeyPress
	KeyRelease
	LeaveNotify
	MotionNotify
·Mouse Movement	X_GetMotionEvents
	X_QueryPointer
	X_TranslateCoords
·Pointer	X_ChangePointerControl
	X_GetPointerControl
	X_GetPointerMapping
	X_SetPointerMapping
	X_WarpPointer

$\boxed{\text{MISC}}$	
·Access	X_ChangeHosts
	X_ListHosts
	X_SetAccessControl
·Bell	X_Bell
·Client Control	X_KillClient
	X_SetCloseDownMode
·Error	Error
·Extension	Extension-request
	X_ListExtensions
	X_QueryExtension
·Ipc	ClientMessage
	ColormapNotify
	SelectionClear
	SelectionNotify
	SelectionRequest
	X_ConvertSelection
	X_GetSelectionOwner
	X_SendEvent
	X_SetSelectionOwner
·Noop	X_NoOperation
·Screen Saver	X_ForceScreenSaver
	X_GetScreenSaver
	X_SetScreenSaver

TEXT	
·Font	X_CloseFont
	X_GetFontPath
	X_ListFonts
	X_ListFontsWithInfo
	X_OpenFont
	X_QueryFont
	X_QueryTextExtents
	X_SetFontPath
·Text	X_ImageText16
	X_ImageText8
	X_PolyText16
	X_PolyText8
WINDOW	
·Color	X_AllocColor
	X_AllocColorCells
	X_AllocColorPlanes
	X_AllocNamedColor
	X_CopyColormapAndFree
	X_CreateColormap
	X_FreeColormap
	X_FreeColors
	X_InstallColormap
	X_ListInstalledColormaps
	X_LookupColor
	X_QueryColors
	X_StoreColors
	X_StoreNamedColor
	X_UninstallColormap

·Focus	X_GetInputFocus
	X_SetInputFocus
·Grab	X_AllowEvents
	X_ChangeActivePointerGrab
	X_GrabButton
	X_GrabKey
	X_GrabKeyboard
	X_GrabPointer
	X_GrabServer
	X_UngrabButton
	X_UngrabKey
	X_UngrabKeyboard
	X_UngrabPointer
	X_UngrabServer
·KeyBoard	X_ChangeKeyboardControl
	X_ChangeKeyboardMapping
	X_GetKeyboardControl
	X_GetKeyboardMapping
	X_GetModifierMapping
	X_QueryKeymap
	X_SetModifierMapping
·Property	PropertyNotify
	X_ChangeProperty
	X_DeleteProperty
	X_GetAtomName
	X_GetProperty
	X_InternAtom
	X_ListProperties
	X_RotateProperties

·Window	CirculateNotify
	CirculateRequest
	ConfigureNotify
	ConfigureRequest
	CreateNotify
	DestroyNotify
	Expose
	GraphicsExpose
	GravityNotify
	KeymapNotify
	MapNotify
	MapRequest
	MappingNotify
	NoExpose
	ReparentNotify
	ResizeRequest
	StreamInit
	UnmapNotify
	VisibilityNotify
	X_ChangeGC
	X_ChangeSaveSet
	X_ChangeWindowAttributes
	X_CirculateWindow
	X_ConfigureWindow
	X_CopyGC
	X_CreateGC
	X_CreatePixmap
	X_CreateWindow
	X_DestroySubwindows
	X_DestroyWindow
	X_FreeGC

X_FreePixmap

X_GetGeometry

X_GetWindowAttributes

X_MapSubwindows

X_MapWindow

X_QueryTree

X_ReparentWindow

X_SetClipRectangles

X_SetDashes

X_UnmapSubwindows

X_UnmapWindow

Appendix B

Models For Each X Message

This appendix lists the predictive models used for some of the most important X messages.

Each message is preceeded by a compressed tag, which fully determines the message opcodes. Tag values are predicted using a first order model indexed by the previous tag value, so that tags in common sequences of messages are well compressed.

Tables 16 to 28 show the predictive models used by each field in each X message. The first three tables show the default models used for each of the three types of X messages (requests, events, and errors) when no specialized models exist. The rest of the tables show the specialized models for those messages which I optimized. The non-generic messages are listed in the order in which they appear in the list of the top twenty messages in Section 5.4.3.

In these tables, the "Preconditioning" column indicates how values were preconditioned before being predicted. The first example is for a sequence number "$\Delta(PrevSeqNo)$" which means that at the encode end, the sequence number from the previous message is substracted from the sequence number of the current message, and the difference is predicted encoded and transmitted. At the decode end, the difference is received, decoded, and added to the previous sequence number. Other Δ fields are handled in exactly the same way.

The "Predictive Model" column indicates which contexts were used to index models for prediction. A common context is "Tag, Offset". This context implies that a hierarchical model is instanced, where the top-level (most specific) model is indexed by both the type of the message and the offset of the field in the model. If the value is not found in this model and an escape is transmitted, the next model is indexed by just the type of the message. If an escape is transmitted in this context as well, the value is broken into bytes and predicted using a model which makes zero order predictions on byte values, and which is constrained to include all byte values. Symbols are never deleted from this model-of-last-resort.

Analogous hierarchical models are built for other lists of context specifiers. Here is an explanatory list of all the context specifiers used (in order of introduction):

Tag The message type.

Offset The byte offset of the field in the message.

Width The value of the width field from the current message.

Length The value of the length field from the current message.

XCOORD A constant used to keep X coordinates separate from other types of data.

YCOORD As XCOORD, but for Y coordinates.

RXCOORD A constant used to keep relative X coordinates separate from other types of data.

RYCOORD As RXCOORD, but for Y coordinates.

Prev 3 X Coords This is equivalent to a list of the three coordinates in that a hierarchical model is built with a model indexed by all of these at the top level, then just two of them, then just one and so on.

Prev 3 Y Coords As above but for Y coordinates.

RWIDTH A constant used to keep rectangle widths separate from other types of data.

RHEIGHT A constant used to keep rectangle heights separate from other types of data.

S_{n-1} **etc.** Previous characters in the current string

The models are listed here.

Generic Request

Field	Preconditioning	Predictive Model
Opcode		Tag, Offset
Data		Tag, Offset
Length		Tag, Offset
Data D_n		Tag, Offset

Table 16:

Generic Error

Field	Preconditioning	Predictive Model
Error		Tag, Offset
Error Code		Tag, Offset
Sequence Number	$\Delta(PrevSeqNo)$	Tag, Offset
Resource ID		Tag, Offset
Minor Code		Tag, Offset
Major Code		Tag, Offset
Pad		

Table 17:

Generic Event

Field	Preconditioning	Predictive Model
Event type		Tag, Offset
Detail		Tag, Offset
Sequence Number	$\Delta(PrevSeqNo)$	Tag, Offset
Data $D_{n=1 \cdots 7}$		Tag, Offset

Table 18:

X_PutImage

Field	Preconditioning	Predictive Model
Opcode		Tag, Offset
Format		Tag, Offset
Length		Tag, Offset
Drawable		Tag, Offset
Graphics Context		Tag, Offset
Image Width		Tag, Length
Image Height		Tag, Width, Length
Dst-X	$\Delta(PrevX + PrevWidth)$	Tag, Offset
Dst-Y	$\Delta(PrevY$ $+PrevHeight - Height)$	Tag, Offset
Left pad		Tag, Offset
Depth		Tag, Offset
Image		Hash(Width, Height), Prev 3 Glyphs *see caption*

Table 19: The model used for images depends on the size and depth of the image. Small bi-level images (width and height not more than 64) use a hash of the width and height fields and the previous three glyphs to predict the entire image as an atomic entity (as shown in the table). Larger bi-level images are predicted pixel by pixel using some of the previous pixels from the same image for prediction as shown in Figure 22. All of the images in the trace suite were bi-level.

X_PolyLine

Field	Preconditioning	Predictive Model
Opcode		Tag, Offset
Coord mode		Tag, Offset
Length		Tag, Offset
Drawable		Tag, Offset
Graphics Context		Tag, Offset
Point X_0		XCOORD, Tag
Point Y_0		YCOORD, Tag
Point $X_{n>0}$	$\Delta(X_{n-1})$	RXCOORD, Prev 3 X coords
Point $Y_{n>0}$	$\Delta(Y_{n-1})$	RYCOORD, Prev 3 Y coords

Table 20:

X_PolySegment

Field	Preconditioning	Predictive Model
Opcode		Tag, Offset
Pad		
Length		Tag, Offset
Drawable		Tag, Offset
Graphics Context		Tag, Offset
Point X_0		XCOORD, Tag
Point Y_0		YCOORD, Tag
Point $X_{n>0}$	$\Delta(X_{n-1})$	RXCOORD, Prev 3 X coords
Point $Y_{n>0}$	$\Delta(Y_{n-1})$	RYCOORD, Prev 3 Y coords

Table 21:

X_PolyFillRectangle

Field	Preconditioning	Predictive Model
Opcode		Tag, Offset
Pad		
Length		Tag, Offset
Drawable		Tag, Offset
Graphics Context		Tag, Offset
Rect Origin X_0		XCOORD, Tag
Rect Origin Y_0		YCOORD, Tag
Rect Width		RWIDTH, Tag
Rect Height		RHEIGHT, Tag
Rect Origin $X_{n>0}$	$\Delta(X_{n-1})$	RXCOORD, Tag
Rect Origin $Y_{n>0}$	$\Delta(Y_{n-1})$	RYCOORD, Tag

Table 22:

X_ImageText8

Field	Preconditioning	Predictive Model
Opcode		Tag, Offset
String len		Tag, Offset
Length	Fully determined by string len	
Drawable		Tag, Offset
Graphics Context		Tag, Offset
X	$\Delta(PrevX)$	Tag, Offset
Y	$\Delta(PrevY)$	Tag, Offset
String S_0		Tag, Offset
String S_1		Tag, S_{n-1}
String S_2		Tag, S_{n-1}, S_{n-2}
String $S_{n>2}$		Tag, S_{n-1}, S_{n-2}, S_{n-3}

Table 23:

X_FillPoly

Field	Preconditioning	Predictive Model
Opcode		Tag, Offset
Pad		
Length		Tag, Offset
Drawable		Tag, Offset
Graphics Context		Tag, Offset
Shape		Tag, Offset
Coord Mode		tag, Offset
Pad		
Point X_0		XCOORD, Tag
Point Y_0		YCOORD, Tag
Point $X_{n>0}$	$\Delta(X_{n-1})$	RXCOORD, Prev 3 X coords
Point $Y_{n>0}$	$\Delta(Y_{n-1})$	RYCOORD, Prev 3 Y coords

Table 24:

	X_CopyArea	
Field	Preconditioning	Predictive Model
Opcode		Tag, Offset
Pad		
Length		Tag, Offset
Src-Drawable		Tag, Offset
Dst-Drawable		Tag, Offset
Graphics Context		Tag, Offset
Src-X		Tag, Offset
Src-Y		Tag, Offset
Dst-X	$\Delta(Src - X)$	Tag, Offset
Dst-Y	$\Delta(Src - Y)$	Tag, Offset
Width		Tag, Offset
Height		Tag, Offset

Table 25:

	X_PolyText8	
Field	Preconditioning	Predictive Model
Opcode		Tag, Offset
Pad		
Length		Tag, Offset
Drawable		Tag, Offset
Graphics Context		Tag, Offset
X	$\Delta(PrevX)$	Tag, Offset
Y	$\Delta(PrevY)$	Tag, Offset
String S_n		Tag, S_{n-1}, S_{n-2}, S_{n-3}

Table 26:

	MotionNotify	
Field	Preconditioning	Predictive Model
Event type		Tag, Offset
detail		Tag, Offset
Sequence Number	$\Delta(PrevSeqNo)$	Tag, Offset
Time Stamp	$\Delta(PrevTimestamp)$	Tag, Offset
Root window		Tag, Offset
Event Window		Tag, Offset
Child Window		Tag, Offset
Root X	$\Delta(2 * Prev(RootX)$	
	$-Prev(Prev(RootX)))$	Tag, Offset
Root Y	$\Delta(2 * Prev(RootY)$	
	$-Prev(Prev(RootY)))$	Tag, Offset
Event X	$\Delta(RootX)$	Tag, Offset
Event Y	$\Delta(RootY)$	Tag, Offset
Key Button Mask		Tag, Offset
Same Screen		Tag, Offset
Pad		

Table 27:

	NoExpose	
Field	Preconditioning	Predictive Model
Event type		Tag, Offset
detail		Tag, Offset
Sequence Number	$\Delta(PrevSeqNo)$	Tag, Offset
Minor Opcode		Tag, Offset
Major Opcode		Tag, Offset
Pad		

Table 28:

Bibliography

[1] Adobe Systems Incorporated, "POSTSCRIPT language reference manual," Addison-Wesley Publishing Company, Inc. 1985

[2] Andrews, Edmund L., "A. T. & T. sees the future in games," The New York Times, Business Day, pages D1 and D5, July 29, 1993

[3] Arden, Michelle J., James Gosling, and David S. H. Rosenthal, "The NeWS book : an introduction to the Networked extensible Window System," Springer Verlag, 1989.

[4] Baker, Mary G., John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John L. Osterhout, "Measurements of a distributed file system," Proceedings of the Thirteenth ACM Symposium on Operation System Principles, October 1991, pp 198-212

[5] Bell, Timothy C., John G. Cleary, and Ian H. Witten, "Text Compression," Prentice Hall, Englewood Cliffs, NJ, 1990

[6] Blaze, Matt, "NFS Tracing by Passive Network Monitoring," Proceedings of the 1992 USENIX Winter Technical Conference.

[7] Card, Stuart K, Thomas P Moran, and Allen Newell, "The psychology of human-computer interaction," Hillsdale, N.J., L. Erlbaum Associates, 1983

[8] Cameron, R.D., "Source encoding using syntactic information source models," IEEE Transactions on Information Theory. (34)4 pp 843-850, July 1988

[9] Cleary, J. and I. Witten, "Data compression using adaptive coding and partial string matching," IEEE Trans. Communications, COM-32, No 4., 396-402 (1984)

[10] Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest. "Introduction to algorithms," The MIT Press, Cambidge MA and McGraw-Hill Book Company, New York, 1990

[11] Cornelius, David, "XRemote: a serial line protocol for X" 6th Annual X Technical Conference, Boston, MA, 1992

[12] Danskin, John and Pat Hanrahan, "Profiling the X Protocol," *1994 SigMetrics conference on measurement and modeling of computer systems.* Full paper in Technical Report CS-TR-442-94, Department of Computer Science, Princeton University, Princeton, NJ, January 1994.

[13] Danskin, John and Pat Hanrahan, "Compression Performance of the Xremote Protocol," *1994 Data Compression Conference.* Full paper in Technical Report CS-TR-441-94, Department of Computer Science, Princeton University, Princeton, NJ, January 1994.

[14] Droms Ralph and Wayne R. Dyksen, "Performance measurements of the X window system communication protocol," *Software Practice and Experience* vol. 20, no. S2, May 1991.

[15] Foley, J. D., and A. Van Dam, "Fundamentals of Interactive Computer Graphics," Addison Wesley, 1982, p 180

[16] Fulton Jim, and Chris Kent Kantarjiev, "An update on low bandwidth X (LBX)," Proceedings of the 7th Annual X Technical Conference, January 1993, O'Reilly and Associates.

[17] Gettys, James, Philip L. Karlton, and Scott Mcgregor, "The X Window System, Version 11," *Software Practice and Experience* vol. 20(S2), S2/35-S2/67, October 1991

[18] Griswold, Ralph E., and Madge T. Griswold, "The icon programming language," Prentice Hall, Englewood Cliffs NJ, 1990

[19] Guazzo, M., "A general minimum redundancy source coding algorithm," IEEE Trans. Information Theory, IT-26 (1), 15-25, January 1980.

[20] Held, Gilbert. "The Complete Modem Reference," John Wiley & Sons, Inc. 1991

[21] Hennessy, John L., and David A. Patterson, "Computer architecture: a quantitative approach," Morgan Kaufmann Publishers, Inc., San Mateo California, 1990

[22] Jacobson, V., "Compressing TCP/IP headers for low-speed serial links," SRI International, Menlo Park, CA, Feb. 1990., RFC-1144

[23] Inglis, Stuart, and Ian H. Witten, "Compression-based template matching," Proceedings 1994 Data Compression Conference, March 29-31 Snowbird Utah pp 106-115.

[24] H. Kalmus, D. B. Fry and P. Denes, "Effects of Delayed Visual Control on Writing, Drawing and Tracing," Language Speech, 1960, v3, pp 96-108

[25] Katajainen, J., M., Penttonen, and J., Teuhola, "Syntax-directed compression of program files," Software-Practice and Experience, 16(3), 269-276

[26] Manasse, Mark, personal communication, Jan 93.

[27] Massalin, Henry, "Superoptimizer — a look at the smallest program," Second International Conference on Architectural Support for Programming Languages and Operating Systems, pp 122-126, 1987

[28] McConnell, Kenneth R., Dennis Bodson, and Richard Schaphorst, "FAX: digital facsimile technology and applications," Second Edition, Artech House, Boston, 1992

[29] Moffatt, Alistair, "Linear time adaptive arithmetic coding," Trans. Information Theory, Vol 36, No. 2, March 1990

[30] Moffatt, A., "A note on the PPM data compression algorithm," Research Report 88/7, Department of Computer Science, University of Melbourne, Parkville, Victoria, Australia.

[31] Moffatt, Alistair, "Word-based text compression," Software-Practice and Experience, Vol. 19(2) 185-198 February 1989

[32] Netravali, Arun, N. and Barry G. Haskell "Digital pictures, representation and compression," Plenum Press, New York, 1988

[33] Ousterhout, John K., Hervè Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson "A trace-driven analysis of the UNIX 4.2 BSD file system," Proceedings of the Tenth ACM Symposium on Operating Systems Principles December 1985, pp 15-24

[34] Papadakis, Thomas, "Skip lists and probabilistic ananalysis of algorithms," PhD. Thesis in Computer Science, University of Waterloo, Ontario Canada, 1993

[35] Patterson, David, A., and John L. Hennessy, "Computer organization & design: The hardware/software interface" Morgan Kaufmann Publishers, San Mateo California, 1994.

[36] Peterson, James L., "XSCOPE: A debugging and performance tool for X11" Information Processing 89: Proceedings of the IFIP 89 Congress, August 1989, pp 49-54

[37] Pike R. "The blit: a multiplexed graphics terminal," AT&T Bell Laboratories Technical Journal, Vol. 63, No. 8, pp 1607-1631, October 1984

[38] Pike R. Personal communication, Fri Jul 22 01:51:31 1994.

[39] Postel J. Ed., "Internet Protocol Specification," SRI International, Menlo Park, CA, Sept. 1981. RFC-791

[40] Postel J. Ed., "Transmission Control Protocol Specification," SRI International, Menlo Park, CA, Sept. 1981. RFC-793

[41] Postel J., "User Datagram Protocol," SRI International, Menlo Park, CA, Aug. 1980. RFC-768

[42] Rissanen, J. J., and G. G. Langdon, "Arithmetic coding," IBM J. Research and Development, 23(2), 149-162, March 1979.

[43] Romkey, J., "A Nonstandard for Transmission of IP Datagrams Over Serial Lines," SRI International, Menlo Park, CA, June 1988, RFC-1055

[44] Rost, Randi J., Jeffrey D. Friedberg, and Peter L. Nishimoto "PEX: a network-transparent 3D graphics system," IEEE Computer Graphics and Applications, v9, n4, July 1989, pp 14-26

[45] Rubin, Paul and Jay Fenlason, "Gawk - pattern scanning and processing language," man page for gawk, Free Software Foundation, 1993, ftp from prep.ai.mit.edu

[46] Rubin, F., "Arithmetic stream coding using fixed precision registers," IEEE Trans. Information Theory, IT-25 (6), 672-675, November 1979.

[47] Scheifler Robert W., "The X Window System Protocol," M.I.T. Laboratory for Computer Science. 1988.

[48] Scheifler, Robert W. and Jim Gettys, "The X Window System," *Transactions on Graphics* 5(2), 79-109, April 1986, and *Software Practice and Experience* vol. 20(S2), S2/5-S2/34, October 1991

[49] Shannon, C. E., "A mathematical theory of communication," Bell System Technical Journal, 27, 398-403, July 1948

[50] Simpson, W. A. "Point-to-Point Protocol (PPP) for the transmission of multi-protocol datagrams over point-to-point links," SRI International, Menlo Park, CA, May 1992. RFC-1331

[51] Storer, James A., "Data compression: methods and theory," Computer Science Press, Rockville Maryland, 1988.

[52] Thomas, S. W., J. McKie, S. Davies, K. Turkowski, J. A. Woods, and J. W. Orost. "Compress (version 4.0) program and documentation," available from joe@petsd.uucp, 1985.

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

[54] Ziv, J. and Lempel, A. "Compression of individual sequences via variable-rate coding," IEEE Trans. Information Theory, IT-24 (5), 530-536, September 1978