An Execution Profiler for Window-oriented Applications

Aloke Gupta and Wen-Mei W. Hwu

Center for Reliable and High-performance Computing
Coordinated Science Laboratory
University of Illinois, Urbana-Champaign
1308 W. Main Street
Urbana, Illinois 61801
hwu@crhc.uiuc.edu

SUMMARY

Execution profiles are important in analyzing the performance of computer programs on a given computer system. However, accurate and complete profiles are difficult to arrive at for programs that follow the client-server model of computing, as in the popular X Window System. In X Window applications, considerable computation is invoked at the display server and this computation is an important part of the overall execution profile. The profiler presented in this paper generates meaningful profiles for X Window applications by estimating the time spent in servicing the messages in the display server. The central idea is to analyze a protocol-level trace of the interaction between the application and the display server and thereby construct an execution profile from the trace and a set of metrics about the target display server. Experience using the profiler for examining bottlenecks is presented.

Key Words: execution profiling, window systems, distributed processing, client-server computing, X Window System, protocol-level profiling.

INTRODUCTION

Execution profiles are an important aid in analyzing the performance of computer programs on a given computer system [1, 2]. Such profiles are helpful in providing information about the dynamic, or run-time, behavior of the program. This run-time information can lead to insights

[0]To appear in *Software Practice & Experience*

about the performance bottlenecks in a program, which allows the programmers to better focus their efforts when tuning the program performance.

However, accurate and complete profiles are difficult to arrive at for programs that follow the *client-server* model of computing. In this model, which is followed by programs in the X Window System [3, 4], the *client* programs request various services from *servers* by exchanging messages with them. In the client program, the routines that invoke the computation at the server are merely *stub* routines that send the appropriate request message to the server. The request may invoke substantial computation at the server, but the execution time of this computation may not be reflected in the profiled execution time of the stub routine.

This paper describes **Xprof**, an execution profiler for X Window applications. The X Window System follows the client-server paradigm. A *display server*, the *X server*, manages the actual display hardware and controls access to the graphics and windowing functions on the display. It also receives the user input from the mouse and keyboard. The application programs, or *clients*, achieve graphical and windowing functions by means of high-level messages exchanged with the display server by following a network protocol [5]. The display server alone has access to the actual display hardware and renders the high-level requests on it. The interaction between the client and server programs is network transparent in the sense that the communication protocol is followed even when the client program runs on the same processor host as the display server.

Distributed systems have come into widespread use recently [6, 7]. Such systems often consist of groups of autonomous computers, or *nodes*, connected together over a local area network, or *LAN*. The node computers often perform dedicated specialized tasks, but are also flexible enough to share different tasks among themselves. As shown in Figure 1, the

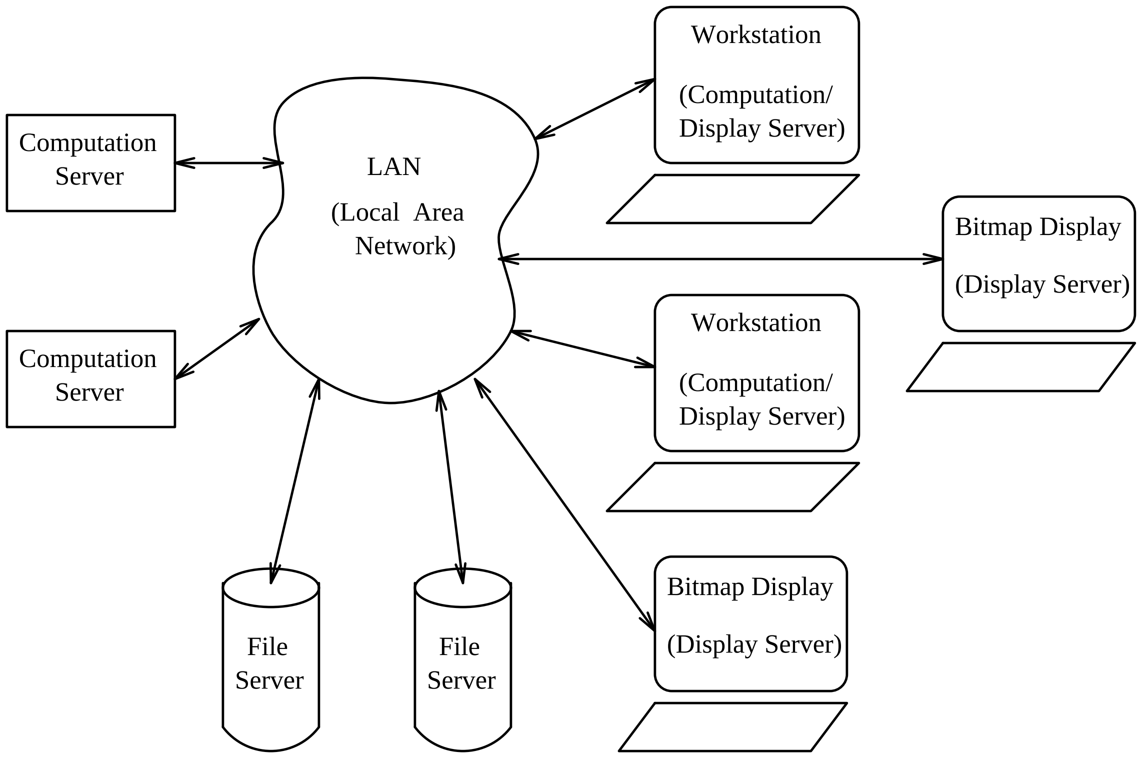

Figure 1: Distributed System Model.

distributed system may contain, among others, *file servers* for maintaining the disk storage in the system, *computation servers*, which are fast numerical processing oriented machines, and *display servers*, which are the window-oriented display units with or without the ability to perform user computations. In Figure 1 the display servers that also have general purpose computation capability are referred to as *workstations*, and the ones that are optimized as dedicated display servers are referred to as *bitmap displays*. In the X Window domain, the latter are commonly referred to as *X terminals*.

A meaningful profile of an X Window application program, or client, must account for the following three aspects:

1. The execution profile of the client program itself. This can be measured by a traditional

execution profiler [1, 2].

2. The time spent in servicing the requests at the display server.

3. The time spent in transferring the client-server communication messages over the connection between them. This is especially meaningful when the two programs are run on different machines and connect over a network.

The profiler described in this paper, Xprof, estimates the time spent in the display server and in the network connection and constructs an execution profile of the requests made by a client program. It achieves this by analyzing a trace of the interaction between the client and the display server programs at the X protocol level. It assigns a computation cost to each request on the basis of its attributes by consulting a set of parameters about the display server. The network time for each request is estimated on the basis of the size of the request message and the speed and latency of the network connection.

The principal advantages of this *protocol-level profiling* strategy, which may be applied to analyze all systems that follow the client-server paradigm are as follows. First, one can identify the most time consuming part of the client application by taking all the aspects into account, including the time spent in the server and the network. Second, by combining the results of Xprof with the results of a conventional execution profile of the application program, one can identify how the computation is being distributed between the client and the display server. Third, our technique permits cross-display-server profiling. One can take a trace from a particular client-server configuration and generate profiles for other display servers for the same client. This allows application developers to tune their applications for many display servers at the same time. Also, system designers can use the tool to predict the performance of applications on new or hypothetical hardware.

Other advantages of this strategy include the following. Since the trace collection is done at a protocol level, there is no need to recompile the client or server programs for tracing. This feature is especially useful since the user does not have to recompile the X server or the X libraries, both of which are fairly large and complex pieces of software, for profiling. Also, even though the tracing procedure causes some slow down in the processing of the requests, this may not matter for the client programs that tend to make asynchronous requests. The trace collector may be run on a third processor host to minimize the conflict for computation resources. Furthermore, for most client programs, if the tracing program is slow, it affects the arrival distribution of the messages but not their information content. Thus the post-processing done on the trace by Xprof can still provide a meaningful picture of the computation invoked by them.

REVIEW OF PROFILER STRATEGIES

Client profile Procedure-level profilers such as *Prof* and *Gprof* are frequently used to derive the execution profiles of conventional programs [1, 2]. These profilers entail recompiling the source code of the program to insert profiling code within the object code and are useful in studying the computation bottlenecks within the client program. As shown in Figure 2, procedure-level profilers generate information about the procedure calling pattern within a program. However, for a client program in the client-server model, these procedure-level profilers lose the information about the execution time of requests at the server. This is because the routines that invoke the computation at the server, such as procedure *P9* in Figure 2, are merely stub routines that send the appropriate request messages to the server.

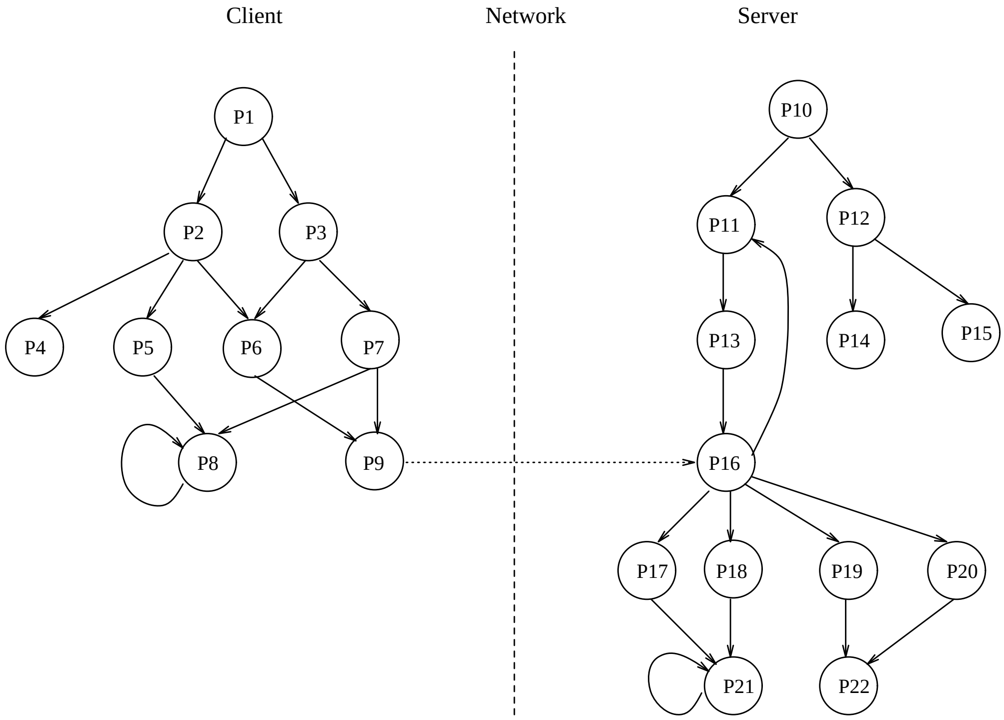

Figure 2:

Server profile The traditional profilers can also be used to instrument the display server. There are several disadvantages to this approach. First, the display server is usually a fairly large program and its size can grow appreciably when it is recompiled for profiling. Secondly, the server profile fails to give any information about the link between the requests from a specific client and the corresponding execution in the server. Thirdly, the profile usually gives a total information about the functions invoked and the total time spent in them. Since requests are frequently made with different attributes, it is not possible to analyze the distribution of the weight of each request as executed in the server.

In an effort to identify server bottlenecks, we instrumented the X Window display-server for a Sun 4/IPC Sparcstation using the popular procedure-level profiler Gprof. However, the profiles generated were not very meaningful. It turned out that the procedure calling pattern within the X Window server is quite complex with a number of self-referential loops in the calling graph and with multiple calling paths for many of the leaf procedures. Gprof uses a statistical approach for estimating the profile in order to keep profiling overhead low. In doing so, it propagates execution time up along the calling graph *equally* along each calling path. For a simple calling sequence this approach works well, but, for the X server profiled, the resulting profile could not be used to draw any meaningful conclusions.

Another approach is to measure the execution times of frequently invoked requests. In this strategy, followed by the X Window program *x11perf*, a special measurement program measures the runtimes for the requests for a set of values of the possible attributes. The information collected is very useful for comparing the performance of two different display servers. However, the data obtained are of limited utility for gauging the performance of a given application program since the user has to make a judgement about which of the requests are critical to the program and for which attribute values. Xprof makes a partial use of this approach by using a measurement program to generate a set of parameters for the target display server and making using of these, in addition to a protocol trace, to construct an execution profile.

Network profile Traditionally, network traffic is studied by measuring the load on a network by using a network monitor that logs all the packets on the network [8, 9]. Such measurements can give a good idea of the transport time of the request messages and the overall distributions of arrival time and byte-size of the packets. As with x11perf, it is difficult to relate such measurement to the performance of the actual application programs. However, there is a close correlation between the X protocol traffic and the actual traffic on the underlying network

[10]. Therefore, the network aspect of an X Window application may also be deduced from the protocol trace. Such a study has been done by Linton and Dunwoody [11].

Xprof Xprof automates the process of evaluating the performance of an application program on a target display server by consulting a set of performance parameters collected by an associated measurement program *Xmeasure*. Thus, it combines the information about the client-server interaction, in the trace, with the information about the display server to arrive at a meaningful execution profile. It also estimates the the time spent in network communication on the basis of the size of each request, in bytes, and the speed and latency of the network. It is thus able to arrive at a meaningful execution profile of the application with respect to the display server processing and the network communication overhead and identifies the contribution of each request type to this execution time.

MESSAGES IN THE X WINDOW SYSTEM

The X protocol supports a rich variety of message types for client-server communication [5, 3]. There are, broadly speaking, four broad categories of messages, i.e., *Requests*, *Replies*, *Events*, and *Errors*. Request messages are sent by the client program to the display server to request various windowing and graphics functions. Replies are sent from the display server to the client programs in response to requests that ask for some information from the server. Events are sent from the server to the client programs and are usually a consequence of real-time activities of the user, such as mouse movements and key presses. Lastly, errors are warning messages of various types that are sent from the server to the client. Figure 3 shows this broad hierarchy of messages types. The subtypes of each message category are not enumerated because the number of message types defined in the X Window System is over two hundred.

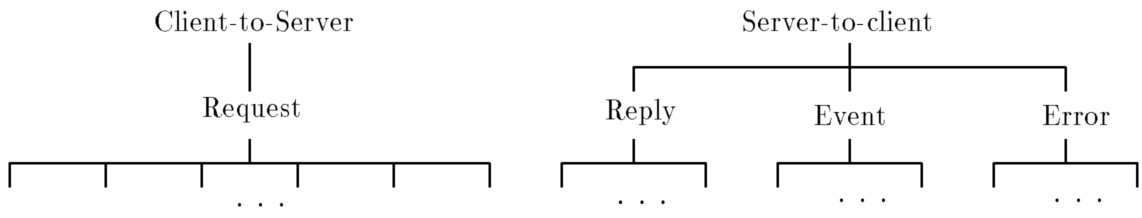

Figure 3: X protocol messages.

The X Window protocol manual describes the details of each message type [5].

Requests The Request messages invoke computation on the server, as requested by the client. These messages are analyzed in detail by Xprof for their statistical distribution and for the processing invoked on the server. Asynchronous, or *one-way*, request messages form the bulk of the messages traded in a typical X Window session. Since they do not require a reply from the server they can be pipelined on the network connection. The synchronous, or *round-trip*, messages, on the other hand, block until a reply is received and thus incur the overhead of network latency.

Message attributes Each of the messages has a number of attributes associated with it, e.g., the *byte-size* of each message is simply the actual size of the messages, in bytes. Event and Error messages are always 32 bytes long, but Requests and Replies can range in length from 32 bytes to 64 KBytes depending on their information content. Other attributes depend on the type of the message, e.g., the *CopyArea* request has associated with it the information about the location and size of the source and the location of the destination of the area copy. Similarly the line drawing request, *PolyLine*, invokes the attributes regarding the line length, line width, fillstyle etc.

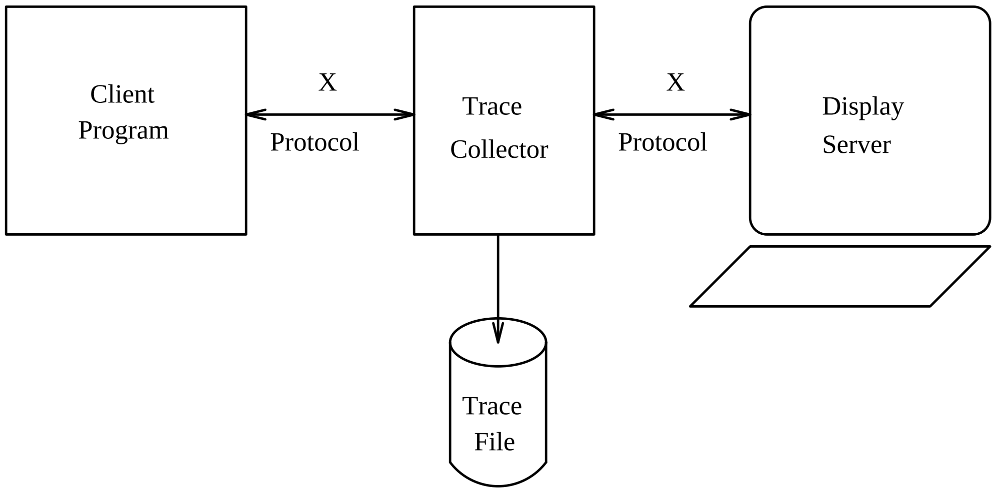

Figure 4: Trace Collection.

TRACE COLLECTION: XSCOPE

As discussed earlier, in the X Window System application programs, or *clients*, communicate with a *display server* program to request windowing and graphics services on the display. The communication is specified by a high-level protocol. A trace of the protocol messages is enough to characterize the computation invoked by the client program at the display server. Xprof is designed to analyze such a trace. An advantage of this approach is that there is no need to recompile the applications, or the display server, for collecting the profiles. An existing program, *Xscope*[1], was selected as the trace collection program. It is distributed with the source code of the X Window System and is thus available on all the X Window platforms.

As shown in Figure 4, the tracing program, Xscope, is set up to communicate with the display server and to act as a "dummy server". The client programs communicate with it as if dealing with an X server. Xscope passes on all the messages to and from the actual display server after logging them in a file. The degree of detail of the trace collection may be set up as a

[1]Xscope was written by James Peterson of MCC.

command line option. Each of the three programs in Figure 4 may run on their own computation hosts. The slow down of the client-program, caused by the trace collection, depends on the speed of the trace-collector host. In practice, clients that make high-level requests, such as geometrical figures, incur very little performance degradation, but clients that request large data transfers with the server may be slowed down by an order of magnitude.

TRACE ANALYSIS: XPROF

The protocol-level trace, collected by Xscope, is analyzed by *Xprof*, the trace analyzer and profiler program. This program constructs a statistical analysis of the messages exchanged and also constructs an execution profile of the session on the basis of parameters describing the target display server and the network connection.

After running Xprof on a trace, the end-user may chose to refine the trace analysis in order to bring out the details of interest. These refinements would be made in terms of better selection of the sizes of the data structures that are used to accumulate statistics or by supplying more precise values of the profiling parameters for the critical requests. These steps are discussed, in greater detail, in the section entitled “Refining the measurements”. The analysis process is summarized in Figure 5.

Profile generator For an application program running in a client-server environment, the total execution time, T of the program can be expressed as the sum of the total time spent in the client program itself and the time spent in servicing the requests, i.e.,

$$T = T_{client} + T_{server} \tag{1}$$

When the client and server programs execute, asynchronously, on different computation

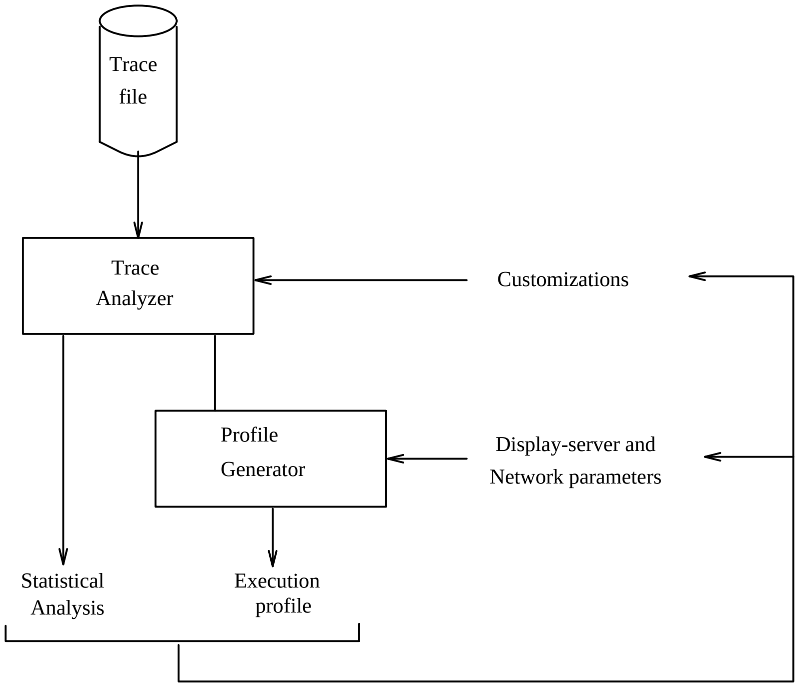

Figure 5: Overview of trace analysis.

hosts, their activities go on with some degree of concurrency and so the actual execution time would be less than the term T calculated above. Therefore, the above equation is actually an approximation of the total program execution time.

So,

$$T = T_{client} + T_{server} - T_{overlap} \tag{2}$$

The time spent in the server, T_{server} is the sum of the time spent in servicing the client requests and the time spent in processing real-time events such as mouse movement and key strokes, i.e.,

$$T_{server} = T_{requests} + T_{events} \tag{3}$$

For an X Window application, Xprof estimates the $T_{requests}$ in Equation 3 on the basis of the contribution of each type of request. Let R be the set of all request messages sent to the display server and let r_i be the ith message. If T_{r_i} is the time spent in servicing the message r_i then the total time of processing requests, $T_{requests}$, is given by the following equation:

$$T_{requests} = \sum_{r_i \in R} T_{r_i} \tag{4}$$

T_{r_i} can be expressed as the sum of the time actually spent in executing the requested operation on the display server, i.e., $T_{r_i}^{server}$. and the time spent in transporting the request message across the network, i.e., $T_{r_i}^{net}$.

$$T_{r_i} = T_{r_i}^{server} + T_{r_i}^{net} \tag{5}$$

Server time For computing the server time term $T_{r_i}^{server}$ the information content, or attributes, of each message must be taken into account. A particular invocation of a request may be made from a wide range of values for various attributes of the message, e.g., in order to draw a line, the width and the length of the line drawn are both important in determining the execution time of the request. Other attributes include the line-style, i.e., whether to draw the line continuous or dashed. Thus,

$$T_{r_i}^{server} = f(attributes_{r_i}) \tag{6}$$

The computation of $T_{r_i}^{server}$ is discussed, in greater detail, in the section "Profiler Details" The statistical analysis of messages is also discussed there. The application of the computation is studied in the section "Results".

Network time The network time term, $T_{r_i}^{net}$ is relatively easy to compute in terms of the size of each request in bytes and the average network speed and latency. The network latency matters only for synchronous requests, which block until they receive a reply from the display server. With each request one can associate a boolean variable, $blocking_{r_i}$, which is true if the request type is synchronous and false otherwise. Then, for the ith request,

$$T_{r_i}^{net} = (bytesize_{r_i}/netspeed) + (blocking_{r_i} * netlatency) \tag{7}$$

Steps in processing the trace input The profiler, Xprof, thus analyzes the protocol-level trace and makes use of the metrics supplied to it about the target display server and the network connection. For each instance of the requests, as seen in the trace, Xprof goes through the following steps.

Step 1: Read in the timestamp, the byte-size of the request message and the relevant attributes. Compute the operation size, or *op-size*, for the message.

Step 2: Update the histograms of byte-size, op-size, and arrival time distributions.

Step 3: Compute $T_{r_i}^{net}$ for the request on the basis of its byte-size and enter it in the data structure for this request type.

Step 4: Compute $T_{r_i}^{server}$ for the request on the basis of its op-size and other attributes and enter it in the data structure for this request type.

Some messages affect the state of the display server, e.g., messages that change the *graphics context* affect the attributes of future graphical requests. Xprof maintains the server state and computes the attributes of affected requests from it.

When the trace analysis is complete, Xprof prints out the statistical distribution of the messages and a summary of the time spent in serving each type of request.

COLLECTION OF SERVER METRICS: XMEASURE

The Xmeasure program is used the collect the server parameters for a given display server. It runs measurements for each of the requests defined in the X Window protocol, for a wide range of attribute values. This program is thus similar to the X Window program *x11perf* and is designed to output its results in a format suitable for parsing by Xprof. For each request, the measurements are made for a wide range of attribute values critical to that request. Each measurement is made by requesting a large number of operations within two, carefully measured, synchronization points. The rate of the operation execution is printed out along with the attributes.

For each request type, the key attribute identified is the *op-size*, which is defined appropriately for the request. The op-size is a measure of the grain of the computation invoked on the server and thus different from the "byte-size" of the request packet. For instance, for a data transfer request, such as *PutImage*, or *CopyArea*, the op-size would be the area of the target. For a line drawing request the line length is taken to be its op-size. Table 1 shows some measurement results for typical request invocations on three popular color workstations, i.e., Sun 4/IPC, DECStation 3100, and HP 9000.

The xmeasure results, or server parameters are supplied to Xprof in the form of a description language. Each entry in the parameters description file has the following format:

$$Request\_Name \quad [attribute\_1 =< value\_1 >] \quad [attribute\_2 =< value\_2 >] \quad \ldots$$

$$\ldots \quad [attribute\_n =< value\_n >] \quad (opsize, rate) \qquad (8)$$

where attributes 1 through n are the attributes appropriate to that message and the "rate" is the number of operations per second that were measured for the specified op-size. Any

Table 1: Xmeasure measurements for common client requests

Request	Size	Typical Attributes	Time per operation (ms)		
			Sun 4/IPC	DS 3100	HP 9000/350
PutImage	10x10	depth=8	0.29	0.84	1.46
	100x100	depth=8	10.50	11.39	51.23
	300x300	depth=8	89.45	96.25	537.63
PolyLine	length=10	width=0	0.035	0.046	0.105
	length=100	width=0	0.043	0.056	0.159
	length=300	width=0	0.087	0.092	0.361
PolyText8	strlen=8	font=6x13	0.32	0.38	2.60
	strlen=32	font=6x13	1.07	1.13	6.71
ClearArea	10x10	depth=8	0.43	0.31	3.36
	100x100	depth=8	1.69	1.48	3.37
	300x300	depth=8	10.19	9.51	5.21
CopyArea	10x10	depth=8	0.31	0.38	6.55
	100x100	depth=8	1.86	1.84	4.81
	300x300	depth=8	15.36	15.36	6.79
PolyFillRectangle	10x10	fillstyle=Solid	.049	.055	.189
	100x100	fillstyle=Solid	1.17	1.07	0.49
	300x300	fillstyle=Solid	9.58	9.01	2.84

number of entries may be given for a particular request, say for different values of op-sizes and attributes.

It turns out that the op-size is adequate to characterize the performance of most of the Request types. The graphics requests [2] are a notable exception to this general observation. At an early stage of the design, it was decided to limit the types of possible graphics attributes, handled by Xprof, to four. These are as follows:

1. **Gxmode** refers to the boolean function that is used to combine source and destination pixels. Typically an application will either choose to replace the destination pixel with a completely new value, or, combine the old value with the new value of the destination and write it back. The second type of operation is usually more expensive than the first type

[2] There are 8 graphics requests, i.e., *PolyPoint, PolyLine, PolySegment, PolyRectangle, PolyArc, FillPoly, PolyfillRectangle*, and *PolyFillArc*.

because of the extra memory access involved. Therefore, this attribute is maintained for these two types.

2. **Linewidth** is the width of a line, in pixels. Zero width has a special meaning in the X Window System and offers a hint to the server that it may use a hardware algorithm, if any, to draw a line of width 1. All other linewidths are generally drawn by a software algorithm. Any number of linewidths could be invoked by an application. Owing to practical considerations, this variable is allowed to have up to four values. Computation time for other line widths is interpolated from the runtimes for the available line widths.

3. **Fillstyle** may call for solid filling, in the default case, or specify filling a region with a standard tile or with a supplied *pixmap*. Again, one value of this attribute is allowed to have value of solid fill and the others are all clubbed together.

4. **Linestyle** may require solid lines or various types of dashed lines. The solid linestyle is treated as one value of this attribute and the others are treated together.

Figure 6 shows the entries for the requests for creating windows and for line drawing. These were gathered from an actual measurement run for the Sun 4/IPC Sparcstation.

```
CreateWindow            (0,4717.030)
         ...
PolyLine gxmode=GXcopy linestyle=LineSolid      fillstyle=FillSolid          \
         linewidth=0    (100,19161.61)
PolyLine gxmode=GXcopy linestyle=LineSolid      fillstyle=FillSolid          \
         linewidth=0    (300,10428.97)
PolyLine gxmode=GXxor  linestyle=LineSolid      fillstyle=FillSolid          \
         linewidth=0    (100, 8423.40)
PolyLine gxmode=GXxor  linestyle=LineSolid      fillstyle=FillSolid          \
         linewidth=0    (300, 3309.19)
         ...
PolyLine gxmode=GXxor  linestyle=LineDoubleDash fillstyle=FillOpaqueStippled \
         linewidth=10   (100,   45.31)
PolyLine gxmode=GXxor  linestyle=LineDoubleDash fillstyle=FillOpaqueStippled \
         linewidth=10   (300,   20.67)
         ...
PolyText8 fontname=6x13 ( 8, 3121.67)
PolyText8 fontname=6x13 (32,  934.63)
         ...
```

Figure 6: Typical entries for server parameters.

PROFILER DETAILS

Server time The approach followed in Xprof is to estimate the server execution time, i.e., $T_{r_i}^{server}$, for a given request by interpolating from a supplied list containing information about execution speeds of the requests for typical values of op-size and other attributes. This information is provided to Xprof in the format discussed earlier and is typically generated by running the program Xmeasure on the target workstation.

Thus, the problem of estimating the cost of a request reduces to one of selecting and interpolating from values supplied in a list of information about the costs of a set of standardized requests. Since there is a very large number of possible attributes for each request and each could have possibly limitless values, it is necessary to limit the range of attributes that are actually measured and used. The design choice made in Xprof is to use the op-size as the sole attribute for the vast majority of requests. The graphics requests are measured for all four

```
typedef union _MsgCost {
  CostCell *window;          /* Pointer to a list of costs   */
  CostCell **gfx;            /* Array of graphics cost lists */
  CostCell **txt;            /* Array of text cost lists     */
} MsgCost;

typedef struct _CostCell {
  float   size;              /* Size for which this measurement was made   */
  float   speed;             /* Speed in size units per second             */
  struct _CostCell *nextcost; /* Next data point for size and speed thereof */
} CostCell;
```

Figure 7: C data structures for maintaining the measured cost of a message in Xprof.

attributes discussed earlier. In the current implementation the linewidth is allowed to have up to 4 values and the other attributes are allowed to have up to two values each. Thus, 32 variations of the graphics attributes are possible for each value of op-size chosen. The text rendering requests are also maintained for upto 32 possible fonts.

As described earlier, for each set of attribute values, Xmeasure makes many different measurements for the possible values of op-sizes. Thus, it is necessary to devise a way of storing and retrieving the measured information. Xprof maintains the display-server measurements in an array of lists as shown in Figure 7 and Figure 8.

Figure 8 shows the request *CreateWindow* as representative of most request messages, which have associated with them a linked list of size and speed pairs. Graphics requests, such as *PolyLine*, have an array of lists — one list for each combination of allowed attribute values. Initially, each list is empty. During initialization, the entries, as shown in Figure 6 are read and the size-pair entry is entered in the appropriate list, which is maintained in ascending order of size for easy searching. In terms of the C language, there is an array of pointers called *MsgCost* that has one entry for each request. For the graphics and text requests, the array entry points

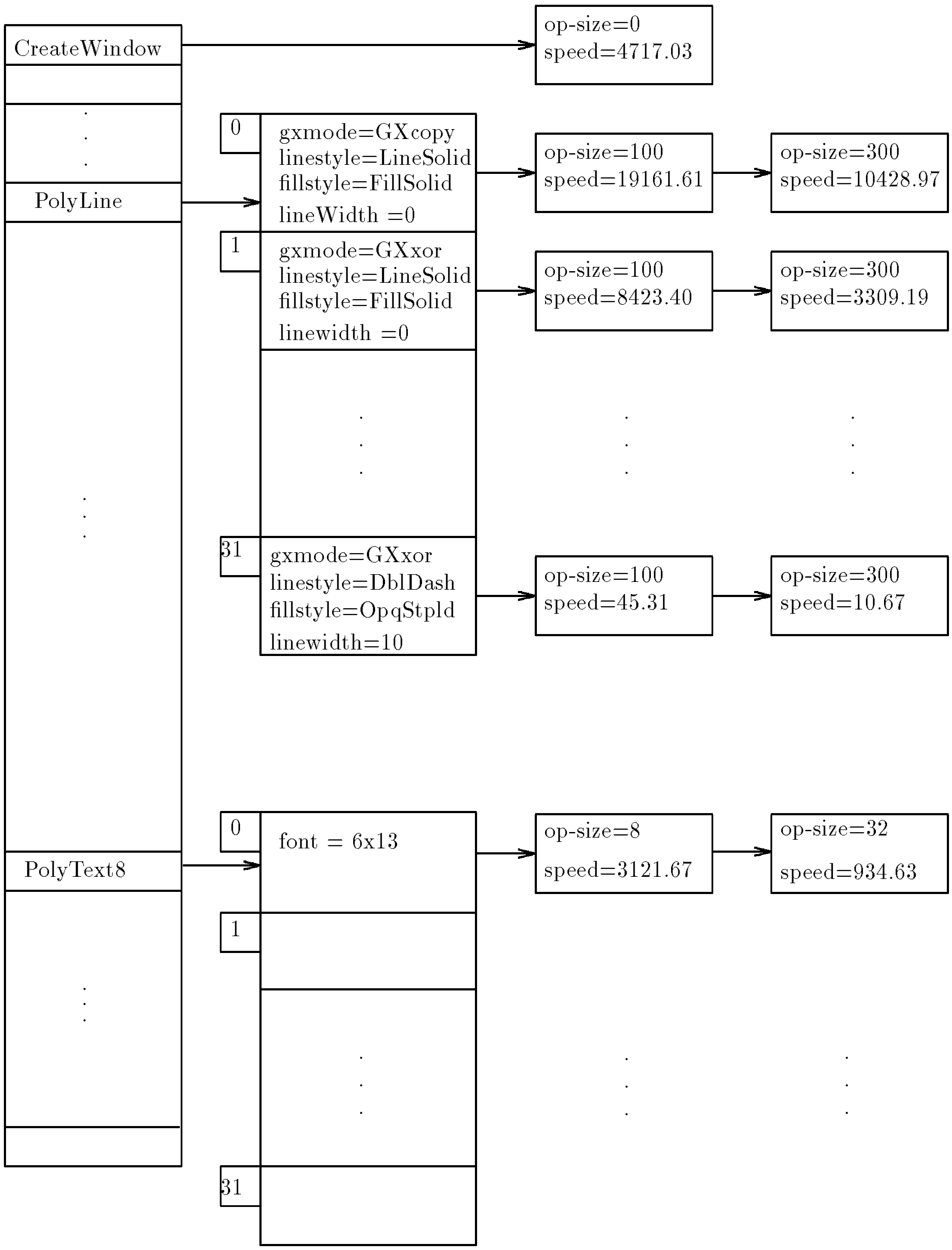

Figure 8: Message costs as maintained by Xprof. Data shown are for the earlier example, as measured for the Sun 4/IPC workstation.

to an array of lists comprised of the *CostCell* structure. All other requests have an entry that points to a single list of *CostCell* structures.

During trace analysis, for each request encountered, Xprof searches for an entry matching its size and attributes in the *MsgCost* array. There are three possible outcomes of this search. First, an exact match may be found for the given request's attributes and op-size. In this case the T_{r_i} term is easily computed from the *speed* of the matching entry. Second, an exact match for the attributes may be found, but the entry for the exact op-size may not be found in the linked list. In this case, the solution is to interpolate the term for speed from the entries that match the desired op-size most closely. Third, in the worst case, there may be no exact match for the attributes desired. In this case, for each of the nonmatching attributes, Xprof substitutes another, measured, attribute on the basis of heuristics. For example, in the current implementation of Xprof, the gxmode *gxand* would be replaced by the more common gxmode *gxxor*, which is also a two operand function. Similarly the gxmode *gxset* would be replaced by *gxcopy*, which is a one operand operation too. In this way a set of attributes is obtained for which there is an entry in the MsgCost array and a cost is computed as in the earlier cases. A warning message is printed out detailing the substitutions made.

Example The following example illustrates the computation of T_{r_i}. Say, Xprof sees the following three *PolyLine* requests in the trace. In each case the op-size refers to the length of the line in pixels. Also, PolyLine is a graphics request and so the various graphics attributes need to be taken into account. Figure 8 is used to calculate the computation time for each case.

1. Op-size=100, gxmode=GXcopy, linestyle=LineSolid, fillstyle=FillSolid, linewidth=0: In this case, the matching entry in the PolyLine cost array is at index 0. From Figure 8, the

appropriate list entry for op-size of 100 yields a speed of 19161.61 operations per second. From this, the time spent can be computed as 1/19161.61, i.e.,

$$T_{r_1} = 0.0522ms$$

2. Op-size=200, gxmode=GXcopy, linestyle=LineSolid, fillstyle=FillSolid, linewidth=0:

   Again, the index is 0. However there is no exact match for the op-size, since there is no list entry for lines of these attributes and length 200. The solution is to interpolate from the supplied speeds for op-size. 100 and 300. Op-size 100 implies an execution time of 0.0522 ms and op-size 300 implies 0.0959 ms. Therefore,

$$T_{r_2} = 0.0522 + \frac{0.0959 - 0.0522}{300 - 100} \times (200 - 100) = 0.07385ms$$

3. Op-size=100, gxmode=GXxor, linestyle=LineDoubleDash, fillstyle=FillStippled, linewidth=10:

   For this set of attributes, there is no entry corresponding to the fillstyle, i.e., FillStippled. According to the substitution heuristic, this attribute is to be substituted by the attribute FillOpaqueStippled. This substitution leads to a match at index 31. From Figure 8 we find that an operation of size 100 is executed with a speed of 45.31 operations per second. From this,

$$T_{r_3} = 1/45.31 = 22.07ms$$

Statistical Distributions Xprof collects the statistical distributions of the interarrival time and op-size distributions of each X message type, i.e., all requests, replies, events, and errors. The grain size for the measurement can be set at runtime as discussed below.

```
typedef struct {
  Boolean  invoked;         /* Has this structure been invoked before?     */
  long     number;          /* Total number of these messages seen         */
  long     total_bytes;     /* Total number of bytes seen for this message */
  long     last_time;       /* The time stamp of previous message          */
  Grain    size_grain;      /* Size grain for this measurement             */
  Detailed detailed;        /* Are we maintaining detailed information? The
                               following are updated only if this is set   */
  long     *iat_distbn;     /* Interarrival time distribution              */
  long     min_iat, max_iat; /* Range of values of the raw data            */
  long     *size_distbn;    /* Size distribution                           */
  long     min_size, max_size;
} MsgStats;
```

Figure 9: C language structure to maintain message statistics.

Figure 9 shows the data structure employed to collect the statistics for each of the message types. Since each X message type has a copy of *MsgStats* associated with it, the total number of instances of the MsgStats data structure is over two hundred. Thus, it is important to keep the size of the data structures within reasonable bound. In order to achieve this, the two arrays for collecting the distributions of inter-arrival time and size, i.e., *iat_distribution* and *size_distbn*, are allocated dynamically from heap memory at run time.

The distribution of inter-arrival time can be expected to have a very wide range of values. Therefore, early in the design process, it was decided to collect the corresponding histogram on a log scale. The dimension of the corresponding array was set at 32. Given that the grain size for the measurement of time, on a Unix system, is 10ms, this choice is enough to cover interarrival times of, approximately, up to a year, which should be adequate for most applications. This choice of size of the iat_distbn array implies an overhead of about 128 bytes [3] for each data structure, which is quite reasonable.

[3] Assuming that the computer uses 4 bytes for each *long* integer

The choice of the size of the histogram array for the operation size, or *op-size*, is a trickier proposition. The range of op-sizes is different for each message type. Also, the op-sizes are distributed fairly uniformly within that range. Thus, to be meaningful, these measurements should allow for the different ranges and also be measured on a linear scale. The design choice made was to set the array size of *size_distbn* at 4096, which is changeable at run-time, and allow for different grain-size of measurement for each request. A good choice of grain-size for a request would thus be one that distributes its range uniformly over the array.

Since each request has a different grain for its size measurement, the information about the grain is also maintained in the MsgStats, in the *size_grain* variable, and the size histogram is interpreted only with reference to this grain. Default size-grains are set up at initialization time and may be set by the user. To illustrate the choice of a suitable size-grain, it may be noted that some requests, such as *PolyLine* generally request small operations and so a size-grain of 1 is adequate for such requests. The *PutImage* request, on the other hand, can request the copying of data of up to 64 Kilobytes and would need a size-grain of 16, for the above choice of 4096 buckets, to accommodate all possible values. As an extreme example, the *ClearArea* request can request the clearing of very large sections of the display screen. For clearing an entire screen of a display that is 1024 by 1024 pixels with 8 bits per pixel, the server needs to process 1 Megabyte of data, which implies that a grain-size of 256 is needed for this operation. Xprof has built-in default values for the grain of each message type, which are adequate for most cases.

Given a choice of 4096 for the number of buckets in the size distribution, this array accounts for 16 Kilobytes [4] of heap memory per message measured. Since over 200 instances of this

[4] Assuming that the computer uses 4 bytes for each *long* integer

structure may be needed, the total space usage amounts to over 3 Megabytes. In order to reduce this, worst-case, memory requirement two further optimizations are made.

First, the variable *detailed* determines whether the user is interested in collecting the histogram at all. If not, the distribution arrays are not allocated, or maintained, at run time. This may be true if the user is not interested in certain requests or is only interested in the execution profile and not in the message distributions. This variable can be set for each message type individually.

Second, the boolean variable *invoked*, which is false by default, is used to track whether the message has been encountered at all in the trace. The allocations of the size_distbn array, from heap memory, are actually made the first time the message is seen. Since a typical X Window session uses only a subset of the possible message types, this feature can save a lot of heap memory. In practice, 30-50 message types are typically seen in a trace. This implies a memory usage of 480-800 Kilobytes of usage, which is a vast improvement over the worst case usage of over 3 Megabytes calculated earlier.

Thus, the customizable parameters, for the histograms, are the size-grains for the operation size — on a per-message basis, the choice of whether to maintain the detailed histograms, and the sizes of the histograms arrays. Default values for each are built into Xprof and are customizable by the user.

REFINING THE MEASUREMENTS

Since Xprof is a trace-driven profiler, it is possible to rerun it on the same trace in order to bring out information of interest to the user. For instance, after running Xprof once on the trace input, the user may find that the trace involves requests with combinations of attributes that are not covered in the server parameters list. For such a case, the user may choose to collect the necessary data, by running Xmeasure on the target display server for the necessary combination of attributes, and then augment the server parameter list. Then, Xprof may be rerun to generate a more accurate profile of the trace.

Several run-time variables may be tweaked in order to refine the statistical analysis of the messages. Some of these are discussed below. First, the array size of the *size_distbn* array may be changed at run-time. This choice is driven by the available physical memory to run Xprof. The default choices embedded in Xprof reflect the resources available on current generation machines. Second, the *size_grain* may be modified on a per message basis. This choice depends on the range of sizes, seen in the trace for each message type, and may thus be refined after the trace has been analyzed once through. Third, the *detailed* variable mentioned earlier may be used to selectively turn off the statistical measurements, but not the profiling, of certain requests. Such a choice would not affect the computation requirement of Xprof, but may reduce its dynamic memory usage substantially.

Each request type has an associated *action* function that processes each instance of the request, as seen in the trace, by following the steps described in the earlier section "Trace Analysis: Xprof". If, for some reason, the user wants to rewrite the actions, a template file is included with the source code. Thus, users could extend Xprof to support future extensions to the X protocol or, say, change the definitions of the op-size for a request, as they choose.

In some cases, users may want to set up Xprof to consume trace data in real-time. In order to support such a usage, Xprof captures the following signals in the Unix environment. [5]

[5]These signals are communicated to Xprof by using the *kill* command from a Unix shell, e.g., for sending SIGHANGUP, the user would type: *kill -1 [Xprof-process-number]*, and for sending SIGKILL: *kill -2 [Xprof-process-number]*.

1. SIGHANGUP: This signal causes Xprof to print out the results accumulated up to the current point.

2. SIGKILL: This causes Xprof to reset its data structures to their initial values, clear all histogram arrays, and to reread the server parameter file.

RESULTS

The output generated by Xprof consists of the estimated execution profile for the requests, at the server, and the statistical distribution for the messages and the message categories, i.e., Requests, Replies, Events, and Errors.

In the following tables, detailed profiler results are shown for the Xtex program, as computed by Xprof. The application was run on a Sun 4/IPC Sparcstation system with release 4 of version 11 of the X Window System. Xtex is a previewer for documents formatted by the LaTeXdocument processing software. The trace was collected for the sequential display of all the pages of a 19 page technical report. Trace analysis was done using the server parameter list for the Sun 4/IPC Sparcstation — as collected by the measurement program Xmeasure. The network parameters were set to a data transfer speed of 100 KBytes/sec and a latency of 10 ms.

Execution profile for requests The execution profile consists of a list of all the requests that are made during the execution of the program, with the total estimated time of execution for each. This time is broken down in terms of the computation and communication parts. The number of messages received, in each category, and the mean execution time per request are also printed out.

Table 2: Excerpt of the execution profile of Xtex.
Network speed = 100.00 KBytes/sec, Latency = 10.00 ms

Request Name	Time (ms)	%of total	Compute part(%)	Network part(%)	No. of msgs	%of total	Time/call (ms)
PolyFillRectangle	3137.490	50.68%	47.22%	3.45%	508	5.92%	6.176
PolyText8	2130.350	34.41%	7.12%	27.29%	7389	86.12%	0.288
MapSubwindows	102.985	1.66%	1.66%	0.01%	5	0.06%	20.597
MapWindow	100.388	1.62%	1.59%	0.03%	23	0.27%	4.365
QueryFont	227.464	3.67%	1.07%	2.60%	16	0.19%	14.216
...							
Grand Total	6191.332	100.00%	62.12%	37.88%	8580	100.00%	0.722

Table 2 shows the execution profile for Xtex for the five most time consuming requests, which together account for over 90% of the execution time. Not surprisingly, the text rendering messages, *PolyText8*, account for a large number of the messages: over 86%. Yet, the computation part for these messages is responsible for only about 7% of the profiled execution time. The dominant message, from the viewpoint of the display server, is the *PolyFillRectangle* request. It turns out that, in the design of Xtex, this request is invoked to clear a page before rendering text on it. Clearly, text rendering itself is not the computation bottleneck for this application. However, when we look at the network part, the PolyText8 requests take up 27% out of the 38% of the time spent in network communication. The overall performance of this program, on the hardware studied, could be improved by reducing the computation cost of clearing a page, and the network cost of communicating the text rendering requests.

In order to get an idea of the server-side computation, the user may be interested in looking only at the computation profile. Table 3 shows the estimated profile with network speeds and latency values that effectively make the network component irrelevant. Such an analysis emphasizes the computation bottlenecks in the profile.

Table 3: Excerpt of the execution profile of Xtex.
Network speed = 1000000.00 KBytes/sec, Latency = 0.00 ms

Request Name	Time (ms)	%of total	Compute part(%)	Network part(%)	No. of msgs	%of total	Time/call (ms)
PolyFillRectangle	2923.832	76.02%	76.02%	0.00%	508	5.92%	5.756
PolyText8	440.879	11.46%	11.46%	0.00%	7389	86.12%	0.060
MapSubwindows	102.585	2.67%	2.67%	0.00%	5	0.06%	20.517
MapWindow	98.548	2.56%	2.56%	0.00%	23	0.27%	4.285
QueryFont	66.184	1.72%	1.72%	0.00%	16	0.19%	4.136
...							
Grand Total	3846.058	100.00%	99.99%	0.01%	8580	100.00%	0.448

Message statistics Xprof prints out the statistical distribution for the message categories, as well as for the individual messages. In addition, it can be set up to print out the detailed histograms from which these statistics are derived. The statistics are printed for the interarrival time and size distributions of the messages.

Message categories Table 4 shows the overall distributions for the Request messages, in the Xtex trace. The interarrival distribution has a large number of zero entries in it owing to the buffering of messages within the X library, which makes a lot of messages arrive together at the server. Hence, the arrival distribution for the actual message packets can be arrived at by discarding the zero values. This distribution is also computed and printed. For the Request messages, the sizes refer to the actual byte sizes of the requests. The total bytes for each request message are also computed and printed as shown in Table 5. Similar printouts are made for the other categories of messages, i.e., Replies, Events, and Errors, but, in order to save space, those are not shown here.

Table 4 shows that, the Xtex message are buffered frequently by the X protocol. Most message are relatively small, with a mean size of about 24 bytes. Table 5 shows that the PolyText8 request accounts for over 80% of the network traffic for Xtex. This explains why

Table 4: Message statistics for Xtex

```
               ***** Statistics for Requests *****
Inter-arrival time distribution (ms):
               Number     Range          Mode  Median       Mean    Std. Dev.
(All points)     8580       0-8990          0       0      13.22       165.58
(Zeros removed)   150      30-8990        630     310     756.13      1003.27

Size distribution:
               Number     Range          Mode  Median       Mean    Std. Dev.
(All points)     8580       4-96           24      23      24.07         9.41
```

Table 5: Total bytes for each request in the trace of Xtex

Request messages	Total Bytes		Number	
PolyFillRectangle	21368 bytes	(10.35%)	508	(5.92%)
PolyText8	168964 bytes	(81.80%)	7389	(86.12%)
MapSubwindows	40 bytes	(0.02%)	5	(0.06%)
MapWindow	184 bytes	(0.09%)	23	(0.27%)
QueryFont	128 bytes	(0.06%)	16	(0.19%)
...				
Grand Total	206548 bytes		8580	

these requests have a relatively high network component in the execution profile. The average size of these requests is about 23 bytes, which is pretty close to that for the overall profile.

Individual messages The last section of the Xprof output lists the distributions of each message type individually. Table 6 shows the distribution for the PolyText8 request, for Xtex. As noted earlier, for the overall request distribution, the effect of buffering of the messages can be seen here in the large number of entries for zero arrival time. In the size distribution, the op-size, for this request, is the length of the requested string of text. The distribution shows that the text requests are made, on the average, for very short string lengths of about 3.5 characters. Since, as noted earlier, the average PolyText8 message is about 23 bytes long, this means that the message is not very efficient at transmitting the strings. Longer string lengths in each request might improve the network performance.

Table 6: Statistics for PolyText8 messages in the trace of Xtex

```
                ***** Statistics for PolyText8 *****
Inter-arrival time distribution (ms):
                Number     Range          Mode  Median      Mean  Std. Dev.
(All points)      7389       0-21340         0       0     13.99     284.42
(Zeros removed)    108     150-21340       630     310    957.41    2152.07

Size distribution:
                Number     Range          Mode  Median      Mean  Std. Dev.
(All points)      7389       1-43            3       2      3.46       2.35
```

Table 7: Cross-server profiler results for Xtex.

Request Name	Message distribution		Execution Profile					
			Sun 4/IPC		DecStation 3100		HP 9000/350	
	No. of messages	% of total	Time	% of total	Time	% of total	Time	% of total
PolyFillRectangle	508	5.92%	2.92s	75.99%	2.67s	72.39%	1.14s	14.98%
PolyText8	7389	86.12%	0.44s	11.50%	0.59s	16.20%	4.28s	56.32%
MapSubWindows	5	0.06%	0.10s	2.67%	0.03s	0.72%	0.14s	1.84%
MapWindows	23	0.27%	0.99s	2.56%	0.12s	3.20%	0.33s	4.39%
QueryFont	16	0.19%	0.66s	1.72%	0.11s	2.92%	0.42s	5.60%
All Messages	8580		3.85s		3.69s		7.60s	

Cross-server profiling Table 7 is a summary of a cross-server profiling study of the performance of the Xtex trace on several different architectures. In addition to the Sun 4/IPC, Xprof was run on the trace with server parameter lists for the DECStation 3100 and HP 9000/350 computer systems, each of which is a color workstation, with 8 bit color and running release 4 of version 11 of the X Window System. In order to emphasize the computation part at the display server, the profile was run for network parameters that effectively make the network component irrelevant.

The data show that the PolyFillRectangle requests are the computation bottleneck for both the Sun and DEC machines. For the HP, however, the PolyText8 requests are dominant in the profile. Note that on the HP, the text rendering is about 8 times slower than on the other

Table 8: Client-server profiles for the Sun4/IPC.

Application	Client Time (Gprof)	Server Time (Xprof)	Total Profile (Gprof+Xprof)	Actual Time (wallclock)	Ratio (Profile/Actual)
XImage	255.0 s	57.84 s	312.84 s	320.0 s	97.8 %
Xtex	2.5 s	3.85 s	6.37 s	7.0 s	91.0 %

machines. So, its profile is skewed towards the text rendering function. However, because it has a fast implementation of PolyFillRectangle, its total time for Xtex is only about 2 times that for the other two machines. This example clearly demonstrates the importance of correctly identifying the critical server functions, for a given workload, to optimize the server performance. For all three machines, just a few requests account for 70-90% of the computation time on the display server.

Client-server load partitioning In order to get an idea of the distribution of computation between the client and server programs, the client program may be profiled by a conventional procedure-level profiler. The client profile time may then be compared to the server profile time, as estimated by Xprof. Table 8 shows the results of such a measurement. In addition to Xtex, data are shown for Ximage, which is a scientific visualization tool that is used to display the result of scientific computation in the form of color pictures. It can be set up to display a succession of such pictures as an animation sequence. The data set chosen was a sequence of 60 pictures, each 300x300 pixels in 8 bit color. The sequence was run through 10 times in order to generate the trace.

The data in Table 8 demonstrate that Xprof complements the client-side profile by providing an accurate server-side profile. For the applications shown, the sum of the client time, as measured by Gprof, and the server-time, as measured by Xprof, is very close to the actual wall

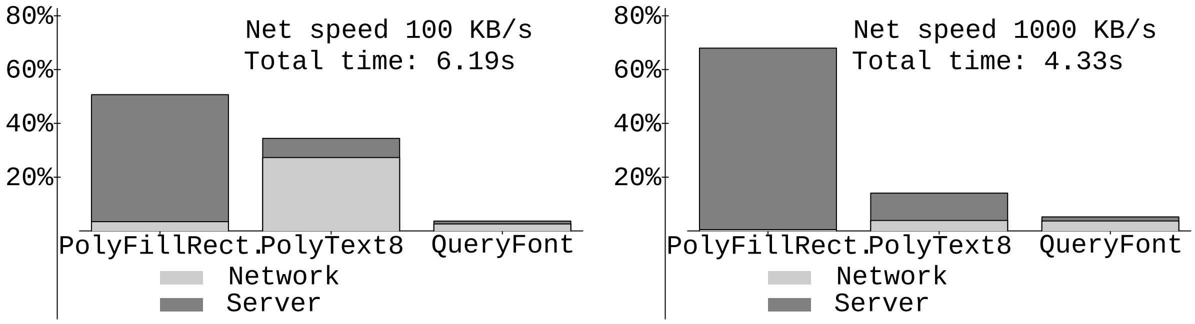

Figure 10: Effect of network speed on Xtex profile

clock time. Since both the application and the display server were run on the same cpu host, the network time is not relevant to this measurement.

Effect of network speed In the client-server model of computing, processing is partitioned between the client and server programs. However, the overall performance also depends on the network connection between the two processes. In this section we study the impact of the network parameters on the Xtex application. The profiles are constructed for the Sun4/IPC with the client and server communicating over a network with latency of 10ms and a speed of either 100 KBytes/s or 1000 KBytes/s. The y axis in Figure 10 is normalized to the total processing time since the objective is to see how the overall request service time is divided between the network and the display server.

Figure 10 shows that Xtex is not affected too much by the network speed for the trace studied. The critical function *PolyFillRectangle* is not affected much by the network speed at all. This is because it is a graphics request without much data content. The text rendering request *PolyText8* is affected more by the network speed. In fact this request constitutes over 80% of the request bytes. Overall, this application slows down about 50% for a 10 times slowdown in the network.

LIMITATIONS OF XPROF

The profiling approach developed in this paper makes use of a protocol-level trace, which has no knowledge of the lines of code that gave rise to the protocol messages. So, the current implementation of Xprof cannot connect the critical server functions to specific lines of client code. In order to collect such information, the trace would need to be generated by an instrumented version of the client program, which would emit the source code information along with the protocol messages. This would involve the recompilation of the client program source code with a compiler that inserts the appropriate trace routines in the object code.

There are several extensions to the server cost model that could add to the profiling information available from Xprof. Currently, the cost of server events, such as mouse movements, is ignored. The server cost model could be extended to include the cost of processing the real-time events. This aspect will become more important as sophisticated input mechanisms, such as speech and handwriting recognition become widely used for computer input. Also, the computation of the request service time only takes the direct cpu cost into account. However, X clients can create *X resources* in the server, which use up server memory. If enough resources are created, the virtual memory needed by the server may exceed the physical memory available. The resulting swapping may affect execution time significantly. Thus the user may be interested in the dynamic allocation profile of the X resources. The trace analyzer in Xprof could be extended to generate such a dynamic profile to help the user understand the server-resource consumption pattern for a certain client. For this, the server parameter information would need to be augmented with information such as the number of bits per pixel and the bytes per word for the target display-server.

The estimation of the cost of a request, as described in the section "Profiler Details" may involve two kinds of approximations, i.e., interpolation of the operation size or the substitution of an attribute by another. In either case a warning message is printed out. These approximations could yield erroneous results and must be understood well by the user. First, the interpolation function is linear. If the actual cost function is non-linear with respect to the operation size, the computed cost may be wide off the mark. The best defense against this is to measure the actual speed of the request for the needed operation size and augment the server parameter information. Second, the substitution of a non-measured attribute by a measured attribute is done on the basis of certain heuristics. These heuristics may not be valid in some instances. For example, there may exist special hardware support for one class of attributes but not for others. The user should examine the warning messages carefully and alter the substitution heuristics, if necessary. A more precise solution would be to measure the costs of the requests for the attributes of direct interest.

The network cost model, in the current implementation of Xprof, gives the user an idea of the order of magnitude of the communication time on the basis of supplied network speed and latency. A more sophisticated network simulation could model the performance of actual network protocols, such as the collision sensing protocols in wide use in local area networks today.

CONCLUSIONS

In this paper we describe Xprof, a protocol-level profiling profiler that generates meaningful profiles of X Window applications. The profiler estimates the time spent in servicing the request messages in the display server and the network connection by analyzing the protocol-level trace of messages exchanged between the application and the display server. In addition, the statistical distributions of the arrival-time and the operation sizes of the requests are analyzed. The resulting profile provides a detailed picture of the server-side execution of the application program.

The profiles generated by Xprof may be useful to many different audiences. For instance:

1. Xprof supports cross-display-server profiling. Thus, users of display servers, such as workstations, could evaluate the performance of different workstations for their own applications by profiling traces of interest to them for several target servers. All they need is the server parameter data for each workstation, which can be generated by Xmeasure in a standardized manner.

2. Developers of X Window based software can identify bottlenecks in their software and tune it for different platforms. Conventional profilers do not give a coherent picture of the overall execution profile of a client-server program.

3. Designers of display servers can get a good idea of the critical requests made by typical applications and tune their systems to execute such requests faster.

4. Administrators of distributed systems can get a better idea of the partitioning of computation between the client and display server programs and also the network load imposed by typical applications.

As distributed systems come into widespread use, the client-server paradigm of computing will become increasingly important. The protocol-level profiling methodology followed in Xprof may be used to design profiles for any general client-server system. Information gained from such profiles would be of great help in designing strategies for task partitioning and load balancing.

ACKNOWLEDGEMENTS

The authors would like to thank the referees for their detailed and insightful comments. This research has been supported by the Joint Services Engineering Programs (JSEP) under Contract N00014-90-J-1270, Dr. Lee Hoevel at NCR, the AMD 29K Advanced Processor Development Division, Matsushita Electric Industrial Corporation Ltd., Hewlett-Packard, and the National Aeronautics and Space Administration (NASA) under Contract NASA NAG 1-613 in cooperation with the Illinois Computer laboratory for Aerospace Systems and Software (ICLASS).

References

[1] S. Graham, P. Kessler, and M. McKusick, "An execution profiler for modular programs," *Software Practice and Experience*, vol. 13, pp. 671–685, 1983.

[2] AT&T Bell Laboratories, Murray Hill, N.J., *UNIX programmer's manual*, January 1979.

[3] R. Scheifler and J. Gettys, "The X window system," *ACM Transactions on Graphics*, vol. 5, pp. 79–109, April 1986.

[4] J. Gettys, P. Karlton, and S. McGregor, "The X Window system, version 11," *Software Practice and Experience*, vol. 20, pp. S2/35–S2/67, October 1990.

[5] R. Scheifler, *X Window System Protocol: X Version 11, Release 4*. MIT X Consortium, Massachusetts Institute for Technology, Laboratory for Computer Science, 1988.

[6] G. Champine and D. Geer, Jr., "Project Athena as a distributed computer system," *IEEE Computer*, vol. 23, pp. 40–51, September 1990.

[7] S. Mullender, G. van Rossum, A. Tannenbaum, R. van Renesse, and H. van Staveren, "Amoeba: A distributed operating system for the 1990s," *IEEE Computer*, vol. 23, May 1990.

[8] M. Lorence and M. Satyanarayanan, "IPwatch: A tool for monitoring network locality," *Operating Systems Review*, vol. 24, pp. 58–80, January 1990.

[9] D. Mirchandani and P. Biswas, "Ethernet performance of remote DECwindows applications," *Digital Technical Journal*, vol. 2, pp. 84–94, Summer 1990.

[10] R. Droms and W. Dyksen, "Performance measurements of the X window system communication protocol," *Software Practice and Experience*, vol. 20, pp. S2/119–S2/136, October 1990.

[11] J. Dunwoody and M. Linton, "A dynamic profile of window system usage," *Proceedings of the 2nd International Conference on Computer Workstations*, pp. 90–99, March 1988.