A Dynamic Profile of Window System Usage

J. Craig Dunwoody and Mark A. Linton
Stanford University

Abstract

Window systems are in widespread use on workstations. The interface between application programs and the window system can significantly affect interactive performance. In a server-based window system, this interface consists of an inter-process communication protocol between client processes and a window server. Optimizing this interface requires knowledge of how application programs use a window system. To obtain a dynamic profile of user input and application requests, we have built an instrumented version of the X window system[6] and installed it on a number of workstations.

We have found that, in our environment, over 90 percent of all requests are to display text, draw a rectangle, or fill a rectangle. Most text strings, vectors, windows, and area fills are relatively small; most area copies are large. The X client library reduces communication overhead by batching an average of four requests to the server in a single message. Overall, our data show that current server-based window system protocols can be quite efficient in actual use.

1 Introduction

A window system[2] has become a standard software component on general-purpose workstations. Window systems provide an intuitive visual interface to multitasking, a standard I/O interface for graphics applications, and access to a collection of generic tools such as terminal emulators and text editors. Since the window system provides a central point of access to graphical I/O facilities, it is important that it be efficient.

Two approaches to window system organization are in widespread use. In a library-based window system, an application program performs graphics I/O operations by calling library routines that directly use the input and display device interfaces provided by the operating system. Accesses to the display by independent application processes are synchronized by a central lock manager (often part of the operating system). Examples of library-based window systems include SunWindows[8] and MEX[5]. In a server-based window system, each application program communicates via messages with a window server process that performs all graphics I/O operations on behalf of its clients. Examples of server-based window systems include VGTS[3], X[6], and NeWS[7].

An important advantage of server-based window systems is that application programs can run on a remote system and send messages across a network to perform graphics I/O operations. Also, if processes cannot share libraries (as is the case in UNIX), a server uses less memory, and a server can be updated without relinking application programs.

A major concern in the design of a server-based window system is the protocol between application programs and the server. A library-based window system will usually have less overhead than a server because obtaining a lock is usually faster than sending a message, but the message overhead may be insignificant depending on the design of the protocol and how it is used by clients.

To characterize the dynamic usage patterns of the X window system, we have instrumented the X server to collect information about input events and application requests. We have installed this modified server on workstations that are used for software development, document preparation, and VLSI CAD. Although the profile of use that we have gathered is for a particular implementation (X Version 10, Release 4), our approach can be applied to any server-based window system.

2 Background

A window system manages concurrent access to graphics devices in much the same way that an operating system manages concurrent access to other machine resources. In our terminology, a window system is independent from a window manager, which is a program that determines the user interface and policy for screen space allocation.

The difference between library-based and server-based window systems is in how they support concurrency. In SunWindows, for example, a simple kernel locking mechanism is used to synchronize access to the display device so that drawing commands issued by independent processes do
not interfere with each other. Locking operations are expensive because they require a system call, and overall system performance depends on each application program choosing an appropriate granularity for locking. These problems are avoided in Mex[1], which relies on a graphics resource manager in the kernel to keep track of multiple graphics processes and provide fast context switching between them.

In server-based window systems, a single program "owns" the graphics devices; all other programs must send requests to this server to access a device. These requests are sent as messages over an inter-process communication (IPC) connection. A client program may run on the same machine as the window server or on a different machine. Just as in a library-based system, application programs call library routines to perform graphics I/O operations. In a server-based system, however, the library routines are simply stubs that package requests into messages and send them to the server.

The current implementations of the X and VGTS window servers are single-threaded; that is, the server processes one request at a time. The basic loop of these servers is to wait for either input or a client request and then handle the event (typically giving priority to input). Thus, synchronization of display operations occurs entirely as a side effect of the communication between applications and the server. If multiple threads of control can run within the server, additional synchronization must be performed while processing requests. This synchronization is still likely to be cheaper than that required in a library-based system because fewer context switches are necessary.

The principal advantage of library-based window systems is the potential for achieving better graphics performance by going directly to the device interfaces. This potential is not always realized, however, because of locking overhead. Furthermore, measurements of VGTS have shown that a high-level graphics protocol and batching of requests can make the overhead of communication between applications and the server negligible. This result is relatively insensitive to the bandwidth of the connection between the client and server; even remote client processes can obtain good graphics performance. Due to load sharing between the remote and local CPUs, remote clients may actually perform better than if they were local. The minimal loss of performance, combined with the flexibility of network access, has made the server-based approach more popular.

3 Method

The goal in instrumenting any system is to minimize the effect on performance and reliability during normal operation while collecting enough information to be of use. We therefore focused on the outermost loop of the X server, which is an infinite loop that waits for events. If an input event is available from the keyboard or mouse, the server reads the event and sends it in a message to the appropriate client. Input events that are not of interest to any client are ignored. If a request from a client has been queued, the server dequeues the request and calls the appropriate internal routine. If a message is pending from a client, the server reads and queues as many requests as possible.

In deciding what data to collect, we tried to obtain information that would answer the following questions:

1. Which X operations are used most heavily?
2. How many clients does the server typically have?
3. How many client requests are received in a single message?
4. How many bytes are sent in a text operation?
5. How long are vectors?
6. How large are area fills and copies?
7. How large are windows? How much of a window typically needs to be redrawn when an obscured part becomes visible?
8. How many requests are received in a single message from a client?

3.1 Collecting Data in the Server

We added code to the server’s main loop to record information in a profile area for each input event and client request. Only input events that are actually sent to some client are recorded. The profile area is a fixed-size structure consisting of two parts: a global area for client-independent data, including input events, and a per-application area for information about output requests.

We initially recorded all information in a global area, but we found that our results were skewed by "desk-accessory" applications. Two popular programs in our environment are clock, which uses polygons to simulate a seven-segment digital clock display, and iostat, which displays a bar graph of the system load, updating it every five seconds. The use of per-application areas allows us to ignore these desk-accessory programs and to study the variations in request patterns across application programs.

Both the global and the per-application profile areas use arrays of buckets to represent distributions of event times and operation parameter sizes. For wide-ranging quantities such as screen areas, the buckets are arranged on a \( \log_2 \) scale.
The global profile area is defined by the following C structure:

```c
struct GlobalStats {
    int startT ime;
    int buttons[NumTimes];
    int enters;
    int motions;
    int exposures[NumAreaSizes];
    int clientTimes[MaxClients];
};
```

The `startTime` field contains the time when the buffer was last zeroed. The `buttons` field is an array of buckets representing the interarrival time distribution of keyboard key and mouse button down events using a log scale. Each event contains a timestamp in ten-millisecond ticks. When a key or button down event occurs t ticks after the previous one, `buttons[log(t)]` is incremented. The `enters` field is incremented each time a client is sent a window entry event (indicating that the mouse cursor has just entered a window), and the `motions` field is incremented when a mouse motion event is sent to a client.

The `exposures` field contains the distribution of the size (in pixels) of rectangular areas that are exposed by manipulating windows. An exposure event is sent to a client when a previously obscured area becomes visible and needs to be redrawn. Finally, the `clientTimes` field contains the distribution over time of the total number of clients. `clientTimes[i]` contains the number of seconds during which the server had a total of `i` clients.

The per-application profile area is an array of C structures defined as follows:

```c
struct ClientStats {
    int opCounts[NumOpcodes];
    int texts[NumTextSizes];
    int vectors[NumVectorSizes];
    int polygonAreas[NumAreaSizes];
    int polygonVertices[NumVertexSizes];
    int windows[NumAreaSizes];
    int copies[NumAreaSizes];
    int fills[NumAreaSizes];
    int reads[NumReadSizes];
    int instanceTimes[MaxClients];
};
```

The `opCounts` field contains a count for each request code in the X protocol. The `texts` field contains the distribution of the length (in characters) of text drawing requests. Each bucket contains a count for a range of four lengths: when a text operation of length `c` is received, `texts[c/4]` is incremented.

The `vectors` field contains the distribution of vector lengths. The `polygonAreas` and `polygonVertices` fields contain the distributions of bounding box areas and number of vertices in polygon drawing requests, respectively. The `windows`, `copies`, and `fills` fields contain the size distributions of requests to create a window, copy an area, and fill an area. The `XPixFill` (solid color fill) and `XTileFill` (patterned fill) operations are both counted as fill requests.

The `reads` field contains the distribution of request message sizes as received by the server. When a message is pending from a client, the server reads as much data as possible and records the number of bytes read using the appropriate bucket.

The `instanceTimes` field contains the distribution over time of the number of instances of a particular application program. `instanceTimes[i]` contains the number of seconds during which `i` copies of the application program corresponding to this profile area were running.

We added a new operation, `XIdentity`, to the X protocol to allow an application program to identify itself to the server when it opens a connection. We dedicated one per-application profile area for each of five frequently-used application programs, and we modified each program to call `XIdentity` after it connects to the X server. Requests from applications that do not use `XIdentity` are recorded in an "Others" profile area.

We added two new operations to the X protocol to access the server profile information. `XGetProfile` is used to zero the profile areas, disable profile collection, or enable collection. `XGetProfile` sends the contents of the profile area to the client program. To gather the data, we wrote a program that connects to an instrumented X server, extracts the profile data, and then zeros the profile area. We used this program to gather data from each instrumented server once a day. We accumulated the information for each workstation in separate files, then summed to obtain an overall profile.

### 3.2 Applications

We collected individual profiles for the following five application programs:

- **xterm**
  Emulates a VT100 terminal in a window.

- **xemacs**
  A modified version of the Emacs text editor that supports multiple X windows and mouse input.

- **magic**
  A VLSI layout editor integrated with a collection of design tools. The editor enforces "Manhattan geometry" – all edges in the layout must be aligned with the horizontal or vertical axis. Because of this restriction, the editor can render any layout using only rectangular area fill operations. Because `magic` was originally written without a window system, it uses a single X window.
4 Results

We installed our instrumented X server on ten DEC VAXstation 11 workstations in our building. Six of the workstations have monochrome displays and are primarily used for software development and document preparation. The other four workstations have color displays with GPX graphics accelerators and are used primarily for VLSI layout. We collected profile data from each server over a period of two months.

4.1 Application Usage

Figure 1 shows the distribution of the total number of clients connected to the server and the number of xterm clients. On the average, a server has a single client 32 percent of the time. This single client is most likely the login xterm that runs when no one is logged on. Because many users do not log off of their workstations when they go home, 32 percent is not so much a measure of idle workstation time as it is an indication of how many workstations are in public areas.

When a workstation is in use, there are usually 9 or more clients. Of these clients, typically 5 or more are copies xterm. The remaining clients are often a copy of xemacs or magic and several desk-accessory applications. This profile is a result of our Unix environment, which is still oriented toward the tty-style interaction supported by xterm.

Table 1 shows the distribution of accumulated running time and observed client requests among the applications. Although xterm accumulated the most running time by a wide margin, it is responsible for a much smaller fraction of requests. This suggests that xterm windows are usually idle. The accumulated time for magic and vem is low because data were only obtained from these applications during the last week of the experiment. Even so, magic and vem accumulated over 24 and 5 hours of running time, respectively.

The applications in the Others category account for a disproportionate number of server reads and client requests relative to their accumulated running time. This is misleading because of the desk-accessory applications. When a user leaves a workstation without logging out, programs such as dclock and istat continue to generate requests.

As would be expected, the two text-oriented applications, xterm and xemacs, account for most of the text drawing and area copy (scrolling) requests. The Others category dominates area fills, polygons, and vectors because of the periodic updates by dclock and istat.

Table 2 shows the distribution of requests to the server from each application and overall. Operations that did not account for more than five percent of the requests from any program are not shown.

Overall, four graphics output operations account for over 90 percent of the requests: XText, XDraw (polygon drawing), XTileFill (patterned area fill), and XPixFill (solid area fill). Over 99 percent of the XDraw requests are rectangles.

A notable difference between magic and vem is the choice of fill operations. For simplicity, magic always uses XTileFill. A solid tile is used for solid fills. Because XPixFill may be more efficient on some hardware, vem uses XPixFill for the common case of a solid fill. Although magic could be changed to use XPixFill also, it would make more sense to have the server optimize the fill operation.

The request distribution highlights another anomaly with respect to idraw. To update part of a picture, idraw needs to clip output to a portion of its window. In X Version 10, clipping is done with a "transparent" window. Operations on these windows account for nearly 50 percent of the requests from idraw. Special clipping operations are provided in X Version 11, removing the need for transparent windows.

Without idraw’s clipping, the windowing operations would be insignificant in the distribution. This usage is also a reflection of our current applications, which use at most a few windows. We are developing new applications that use large window hierarchies and can therefore be expected to request windowing operations more frequently.

4.2 Input Events

Table 3 shows the distribution of input events delivered to client programs. Input events of all types are very infrequent compared to requests from application programs. Keystroke and mouse button events occurred significantly more frequently than motion events. Window entry events, which indicate that the mouse cursor has entered a new window, are surprisingly frequent relative to motion events, even in our environment of mostly single-window application programs.

<table>
<thead>
<tr>
<th>Event</th>
<th>Number (millions)</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key/Button down</td>
<td>3.4</td>
<td>43</td>
</tr>
<tr>
<td>Mouse motion</td>
<td>2.0</td>
<td>25</td>
</tr>
<tr>
<td>Window entry</td>
<td>0.7</td>
<td>8</td>
</tr>
<tr>
<td>Window exposure</td>
<td>1.8</td>
<td>23</td>
</tr>
</tbody>
</table>

Table 3: Input events

The X server sends an exposure event to a client when all or part of one of the client’s previously obscured windows has become visible and should be redrawn. Exposure
Figure 1: Distribution of number of clients

<table>
<thead>
<tr>
<th>Quantity</th>
<th>Number (millions)</th>
<th>Percentages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>idraw magic xterm</td>
<td>others</td>
</tr>
<tr>
<td>Client-seconds</td>
<td>213.6</td>
<td>0.2 0.0 0.0</td>
</tr>
<tr>
<td>Server Reads</td>
<td>43.9</td>
<td>0.8 0.0 0.1</td>
</tr>
<tr>
<td>Client Requests</td>
<td>195.9</td>
<td>0.6 0.3 0.2</td>
</tr>
<tr>
<td>Area Copies</td>
<td>1.1</td>
<td>0.0 0.0 0.0</td>
</tr>
<tr>
<td>Area Fills</td>
<td>67.1</td>
<td>0.4 0.5 0.6</td>
</tr>
<tr>
<td>Polygons</td>
<td>13.7</td>
<td>0.0 0.0 0.1</td>
</tr>
<tr>
<td>Filled Polygons</td>
<td>5.0</td>
<td>0.2 0.0 0.0</td>
</tr>
<tr>
<td>Vectors</td>
<td>44.3</td>
<td>0.2 0.6 0.1</td>
</tr>
<tr>
<td>Texts</td>
<td>57.3</td>
<td>0.0 0.0 0.1</td>
</tr>
</tbody>
</table>

Table 1: Running time and client requests

<table>
<thead>
<tr>
<th>Operation</th>
<th>idraw magic xterm xterm Others Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>XCreateTransparency</td>
<td>12 0 0 0 0 0 0</td>
</tr>
<tr>
<td>XDestroyWindow</td>
<td>12 0 0 0 0 0 0</td>
</tr>
<tr>
<td>XMapWindow</td>
<td>12 0 0 0 0 0 0</td>
</tr>
<tr>
<td>XSelectInput</td>
<td>12 0 0 0 0 0 0</td>
</tr>
<tr>
<td>XPixFill</td>
<td>2 0 61 4 2 19 14</td>
</tr>
<tr>
<td>XTileFill</td>
<td>22 50 4 0 2 28 20</td>
</tr>
<tr>
<td>XText</td>
<td>2 0 7 76 81 9 29</td>
</tr>
<tr>
<td>XDraw</td>
<td>16 42 8 17 7 35 28</td>
</tr>
<tr>
<td>XStippleFill</td>
<td>0 0 11 0 0 0 0</td>
</tr>
</tbody>
</table>

Table 2: Request distribution
events are necessary because the server does not store obscured portions of windows as offscreen images. This choice was made because an offscreen image may require a large amount of memory and it is slow or impossible to draw and move offscreen images on some graphics devices. In X Version 11, clients can request that the obscured portion of a window be saved; the server may ignore this request.

Figure 2 shows the size distribution of window fragments in exposure events. Because 46 percent of exposed areas are smaller than 1024 pixels in area, a good strategy for the server would be to retain images for small fragments only. Memory consumption and transfer speed are not problems for small areas, and the benefits of retained images are greatest for small fragments, for two reasons: First, when redrawing a small fragment, an application may need to do a significant amount of work to determine just what to draw. More typically, the application draws more than is necessary. Second, client-server communication is relatively inefficient when updates involve only a small number of requests because batching is less effective.

Figure 3 shows the distribution of key/button event interarrival times. The peak corresponds to a burst typing rate of 3 to 6 characters per second.

4.3 Graphics Output

Figure 4 shows the distribution of text string lengths. Under xemacs and xterm, text strings are quite short because of single-character echoing. Figure 5 shows the distribution of vector lengths. magic tends to draw relatively short vectors, while xdraw tends toward longer vectors. The large number of very short vectors in the overall distribution is due to istat, which constantly draws single-pixel vectors when the load is zero at night.

Figure 6 shows the overall distribution of sizes for windows and area copies. The surprisingly large fraction of small windows is partly due to the fact that many of our application programs use small subwindows to create banners and borders for each large window. The window manager also creates many small windows for pop-up menus and icons. Area copies are typically large, since they are mostly used to scroll all but the bottom few lines of text in a large window.

Figure 7 shows the distribution of area fill (XPixFill and XTileFill) sizes. The predominance of small fills in the overall distribution is due to the updating of clock and istat windows. The VLSI layouts rendered by magic have a fairly even rectangle size distribution, while the larger fills performed by xterm are on the order of the area covered by a line of text.

The implementation of a graphics operation is generally divided into two phases. The first is the setup phase, in which termination conditions are computed and, if special graphics hardware is to be used, registers are loaded. The time to complete this phase is typically independent of the size of the graphics operation. In the second phase, the operation is actually executed. The time to complete this phase is typically proportional to the size of the operation. Because of the small size of most graphics operations, the setup phase often dominates the cost of an operation. This is especially true when using fast graphics hardware with complex setup requirements. Our data suggest that the best way to improve real-world graphics performance is to concentrate on reducing this setup overhead.

4.4 Protocol Efficiency

Figure 8 shows the distribution of server read sizes in bytes. The minimum request is 24 bytes in X Version 10. On the average, the server received 4 requests in each read. From the overall distribution, it would appear that it is rarely useful to batch a large number of requests. However, the distribution is skewed by single-character echoing in text applications and background updates from clock and istat, which involve a small number of operations. The distribution for magic clearly shows that batching is very important for some applications.

The bimodal distribution of read sizes from magic is caused by a mixture of small changes and full window updates. The reason magic batches more operations than xemacs is that a typical circuit contains more rectangles than a typical editor buffer contains lines.

Using our profile data, along with some measurements of the cost of inter-process communication and the individual graphics operations [4], we can obtain a rough estimate of the overhead incurred by the X protocol. The worst-case overhead occurs during input echoing, where drawing is cheap and effective batching is not possible. Sending a single X request takes about 2.8 milliseconds on a VAXstation II/GPX, and drawing a single character takes 0.17 milliseconds. Thus, the communication overhead for echoing a character is 2.8/2.97 or 94 percent of the total time. This overhead is not a problem, however, because high throughput is not required in input echoing. Given a burst typing rate of six characters per second, the cost of echoing is still only 6*2.8/1000 or 1.7 percent of the CPU.

The lowest overhead is achieved during updates of large graphical structures, in which graphics operations are more expensive and batching is most effective. For example, consider a window update in magic. Assuming that area-fill requests are batched into 2K-byte messages, 85 can be sent in 3.8 milliseconds. For a moderate-size area fill (20,000 pixels), which takes about 3 milliseconds on the GPX, the protocol overhead is 3.8/(85*3+3.8) or 1.5 percent.

The X Version 11 protocol has several changes that improve performance over Version 10. First, instead of fixed 24-byte requests, X11 has variable-size requests with a 4-byte minimum header. Second, X Version 10 graphics requests must carry state information such as color and line style, while in X11 this state information may be stored in
Figure 2: Distribution of exposure event sizes

Figure 3: Key/button event interarrival times

Figure 4: Distribution of text string lengths
Figure 5: Distribution of vector lengths

Figure 6: Distribution of window and area copy sizes

Figure 7: Distribution of area fill sizes
the server and requests need only contain a reference to this information. Finally, X11 supports requests such as PolyLine and PolyRectangle that perform an operation multiple times, thereby amortizing the cost of the request header. The result of these changes is that more requests can be batched into a message of a given size, and the protocol overhead is thus lower.

Not apparent in our profile data is a problem in the processing of exposure events. If some windowing operation requires a large number of windows to be redrawn, the X server will generate a large number of exposure events. The client enters a loop consisting of reading an event and redrawing part of a window in response. Because reading an event causes all pending output to be flushed, this loop renders batching ineffective and thus drastically increases protocol overhead. The problem is especially apparent when a program generates a subwindow hierarchy and then maps it to the display, causing the X server to generate a flood of exposure events. Application programs can avoid this problem by queuing exposure events until an event read would block or a non-exposure event is received, then performing all of the redraws at once. Given that an increasing number of applications are using large window hierarchies, a better long-term solution would be to change the protocol.

Overall, our analysis indicates that a server-based system can be efficient. Even if a library-based system is an order of magnitude more efficient, the difference between 1 percent and 0.1 percent overhead is worth the benefits of the server approach.

5 Conclusions

Instrumenting a window server to provide a dynamic profile of window usage is straightforward. It is important to collect data on a per-application basis because a long-running program, such as a clock or load monitor, can skew results. Also, request patterns vary significantly among the different application programs.

Our measurements indicate that over 90 percent of client requests are for drawing text, rectangles, and filled rectangles. Looking at specific applications, we found that our software development tools mostly draw text, while VLSI CAD applications most frequently draw vectors, rectangles, and filled rectangles. Text strings, vectors, and area fills are heavily biased toward small sizes for most applications. It is therefore very important to keep setup overhead low for these graphics operations. Area copy sizes are typically large, on the order of one quarter of the total screen area. Most windows are small.

The distribution of server read sizes shows that effective use is made of batching multiple requests into a single message under certain circumstances, such as when a VLSI CAD program draws a circuit. Text-based applications make less use of batching because they tend to have less data to write. Frequently, a text editor or terminal emulator sends just a single request to echo a character.

Overall, our results show that client-server communication overhead can be quite low in a server-based window system. In the case of input echoing, where low delay is most important, communication overhead is high but CPU usage is low because IPC message times are much smaller than input event interarrival times. In the case of window updates, where high throughput is most important, overhead is low because many graphics output requests can be batched into a single IPC message.

Our profile data are strongly influenced by the behavior of the application programs that we use, many of which have simply been ported from a non-window environment to run under a terminal emulator. We expect that window system usage will change as new tools that take full advantage of windows are developed. Our approach to profiling window system usage, however, will continue to be applicable. We believe that profiling facilities of this type should be a standard part of future window systems.
References


