PERFORMANCE ASPECTS OF COMPUTERS WITH GRAPHICAL USER INTERFACES

BY

ALOKE GUPTA

B.Tech., Indian Institute of Technology, 1985
M.S., University of Illinois at Urbana-Champaign, 1990

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1993

Urbana, Illinois

PERFORMANCE ASPECTS OF COMPUTERS WITH GRAPHICAL USER INTERFACES

Aloke Gupta, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 1993
Wen-Mei W. Hwu, Advisor

Graphical interfaces and windowing systems are now the norm for computer-human interaction. Also, advances in computer networking have given computer users access to immense distributed resources accessible from anywhere on the network. In this setting, the desktop, or personal, computer plays the role of a *user-interface engine* that mediates access to the available resources. Interface paradigms, such as the "desktop metaphor" and "direct manipulation," provide the user with a consistent, intuitive view of the resources. Traditional computer research has focused on enhancing computer performance from the numerical processing and transaction processing perspectives. In the research described in this thesis a systematic framework is developed for analyzing and improving the performance of window systems and graphical user interfaces. At the system level a *protocol-level profiling* strategy has been developed to profile the performance of *display-server* computers. A sample protocol-level profiler, *Xprof*, has been developed for applications under the X Window System. At the microarchitecture level the memory access characteristics of windowing programs are studied. Cache tradeoffs for a frame-buffer cache are presented. A cache organization is proposed to improve the frame-buffer performance.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Wen-Mei Hwu, for his support and advice over the years. I would like to thank my wife, Molly Shor, for her love and moral support. I would like to thank my parents, Amrit and Nina Gupta, for their love, care and support.

Thanks go to the members of the IMPACT group, with whom I have had many fruitful discussions over the years. A partial list includes Sadun Anik, Roger Bringmann, Pohua Chang, William Chen, David Gallagher, Andy Glew, Grant Haab, Rick Hank, John Holm, Michael Hsiao, Tokuzo Kiyohara, Scott Mahlke, David McCracken, Roland Ouellette, and Nancy Warter.

I would like to acknowledge Professor Michael Loui, and Liz Rudnick for helping me improve the presentation of this thesis material.

Finally, I would like to thank all of the members of the Illini Folk Dance Society for helping make my stay in Urbana so enjoyable.

TABLE OF CONTENTS

Page

1. INTRODUCTION 1
1.1 Graphical User Interfaces 2
1.2 Distributed Computing 6
1.3 The X Window System 9
1.4 Organization of Thesis 11

2. PROTOCOL-LEVEL PROFILING 14
2.1 Review of Profiler Strategies 17
2.1.1 Client profile 17
2.1.2 Server profile 18
2.1.3 Network profile 20
2.1.4 Xprof: protocol-level profiling 20
2.2 Protocol-level Profiling 21
2.2.1 Server time 22
2.2.2 Network time 23

3. XPROF: AN EXECUTION PROFILER FOR THE X WINDOW SYSTEM 24
3.1 Messages in the X Window System 24
3.1.1 Requests 25
3.1.2 Request attributes 26
3.2 Trace Collection 26
3.3 Trace Analysis: Xprof 28
3.3.1 Steps in processing the trace input 28
3.4 Collection of Server Metrics: Xmeasure 30
3.5 Xprof Details 34
3.5.1 Server time 34
3.5.2 Statistical distributions 39
3.6 Refining the Measurements 42

4. RESULTS FROM XPROF ANALYSIS 44
4.1 Execution Profile for Requests 45
4.2 Message Statistics 47
4.2.1 Message categories 47
4.2.2 Individual messages 48
4.3 Cross-server Profiling 49
4.4 Client-server Load Partitioning 53
4.5 Effect of Network Speed 53
4.6 Limitations of Xprof 57
4.7 Xprof Audiences 59

5. FACTORS IN MEMORY-SYSTEM DESIGN 60
5.1 Cache Memory 60
5.2 Issues in Cache Design 63
5.3 The Video RAM 66

6. MEMORY ACCESS CHARACTERISTICS OF DISPLAY-ORIENTED PROGRAMS 72
6.1 Instruction-level Trace Collection 72
6.2 Benchmark Programs 74
6.3 Dynamic Instruction Distributions 75
6.4 Working Sets 76
6.5 Memory Traffic 77
6.6 Sequential-access Length Distributions 79
6.7 Summary of Access Characteristics 81

7. DESIGN OF A FRAME-BUFFER CACHE 82
7.1 Introduction 82
7.2 Guiding Principles 85
7.2.1 Cache size 85
7.2.2 Write policy 86
7.2.3 Associativity 86
7.2.4 Write-allocate policy 87
7.2.5 The need for flushing mechanisms 87
7.2.6 Write buffers 88
7.3 A Suggested Frame-buffer Cache 88
7.4 Line Size Effects 89
7.5 Cache Size Effects 90
7.6 Associativity Effects 91
7.7 Flushing Effects 92
7.8 Effect of Allocate-no-fetch Policy 93
7.9 Cache Configurations 94
7.10 Future Work 95

7.11 Conclusions 97

8. CONCLUSIONS 99

REFERENCES 101

VITA 109

LIST OF TABLES

Table Page

3.1: Xmeasure measurements for common client requests. 31
4.1: Excerpt of the execution profile of Xtex. 46
4.2: Excerpt of the execution profile of Xtex. 46
4.3: Message statistics for Xtex. 47
4.4: Total bytes for each request in the trace of Xtex. 48
4.5: Statistics for PolyText8 messages in the trace of Xtex. 49
4.6: Cross-server profile for Xtex. 50
4.7: Cross-server profile for Ximage. 51
4.8: Cross-server profile for Xtetris. 51
4.9: Cross-server profile for Xmagic. 52
4.10: Client-server load partitioning for the Sun4/IPC. 53
5.1: Selected parameters for the Fujitsu MB81461-12 VRAM chip. 71
6.1: Benchmark programs. 75
6.2: Dynamic instruction distribution. 76
6.3: Memory access working sets. 77
6.4: Memory traffic. 78
6.5: Frame buffer sequential-access lengths (in bytes). 80
7.1: Access time for suggested frame-buffer cache. 89
7.2: Access time for frame-buffer caches. 95
7.3: Traffic ratios for frame-buffer caches. 95

LIST OF FIGURES

Figure Page

1.1: Distributed system model. 7
1.2: The X Window System. 10
1.3: Software architecture of the X Window System. 12
2.1: Procedure-level profilers in a client-server environment. 18
3.1: X protocol messages. 25
3.2: Protocol trace collection. 27
3.3: Overview of trace analysis. 29
3.4: Typical Xmeasure entries for server parameters. 33
3.5: C language data structures for the cost model of messages. 35
3.6: Message costs as maintained by Xprof. 36
3.7: C language data structure for message statistics. 39
4.1: Effect of network speed on Xtex profile. 54
4.2: Effect of network speed on Ximage profile. 55
4.3: Effect of network speed on Xtetris profile. 55
4.4: Effect of network speed on Xmagic profile. 56
5.1: Memory hierarchy in computer systems. 61
5.2: Cache organization in typical systems: "Harvard" architecture. 62
5.3: Organization of a 256-Kbit (64K x 4) VRAM chip. 68
6.1: Relationship of frame-buffer addresses and screen real estate. 80
7.1: Memory system model. 84
7.2: Effect of line size on access time. 90
7.3: Effect of cache size on access time. 91
7.4: Associativity effects. 92
7.5: Flushing effects. 93
7.6: Effect of allocate-no-fetch policy. 94

1. INTRODUCTION

Graphical user interfaces, or GUIs, have emerged as the standard means of human-computer interaction. The widespread availability of inexpensive personal computers and workstations has led to the design of computers oriented to interactive, visual activity between the computer and the user. Traditional computer research has focused on the design of computers for numerical and transaction-oriented applications. However, in this thesis research, the focus is on the design of the computer as a user-interface engine, or *display server*.

Three major trends have fueled the importance of designing computers with attention to the visual interface. First, personal computers and workstations have come into widespread use. Unlike the time-shared machines of the early days, these are typically dedicated to providing service to a single user. Consequently, much of the available computing power may be dedicated to the user-interface mechanism. Second, this emphasis on user-interface technology has led to a new paradigm of computer-human interfaces in which computers are made accessible to nontechnical users. In this paradigm, which is

still evolving, the computer interface is presented to the user in the form of real-world metaphors, such as the *desktop metaphor*, thus enriching the human-machine dialog. Examples of this evolution in user-interface technology include GUIs, window systems, audio input, and pen-based computing. Consistent *look-and-feel* specifications hide the complexities and details of the underlying hardware and ease the migration of applications across different architectures and operating systems. Third, widespread use of computer networks encourages access of resources anywhere on the network and from anywhere on the network. Thus, users can access vast resources from within their personal computers and have access to a "metacomputer" that spans the computation capabilities of the fastest supercomputer and the data availability of every on-line database. The Internet is probably the best known and most widely used research computer network today. Private networks, such as Compuserve, have brought networked resource access to businesses and individuals. For instance, the Easy Sabre system on Compuserve enables users to query airline flight reservation information and to make their own flight reservations on the major U.S. airlines from their homes. Also, wireless computer networks, such as the Ardis system built by IBM and Motorola, can be accessed from within their portable computers.

1.1 Graphical User Interfaces

In the early days of computing, computer interfaces were geared towards highly skilled technicians who had a detailed knowledge of the machine language. Now, graphical user

interfaces, or GUIs, provide ordinary users with easy access to a wide variety of computer applications.

The first generation of computers, in the 1950s, were *batch processing* systems. Users typically submitted their jobs, on punch cards, to an operator who then queued the jobs on a single-tasking computer. After a job was completed, the output would be returned to the user. By the early 1960s, *multiprogrammed* computers were able to support multiple user programs concurrently. *Time-sharing* systems were developed to support on-line sessions of multiple users. In such systems, users could interact with the computer system over a teletype *terminal*. This was the origin of the familiar *command-line interface*, or CLI, in which a shell program mediates between the user and the computer system. Such systems made it possible for users to develop programs and to locate and correct errors interactively instead of having to wait several hours for a job to complete in a batch processing environment. By the mid 1970s, advances in microprocessors and in computer networking were making it possible to provide *personal computers*, or PCs, and *personal workstations* on the desks of individual users. This trend gave the impetus to the development of graphical user interfaces. At first, *menu systems* were provided to abstract the user from the symbolic, mnemonic, details of the command line interfaces. Gradually, the availability of inexpensive bit-mapped displays has led to the widespread adoption of graphical user interface standards such as Apple Computer's Macintosh GUI, Microsoft's Windows, IBM's Presentation Manager, Next Corporation's NextStep, SunSoft's OpenLook, and the Open Software Foundation's Motif. *Window*

systems, such as the X Window System, developed at MIT, have provided the operating system foundations for building such interfaces.

The foundations of graphical user interfaces were laid in the mid 1970s in the Alto project at Xerox PARC, the Palo Alto Research Center [1, 2]. The unique innovations of the Alto included the use of a high-resolution, bit-mapped, raster-scan display designed to resemble a sheet of paper. The Alto was meant to be used in a networked environment in which servers, such as file-servers and shared printers, would provide services over an Ethernet [3, 4] network. The user interface was built around the *desktop metaphor*, which views the computer display as a virtual desktop on which user processes are arranged much as overlapping sheets of paper lie on a real desktop. Input devices included the standard keyboard and a *mouse*[1] pointing device. *Icons* were used to represent computing objects in the form of familiar real-world objects. Thus, instead of seeing files and directories, users were presented with icons resembling folders and file cabinets. The icons were manipulated by invoking a *direct manipulation* [7, 8] metaphor. For instance, clicking on a folder would open a word processor for modifying the folder. In contrast, all other systems at the time followed the *tool* metaphor in which a user first opens an application, or tool, and then acts upon a data object.

The Alto was a prototype and not a commercial product. Eventually, Xerox marketed the Xerox Star computer. Many of the ideas from the Alto and the Star found their way into the Apple Lisa [9], and its successor the Apple Macintosh [10, 11], which was

[1]The mouse was invented by Doug Engelbart's group at the Stanford Research Institute [5, 6].

introduced in 1984. The overwhelming commercial success of the Macintosh brought wide acceptance of the desktop metaphor and also led to the development of the Microsoft Windows system in the competing market for IBM PCs.

In the Unix workstation domain, proprietary window systems and GUIs prevailed for a while. The industry eventually accepted the X Window System [12, 13, 14, 15] from MIT as an industry standard. One characteristic of Unix workstation users is that they are frequently developers, or *hackers*, who prefer the tool-oriented metaphor over the desktop metaphor. Nevertheless, GUI standards such as Motif and Open Look have been developed to provide intuitive user interfaces as a layer over the underlying window system. The X Window System is discussed in greater detail in Section 1.3.

Future directions in user interfaces will certainly enrich the human-computer interaction in new and creative ways. Emerging interaction devices such as the *DataGlove* [16] allow a computer to sense hand position and orientation in three dimensions. *Virtual realities* [17, 18] are computer-generated environments with realistic appearance, behavior and interaction techniques. The emerging discipline of *scientific visualization* [19, 20, 21] employs computer graphics techniques to render results of scientific computations in the form of visual images. *Multimedia computing* [22, 23, 24, 25, 26, 27, 28] aims at integrating live video and speech input into computing environments. One consequence of such an integration is the blurring of the the dividing line between computers and consumer electronics devices. Data compression standards are being developed to support transfer of image data over existing low-bandwidth channels [29, 30, 31].

1.2 Distributed Computing

The widespread proliferation of computer networks has made distributed computing accessible to a large population of users. Thus, a user has access not only to the resources on the desktop, but also to distributed resources worldwide. For instance, a researcher in oceonography at the Oregon State University, located in Corvallis, Oregon, can submit computation jobs to the supercomputer center at the University of Illinois, located in Urbana, Illinois, and receive the results over the Internet network. Similarly, e-mail standards and file transfer protocols allow widespread dissemination of information to the interested communities. Electronic banking machines, airline reservation systems, and on-line electronic bulletin boards are common examples of the utility of distributed computing systems.

The *client-server* model of computing [32, 33] allows transparent access to resources distributed across a computer network. In this model resources are managed by *servers*, which provide high-level services to application programs, or *clients*. The client programs communicate with the servers by means of a high-level protocol. Thus, the client-server model allows computers of widely different types to share resources as long as they follow the appropriate communication protocol. Common examples of this model include the Network file system, or NFS [34], which allows for distributed file access, and the X Window System, which allows distribution of window system components. As shown in Figure 1.1, a distributed system may contain, among others, *file servers* for maintaining the disk storage in the system, *computation servers*, which are fast numerical processing

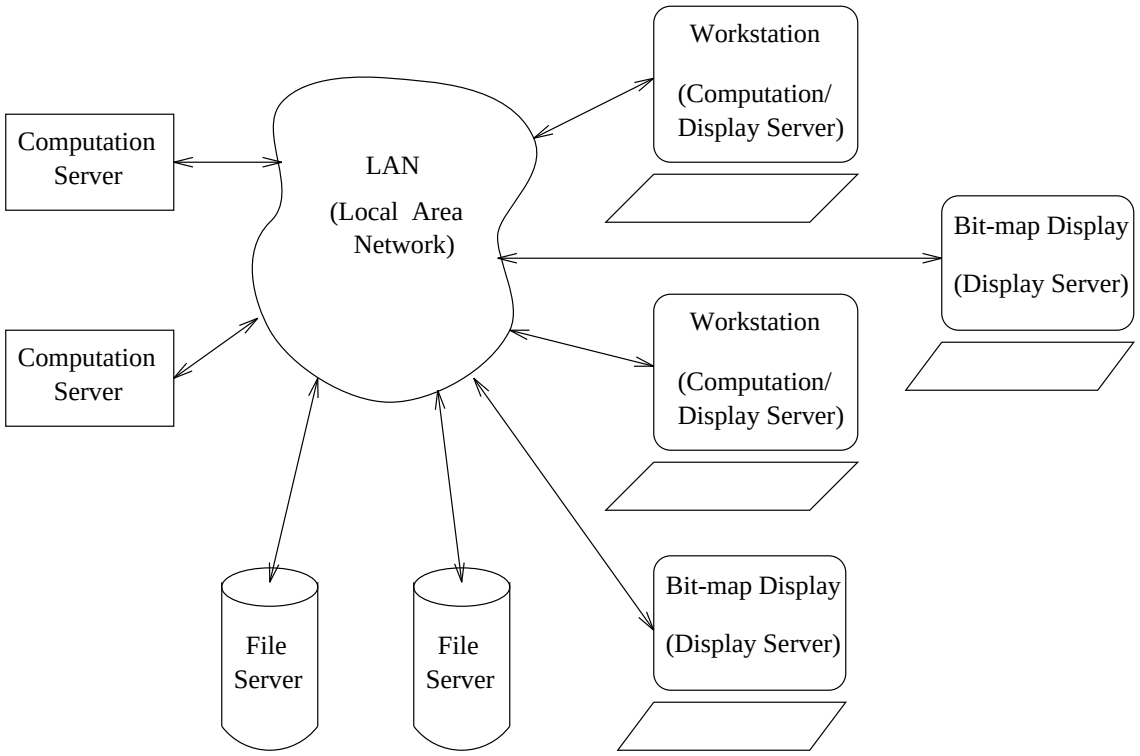

Figure 1.1: Distributed system model.

oriented machines, and *display servers*, which are the window-oriented display units with or without the ability to perform user computations. In Figure 1.1 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 *bit-map displays*. In the X Window domain, the latter are commonly referred to as *X terminals*.

Distributed operating systems have evolved over time [35]. These enable resource sharing over networks, to varying degrees of transparency. As yet, there is no standard distributed operating system, but various solutions have been proposed and implemented over time and have influenced the nature and configuration of distributed computers. The Athena project [36] at MIT is one of the largest centrally managed educational

networks of heterogeneous workstations. Popular innovations from Athena, which is designed to support several thousand workstations, include the X Window System and the authentication protocol Kerberos. The network protocol used in Athena is the popular Ethernet, which came out of the Xerox Alto/Star project discussed earlier in Section 1.1. Network services available in Athena include name service, file service, printing, e-mail, notification service, service management, authentication, and installation and update. Athena was established in 1983 and is still growing.

The Andrew [37, 38] project at Carnegie Mellon University is similar to Athena in being a system for supporting educational and research computing. A distributed file system called the Andrew File System, or AFS, has been developed in the Andrew project. In this file system, scalability and security in a distributed environment were primary design concerns. Andrew and Athena have shared ideas over time. For instance, Andrew now uses the X Window System and Athena uses the Andrew File System. In the wider research and commercial setting, the de facto standard in distributed file sharing is the Sun Network File System, or NFS [34].

Athena and Andrew address the requirements of resource sharing over a local network. Some other projects have studied the sharing of resources over networks that span a much larger geographical extent. Amoeba [39, 40], which has been developed at the Free University and the Centre for Mathematics and Computer Science in Amsterdam, is designed to run in both local and wide area networks. An experimental system currently connects Amoeba systems in wide area networks spanning several countries in Western

Europe. The Grapevine system [41], which is used extensively within the Xerox corporate environment, is another example of a transnational distributed system.

Other examples of distributed operating system projects include Sprite [42] at the University of California, Berkeley, Eden [43, 44] at the University of Washington, Mach [45] at Carnegie Mellon University, V [46] at Stanford University, Locus [47] at the University of California, Los Angeles, and Clouds [48] at the Georgia Institute of Technology. Other projects in distributed systems have evolved algorithms for distributed object management [49, 50, 51].

1.3 The X Window System

The X Window System [12, 13, 14, 15] has emerged as the de facto standard in the Unix workstation domain. It was developed in the early 1980s as part of project Athena at MIT. Portability was an early design concern in X Window since MIT was using a diverse group of computer workstations from different vendors. It was designed as a distributed, device-independent, network-transparent windowing and graphics system. Being an open standard, it gained wide acceptance in industry. Currently, the development of the X Window standard is administered by the X Consortium, whose members include all of the major workstation vendors such as Sun, HP, DEC, IBM, SGI, and Tektronix.

As shown in Figure 1.2, the X Window System follows the client-server model of computing. A *display server,* the *X server* [52], manages the actual graphics-display hardware and controls access to the graphics and window functions on the display. It

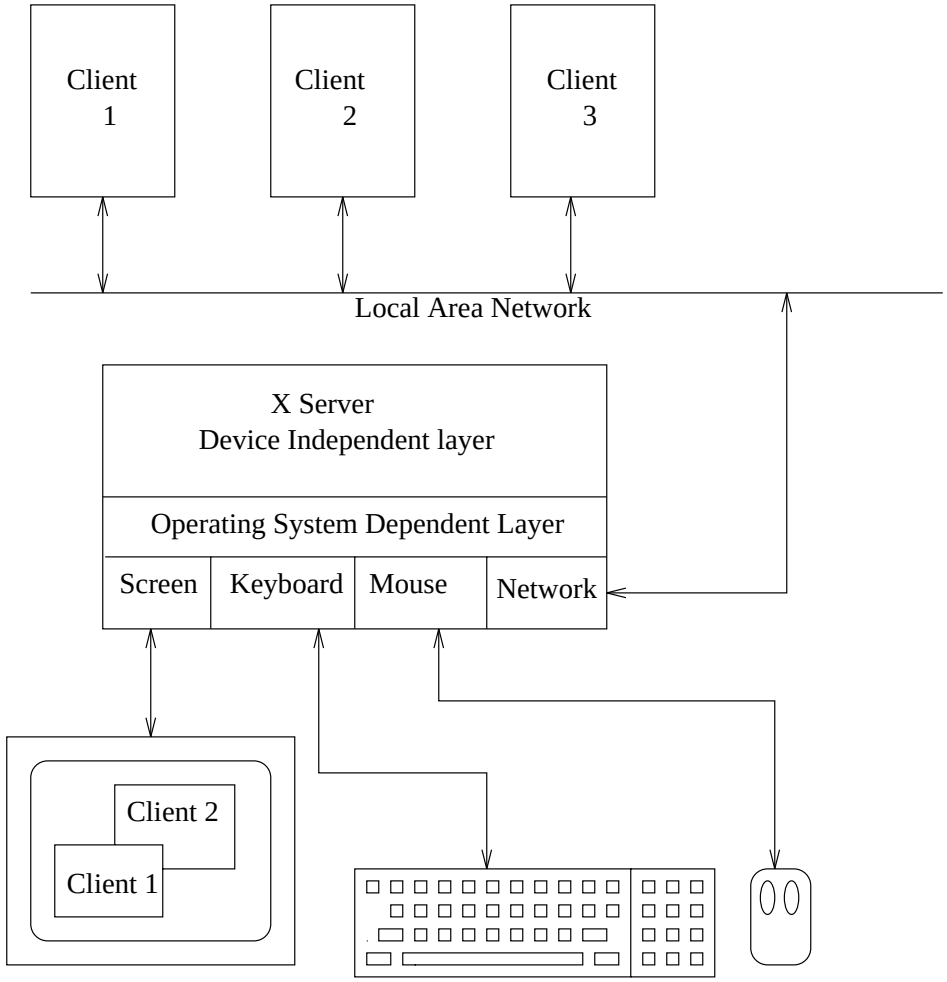

Figure 1.2: The X Window System.

also receives user input from input devices such as the keyboard and mouse. The application programs, or *clients,* achieve graphics and window functions by means of high-level messages exchanged with the display server; the messages follow a network protocol, the *X protocol* [53]. 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.

An important characteristic of the X Window System design is that the base window system provides *mechanism, not policy* [12]. The base window system provides mechanisms such as graphics primitives. Window management policies are enforced by a *window manager* [54], which is simply another user process and can be freely chosen by a user. User interface characteristics are enforced by *toolkits* and GUI specifications, such as Motif, which are built as a layer on top of the base system. Also, there is a built-in mechanism for extending the core protocol. Any proposed extensions are submitted to the X Consortium, and useful ones are included in future distributions. Some current extensions include PEX [55], or PHIGS Extensions to X, which is a 3-dimensional imaging model. The Display Postscript, or DPS, imaging model is also supplied as an extension with some X implementations. This versatility of X is an important reason for its widespread popularity. Figure 1.3 shows the software model of the X Window System.

1.4 Organization of Thesis

The rest of this thesis is organized as follows. Chapter 2 develops a new *protocol-level profiling* strategy for analyzing client-server interaction. Chapter 3 describes a protocol-level profiler, *Xprof*, which has been developed in this research for generating meaningful execution profiles of X Window applications. Chapter 4 presents results from Xprof analysis of some common window-oriented applications. Chapter 5 reviews factors in memory system design. Chapter 6 discusses the frame-buffer memory access characteristics for

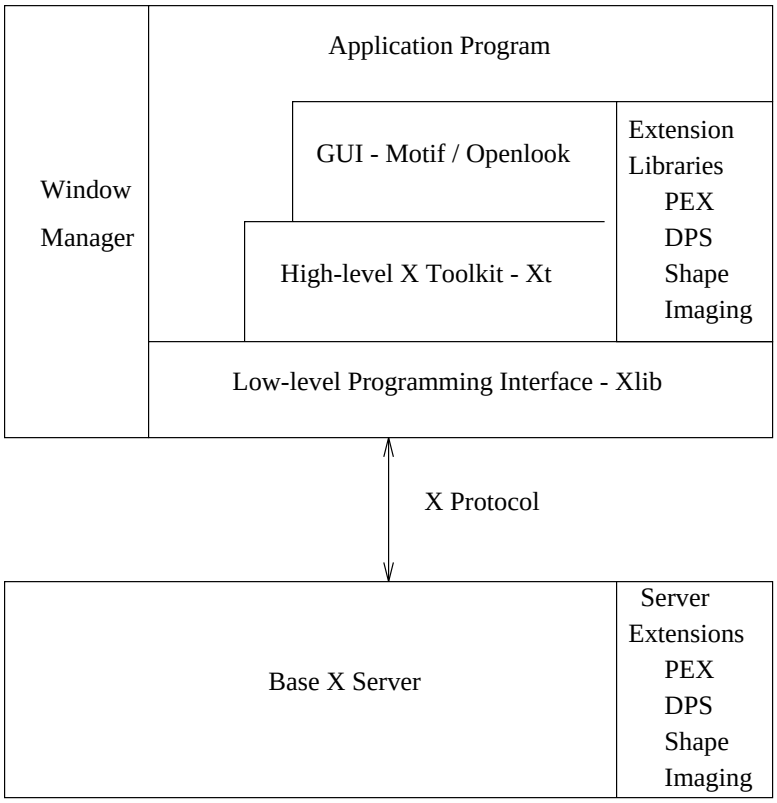

Figure 1.3: Software architecture of the X Window System.

some common X Window programs. Trace-driven simulation is done to analyze this memory behavior for extensive low-level traces collected on a DECstation 3100 workstation. This discussion motivates the development of different caching strategies studied in Chapter 7. Finally, Chapter 8 summarizes the contributions of this research.

2. PROTOCOL-LEVEL PROFILING

The widespread use of workstations and personal computers as *display-servers* motivates a desire to identify the critical design issues for supporting display-oriented applications. Execution profiles are important in analyzing the performance of computer programs on a given computer system. 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 about the performance bottlenecks in a program, allowing the developers to better focus their efforts when tuning the performance of a program or of the computer system.

However, accurate and complete profiles are difficult to achieve for programs that follow the *client-server* model of computing. In this model, which is followed by programs in the X Window System, 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.

A meaningful profile for a client program, in a client-server model, must account for several aspects. The first aspect is the execution profile of the client program itself. This can be measured by a traditional execution profiler [56, 57]. The second aspect is the server-time spent in servicing the requests at the server. The final aspect is the time spent in transferring the requests, and results, between the client and server. This is especially meaningful when the two programs are run on different machines and exchange messages over a network.

In the course of this research a new profiling strategy has been developed for analyzing client-server interaction. The central idea of this *protocol-level profiling* strategy is to analyze a protocol-level trace of the interaction between the client application and the server and thereby construct an execution profile from the trace and a set of metrics about the display server.

A protocol-level profiler, *Xprof*, has been developed for generating meaningful profiles for X Window applications [58, 59, 60]. Xprof estimates the time spent in the display server and 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 a protocol-level profiling strategy are as follows. First, one can identify the most time consuming part of the client application by taking all aspects into account, including the time spent in the server and the network. Second, by combining the results 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, this technique permits cross-server profiling. One can take a trace from a particular client-server configuration and generate profiles for other servers and the same client. This allows application developers to tune their applications for many different 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 may be no need to recompile the client or server programs for tracing. This feature is especially useful in the X Window System 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 postprocessing of the trace can still provide a meaningful picture of the computation invoked by the clients.

In the following section, several profiling strategies are reviewed in the context of the X Window system.

2.1 Review of Profiler Strategies

2.1.1 Client profile

Procedure-level profilers such as *Prof* [56] and *Gprof* [57] are frequently used to derive the execution profiles of conventional programs. 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 a client program. As shown in Figure 2.1, procedure-level profilers generate information about the procedure calling pattern within a program. The information is in the form of the dynamic call graph for the functions within the program. The call graph consists of nodes, which represent each function, and edges, which represent calls from one function to another. The call graph is annotated to reflect the execution time spent in each function and the call frequency along each edge. 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 2.1, are merely stub routines that send the appropriate request messages to the server.

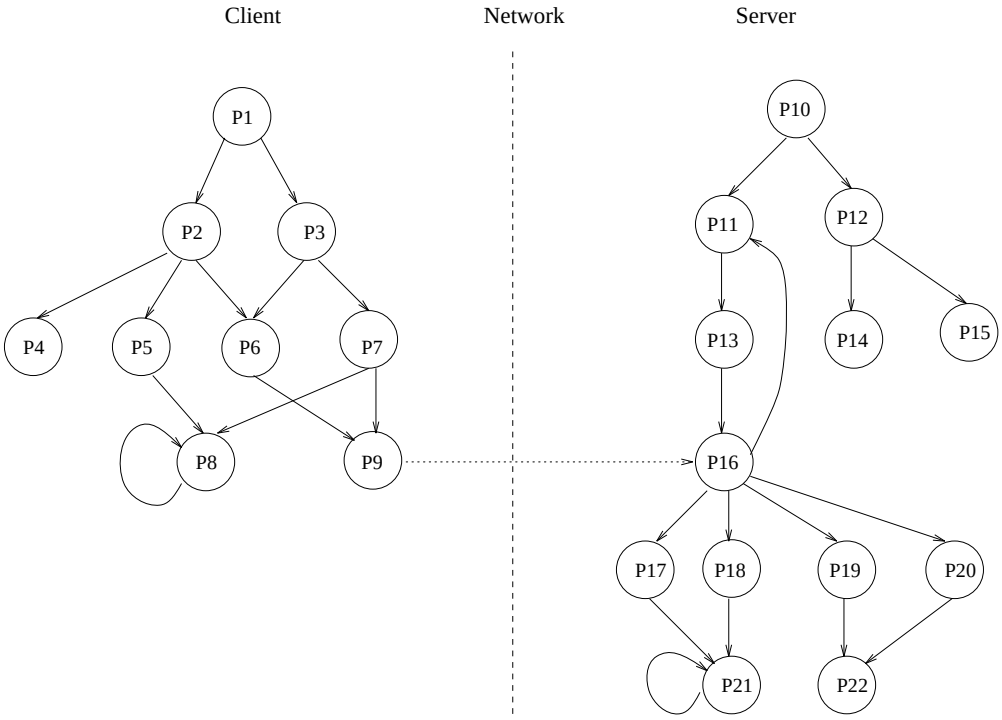

Figure 2.1: Procedure-level profilers in a client-server environment.

2.1.2 Server profile

The traditional procedure-level 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. Second, 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. Third, the profile usually gives overall 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, which is used by the X Window program *x11perf*, [1] a special measurement program measures the run times of 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: a measurement program is used to generate a set of parameters for the target display server, and the parameters are used in addition to a protocol trace to construct an execution profile.

[1] X11perf is distributed with the X Window source code available from MIT.

2.1.3 Network profile

Traditionally, network traffic is studied by measuring the load on a network by using a network monitor that logs all of the packets on the network [61, 62]. Such measurements can give a good idea of the transport time for 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 [63]. 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 [64].

2.1.4 Xprof: protocol-level profiling

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 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 for an 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.

2.2 Protocol-level Profiling

Protocol-level profiling involves the analysis of a protocol-level trace, which characterizes the interaction between the client and server programs. From this trace, the profiler 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 server and the network connection.

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.,

$$T = T_{client} + T_{server} \tag{2.1}$$

When the client and server programs execute, asynchronously, on different computation hosts, their activities go on with some degree of concurrency, and thus 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.

Therefore,

$$T = T_{client} + T_{server} - T_{overlap} \tag{2.2}$$

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, which are sent from the server to the client. For instance, in the context of the X Window system, mouse

movement and key strokes would result in event messages. Thus,

$$T_{server} = T_{requests} + T_{events} \tag{2.3}$$

For an X Window application, Xprof estimates the $T_{requests}$ in Equation (2.3) 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:

$$T_{requests} = \sum_{r_i \in R} T_{r_i} \tag{2.4}$$

The time 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., $T_{r_i}^{net}$.

$$T_{r_i} = T_{r_i}^{server} + T_{r_i}^{net} \tag{2.5}$$

2.2.1 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. For example, 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,

$$T_{r_i}^{server} = f(attributes_{r_i}) \tag{2.6}$$

2.2.2 Network time

The network time term, $T_{r_i}^{net}$ may be computed 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,

$$T_{r_i}^{net} = (bytesize_{r_i}/netspeed) + (blocking_{r_i} * netlatency) \tag{2.7}$$

3. XPROF: AN EXECUTION PROFILER FOR THE X WINDOW SYSTEM

In this chapter we describe Xprof, a protocol-level profiler that generates meaningful profiles of X Window applications. The profiler estimates the time spent in servicing request messages in the display server and in 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.

3.1 Messages in the X Window System

The X protocol supports a rich variety of message types for client-server communication [53, 12]. There are, broadly speaking, four 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

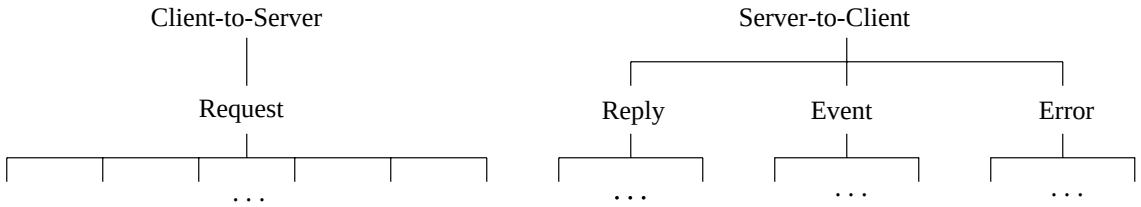

Figure 3.1: X protocol messages.

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. Last, errors are warning messages of various types that are sent from the server to the client. Figure 3.1 shows this broad hierarchy of message 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. The X Window protocol manual describes the details of each message type [53].

3.1.1 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.

3.1.2 Request attributes

Each of the messages has a number of attributes associated with it; for example, 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 K depending on their information content. Other attributes depend on the type of the message; for example, the *CopyArea* request has associated with it the information about the location and size of the source and the location of the destination. Similarly the line drawing request, *PolyLine*, invokes the attributes such as the line length, line width and fillstyle.

3.2 Trace Collection

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 X Window platforms.

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

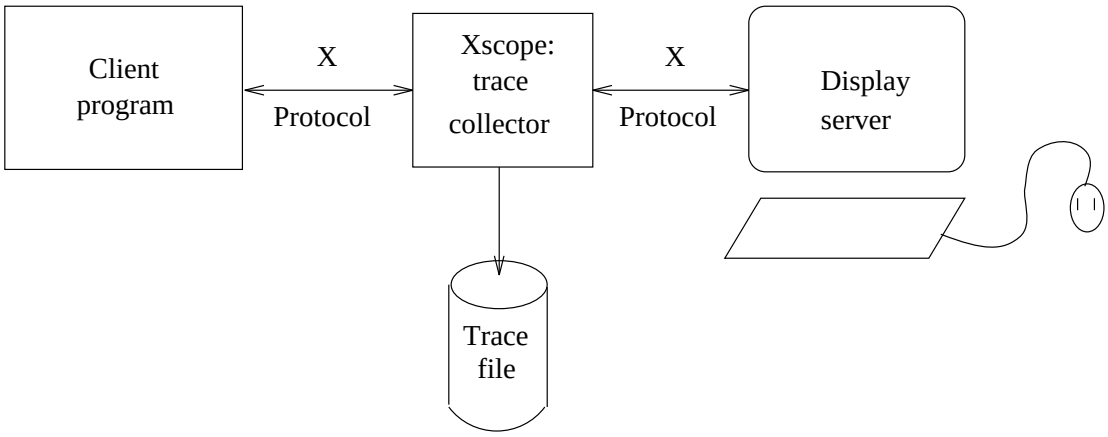

Figure 3.2: Protocol trace collection.

As shown in Figure 3.2, the tracing program, Xscope, is set up to communicate with the display server and to act as a "pseudo-server." The client programs communicate with it as if dealing with an X server. Xscope passes on all of 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 command line option. Each of the three programs in Figure 3.2 may run on its own computation host. The slowdown 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 for 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.

3.3 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 sizes of the data structures that are used to accumulate statistics or by supplying more precise values of the profiling parameters for critical requests. These steps are discussed, in greater detail, in Section 3.6. The analysis process is summarized in Figure 3.3.

3.3.1 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 request, 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.

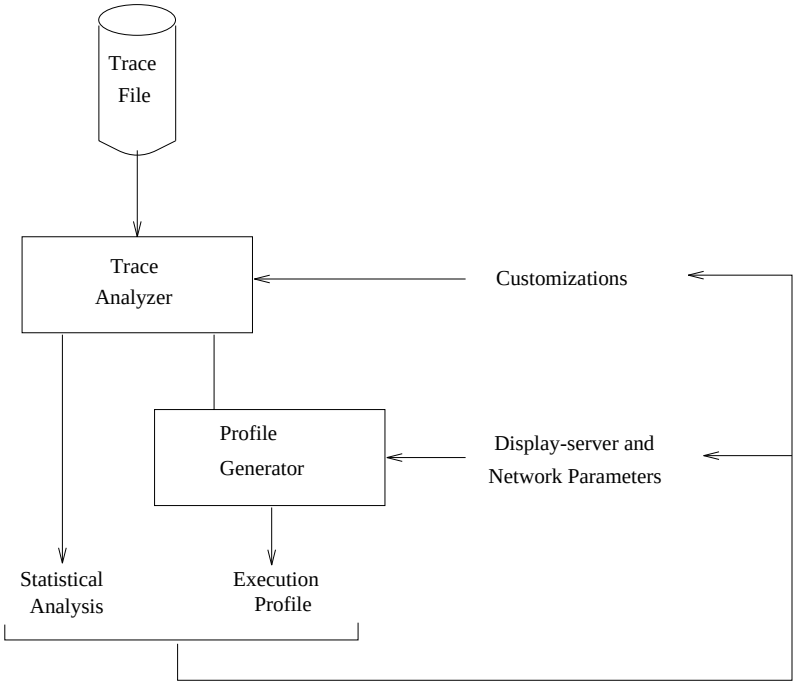

Figure 3.3: Overview of trace analysis.

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.

3.4 Collection of Server Metrics: Xmeasure

The Xmeasure program is used to 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*[2] 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 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 3.1 shows some measurement results for typical request invocations on three popular color workstations, i.e., Sun 4/IPC, DECStation 3100, and HP 9000.

[2]x11perf is distributed with the X Window source code available from MIT.

Table 3.1: 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

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:

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

$$\ldots \quad [attribute\_n =< value\_n >] \quad (opsize, rate) \tag{3.1}$$

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 number of entries may be given for a particular request, e.g., 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[3] 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 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 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 one exists, 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 run times for the available line widths.

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

```
CreateWindow            (0,4717)
          ...
PolyLine gxmode=GXcopy linestyle=LineSolid      fillstyle=FillSolid          \
          linewidth=0    (100,19161)
PolyLine gxmode=GXcopy linestyle=LineSolid      fillstyle=FillSolid          \
          linewidth=0    (300,10428)
PolyLine gxmode=GXxor  linestyle=LineSolid      fillstyle=FillSolid          \
          linewidth=0    (100, 8423)
PolyLine gxmode=GXxor  linestyle=LineSolid      fillstyle=FillSolid          \
          linewidth=0    (300, 3309)
          ...
PolyLine gxmode=GXxor  linestyle=LineDoubleDash fillstyle=FillOpaqueStippled \
          linewidth=10   (100,   45.3)
PolyLine gxmode=GXxor  linestyle=LineDoubleDash fillstyle=FillOpaqueStippled \
          linewidth=10   (300,   20.7)
          ...
PolyText8 fontname=6x13 ( 8, 3121)
PolyText8 fontname=6x13 (32,  934)
          ...
```

Figure 3.4: Typical Xmeasure entries for server parameters.

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 instance of this attribute is allowed to have a 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 line style is treated as one value of this attribute and the others are treated together.

Figure 3.4 shows entries for requests to create windows and to draw lines. These were gathered from an actual measurement run for the Sun 4/IPC Sparcstation. For example, the first entry in the figure states that the CreateWindow requests could be serviced at the rate of 4717.03 requests per second.

3.5 Xprof Details

3.5.1 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 standard requests. Since a very large number of attributes are possible for each request and each could have 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 attributes discussed earlier. In the current implementation the linewidth is allowed to have up to four 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 up to 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

```
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;                /* OpSize for this measurement    */
    float   speed;               /* Speed in size units per second */
    struct _CostCell *nextcost;  /* Next data point in the list    */
} CostCell;
```

Figure 3.5: C language data structures for the cost model of messages.

measurements in an array of lists as shown in Figures 3.5 and 3.6. The data shown in Figure 3.6 are for measured values for a Sun 4/IPC workstation as shown earlier in Figure 3.4.

Figure 3.6 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 3.4, are read, and the size-pair entry is then 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 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

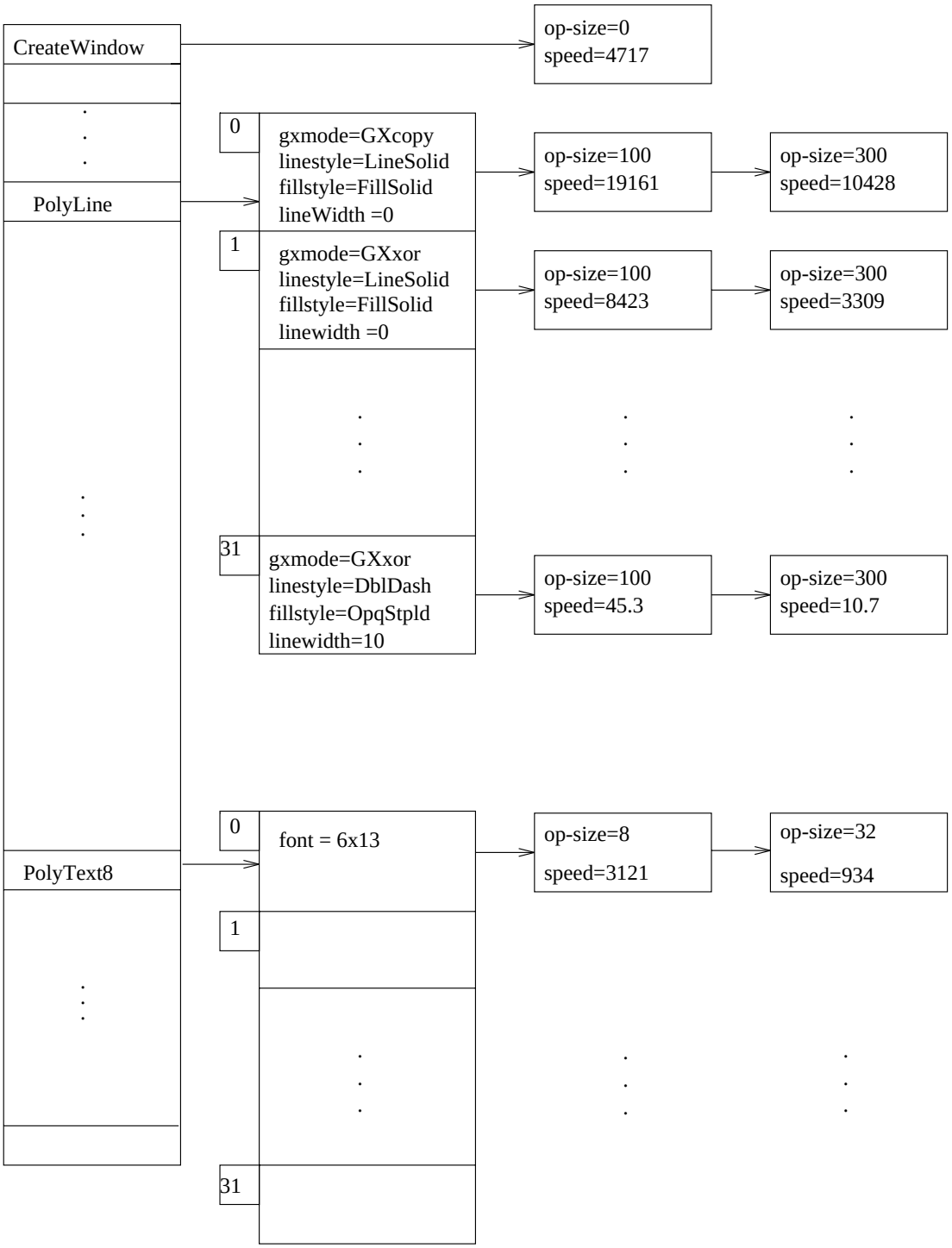

Figure 3.6: Message costs as maintained by Xprof.

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 as well. 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}. Suppose that 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 the various graphics attributes have to be taken into account. Figure 3.6 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 3.6, the appropriate list entry for op-size of 100 yields a speed of 19161 operations per second. From this, the time spent can be computed as 1/19161, i.e.,

   $$T_{r_1} = 0.052 \text{ ms}$$

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 having 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,

   $$T_{r_2} = 0.0522 + \frac{0.0959 - 0.0522}{300 - 100} \times (200 - 100) = 0.074 \text{ ms}$$

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 replaced by the attribute *FillOpaqueStippled*. This substitution leads to a match at index 31. From Figure 3.6 we find that an operation of size 100 is executed with a speed of 45.3 operations per second. From this,

   $$T_{r_3} = 1/45.3 = 22.1 \text{ ms}$$

```
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 msg */
  long     last_time;       /* The time stamp of previous message      */
  Grain    size_grain;      /* Size grain for this measurement         */
  Detailed detailed;        /* Are we maintaining detailed information?
                               If so, the following are updated too.   */
  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 3.7: C language data structure for message statistics.

3.5.2 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 run time as discussed below.

Figure 3.7 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 a reasonable bound. In order to achieve this, the two arrays for collecting the distributions of interarrival times and operation sizes, i.e., *iat_distribution* and *size_distbn*, are allocated dynamically from heap memory at run time.

The distribution of interarrival times 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 to 32. Given that the grain size for the measurement of time on a Unix system is 10 ms, this choice is enough to cover interarrival times for 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[4] for each data structure, which is quite reasonable. The choice of the size of histogram array for the operation size, or *op-size*, is trickier. 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* to 4096, which is changeable at run time, and to allow for a 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 involve small operations; therefore a size-grain of one is adequate for such requests. The *PutImage*

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

request, on the other hand, can involve the copying of data of up to 64 K and would require 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 involve 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 has to process 1 M of data, which implies that a size-grain 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, the MsgStats array accounts for 16 K[5] of heap memory per message measured. Since over 200 instances of this structure may be needed, the total space usage amounts to over 3 M. 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 interested only 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

[5] Assuming that the computer uses four bytes for each *long* integer.

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 K, which is a vast improvement over the worst case usage of over 3 M 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 histogram arrays. Default values for each are built into Xprof and are customizable by the user.

3.6 Refining the Measurements

Since Xprof is a trace-driven profiler, it is possible to rerun it on the same trace 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 augmenting the server parameter list. Then, Xprof may be rerun to generate a more accurate profile of the trace.

Several runtime 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 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 might 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 Section 3.3. 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, for example, 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. To support such a usage, Xprof captures the following signals in the Unix environment:[6]

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 reread the server parameter file.

[6] 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]*.

4. RESULTS FROM XPROF ANALYSIS

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.

Three popular workstations were selected for the following study. These were the Sun4/IPC, DecStation 3100, and HP 9000/350. Each of these supports 8-bit color, has at least 8 M of memory and runs release 4 of version 11 of the X Window system.

The applications programs selected for profiling were Ximage, Xtex, Xtetris, and Xmagic. Ximage is a scientific visualization tool that is used to display the results of scientific computations as 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, i.e., 90 K each. The sequence was run through 10 times to generate the protocol trace. Xtex is a previewer for documents formatted by the LaTeXdocument processing system. The protocol trace was collected for display of a 19-page research report. Xtetris is an interactive game played by the user against the

computer in which the player guides falling blocks to form filled rows. A protocol trace was taken for a session lasting about 10 min. Xmagic is a CAD layout tool that is used in the VLSI design class at the University of Illinois. The trace was collected for the layout of a 4-bit D-latch. All protocol traces were collected on the Sun4/IPC system.

4.1 Execution Profile for Requests

The execution profile consists of a list of all requests that are made during the execution of the program, with the total estimated time of execution for each. This time is divided into 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 4.1 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. Obviously, text rendering itself is not the computation bottleneck for this application. However, when we look at the network part, the PolyText8 requests take up a major proportion of the time spent in the network communication. The overall performance of this program on the hardware

Table 4.1: Excerpt of the execution profile of Xtex.
Network speed = 100.00 K/s, Latency = 10.00 ms

Request Name	Time (ms)	%of total	Compute part	Network part	Number of messages	Time/ call (ms)
PolyFillRectangle	3137	50.7	47.2%	3.5%	508 (5.9%)	6.2
PolyText8	2130	34.4	7.1%	27.3%	7389 (86.1%)	0.3
MapSubwindows	103	1.7	1.7%	0.0%	5 (0.1%)	20.6
MapWindow	100	1.6	1.6%	0.0%	23 (0.3%)	4.4
QueryFont	227	3.7	1.1%	2.6%	16 (0.2%)	14.2
...						
Grand Total6	6191	100.00	62.1%	37.9%	8580	0.7

Table 4.2: Excerpt of the execution profile of Xtex.
Network speed = 1000.00 M/s, Latency = 0.00 ms

Request Name	Time (ms)	%of total	Compute part	Network part	Number of messages	Time/ call (ms)
PolyFillRectangle	2924	76.0	76.0%	0.0%	508 (5.9%)	5.8
PolyText8	441	11.5	11.5%	0.0%	7389 (86.1%)	0.1
MapSubwindows	103	2.7	2.7%	0.0%	5 (0.1%)	20.5
MapWindow	99	2.6	2.6%	0.0%	23 (0.3%)	4.3
QueryFont	66	1.7	1.7%	0.0%	16 (0.2%)	4.1
...						
Grand Total	3846	100.0	99.9%	0.0%	8580	0.5

studied could be improved by reducing the computation cost of clearing a page and the network cost of communicating the text rendering requests.

To gain an idea of the server-side computation, the user may be interested in looking at the computation profile separately. Table 4.2 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 4.3: Message statistics for Xtex.

```
              ***** Statistics for Requests *****
Interarrival 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
```

4.2 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.

4.2.1 Message categories

Table 4.3 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 estimated 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 4.4. Similar

Table 4.4: Total bytes for each request in the trace of Xtex.

Request messages	Total Bytes		Number	
PolyFillRectangle	21,368 bytes	(10.35%)	508	(5.92%)
PolyText8	168,964 bytes	(81.80%)	7,389	(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	206,548 bytes		8,580	

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 4.3 shows that the Xtex messages are buffered frequently by the X protocol. Most messages are relatively small, with a mean size of about 24 bytes. Table 4.4 shows that the PolyText8 request accounts for over 80% of the network traffic for Xtex. This explains why these requests have a relatively high network component in the execution profile. The average size of these requests is about 23 bytes, which is close to that for the overall profile.

4.2.2 Individual messages

The last section of the Xprof output lists the distributions of each message type individually. Table 4.5 shows the distribution for the PolyText8 request for Xtex. As noted earlier, for the overall request distribution, the effect of buffering of 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

Table 4.5: Statistics for PolyText8 messages in the trace of Xtex.

```
              ***** Statistics for PolyText8 *****
Interarrival 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
```

about 3.5 characters. Since the average PolyText8 message is about 23 bytes long, as noted earlier, this means that the message is not very efficient at transmitting the strings. Longer string lengths in each request might improve the network performance.

4.3 Cross-server Profiling

Table 4.6 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 version 11 release 4 of the X Window System. 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

Table 4.6: Cross-server profile for Xtex.

Request Name	Message Distribution		Execution Profile					
			Sun 4/IPC		DecStation 3100		HP 9000/350	
	No. of msgs.	% of total	Time (s)	% of total	Time (s)	% of total	Time (s)	% of total
PolyFillRectangle	508	5.9	2.92	75.9	2.67	72.4	1.14	14.9
PolyText8	7389	86.1	0.44	11.5	0.59	16.2	4.28	56.3
MapSubWindows	5	0.1	0.10	2.7	0.03	0.7	0.14	1.8
MapWindows	23	0.3	0.99	2.6	0.12	3.2	0.33	4.4
QueryFont	16	0.2	0.66	1.7	0.11	2.9	0.42	5.6
All Messages	8580		3.85		3.69		7.60	

than on the other machines. Therefore, 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, only a few requests account for 70-90% of the computation time on the display server.

Table 4.7 shows the profiles for execution of Ximage, the scientific visualization tool. The *PutImage* requests account for over 90% of the processing time on each of the machines even though they constitute only about 15% of the messages. These requests involve the transfer of the actual image to be displayed on the screen and are an important component of the image manipulation functions of Ximage. The *ClearArea* and *CopyPlane* operations account for only about 5% of the processing time yet account for over 60% of the messages. It turns out that they are invoked by functions that manage the user interface of the program, such as the scrollbars and input buttons, and require

Table 4.7: Cross-server profile for Ximage.

Request Name	Message Distribution		Execution Profile					
			Sun 4/IPC		DecStation 3100		HP 9000/350	
	No. of messages	% of total	Time (s)	% of total	Time (s)	% of total	Time (s)	% of total
PutImage	1333	15.3	55.11	95.3	64.92	93.8	350.56	92.4
ClearArea	2782	31.8	1.23	2.1	1.26	1.8	9.16	2.4
CopyPlane	2685	30.7	0.82	1.4	1.86	2.7	12.83	3.4
CopyArea	670	7.7	0.22	0.4	0.46	0.7	4.32	1.1
MapWindow	49	0.6	0.18	0.3	0.29	0.4	0.71	0.2
ImageText8	719	8.2	0.06	0.1	0.07	0.1	0.56	0.2
All Messages	8737		57.84		69.18		379.44	

Table 4.8: Cross-server profile for Xtetris.

Request Name	Message Distribution		Execution Profile					
			Sun 4/IPC		DecStation 3100		HP 9000/350	
	No. of msgs.	% of total	Time (s)	% of total	Time (s)	% of total	Time (s)	% of total
ClearArea	7587	60.3	3.77	57.4	2.85	55.7	25.52	87.9
PolyFillRectangle	4389	34.9	2.10	32.0	1.63	31.9	0.27	0.9%
CopyArea	44	0.4	0.49	7.4	0.50	9.8	0.10	0.4
All Messages	12577		6.57		5.11		29.04	

only a small amount of processing on the display-server. For this application, the Sun 4/IPC and DecStation 3100 are roughly on par on performance. The HP 9000/350 is about 6 times slower than the other two because of its slower PutImage function.

Table 4.8 shows the profile for Xtetris, the interactive game. Again, a few requests account for over 90% of the messages and over 90% of the server-side processing. The total times for processing range from 5.11 s for the DecStation, to 29.04 s, for the HP. Since the trace is for a session lasting about 500 s, the server processing is adequate for the demands posed by the game in each case. The critical functions are *ClearArea*, which

Table 4.9: Cross-server profile for Xmagic.

Request Name	Message Distribution		Execution Profile					
			Sun 4/IPC		DecStation 3100		HP 9000/350	
	No. of msgs.	% of total	Time (s)	% of total	Time (s)	% of total	Time (s)	% of total
PolyFillRectangle	1383	16.8	9.59	92.8	8.80	92.9	3.65	53.8
SetClipRectangles	803	9.8	0.33	3.2	0.18	1.9	0.71	10.5
PolySegment	1316	15.9	0.25	2.4	0.34	3.6	1.31	19.3
PutImage	44	0.5	0.07	0.7	0.03	0.4	0.41	6.0
PolyPoint	995	12.1	0.01	0.1	0.01	0.1	0.08	1.3
PolyText8	803	9.8	0.04	0.4	0.05	0.5	0.33	4.8
ChangeGC	2588	31.4	0.00	0.0	0.00	0.0	0.00	0.0
All Messages	8234		10.33		9.48		6.79	

is invoked to clear the game surface, and *PolyFillRectangle*, which is invoked to draw the game tokens on the screen.

Table 4.9 shows the profile for Xmagic, the VLSI CAD layout tool. The function *PolyFillRectangle* accounts for the bulk of the server-side execution in each of the display-servers. Xmagic invokes this function frequently since its output consists of a large number of rectangular objects. The HP 9000 outperforms the other computers since it has special architectural support for area-fills.

In general, the Xprof profiles clearly point out the critical functions with reference to the server-side processing. Also, since Xprof is capable of doing cross-display-server profiling, it is easy to compare the performance of a trace on many different workstations. The information collected can be used to tune the display-server software and also to achieve better load-balancing between client and server processing.

Table 4.10: Client-server load partitioning for the Sun4/IPC.

Client Program	Client Time (Gprof)	Server Time (Xprof)	Total Profile (Gprof+Xprof)	Actual Time (wallclock)	Ratio (Profile/Actual)
XImage	255.0 s	57.8 s	312.8 s	320.0 s	98%
Xtex	2.5 s	3.9 s	6.4 s	7.0 s	91%

4.4 Client-server Load Partitioning

To quantify 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 4.10 shows the results of such a measurement for Xtex and Ximage.

The data in Table 4.10 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 clock time. Since both the application and the display server were run on the same computation host, the network time is not relevant to this measurement.

4.5 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 10 ms and a speed of either 100 K/s or 1000 K/s. The y-axis in Figure 4.1 is normalized to the total processing

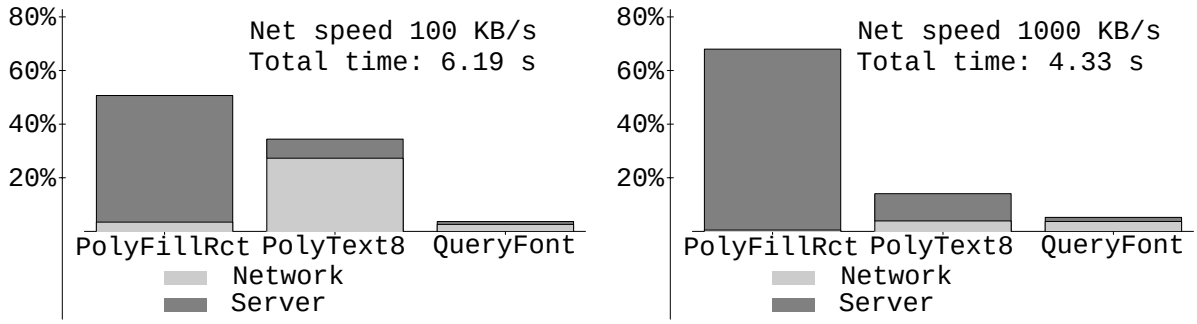

Figure 4.1: Effect of network speed on Xtex profile.

time since the objective is to see how the overall request service time is divided between the network and the display server.

Figure 4.1 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 profile shows that the Xtex requests take 50% longer to execute for the slower network.

Figure 4.2 shows that the Ximage application is deeply affected by the network speed. For the trace studied, the slower network causes the Ximage requests to take 6 times longer to execute. Ximage transfers a large amount of data from client to server in the form of the PutImage requests. The trace studied involves a total transfer of 59.4 M of data. Almost all of this is accounted for by the PutImage requests.

Figure 4.3 shows that Xtetris executes quite well for either network speed. The critical messages for it, i.e., *ClearArea* and *PolyFillRectangle*, are high-level graphics requests.

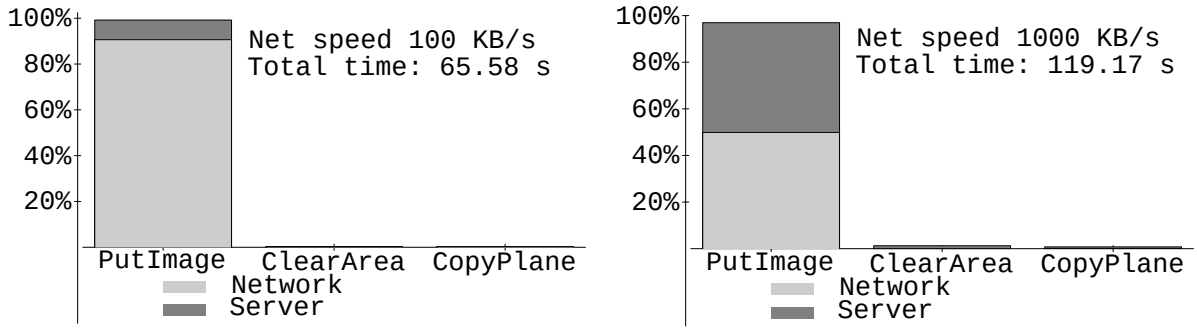

Figure 4.2: Effect of network speed on Ximage profile.

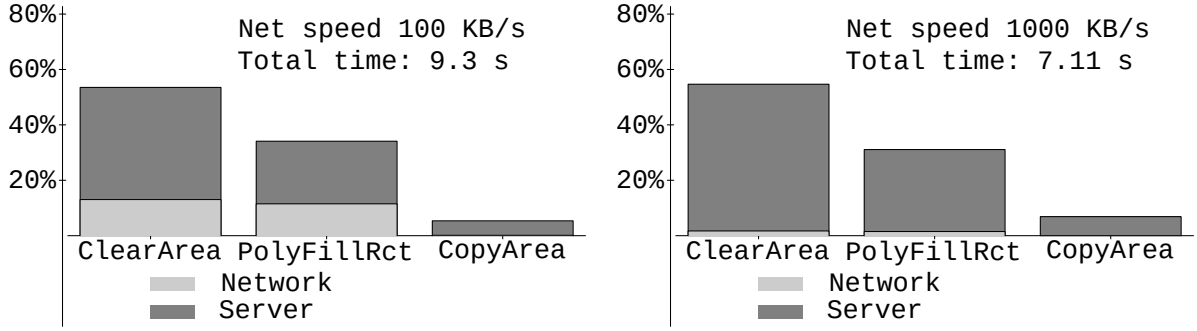

Figure 4.3: Effect of network speed on Xtetris profile.

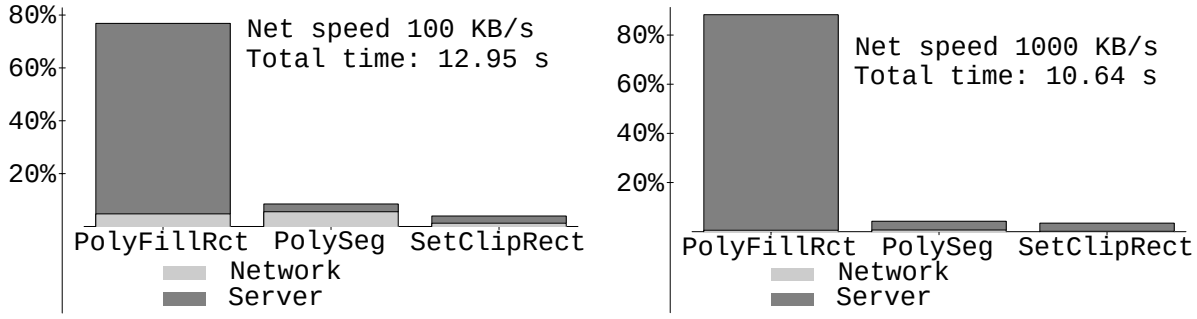

Figure 4.4: Effect of network speed on Xmagic profile.

Therefore, the requests for this application take only about 25% longer for the slower network.

Figure 4.4 shows that the Xmagic program is not seriously affected by the network speed. The important request for this program is *PolyFillRectangle*, which is a tightly encoded request.

Thus, Xprof gives a good idea of the distribution of the request processing between the display-server and the network. The analysis of the impact of the network can be quite valuable in gauging the performance of a client-server-model based program. Future systems are likely to emphasize visualization applications, and the network bottlenecks will become important to the performance of display systems such as the X Window System.

4.6 Limitations of Xprof

The profiling approach developed in this thesis uses a protocol-level trace, which has no knowledge of the lines of code that gave rise to the protocol messages. Therefore, the current implementation of Xprof cannot connect the critical server functions to specific lines of client code. To collect such information, the trace would have 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 takes only 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 for the target display-server.

Estimation of the cost of a request, as described in Chapter 2, may involve two kinds of approximations, i.e., interpolation of the operation size and the substitution of one 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 nonlinear with respect to the operation size, the computed cost may be inaccurate. 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 nonmeasured 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.

4.7 Xprof Audiences

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

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 determine the critical requests made by typical applications and tune their systems to execute such requests faster.

4. Administrators of distributed systems can quantify 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.

5. FACTORS IN MEMORY-SYSTEM DESIGN

Cache memory is frequently used to improve memory access performance [65, 66, 67, 68, 69, 70, 71, 72]. Graphical displays make use of a specialized frame-buffer memory to maintain the bit-map image of the display. However, the benefit of using conventional caches is not clear for frame-buffer accesses since these accesses have large working sets and are mainly writes. As computers with graphical user-interfaces become increasingly popular, it is important to study methods to improve the performance of frame-buffer access. In this chapter and in the following chapters we look at memory-system design with respect to the access characteristics of the frame-buffer.

5.1 Cache Memory

The basic idea behind a cache memory is to exploit spatial and temporal locality of accesses, which is exhibited by typical programs, using a small high-speed memory between the processor and the main memory. This fast memory *caches* recently accessed

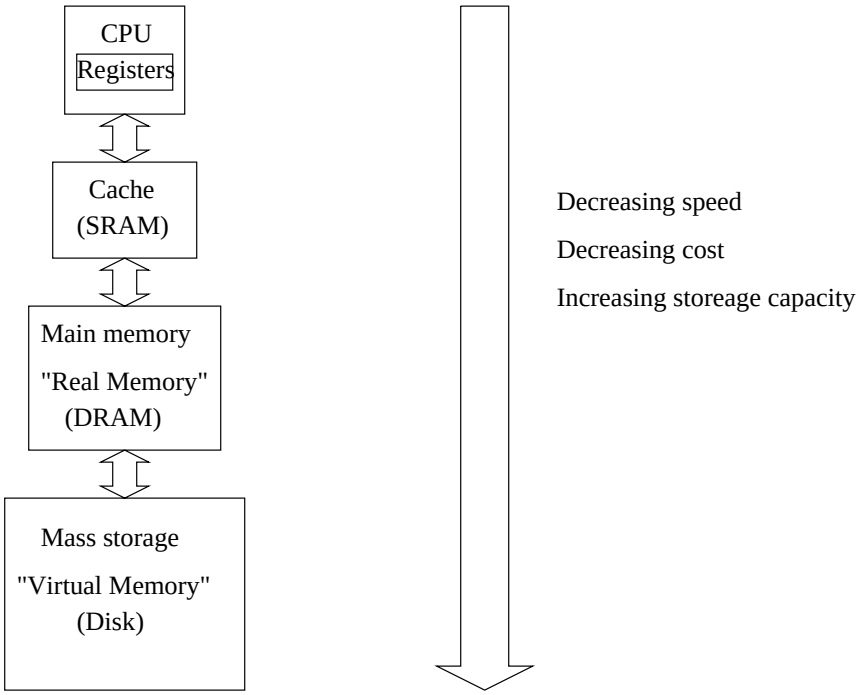

Figure 5.1: Memory hierarchy in computer systems.

items and provides high-speed access to them on reuse. Cache design is tricky since a poorly designed cache can actually degrade the overall system performance.

As shown in Figure 5.1, in typical computer systems, the total memory consists of a hierarchy of different storage technologies. The memory closer to the processor is faster and more expensive than the memory further down in the hierarchy. A balanced memory system design aims at minimizing the system cost while maintaining acceptable performance. Recent RISC architectures have large register sets, but the high instruction execution rates of these processors have made cache design even more critical. In general, cpu speeds have been advancing at rates faster than the rate of improvement in memory access speed. Note that caches are only one way of improving access time.

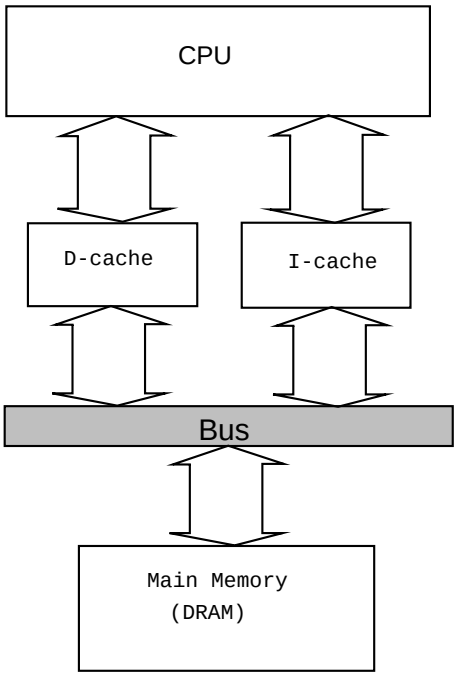

Figure 5.2: Cache organization in typical systems: "Harvard" architecture.

Other strategies, including memory interleaving, page-mode access, write-buffering, synchronous DRAMs, and specialized designs such as Rambus, may be applied to complement the cache design [71].

As shown in Figure 5.2, current computer systems use a split cache configuration with separate caches provided for the instruction (I-cache) and data (D-cache) streams. One motivation for this approach is that pipelined RISC systems may require both an instruction fetch and a memory fetch to complete within a processor cycle. Also, the instruction and data streams may have sufficiently different behaviors to warrant separate organization. Even though the caches are separate, they frequently communicate with the main memory system over the system bus and can thus influence each other's performance by way of the demands placed on the bus. Since the bus may also be used for input-output

operations, it is important to minimize the bus traffic, and this aspect is an important concern in the cache design.

5.2 Issues in Cache Design

An important goal of cache design is to minimize the overall access time and not simply the miss-rate. In this thesis the metrics of performance of a cache are the overall access time and the traffic ratio. These and other terms are defined below. In general, these conform to the terminology used by A.J. Smith in his detailed survey of cache design methodologies [65].

Cache: Cache memories are small, high-speed buffer memories that hold, temporarily, an actively used subset of the contents of the main memory. A cache can either be a *unified* cache or be split into a separate instruction-cache, or *I-cache*, and data-cache, or *D-cache*.

Block: A block, also called a *line*, is the minimum unit of data transfer between the cache and main memory. A *tag* is maintained for each line of data. Proper line size selection is an important part of the cache design.

Tag: The tag of a cache block specifies the block address. A tag has to be maintained for each line of data. This tag memory can be an appreciable part of the chip area that implements the cache. Larger block sizes incur less tag-memory cost.

Set: A set is a grouping of blocks whose tags are checked in parallel when searching for a data item in the cache.

Associativity: The degree of associativity of a cache is the number of blocks in a set. In a *fully-associative* cache the entire cache forms a single set. At the other extreme, a *direct-mapped* cache has sets with one block each. The caches that fall between these extremes are called *set-associative* caches.

In general, set-associative caches are more expensive to implement than direct-mapped caches. Most practical caches are either direct-mapped or 2-way set-associative. According to Hill [68], it is difficult to build a set-associative cache that delivers a higher overall performance than that of a comparable direct-mapped cache.

Hit/miss: A hit is a memory access that is found in the cache, while a miss is one that is not found in the cache. The *hit/miss rate* is the number of hits/misses as a proportion of the total accesses. One objective of cache design is to reduce the miss-rate. The *hit time* is the time to access the item in the cache. The *miss penalty* is the time to replace the cache block with a block fetched from the main memory. This can be divided into the *access latency*, which is the time to access the first word of the missing block, and the *transfer time*, which is the time to transfer the remaining words in the block.

Traffic ratio: The traffic ratio is the ratio of the traffic between the processor and the memory with and without the cache. Thus, if the traffic ratio is less than unity, the cache serves to reduce the memory traffic on the system bus. This has the effect of improving overall system performance.

Replacement Strategy: The replacement strategy is the algorithm for choosing which block in a set will be replaced by a newly fetched block. The most common choices are Least Recently Used (LRU) and Random.

Write strategy: The write strategy is the policy followed when a write is made to the cache. There are two basic choices:

- *Write through* (also called *store through*): The data are written to both the cache block and the lower-level memory.
- *Write back* (also called *copy back*): The data are written only to the cache block. The modified block is written to main memory only when it is replaced. A *dirty* bit has to be maintained to track the blocks which need to be written back.

Both these policies have their own advantages and disadvantages. Also, *write buffers* may be maintained for both of the above strategies. These allow the processor to continue operations while the memory is being updated.

On a write miss some more choices have to be made:

- *Write allocate*: The necessary block is loaded into the cache.

- *No write allocate*: The block is modified only in the main memory and is not loaded into the cache.

Generally, copy-back caches use the write-allocate policy, and write-through caches use no write allocate.

Fetch strategy: The fetch strategy is the strategy for deciding when to fetch a block from the main memory. Demand fetching is the most common policy, but various prefetching algorithms can enhance performance for specific applications. Some recent microprocessors have provided special instructions that allow the compiler to embed prefetch hints in the user code.

Some other interesting issues in cache design include multiprocessor caches, multi-level caches, and virtual vs. real addressed caches.

5.3 The Video RAM

Typical raster-display hardware consists of a frame-buffer memory, a video controller, and a CRT display. The graphics computations are done either on the main cpu or on a specialized graphics processor. The processor computes the graphics primitives and writes the computed image into the frame-buffer. The video controller reads the pixel data from the frame-buffer and drives the display. Thus, both the processor and the video controller contend for access to the frame-buffer memory.

The video controller itself requires a high-bandwidth access to the frame-buffer memory. For instance, for a 1024×864 pixel screen with 8 bits per pixel refreshed at 60 Hz,

the memory bandwidth required for the video controller is $1024 \times 864 \times 60 = 5.3$ M/s. Another way of looking at this is that the video controller must read a fresh byte every 18.84 ns. Conventional DRAMs are unable to sustain this access rate.

The twin requirements of dual-port access and high-bandwidth serial access are met by a novel memory design called *Video RAM*, or *VRAM* [73, 17, 74]. Video RAMs were invented by Texas Instruments in 1981 and now are a standard feature in most raster-scan displays. As shown in Figure 5.3, a VRAM augments a conventional DRAM organization with the addition of a *Serial Data Register*, or *SDR*, which can be parallel loaded with an entire DRAM row. The SDR acts as a high-speed secondary channel for use by the video controller. The first VRAMs, introduced by TI, had a 16K x 1-bit-wide organization. Currently, the popular VRAMs are available in 64K x 4-bit-wide organization and are made by companies such as TI, Fujitsu, NEC, and, Mitsubishi.

Figure 5.3 shows the organization of a 64K x 4-bit-wide VRAM, as typified by the Fujitsu MB81461-12 VRAM [74]. This VRAM consists of four 256×256 DRAM cell arrays and corresponding 256-bit Serial Data Registers, which are also referred to as *Serial Access Memory*, or *SAM*. Each SDR is connected to its corresponding Cell Array by 256 vertical *bit lines*, which are used to transfer data to and from the storage arrays. The SDRs can be parallel loaded by applying the transfer signal, $\overline{TR}$, while a row of memory is being read. The SDR has its own clock, *S-Clock*, which can then be used to transfer data out serially, at high speed to the video controller. This serial transfer can go on asynchronously with processor access to the DRAM arrays, thus providing the

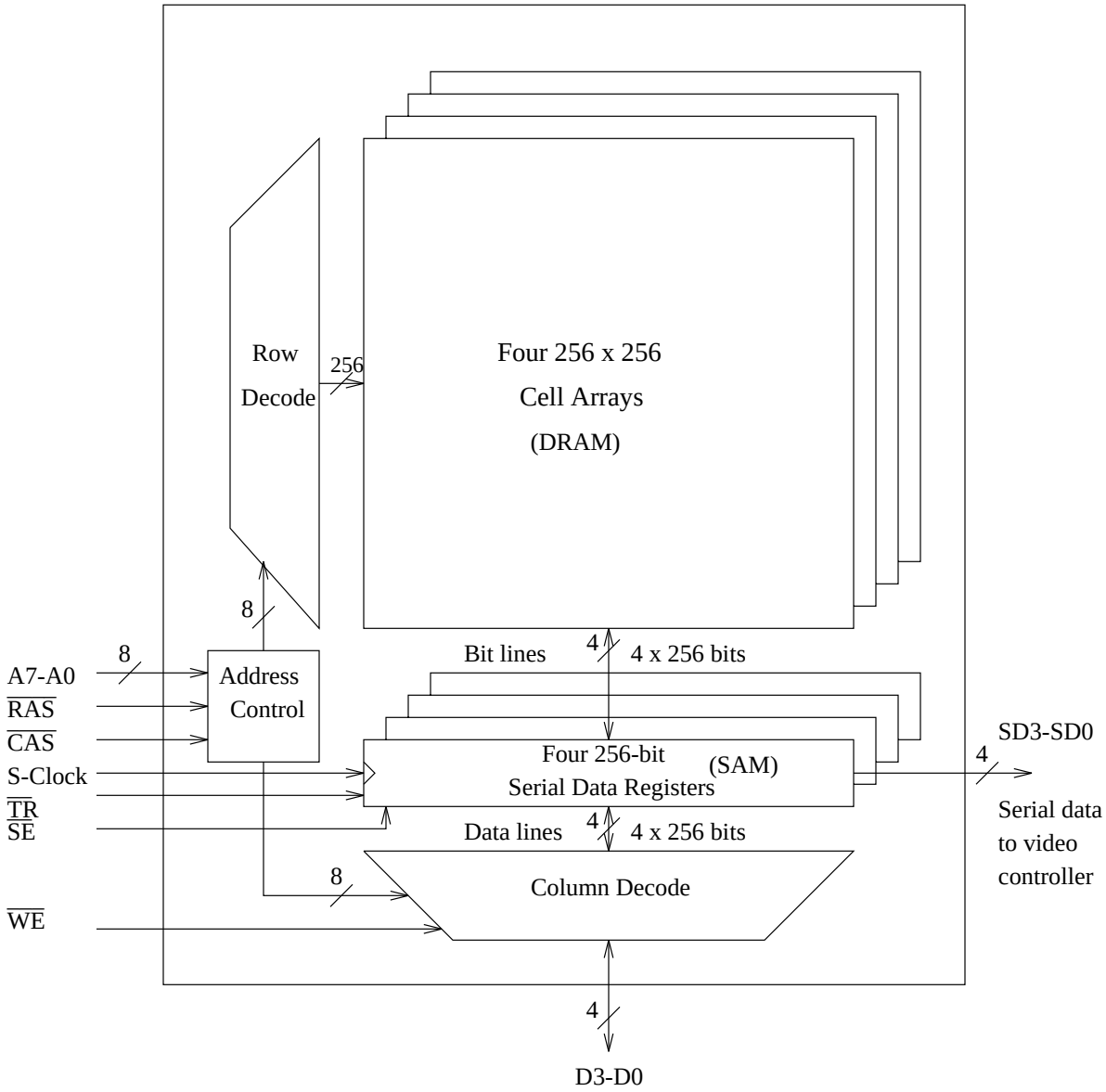

Figure 5.3: Organization of a 256-Kbit (64K x 4) VRAM chip.

desired dual-port access capability. The SDR requires an access to the DRAM array only once every 256 cycles. Thus, a substantial improvement in performance is obtained for a small increase in chip area. However, due to considerations of economies of scale, the VRAM chips typically cost twice as much as corresponding DRAMs.

In Figure 5.3, the following signals account for the usual signals of a 64K x 4 DRAM:

- Eight multiplexed address lines, A7-A0.
- Four bidirectional data lines, D3-D0.
- Row Address Strobe, $\overline{RAS}$.
- Column Address Strobe, $\overline{CAS}$.
- Write Enable, $\overline{WE}$.

To access the data in the DRAM, the following steps are required. First, a row is selected by signaling $\overline{RAS}$ while the desired row address is on the address lines A7-A0. Next, the desired column is selected by signaling $\overline{CAS}$ while the desired column address is on the address lines A7-A0. Thus, the address lines are multiplexed for both the row and column addresses. The selected bits are either written to or read from depending on the status of the write-enable signal $\overline{WE}$.

The DRAM may support a faster access mode, called *page mode*, for successive accesses to memory locations on the same row, or DRAM page. In page-mode access, the row address is presented only once and the $\overline{RAS}$ signal is maintained low while successive columns are accessed by changing the column address bits and activating the $\overline{CAS}$

signal. This mode can save power dissipation and exhibit a faster access time. Typically, page mode accesses are twice as fast as an ordinary memory access cycle.

The addition of the Serial Data Register (SDR) in the VRAM necessitates the following signals:

- Data Transfer, $\overline{TR}$.
- Serial Clock, S-Clock.
- Serial Output Enable, $\overline{SE}$.
- Serial Output Data bits, SD3-SD0.

The data transfer signal, $\overline{TR}$, is used in conjunction with the $\overline{RAS}, \overline{CAS}$, and, $\overline{WE}$ signals to transfer the data from one selected 256-bit row in each cell array to the corresponding SDR. The Serial-clock signal, S-Clock, in conjunction with the Serial Output Enable, $\overline{SE}$, can then be applied to shift the data out of the Serial Output lines SD3-S0.

Table 5.1 shows some parameters for the Fujitsu MB81461-12 VRAM chip.

Table 5.1: Selected parameters for the Fujitsu MB81461-12 VRAM chip.

- DRAM Port:
  - Access Time (t_{RAC}): 120 ns
  - Cycle Time (t_{RC}): 230 ns
  - Page Mode Cycle Time (t_{PC}): 120 ns
- SAM Port:
  - Access Time (t_{SAC}): 40 ns
  - Cycle Time (t_{SC}): 40 ns
- 24-pin DIP and ZIP packages
- 256 refresh cycles every 4 ms

6. MEMORY ACCESS CHARACTERISTICS OF DISPLAY-ORIENTED PROGRAMS

Memory system analysis may be done by following *trace-driven simulation* or *analytical modeling*. In trace-driven simulation, which is employed as the cache-analysis methodology in this research, a detailed address trace of an actual program execution is collected and analyzed. In contrast, analytical models are driven by mathematical models of the high-level behavior of the application programs. Both approaches have their own strengths and weaknesses. Simulation studies have the advantage of being based on real programs, but may be based on traces that are unrepresentative or are too short. Analytical models can yield insights into the high-level tradeoffs but are by their nature an approximation of reality. This research is based on trace-driven simulation of detailed traces collected for X Window programs.

6.1 Instruction-level Trace Collection

Instruction-level profiling techniques are an important aid in the design and analysis of computer architecture [67]. Some examples of instruction-level profiling environments

include *pixie* [75] for the MIPS architecture, *shade* [76] for SPARC, and *goblin* [77] for the IBM RS/6000.

The data presented in this thesis are based on detailed instruction-level traces collected for an X Window server running on a DECstation 3100 workstation. The traces were generated by an instrumented version of the X server compiled with the IMPACT C compiler [78], which was set up to perform local, global, jump, and loop optimizations. The X window server used was version 11, release 4 of the MIT X Window distribution. The frame-buffer manipulation functions, or "color-frame-buffer library," of the X server were instrumented. Thus, the results shown here emphasize the graphics behavior of the X Window System programs.

The experimental computer, the DECstation 3100, has a simple memory-mapped frame buffer without any special graphics processor hardware [79, 80]. The frame buffer supports 864 lines, each with 1024 pixels. Each pixel is represented by a byte of data. All frame buffer operations are handled by the main processor, which is a 16.67 MHz MIPS R2000. Thus, the address traces of the X server contain all of the references to the frame buffer in addition to the non-frame-buffer references.

It is important to note that the results shown here are for the execution of the X Window server when driven by a certain client program. Thus, the instrumentation and analysis are done for the *server-side* aspect of the program execution. Since the X server alone has access to the frame-buffer, its traces give us information about the graphical rendering performed on behalf of the corresponding client program. For instance, for

a computer animation program, the analysis relates to the activity of the X server in rendering the animation sequence on the display. The results do not reflect the activity of the client program such as extracting the animation images from the appropriate disk files. The software architecture of client-server systems and of the X Window System is described in greater detail in Chapter 1.

6.2 Benchmark Programs

Six popular programs were selected as representative of execution under X Window. *Ghostscript* is an interpreter for the Postscript page-description language. The interpretation is done by the client program, and the results here show the activity invoked in rendering the interpreted results into the frame buffer, at the display server. *Ximage* is a scientific visualization tool that involves transfer of image data from the client to the frame buffer for viewing as an animation sequence. *Xtetris* is a popular computer game involving interactive placement of small objects. *Xmagic* is a popular VLSI CAD layout tool. *Xfig* is an interactive tool for drawing and editing figures. *Xterm* is a terminal emulator and is thus used to display text within a window. One characteristic of this benchmark is that scrolling the displayed text requires the copying of vast amounts of data from one part of the screen to another. Thus, this program would be expected to invoke substantial numbers of loads from, as well as stores to, the frame buffer. The remaining programs are oriented mainly to stores.

Table 6.1: Benchmark programs.

Program	*Description*	*Dyn. Instr. Count*
Ghostscript	Postscript previewer	9.2×10^6
Ximage	Scientific visualization / animation	6.4×10^6
Xterm	Terminal emulator	6.3×10^6
Xtetris	Computer game	5.3×10^6
Xmagic	VLSI CAD layout tool	9.5×10^6
Xfig	Interactive drawing tool	7.6×10^6

For the purpose of this paper, the data sets used to collect the traces are as follows. The Ghostscript traces are collected for display of three pages of the Postscript version of a technical report. The Ximage data is for an animation sequence of 100 images, each 300 pixels square, i.e., 90 K of data. The Xterm data are for a 24-line terminal screen displaying and scrolling 60 lines of text. The Xtetris trace is for a game involving the placement of 5 objects. The Xmagic trace is for display of a sequence of 15 layout objects used in a class project. These objects range from a nand gate to a 4-bit adder circuit. The Xfig data are for display of the figure shown earlier in Figure 5.3. Table 6.1 shows the dynamic instruction count for the X server when driven by these benchmarks. These range from about five to nine million instructions. Once again, note that these counts cover only the color-frame-buffer library of the X display-server.

6.3 Dynamic Instruction Distributions

Table 6.2 shows the distribution of dynamic instruction counts for the X Window server for servicing requests from the measured programs. Memory access instructions account for most of the instructions and range from about 21% for Xfig, to about 73%

Table 6.2: Dynamic instruction distribution.

Program	*Load*	*Store*	*Move*	*Branch*	*Alu*	*Falu*	*Other*
Ghostscript	13.92%	16.01%	4.95%	31.74%	32.83%	0.00%	0.54%
Ximage	36.02%	36.54%	1.08%	10.57%	16.81%	0.00%	0.03%
Xterm	29.22%	28.79%	0.61%	10.01%	31.35%	0.00%	0.02%
Xtetris	15.58%	7.13%	5.37%	17.55%	54.31%	0.00%	0.06%
Xmagic	8.30%	14.04%	4.03%	31.01%	42.74%	0.00%	0.30%
Xfig	9.11%	11.74%	8.29%	21.67%	49.03%	0.00%	0.16%

for Ximage. Integer ALU operations are quite frequent too and range from about 16% for Ximage to about 54% for Xtetris. These are used mainly in the address computations for the memory operations. None of the benchmarks invoked floating-point ALU operations. In fact, the X Window server code does not use floating-point operations since it deals with a rasterized frame buffer with integer-addressed pixels. With 3-d applications gaining in popularity, however, floating-point operations may become more important. Three-dimensional X-server extensions such as *PEX* [55] use floating-point operations extensively. For the benchmarks studied, the memory, alu, and control-flow instructions account for over 95% of the processing.

6.4 Working Sets

Table 6.3 shows the memory working sets corresponding to the execution of the benchmark programs. The frame-buffer working sets show the amount of screen real estate occupied by the program. The maximum size of the target frame buffer of the DECstation 3100 is 864 K. The frame-buffer working sets range from about 90 K for Ximage to

Table 6.3: Memory access working sets.

Program	*Working sets*		
	Total	*Frame buffer*	*Non-frame-buffer*
Ghostscript	944 KB	535 KB (57%)	409 KB (43%)
Ximage	211 KB	91 KB (43%)	120 KB (57%)
Xterm	273 KB	266 KB (97%)	8 KB (3%)
Xtetris	214 KB	205 KB (96%)	9 KB (4%)
Xmagic	452 KB	431 KB (95%)	21 KB (5%)
Xfig	1158 KB	683 KB (59%)	475 KB (41%)

about 680 K for Xfig. These data confirm that frame-buffer accesses have large working-sets. Clearly a very large cache size would be needed to contain the full frame-buffer working sets of these programs.

The non-frame-buffer working sets for these programs range from 7.6 K for Xterm to about 475 K for Xfig. In each case, the frame-buffer working sets are a substantial portion of the overall memory trace. It should be noted that the traces are collected only for the graphics code of the X server. Thus, all of the frame-buffer references are covered, but the non-frame-buffer working sets could very well be much larger than what is shown in Table 6.3.

6.5 Memory Traffic

Table 6.4 shows the memory traffic, in megabytes, broken down into the frame-buffer and non-frame-buffer references. The data confirm the notion that most of the frame-buffer accesses are writes. The ratio of frame-buffer memory stores to loads ranges from about 100 for Ximage to about 1.3 for Xterm. The reason for this observation is that

Table 6.4: Memory traffic.
(All figures are in megabytes)

Program	*Total traffic*	*Frame-buffer accesses*		*Non-frame-buffer accesses*	
		Load	*Store*	*Load*	*Store*
Ghostscript	10.3 MB	0.06 (0.6%)	2.41 (23.4%)	4.61 (44.8%)	3.22 (31.3%)
Ximage	18.5 MB	0.09 (0.5%)	9.18 (49.8%)	9.07 (49.1%)	0.11 (0.6%)
Xterm	14.5 MB	5.59 (38.5%)	7.21 (49.6%)	1.73 (11.9%)	0.00 (0.0%)
Xtetris	4.7 MB	0.29 (6.2%)	0.83 (17.6%)	2.94 (62.3%)	0.65 (13.9%)
Xmagic	7.9 MB	1.03 (13.0%)	4.76 (60.3%)	1.83 (23.1%)	0.29 (3.6%)
Xfig	6.2 MB	0.09 (1.5%)	2.83 (45.9%)	2.60 (42.0%)	0.66 (10.7%)

most of the visually oriented applications update the screen frequently but do not usually have to read the screen data. In the case of Xterm, scrolling the screen image requires a large number of reads. Ximage, on the other hand, involves mainly the transfer of a large amount of animation data to the display.

In fact, a handful of functions account for most of the memory traffic for these applications. For Ghostscript over half of the frame-buffer stores, and an equivalent number of non-frame-buffer loads, are accounted for by the *cfbDoBitblt* function, which transfers a precomputed Postscript image to the display.[1] Also, the function *cfbFillBoxTileOdd* accounts for another third of the frame-buffer stores. This function is used to clear portions of the display before writing data on it. In the case of Ximage, the *cfbDoBitblt* function accounts for over 97% of the frame-buffer stores and a corresponding number of non-frame-buffer loads. Clearly, the principal task of this application is in the transfer of the animation images to the display. For Xterm, the *cfbDoBitblt* function is invoked to

[1]In Ghostscript, the Postscript rendering is done by the client program and the server only maps this to the display. In contrast, the *Display Postscript Extension* to X has the server perform both computation and display.

copy data from one part of the display to another during scrolling of the screen image. This accounts for over 85% of the frame-buffer accesses. A smaller proportion of the frame-buffer accesses come from the rendering of text onto the screen. These requests, which are implemented by the *cfbTEGlyphBlt8* function, involve copying of font bit maps from the non-frame-buffer memory to the frame buffer. This function also accounts for over 96% of the non-frame-buffer accesses, which are made from a very small working set of 7.5 K consisting mainly of the font information. In the case of Xtetris, nearly 60% of the frame-buffer accesses are made by *cfbFillBoxSolid* in order to clear the active area on the display. Another 38% is accounted for by the request *cfbUnnaturalStippleFS* to draw small geometric objects on the screen. This function also accounts for over 98% of the non-frame-buffer references. For Xmagic, over 63% of the frame-buffer references are made by the *cfbFillBoxSolid* function to draw the rectangular figures that make up a VLSI CAD layout. In the case of Xfig, too, the *cfbFillBoxTileOdd* and *cfbFillBoxSolid* functions together account for over 97% of the frame-buffer references.

6.6 Sequential-access Length Distributions

Table 6.5 summarizes the sequential-access characteristics of the benchmark programs. It tabulates the statistics about the run lengths of references to consecutive addresses in the frame buffer. The table shows that frame-buffer accesses tend to be of a "sweep" nature with long sequences of accesses to consecutive memory locations. This feature is to be expected given that most of the frame-buffer references are dominated by

Table 6.5: Frame buffer sequential-access lengths (in bytes).

Program	*Loads*			*Stores*		
	Range	*Mean*	*Mode*	*Range*	*Mean*	*Mode*
Ghostscript	1 - 12	4.9	4 (76%)	1 - 752	494.6	556 (82%)
Ximage	1 - 300	291.8	300 (97%)	1 - 308	300.0	300 (99%)
Xterm	1 - 720	703.8	720 (98%)	1 - 744	622.9	720 (83%)
Xtetris	1 - 60	24.4	24 (41%)	1 - 308	93.4	192 (32%)
Xmagic	1 - 412	53.8	4 (23%)	1 - 664	387.5	496 (61%)
Xfig	1 - 4	4.4	4 (89%)	1 - 916	693.4	800 (75%)

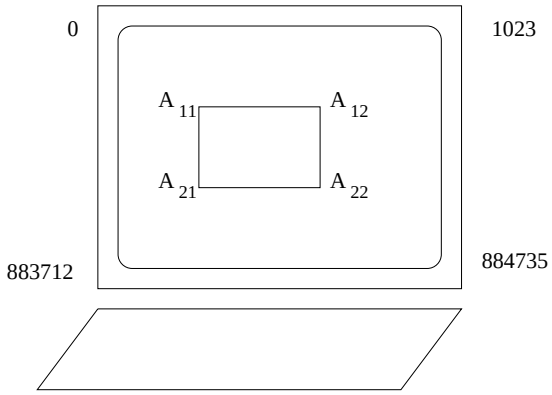

Figure 6.1: Relationship of frame-buffer addresses and screen real estate.

a handful of data-transfer functions, as shown in the previous section. These functions are optimized for the frame-buffer organization.

As shown in Figure 6.1, the frame-buffer memory is organized as a chunk of memory, which is treated as an array of display rows and columns. For the DECstation 3100, the frame-buffer address space consists of 864K of memory organized in 864 rows of 1024 columns each. An operation that clears the entire screen, for example, would sequentially access every byte in the address space. However, most programs access only a subset of the address space in the form of a *window* that occupies a subset of the screen real

estate. Thus, for the window shown with coordinates $(A_{11}, A_{12}, A_{21}, A_{22})$, clearing the window would involve access to address strips each of which is $A_{12} - A_{11}$ pixels wide.

Table 6.5 shows that the applications have frequent long sequential accesses. As pointed out earlier, these have a correspondence with the the screen-image width of the application. For instance Ximage displays a 300x300 image on the screen, thus the sequences of 300 bytes account for almost 99% of the references. The Xterm program, which has the highest proportion of frame-buffer loads in the benchmark set, shows long load runs as well. This program frequently involves a copy of data from one screen line to another. Thus, it has two active screen pointers and copies from one to another. In other words, it exhibits a sequential access to two lines concurrently, with each line being about 720 bytes long.

6.7 Summary of Access Characteristics

The memory reference characteristics for the frame buffer are summarized below. First, the references tend to have very large working sets. In the context of D-cache design, this implies that the frame-buffer references may "crowd-out" other references from the cache. Also, they may result in excessive TLB thrashing. In fact some recent microprocessors support variable-size TLB pages to allow for the mapping of an entire frame-buffer address space in one page [81, 82]. Second, the references are oriented towards writes. Third, the references have a "sweep" behavior with long, consecutive access sequences. Another aspect of the accesses is that copying operations require concurrent maintenance of two pointers to long data structures.

7. DESIGN OF A FRAME-BUFFER CACHE

7.1 Introduction

Caches are frequently employed to bridge the speed gap between processors and memory. This section presents a trace-driven simulation study of various choices in cache-memory design for window-oriented programs. The instruction-level traces were collected on a DECstation 3100 workstation with a simple memory-mapped frame buffer without any special graphics processor hardware. The previous chapter presented some characteristics of the memory accesses and motivated the desire for a special frame-buffer cache.

The focus of this study is on the improvement of frame-buffer memory accesses for *display servers*, i.e., computer systems dedicated to providing the user-interface mechanisms to users. X terminals and inexpensive workstations are popular display servers in the X Window domain. These are frequently designed around common microprocessors

and off-the-shelf components [83, 84, 79, 80, 85, 86]. High-end workstations use specialized graphics processors to offload some of the graphical computation from the main processor [87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98]. These specialized processors can be integrated closely with the frame buffer to improve the memory performance. However, the system cost of the graphics processor can be prohibitive. An enduring tradeoff in the functionality between the special-purpose processor and the general-purpose processor is the "wheel of reincarnation" identified by Myer and Sutherland [99]. Special-purpose hardware is more expensive than general-purpose hardware, and eventually designers of general-purpose hardware build the special-purpose functionality into the next generation of their products, thus making the need for the special-purpose mechanisms obsolete. Accordingly, the desire here is to identify general-purpose memory mechanisms to enhance frame-buffer performance.

The memory-system model is shown in Figure 7.1, which is different from Figure 5.2 in that the data cache is augmented by a separate frame-buffer cache. All memory accesses are serviced by a common bus. Thus, the caches affect each other in the sense that high traffic on any one cache affects the queueing delays for bus service on the other caches. However, these delays are not modeled in the simulation results presented here. Note that the instruction-cache issues are not studied here either. The focus is on studying the tradeoffs in various organizations for the data cache.

As mentioned earlier, the traces were collected on a DECstation 3100 workstation, which is a 16.67 MHz R2000-based workstation. Since it represents the technology of a

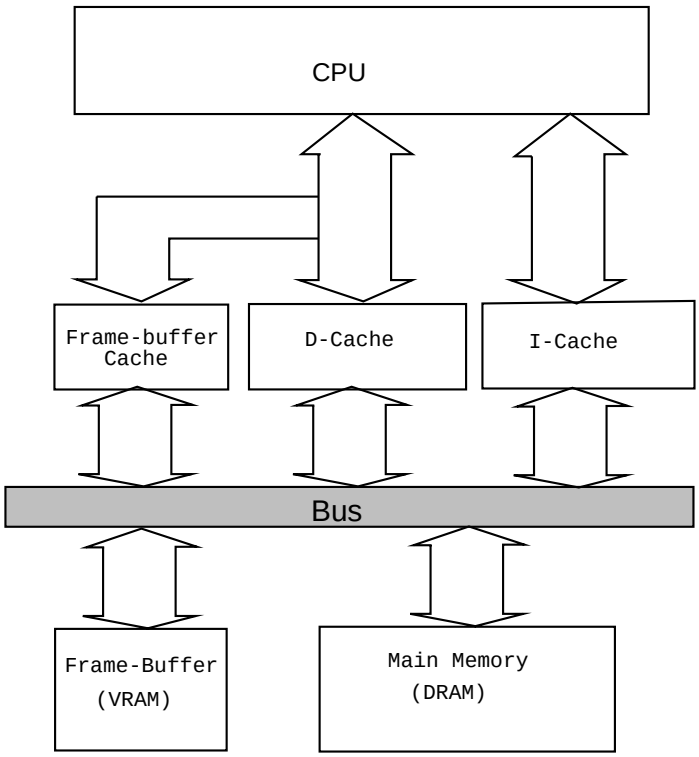

Figure 7.1: Memory system model.

few years ago, the simulations are based on a 33 MHz processor, i.e., a cpu cycle time of 30 ns. The frame-buffer memory is assumed to be built out of 4 banks of interleaved VRAM chips with a 240 ns cycle time and 120 ns burst mode access. This is similar to the Fujitsu MB81461-12 VRAM chip discussed in Chapter 5. With these assumptions, a burst mode access to the VRAM would cost 8 cpu cycles for the first byte and one cycle for each subsequent byte. Most accesses are to words of four bytes each, and the cost of a word access would be 11 cycles, i.e., 2.75 cycles per byte. This access time represents the upper limit on the tolerable access time.

7.2 Guiding Principles

A frame-buffer cache would be useful only if it improves the memory-access time at a reasonable implementation cost. On the basis of the data presented in Chapter 6, the following design principles suggest themselves.

7.2.1 Cache size

The frame-buffer accesses have large working sets, which correspond to the screen size of the displayed image. A naive interpretation of these data is that large cache sizes would be needed to accommodate the working sets in memory. However, several factors work against large frame-buffer caches. First, the total size of the VRAM in a typical color display-server is only about 1 M. Instead of investing in a large frame-buffer cache, the designers could choose to upgrade to a faster VRAM. Second, the embedded

processors used in low-end display servers, are single-chip solutions with limited silicon real estate on the VLSI chip. It is not practical to suggest a large frame-buffer cache for such chips. Third, the desire for a flushing mechanism, in the frame-buffer cache, as discussed later in Section 7.2.5, could add excessive complexity to a large cache.

7.2.2 Write policy

Most of the frame-buffer accesses are writes. Also, the writes come in long, consecutive sequences. Write-through caches do not typically make use of the burst-access characteristics of writes. Copy-back caches are the logical choice for such accesses since the unit of a memory write in such caches is an entire cache line. An additional advantage of a copy-back write policy is that fewer bus transactions would be invoked for a given amount of memory traffic as compared to the write-through policy.

7.2.3 Associativity

Direct-mapped caches are easier to implement than set-associative ones and are the common choice in cache design [67]. However, frame-buffer copies require maintenance of two frame-buffer lines in the cache and benefit from associativity. In a later section we show that for caches larger than a threshold size, the direct-mapped caches are good enough. Below this threshold a set-associative organization is necessary to avoid thrashing.

7.2.4 Write-allocate policy

Since the frame-buffer accesses have been shown to consist mainly of burst writes, cache lines would frequently be overwritten entirely before being written back. Therefore, we study the usefulness of the compiler strategy of giving the cache controller an *allocate-no-fetch* hint. This hint suggests that a cache line be allocated but not fetched from the main memory. Clearly, this hint is potentially very useful when the program intends to overwrite the entire cache line anyway. A drawback is that the compiler needs to have knowledge about the line size of the target cache. However, this may not be too limiting for use by the display-server programs, which are frequently delivered by the vendor as an integral part of the overall system. Some recent processors, such as the MC88110 [100] and HP-PARISC [81], have provided special instructions to support allocate-no-fetch. Our results suggests that such a strategy is effective in improving frame-buffer access time.

7.2.5 The need for flushing mechanisms

Frame-buffer writes have an effect on the visual appearance of the displayed objects. Thus, if frame-buffer accesses are cached, the cache must be kept consistent with the frame-buffer memory. Typically, the frame-buffer contents are displayed on the screen 60-72 times per second. The cache controller must flush the cache contents to the frame buffer memory at least this often. In the following simulation, the effects of such flushing

are studied. As shown later, such a flushing has the effect of increasing the traffic ratio for the cache only slightly.

7.2.6 Write buffers

In any memory-system design, write buffers are frequently used to buffer write accesses to the main memory. Proper write-buffer design can affect the actual access time substantially. In our simulation we compute the range of access times for infinite write buffers and with no write buffer. These can give the designer an idea of the bounds on performance for the buffer design. Actual access time would lie in this range, but would be highly dependent on issues such as the actual arrival rates of the write requests and the overall bus traffic.

7.3 A Suggested Frame-buffer Cache

Table 7.1 shows data for a suggested frame-buffer cache. The data are presented for the frame-buffer access time seen by the benchmark applications. The frame-buffer cache parameters chosen are as follows:

- Cache size: 64 bytes.
- Line size: 16 bytes.
- Associativity: 2-way.
- Write policy: copy-back with flushing.
- Write allocation policy: allocate-no-fetch.

Table 7.1: Access time for suggested frame-buffer cache.

Cache configuration	*Ghostscript*	*Ximage*	*Xterm*	*Xtetris*	*Xmagic*	*Xfig*
No cache	0.31-2.75	0.27-2.75	1.34-2.75	0.90-2.75	0.69-2.75	0.33-2.75
64B, 2-way, ANF	0.25-1.75	0.27-1.72	1.05-2.01	0.31-1.87	0.35-1.89	0.29-1.85

The table shows an improvement in access time for all of the benchmark applications. There is a striking improvement in the values of access time with no write buffers. This is a consequence of the burst-mode access characteristics for filling and for writing cache lines. The impact of various cache parameters in achieving this performance gain is examined in the following sections.

7.4 Line Size Effects

Figure 7.2 shows the access-time curves for an 8 K flushed 2-way-set-associative copy-back cache with allocate-no-fetch policy. The curve for no write buffers shows optimal line sizes of between 8 and 64 bytes for the various applications. Xtetris, which draws small objects on the screen has an optimum line size of 8 bytes. On the other extreme, Ximage, which paints animation frames on the screen, has an optimum at 64 bytes. However, all applications would be well served by a line size choice of 16 or 32 bytes.

The curve for infinite write buffers is monotonically increasing for all applications except Xterm. Since Xterm has a large proportion of frame buffer reads, it benefits from longer burst reads — up to a point. For all other applications, the longer line sizes

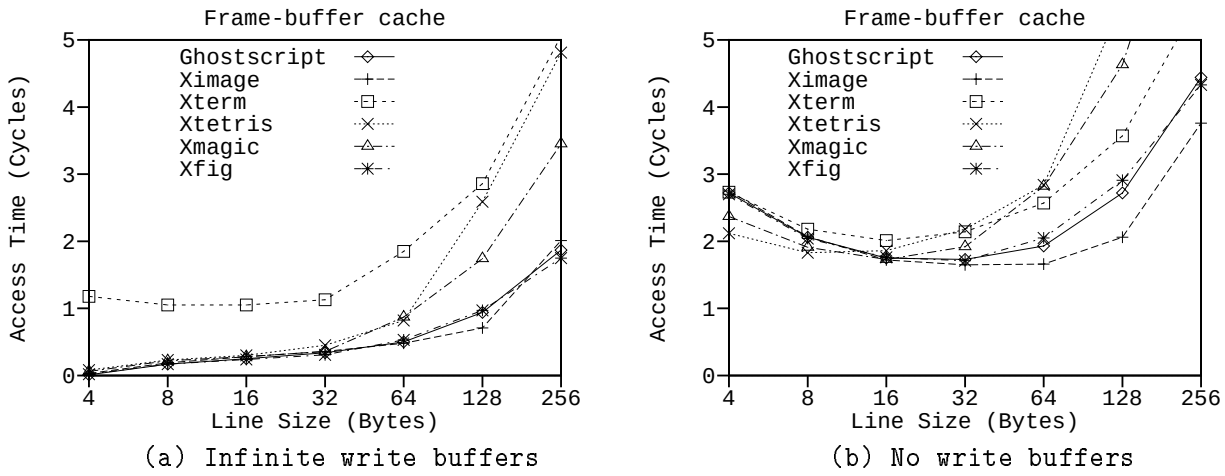

Figure 7.2: Effect of line size on access time.

only increase the probability of cache pollution. This effect is amplified by the allocate-no-fetch policy, which increases the relative penalty of fetching lines that are not to be entirely overwritten. Thus, the optimum line size is smaller than would be expected from the long burst write nature of the frame-buffer writes.

7.5 Cache Size Effects

Figure 7.3 shows the access-time curves for a flushed 2-way-set-associative copy-back cache with allocate-no-fetch policy and a line size of 16 bytes. The curves show that for small cache sizes up to about 4 K all caches exhibit the same access time. The explanation for this behavior lies in the "sweep" nature of the cache accesses. Also, the need for flushing makes larger caches behave equivalently to smaller ones. Xterm shows some improvement for larger cache sizes. This can be explained by the need to scroll

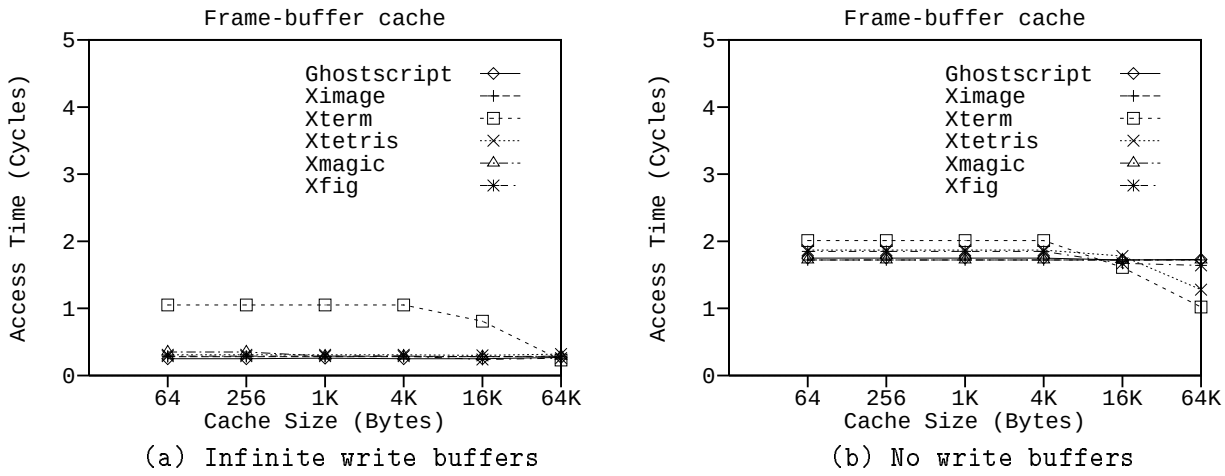

Figure 7.3: Effect of cache size on access time.

data on the screen. For a cache size beyond a certain point, the entire screen image can be contained in the cache and moved efficiently within it; the read operations for the memory copy do not require memory fetches. However, as pointed out in Section 7.2, smaller frame-buffer cache sizes are preferable.

7.6 Associativity Effects

Figure 7.4 shows the access-time curves for a 2-way set-associative cache and a direct-mapped cache. Both curves are for flushed copy-back caches with allocate-no-fetch policy and a line size of 16 bytes. The data show that for the frame-buffer references it is helpful to have a 2-way-set-associative cache when the cache-size is relatively small. The application that requires higher associativity in this case is Xterm. The reason is that Xterm is a "scrolling" application that frequently requests large data copies from one

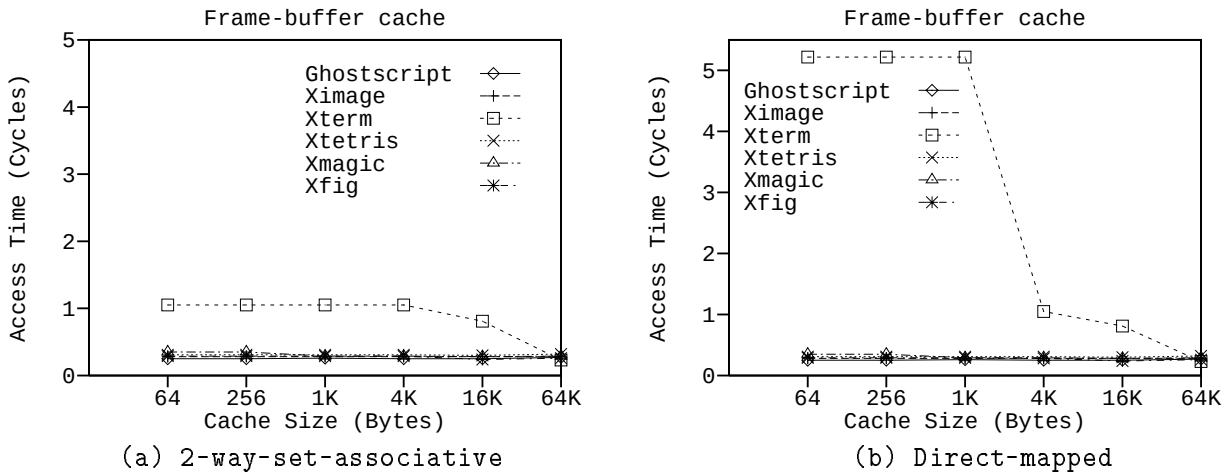

Figure 7.4: Associativity effects.

part of the display to another. To do this, it needs to read from one cache line and write to another. For a small cache, the source and destination are quite likely to map to the same cache line, thereby causing the cache to thrash. For a cache larger than 2K, it is possible to cache the data for two complete display lines. Thus, larger caches can be direct-mapped without causing problems for the data-copy class of applications. We studied 4-way set-associative caches as well, but these show no improvement over the 2-way set-associative caches.

7.7 Flushing Effects

The copy-back frame-buffer caches discussed in this chapter are all assumed to have hardware to flush their contents to the frame-buffer 60 times every second. To quantify the extra memory traffic due to this flushing requirement Figure 7.5 compares the

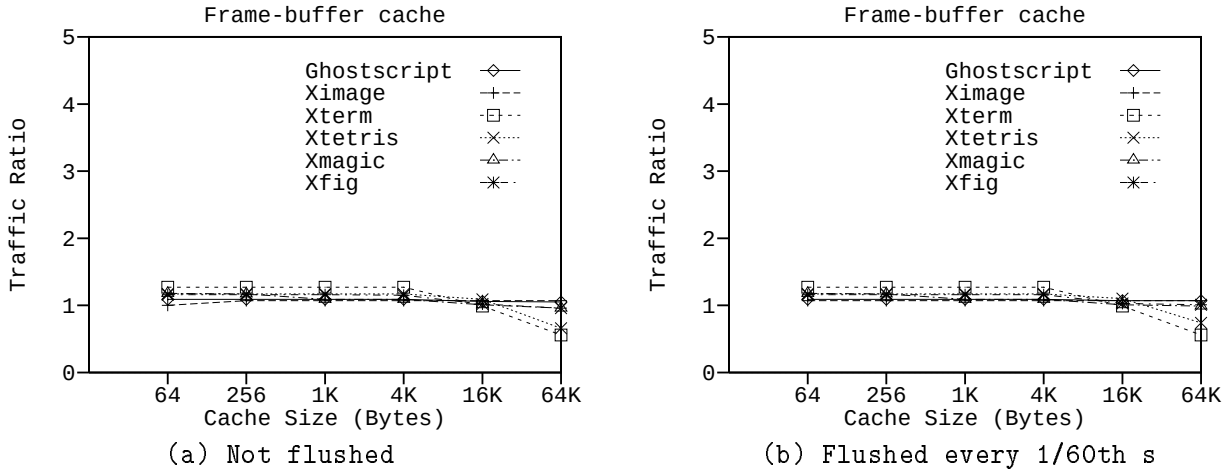

Figure 7.5: Flushing effects.

two traffic-ratio curves for 2-way-set-associative copy-back caches with allocate-no-fetch policy and a line size of 16 bytes.

Interestingly, the flushed caches exhibit memory traffic that is almost identical to that for unflushed caches. The explanation for this behavior is that the frame-buffer caches tend to be "self-flushing" due to the long write accesses that sweep through large working sets. Thus, the performance penalty of adding the flushing hardware is not great.

7.8 Effect of Allocate-no-fetch Policy

Figure 7.6 shows the advantage effected by the allocate-no-fetch policy. The curves are plotted for the traffic-ratios exhibited by 2-way-set-associative flushed copy-back caches with a line size of 16 bytes. The data assume that the compiler can successfully predict each instance in which the allocate-no-fetch hint is useful to the cache controller. It

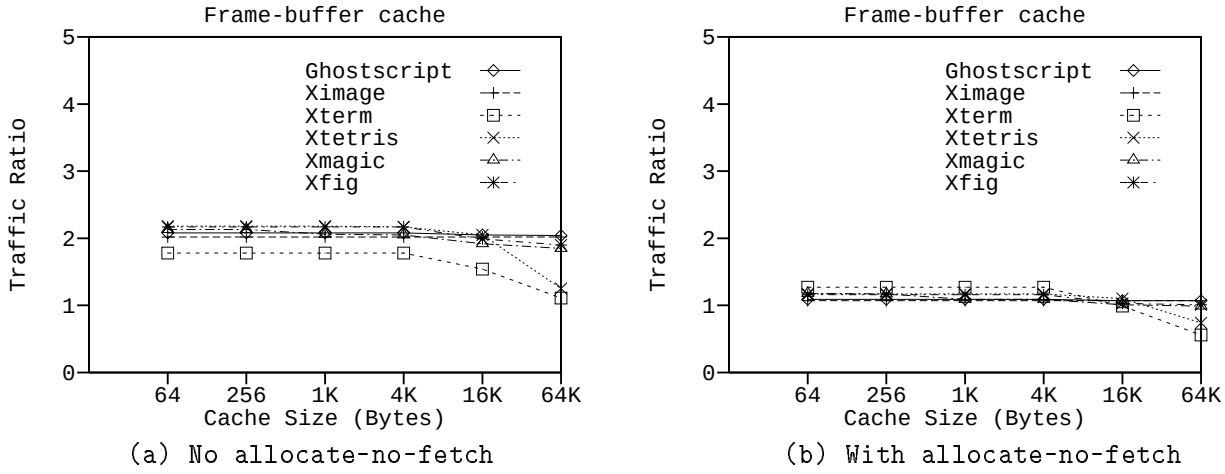

Figure 7.6: Effect of allocate-no-fetch policy.

turns out that each of the benchmark applications benefits from implementation of such a policy.

7.9 Cache Configurations

Table 7.2 shows access-time numbers for some cache configurations in addition to the suggested cache configuration discussed earlier. Table 7.3 shows the traffic ratios corresponding to these caches. These ratios represent the normalized frame-buffer memory traffic seen by the system bus. All data are shown for a flushed 2-way-set-associative copy-back cache with allocate-no-fetch policy and a line size of 16 bytes.

The tables show that the suggested cache (item B) has better access-time than the no-cache case (item A) but the memory traffic is slightly higher. It should be noted that the cache will reduce the number of memory bus transactions since each memory request is made in units of cache line size, which, for the data shown, is four memory words.

Table 7.2: Access time for frame-buffer caches.

Cache configuration	*Ghostscript*	*Ximage*	*Xterm*	*Xtetris*	*Xmagic*	*Xfig*
A. No cache	0.31-2.75	0.27-2.75	1.34-2.75	0.90-2.75	0.69-2.75	0.33-2.75
B. 64B, 2-way, ANF	0.25-1.75	0.27-1.72	1.05-2.01	0.31-1.87	0.35-1.89	0.29-1.85
C. 64B, 2-way	1.68-3.18	1.64-3.09	1.77-2.74	1.75-3.31	1.73-3.26	1.75-3.31
D. 64K, 1-way	1.65-3.12	1.64-3.09	1.01-1.81	1.07-2.01	1.52-2.86	1.54-2.92

Table 7.3: Traffic ratios for frame-buffer caches.

Cache configuration	*Ghostscript*	*Ximage*	*Xterm*	*Xtetris*	*Xmagic*	*Xfig*
A. No cache	1.00	1.00	1.00	1.00	1.00	1.00
B. 64B, 2-way, ANF	1.09	1.07	1.27	1.17	1.18	1.16
C. 64B, 2-way	2.08	2.02	1.78	2.18	2.13	2.17
D. 64K, 1-way	2.04	2.02	1.11	1.25	1.84	1.89

In contrast, most uncached requests are serviced in units of one memory word. Item B shows that without an implementation of the allocate-no-fetch policy, the frame-buffer cache actually performs substantially worse than the uncached case. Item D shows that the larger cache of 64K size does not perform much better than the small cache. As pointed out earlier, there is not much sense in implementing a larger cache size anyway since designers could choose to upgrade the VRAM instead.

7.10 Future Work

Memory system design is usually chosen from a rich design space. Many different choices are usually available to the designers. For instance, the conventional design choice is to use write-through data caches and to mark the frame-buffer cache accesses as uncacheable. Write-buffers are employed to avoid stalling the processor during a memory

write. Depending on the arrival rates of the write accesses, such a design could deliver acceptable performance. It would be interesting to study the utility of a finite number of write-buffers in both cached and uncached designs and derive guidelines for the optimal number of write-buffers for either design. Note that the analysis presented in this thesis does suggest that the cached design has better performance than the uncached design over the range of write-buffer choices. However, clever write-buffer designs, such as *combining write buffers* or *write caches* [72], could make use of the sequential write characteristics of the frame-buffer accesses even when the primary write policy is write-through. But, such write-buffer designs do not improve the access-time for reads, thus they may not deliver adequate performance for applications such as Xterm, which do a lot of frame-buffer reads.

Access time for cached read accesses can also be improved by following policies such as *load forwarding*. In this scheme, the needed data are forwarded to the processor before the line is entirely loaded into the cache. A *prefetching* strategy can be used to decrease the read latency as well. Such a strategy typically requires compiler support. Recent microprocessors have provided special instructions to support prefetching. Many studies have been made of the use of such a strategy for I-cache performance in RISC and superscalar processors, but there is no published work on the utility of prefetching for frame-buffer data.

7.11 Conclusions

This chapter promotes the use of a small set-associative copy-back cache with support for allocate-no-fetch policy to improve frame-buffer performance. The principal advantage accruing from such a cache is that it makes use of burst-access to the VRAM in units of the line size. The traffic seen by the memory bus is not reduced by such a cache though the number of bus transactions are.

A motivation for separate caches is that the frame-buffer references have large working sets of a "sweep" nature and could crowd out the non-frame-buffer references from a unified cache. Also, the frame-buffer references are made by only one program, the X display-server, which can be closely integrated with the cache hardware. The frame-buffer cache should follow a copy-back policy to exploit the burst writes. A copy-back policy necessitates a flushing mechanism that makes the cache consistent with the underlying frame-buffer 60 times a second for visual consistency. However, it turns out that the implementation of such a mechanism does not cause any substantial increase in the memory traffic.

The cache size of 64 bytes is shown to be good enough for the applications studied. Designers may choose to implement a slightly larger cache size, e.g., 256 bytes. The advantage of such a cache is that it could contain 16 lines of size 16 bytes each. Thus, for an application that exhibits reference locality in "tiling" patterns, an entire 16x16 tile could be contained in the cache.

The traces collected in this study are for a frame-buffer with one byte of data per pixel. Future color displays will use a full word of data for each pixel. It is to be expected that the optimal cache and line size parameters for such displays will be 4 times the values suggested in this study, i.e., cache sizes of 256 bytes with 64 byte line sizes. To cover the needs of different color displays, designers could choose to implement, for example, 256 byte frame-buffer caches with 32 byte line sizes.

8. CONCLUSIONS

This thesis research systematically explores the performance aspects of computers with graphical user interfaces. Traditional computer research, in contrast, has focused on the design of computers for numerical and transaction-oriented applications. The widespread use of workstations and personal computers as display-servers motivates the desire to identify the critical design issues for supporting display-oriented applications. In this research we look at performance issues for display-server computers at both system and microarchitecture levels.

At the system level, a new execution profiling strategy, called *protocol-level-profiling*, has been developed for analyzing client-server interaction. The central idea behind this strategy is to construct a profile of the server-side execution by analyzing a protocol-level trace of the client-server interaction. A protocol-level profiler, *Xprof*, has been developed for generating meaningful profiles for X Window applications. Results from this profiling techniques are presented for typical programs running on some typical workstations. Xprof constructs profiles of both the server-execution time and the network-transport overhead incurred on behalf of an application program. It also allows users to compare

the performance of different servers and different networks for execution of the same protocol trace.

At the microarchitecture level, a detailed instruction-level-tracing study investigates the frame-buffer-memory access characteristics of typical programs under the X Window System. The traces are collected by an instrumented version of the X display-server running on a DECstation 3100 workstation. Since X Window follows the client-server model of computing, the instruction-level traces of the X server encapsulate all accesses made to the frame-buffer on behalf of the application programs. The traces are analyzed for processing done on behalf of some typical X applications. The frame-buffer accesses are shown to be mainly writes that tend to occur in long bursts and have large working sets. This behavior occurs due to the fact that a handful of data-movement functions dominate the traces.

Choices in data-cache design are studied. Since the frame-buffer accesses have large working sets, they tend to crowd out the non-frame-buffer data in the data-cache. Therefore, the design of a separate frame-buffer cache is studied. The need for flushing copy-back caches for maintaining visual consistency with the display is simulated. A compiler policy of "allocate-no-fetch" is shown to be useful in reducing the traffic on the frame-buffer cache. It turns out that a small 2-way-set-associative cache with the allocate-no-fetch write policy can improve the overall memory-system performance. The cache does not reduce the frame-buffer traffic seen by the bus, though it does reduce the number of bus transactions. The main advantage of this cache is in improving the access time due to the burst access behavior.

REFERENCES

[1] C. Thacker, E. McCreight, B. Lanson, R. Sproull, and D. Boggs, "Alto: A personal computer," in *Computer Structures: Principles and Examples* (D. Siewiorek, C. Bell, and A. Newell, eds.), New York, NY: McGraw Hill, 1982, pp. 549-572.

[2] J. Johnson, T. Roberts, W. Verplank, D. Smith, C. Irby, M. Beard, and K. Mackey, "The Xerox Star: A retrospective," *IEEE Computer*, vol. 22, pp. 11–29, September 1989.

[3] J. Schoch et al., "Evolution of the Ethernet local computer network," *IEEE Computer*, vol. 15, pp. 10–27, August 1982.

[4] R. Siefert, "Ethernet: Ten years after," *Byte*, vol. 16, pp. 315–321, January 1991.

[5] D. Engelbart and W. English, *A Research Center for Augmenting Human Intellect, FJCC.* Washington, DC: Thompson Books.

[6] W. English, D. Engelbart, and M. Berman, "Display-selection techniques for text manipulation," *IEEE Transactions on Human Factors in Electronics*, vol. HFE-8, pp. 21–31, 1967.

[7] B. Shneiderman, "Direct manipulation: A step beyond programming languages," *IEEE Computer*, vol. 16, pp. 57–69, August 1983.

[8] B. Shneiderman, *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* Reading, MA: Addison-Wesley, 1986.

[9] G. Williams, "The Lisa computer system," *Byte*, vol. 8, pp. 33–50, February 1983.

[10] W. Gregg, "The Apple Macintosh computer," *Byte*, vol. 9, pp. 30–54, February 1984.

[11] C. Rose, B. Hacker, R. Anders, K. Wittney, M. Metzler, S. Chernicoff, C. Espinosa, A. Averill, B. Davis, and B. Howard, *Inside Macintosh*, vol. I. Reading, MA: Addison Wesley, 1985.

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

[13] R. Scheifler, J. Gettys, and R. Newman, *X Window System*. Bedford, MA: Digital Press, 2nd ed., 1990.

[14] 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.

[15] O. Jones, *Introduction to the X Window System*. Englewood Cliffs, NJ: Prentice Hall, 1989.

[16] T. Zimmerman, J. Lanier, C. Blanchard, S. Bryson, and Y. Harvill, "A hand gesture interface device," in *Proceedings of the CHI + GI 1987 Conference*, (New York, NY), pp. 189–192, 1987.

[17] J. Foley, A. van Dam, S. Feiner, and J. Hughes, *Computer Graphics — Principles and Practice*. Reading, MA: Addison-Wesley, 2nd ed., 1990.

[18] J. Foley, "Interfaces for advanced computing," *Scientific American*, vol. 257, pp. 126–135, October 1987.

[19] J. Warner, "Visual data analysis into the 90's," *Pixel*, vol. 1, no. 1, pp. 40–44, January 1990.

[20] T. A. DeFanti, M. D. Brown, and B. H. McCormick, "Visualization: Expanding scientific and engineering research opportunities," *IEEE Computer*, vol. 22, no. 8, pp. 12–25, August 1989.

[21] L. Rosenblum and G. Nielson, "Visualization comes of age," *IEEE Computer Graphics and Applications*, vol. 11, pp. 15–17, May 1991.

[22] A. Narasimhalu and S. Christodoulakis, "Multimedia information systems: The unfolding of a reality," *IEEE Computer*, vol. 24, pp. 6–8, October 1991.

[23] E. Francik, S. Rudman, D. Cooper, and S. Levine, "Putting innovation to work: Adoption strategies for multimedia communication systems," *Communications of the ACM*, vol. 34, pp. 53–63, December 1991.

[24] E. Fox, "Advances in interactive digital multimedia systems," *IEEE Computer*, vol. 24, pp. 9–21, October 1991.

[25] L. O'Gorman and R. Kasturi, "Document image analysis systems," *IEEE Computer*, vol. 25, pp. 5–8, July 1992.

[26] C. Schmandt, M. Ackerman, and D. Hindus, "Augmenting a window system with speech input," *IEEE Computer*, vol. 23, pp. 50–56, August 1990.

[27] R. Kamel, K. Emami, and R. Eckert, "PX: Supporting voice in workstations," *IEEE Computer*, vol. 23, pp. 73–80, August 1990.

[28] D. Wilson, "Wrestling with multimedia standards," *Computer Design*, vol. 31, pp. 70–88, January 1992.

[29] G. Wallace, "The JPEG still picture compression standard," *Communications of the ACM*, vol. 34, pp. 30–44, April 1991.

[30] W. Carlson, "A survey of computer graphics image encoding and storage formats," *Computer Graphics*, vol. 25, pp. 67–75, April 1991.

[31] T. Welch, "A technique for high-performance data compression," *IEEE Computer*, vol. 17, pp. 8–19, June 1984.

[32] A. Sinha, "Client-server computing: Current technology review," *Communications of the ACM*, vol. 35, pp. 77–98, July 1992.

[33] H. Lorin, "A model for recentralization of computing — Distributed processing comes home," *Computer Architecture News*, vol. 18, no. 1, pp. 99–108, March 1990.

[34] R. Sandberg et al., "Design and implementation of the Sun Network File System," in *Proceedings of the Summer Usenix Conference*, (Portland, OR), pp. 119–130, 1985.

[35] A. Tannenbaum and R. Renesse, "Distributed operating systems," *ACM Computing Surveys*, vol. 17, pp. 419–470, December 1985.

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

[37] M. Satyanarayanan, "Scalable, secure, and highly available distributed file access," *IEEE Computer*, vol. 23, pp. 9–210, May 1990.

[38] M. Satyanarayanan and E. Siegel, "Parallel communication in a large distributed environment," *IEEE Transactions on Computers*, vol. 39, pp. 328–348, March 1990.

[39] 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.

[40] A. Tannenbaum, R. van Renesse, H. van Staveren, G. Sharp, S. Mullender, J. Jansen, and G. van Rossum, "Experiences with the Amoeba distributed operating system," *Communications of the ACM*, vol. 33, pp. 46–63, December 1990.

[41] A. Birell et al., "Grapevine: An exercise in distributed computing," in *Communications of the ACM*, pp. 260–274, April 1982.

[42] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch, "The Sprite network operating system," *IEEE Computer*, vol. 21, pp. 23–36, February 1988.

[43] A. Black, "Supporting distributed applications: Experience with Eden," in *Proceedings of the 10th ACM Syposium on Operating System Principles*, pp. 181–193, December 1985.

[44] G. Almes et al., "The Eden system: A technical review," *IEEE Transactions on Software Engineering*, vol. SE-11, pp. 43–58, January 1985.

[45] M. Accetta et al., "Mach: A new kernel foundation for Unix development," in *Proceedings of the Summer Usenix Conference*, pp. 93–112, Usenix, Sunset Beach, CA, 1986.

[46] D. R. Cheriton, "The V distributed system," *Communications of the ACM*, vol. 31, pp. 314–333, March 1988.

[47] B. Walker et al., "The Locus distributed operating system," in *Proceedings of the 9th ACM Symposium on Operating System Principles*, pp. 49–70, October 1983.

[48] P. Dasgupta, R. LeBlanc, Jr., M. Ahamad, and U. Ramachandran, "The Clouds distributed operating system," *IEEE Computer*, vol. 24, pp. 34–44, November 1991.

[49] E. Jul et al., "Fine-grained mobility in the Emerald system," *ACM Transactions on Computer Systems*, pp. 109–133, February 1988.

[50] A. Gupta and W. Fuchs, "Garbage collection in a distributed object-oriented system," *IEEE Transactions on Knowledge and Data Engineering*, vol. 5, April 1993.

[51] A. Gupta and W. Fuchs, "Reliable garbage collection in distributed object-oriented systems," in *Proceedings of the 12th International Computer Software and Applications Conference*, pp. 324–328, October 1988.

[52] S. Angebranndt and T. Newman, "The sample X11 server architecture," *Digital Technical Journal*, vol. 2, pp. 16–23, Summer 1990.

[53] R. Scheifler, *X Window System Protocol: X Version 11, Release 5*. MIT X Consortium, Massachusetts Institute for Technology, Laboratory for Computer Science, 1991.

[54] G. Widener, "The X11 inter-client communication conventions manual," *Software Practice and Experience*, vol. 20, pp. S2/109–S2/118, October 1990.

[55] R. Rost, J. Friedberg, and P. Nishimoto, "PEX: A network-transparent 3d graphics system," *IEEE Computer Graphics & Applications*, pp. 14–26, July 1989.

[56] AT&T Bell Laboratories, Murray Hill, N.J., *UNIX Programmer's Manual*, January 1979.

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

[58] A. Gupta and W. Hwu, "Xprof: An execution profiler for window-oriented applications," Tech. Rep. CRHC-92-02, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL 61801, February 1992.

[59] A. Gupta and W. Hwu, "Xprof: Profiling the execution of X Window programs," in *Proceedings of the 1992 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 253–254, June 1992.

[60] A. Gupta and W. Hwu, "An execution profiler for window-oriented applications," *Software Practice and Experience*, to appear.

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

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

[63] 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.

[64] 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.

[65] A. Smith, "Cache memories," *ACM Computing Surveys*, vol. 14, pp. 473–529, September 1982.

[66] S. Przybylski, *Cache and Memory Hierarchy Design: A Performance Directed Approach.* San Mateo, CA: Morgan Kaufmann, 1990.

[67] J. Hennessy and D. Patterson, *Computer Architecture — A Quantitative Approach.* San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1990.

[68] M. Hill and A. Smith, "Evaluating associativity in CPU caches," *IEEE Transactions on Computers*, vol. 38, pp. 1612–1630, December 1989.

[69] A. Smith, "Line size choices for CPU cache memories," *IEEE Transactions on Computers*, vol. C-36, pp. 1063–1075, September 1987.

[70] A. Smith, "Second bibliography on cache memories," *Computer Architecture News*, vol. 19, pp. 154–182, June 1991.

[71] F. Jones, "A new era of fast dynamic RAMs," *IEEE Spectrum*, vol. 29, pp. 43–49, October 1992.

[72] B. Bray and M. Flynn, "Write caches as an alternative to write buffers," Tech. Rep. CSL-TR-91-470, Computer Systems Laboratory, Stanford University, Stanford, CA 94305, April 1991.

[73] J. Nicoud, "Video RAMs: Structure and applications," *IEEE Micro*, vol. 8, no. 1, pp. 8–27, February 1988.

[74] Fujitsu Microelectronics, Inc., San Jose, CA, *Dynamic RAM Products: 1991 Data Book*, 1991.

[75] M. Smith, "Tracing with *pixie*," Tech. Rep. CSL-TR-91-497, Computer Systems Laboratory, Stanford University, Stanford, CA 94305, November 1991.

[76] Sun Microsystems Laboratories, Inc., Mountain View, CA, *Shade User's Manual*, 1991.

[77] C. Stephens, B. Cogswell, J. Heinlein, G. Palmer, and J. Shen, "Instruction level profiling and evaluation of the IBM R/S 6000," in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pp. 180–189, May 1991.

[78] P. Chang, S. Mahlke, W. Chen, N. Warter, and W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pp. 266–275, May 1991.

[79] T. Furlong, M. Nielsen, and N. Wilhelm, "Development of the DECstation 3100," *Digital Technical Journal*, vol. 2, pp. 84–88, Spring 1990.

[80] J. McCormack, "Writing fast X servers for dumb color frame buffers," *Software Practice and Experience*, vol. 20, pp. S2/83–S2/108, October 1990.

[81] E. DeLano, W. Walker, J. Yetter, and M. Forsyth, "A high-speed superscalar PA-RISC processor," in *Proceedings of the 37th IEEE Computer Society International Conference, (COMPCON Spring 1992)*, pp. 116–121, February 1992.

[82] S. Mirapuri, M. Woodacre, and N. Vasseghi, "The Mips R4000 processor," *IEEE Micro*, vol. 12, pp. 10–22, April 1992.

[83] D. Wilson, "X Windows terminals designers search for a single-processor solution," *Computer Design*, vol. 30, pp. 77–88, August 1991.

[84] A. Socarras, R. Cooper, and W. Stonecypher, "Anatomy of an X terminal," *IEEE Spectrum*, vol. 28, pp. 52–55, March 1991.

[85] M. Dolan and L. Hare, "X Window system servers in embedded systems," in *Proceedings of the 35th IEEE Computer Society International Conference, (COMPCON Spring 1990)*, pp. 314–319, February 1990.

[86] R. Horning, M. Forsyth, J. Yetter, and L. Thayer, "How ICs impact workstations," *IEEE Spectrum*, vol. 28, pp. 58–68, April 1991.

[87] K. Akeley, "The Silicon Graphics 4D/240 GTX supercomputer," *IEEE Computer Graphics and Applications*, vol. 9, pp. 71–83, July 1989.

[88] T. Diede, C. Hagenmaier, G. Miranker, J. Rubinstein, and W. Worley, Jr., "The Titan graphics supercomputer architecture," *IEEE Computer*, vol. 21, pp. 13–30, September 1988.

[89] H.Baeverstad, "Engineering and scientific visualization using high-performance graphics workstations," in *Proceedings of the 34th IEEE Computer Society International Conference, (COMPCON Spring 1989)*, pp. 328–333, 1989.

[90] K. Guttag, J. Aken, and M. Asal, "Requirements for a VLSI graphics processor," *IEEE Computer Graphics & Applications*, vol. 6, no. 1, pp. 32–47, January 1986.

[91] D. Pinedo, "Window clipping methods in graphics accelerators," *IEEE Computer Graphics and Applications*, vol. 11, pp. 75–84, May 1991.

[92] C. Priem, "Developing the GX graphics accelerator architecture," *IEEE Micro*, vol. 10, pp. 44–54, February 1990.

[93] R. Pike, "Graphics in overlapping bitmap layers," *Computer Graphics*, vol. 17, pp. 331–356, July 1983.

[94] R. Pike, "The Blit: A multiplexed graphics terminal," *AT&T Bell Laboratories Technical Journal*, vol. 63, pp. 1607–1631, October 1984.

[95] R. Pike, B. Locanth, and J. Reiser, "Hardware / software tradeoffs for bitmap graphics on the Blit," *Software – Practice and Experience*, vol. 15, pp. 131–151, February 1985.

[96] J. Kelley, "Design experience with a multiprocessor window system architecture." M.S. thesis, University of Waterloo, Waterloo, 1988.

[97] G. Singh, "GRIP: Graphics reduced instruction processor," in *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pp. 143–152, November 1991.

[98] G. Kaplan et al., "Focus report and guide to engineering workstations," *IEEE Spectrum*, vol. 29, pp. 25–61, April 1992.

[99] T. Myer and I. Sutherland, "On the design of display processors," *Communications of the ACM*, vol. 11, pp. 410–414, June 1968.

[100] K. Diefendorff and M. Allen, "Organization of the Motorola 88110 superscalar RISC microprocessor," *IEEE Micro*, vol. 12, pp. 40–63, April 1992.

VITA

Aloke Gupta was born on October 28, 1963 in New Delhi, India. In 1985 he received his B.Tech. degree in Electrical and Computer Engineering from the Indian Institute of Technology, New Delhi, India. During his undergraduate years he was the recepient of a merit scholarship awarded by the National Center for Educational Research and Training, India. His B.Tech. major project involved the building of a robotic manipulator arm and won the Rajiv Bambawle award for the best senior project. His graduate work was conducted at the University of Illinois, Urbana-Champaign, where he received his M.S. degree in 1990 and the Ph.D. degree in 1993. The M.S. research involved the development of a low-overhead garbage collection algorithm for a distributed object-oriented system. The Ph.D. research explored performance aspects of computers with graphical user interfaces. Both degrees were awarded for research done at the Center for Reliable and High-Performance Computing. He worked as a teaching assistant during part of his graduate work and was included in the "List of Teachers Ranked as Excellent" for all five semesters of teaching. Dr. Gupta may be reached at the Department of Electrical and Computer Engineering at the Oregon State University, Corvallis, Oregon.