BUILDING A SCALABLE HIGH-RESOLUTION DISPLAY WALL

YUQUN CHEN

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

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

JANUARY 2001

Abstract

The past three decades have seen dramatic improvements made in computer and networking technologies. But display resolution, an important aspect of modern information systems, has lagged behind. Previous efforts in scaling up display resolution have relied on special-purpose hardware for building both the physical displays and the computer systems that drive the displays. The prohibitively high cost of these efforts limited the scalability of the systems as well as their market adoption. This dissertation describes a *Display Wall* architecture that uses projector tiling and PC clustering to achieve scalable resolution and at the same time render 2D and 3D graphics scenes as fast as or even faster than a fast PC graphics system with a traditional display. Unlike previous approaches that relied on custom-made hardware, this architecture results in low cost per pixel by simply leveraging commodity components such as projection devices, personal computers, graphics accelerators, and off-the-shelf system-area networks. Yet many challenging problems arise when we cluster PCs and tile projector together. The most notable ones are how to build efficient communication transport within the PC cluster, how to coordinate many PCs to display smooth motion graphics, and how to tile projectors without visible seams. This dissertation focuses on three research issues in building a tiled, cluster-based display wall: seamless tiling, efficient cluster communication, and runtime environments for a cluster-based scalable display. For each issue, a novel algorithm is described and its effectiveness evaluated with empirical results.

Acknowledgments

This dissertation would not be possible without great support from my advisor Professor Kai Li, who came up with the idea of a scalable Display Wall in the first place and placed tremendous enthusiasm behind the project for the past two years. I am also indebted to Professor Douglas W. Clark, who shared with me his deep insights into matters both technical and non-technical. I thank Adam Finkelstein and Kenneth Steiglitz for helping me with developing our automatic calibration algorithm, Larry Peterson for serving as a reader, Perry Cook for serving on my thesis committee, and visiting professor Ben Shedd for encouraging the creative side of my work. I also thank our departmental Graduate Coordinator Melissa M. Lawson, *the other mom for every raduate student*, who has gotten some many things done for us. She deserves a life-time achievement award!

I am grateful to many teachers throughout my college and post-graduation years, for educating and inspiring me. I thank physicists Zhichang Yang and Duowu Pan from Fudan University, Shanghai, computer scientists George Markowski and Larry Latour, sociologist Steven Cohn, physicists Kenneth Brownstein and Susan McKay, all from University of Maine, as well as modern dancer and choreographer Aleta Hayes from Princeton University Dance Department. Past three and a half years have been a very difficult, uphill struggle for me. I could not have gone this far, had it not been for warm friendship from my close friends, Stefanos N. Damianakis, Jian Gu, Mariusz Jakubowski, Xiaoxia Lin, Yaoyun Shi, Daniel C. Wang, Wei Zhang, and from my officemates and colleagues, Mao Chen, Amit Chakrabarti, Funyun Cao, Benjamin Gum, Minwen Ji, Lena Petrovic, Emil Praun, Yefim Shuf, Kedar Swaldi, Liming Wang, Bin Wei, and Xiaodong Wen.

Many professors and staff members have made the Computer Science Department a warm and personal place for me to work and hang out. I thank Joe R. Crouthamel, James M. Roberts, Christopher J. Tengi, Sandy Barbu, Rebecca Davies, Ginny Hogan, Patricia Killian, for their kind help throughout my years at Princeton.

The Princeton Scalable Display Wall Project is funded by Department of Energy under grant ANI-9906704 and DE-FC02-99ER25387, and by Intel Corporation under their Intel Research Council and Intel Technology 2000 Equipment grant. Steve Hunt from Intel Microprocessor Architecture Lab, in particular, has shared with us software and equipment.

And lastly, it is the omnipresent Spirit that has inspired Man to seek and pursue his true calling despite temptations and adversities. This dissertation is but an insignificant tribute to Him. One only needs to look up at the night sky to appreciate the magnificence of the Universe. Infinitely much more is still beyond our grasp. The road has just begun.

Und die Seele unbewacht
Will in freien Flügen schweben,
Um im Zauberkreis der Nacht
Tief und tausendfach zu leben.

– ”Beim Schlafengehen,” Herman Hesse

To my parents who gave me fond memories and fostered my curiosity

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 A Case For High Resolution and Immersion

Computers are increasingly becoming the information medium through which humans interact with the physical world and with each other. Using data analysis, visualization, and simulation, we explore and understand the physical world. To communicate with each other, we exchange documents and messages, craft presentations, browse the web, conduct teleconferencing, and chat on-line.

Digital information is presented on a variety of displays, frequently in a content-rich form – for examples, video footage, stock prices, 3D models, and dense network traffic graphs – either because the analytical and inferential abilities of a computer are still much too primitive to distill pithy insights from raw data, or because we humans are simply too unwilling to let a machine think for us.

It follows that the more detail a display is capable of presenting, the better the chances we will have discovering something subtle and significant. This is not to say that detail – or the display resolution – is the only important aspect of an effective display. How to extract pieces of relevant information from raw data, organize and present them in a

clear and meaningful manner on the display is still very much an art, its significance often underestimated [89]. But, everything else being equal, a display with higher pixel density and larger screen surface certainly makes a better information medium.

High resolution and immersion are the two key features of an effective information display. As the technologies for producing thin-film large-area display materials mature, one can imagine that in a not-too-distant future immersive displays with ultra-high resolution will be as ubiquitous as wallpaper and televisions, that we perceive on these displays a high-fidelity snapshot of the real world or virtual reality in the same way we look at our ordinary physical surroundings with comfort and ease.

However, computer displays produced today subtend only a tiny portion of our visual field of view (FOV). Without turning the head, a human can see a wide spatial volume that is 210° horizontal and 130° vertical [3]. Projecting a computer display onto a very large screen increases immersion, but at the cost of much lower display resolution. Our eyes can discern two objects in the center of the retina that are only one-minute arc apart [3]. Besides, we are also free to turn the head and walk about in front of the display medium. To match the sharpest visual acuity everywhere on an immersive display would require many, many millions of pixels.

Let us imagine building just such an immersive display to replace the traditional white-board in an office. This digital white-board could be a cylindrically shaped surface that surrounds 180° of one's horizontal FOV, 2 feet above the ground, and 6 feet tall to match the height of a human being. Assuming a comfortable viewing distance of 4 feet, the display would measure 12.6 feet long and 6 feet tall. In order to match the human visual acuity everywhere on the display surface, we would need 22,500 pixels per square inch [1], or 244,944,000 pixels on the entire display!

A display of 240+ million pixels represents an increase of 77 times in physical res-

[1] An easy way to calculate the pixel resolution is to remember that at a two-feet distance (61 cm), our visual acuity can discern a print quality of 300 dpi [51].

olution from that of the state-of-the-art monitor (2048x1536) [51]. It also demands an improvement of 117 times in the graphics system capacity from what the best commodity graphics accelerator can offer today (1920x1080). Were Moore's Law to hold good for next decade, we would still have to wait for at least nine years for technologies to catch up to our imagination. This is a very, very long time for a society that is increasingly accustomed to fast-paced technological innovations. Providing a relatively inexpensive solution for high-resolution displays can contribute to innovations and scientific discoveries that are being made today instead of a decade later. It was with this motivation that the Princeton Scalable Display Wall project was initiated in 1998. Its goal is to demonstrate that a high-resolution and high-performance scalable display can be built using only commodity components, and that a system built this way is relatively inexpensive and capable of keeping track of rapid advances in the computer industry.

1.2 Technical Challenges

There are three main technical challenges in building an efficient, high-resolution, immersive displays: finding the physical display material, building an efficient digital system to drive the pixels, and writing application software that leverages the high resolution and dense pixels. In addition, there are design issues on how to lay out information on an immersive high-resolution display, as well as psychological questions regarding our perceptual responses to dense pixels and immersion. The initial goal of our research project is to tackle the technical challenges first; the real display has to be built, before designs can be meaningfully experimented and psychological studies carried out. Below is a brief description of the three technical challenges.

Physical display: Today, CRT monitors are the predominant computer displays. Two inherent structural components of a color CRT, the vacuum tube and the shadow mask, make it unsuitable for building a high-resolution immersive display. A CRT that spans 12 feet long and 6 feet high is nearly impossible to build; and even if built, it would be fantastically expensive and too bulky to fit in an office and too fragile to maintain. A larger and higher-pitched shadow mask is also difficult to make and align. Present technology for making the CRT shadow masks limits the spatial resolution to about 88 dpi in advanced displays [67], only half of what we would like for our immersive digital white-board.

Flat panel displays, whose prices have been falling and whose resolutions have been increasing steadily for the past decade, are a likely candidate as the immersive display material. The prevalent flat panel technology uses *amorphous liquid crystal* (AMLCD). However, it is unclear whether in the near term wall-sized AMLCD could be produced, and if could, whether it is economical. A more promising candidate is the *organic LED* (OLED) material [2]. In theory, OLED material can be manufactured in large size and large quantity using processes not unlike producing plastic sheets. It is also bright and durable. At the present time, however, OLED is still undergoing laboratory research and development. We probably will not see mass production of large-area OLED displays for another five years at least.

Graphics subsystem: It is a significant challenge to refresh 250 million pixels at a rate that is fast enough for presenting smooth motions. For motion pictures on regular cinema screens, it has been established that the entire screen be updated at least 24 times a second [69]. The refresh rate of 24 frames per second, or 24 fps, is only the minimum. For larger screens such as those used in IMAX theaters, this rate has been found to be too low to avoid apparent tears when objects move across the screen in very short time. Even at 24 fps, our display system is required to refresh a total of 6 billion pixels per second. It is

an enormous demand on various components of the graphics subsystem: the pixel bus, the frame buffer, the rendering engine, and the system bus. In a typical graphics subsystem, shown in Figure 1.1, primitives generated by an application flow from the system bus to the graphics engine where they get rendered into pixels in a frame buffer. A finished frame is sent to the physical display via a pixel bus. Here are the limits of current systems:

- The system bus can at best deliver 10 million triangles, each of which is specified with between 50 to 100 bytes, from the memory or the CPU to the graphics accelerator.
- The best high-end graphics accelerator in the market can render 10 million 25-pixel, colored triangles per second.
- Present generation of graphics frame buffer and can handle at most 200 million pixels per second refresh.

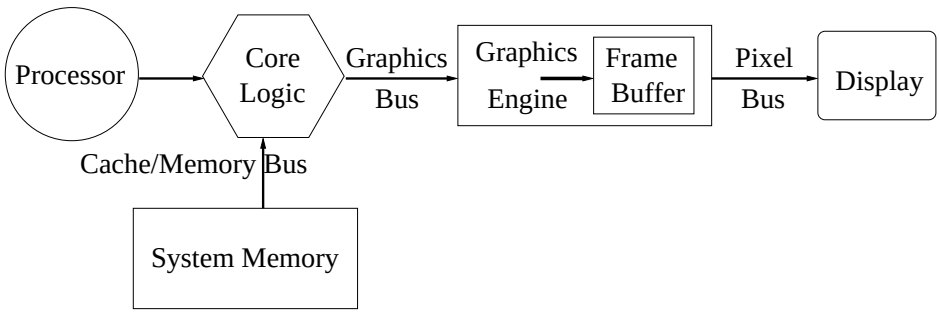


Figure 1.1: A typical system graphics pipeline

Roughly speaking, a factor of 25 exists between what current graphics system can deliver and what an immersive, high-resolution display demands. No single computer system can handle so high a communication bandwidth for updating the pixels and the enormous computation power to generate the pixels for dynamic scenes. A special architecture is required.

Application layer It would be a poor use of 250 million pixels if we only show a super-fine square or a really, really shiny and smooth sphere – very much like watching an plushly-made movie filled with tons of impressive special effects and pretty faces but little content. The applications should generate additional fine details that are not "trivially" generated from coarser data. For realistic modeling, this means that minute details be added and more sophisticated lighting conditions be taken into account. Animated models may require a large amount of computing power for updating the models during each frame. Visualizing scientific and medical data such as CT scans can benefit from higher data sampling precision which in turn demands more computation power for post-processing and analysis.

A single PC lacks sufficient power in its processors and enough bandwidth in its system bus to sustain these kinds of data-intensive computation. Ideally one would need a scalable parallel-processor for this task. Conventional uni-processor applications, in turn, require modification to take advantage of the parallel-processor architecture.

1.3 Architectural Choices

It might be possible to design and build a single-piece graphics hardware engine capable of driving 100 million or more pixels, using the best technologies currently available. But such a brute-force approach would incur prohibitively high cost and take years to design and implement. Just as in many engineering and research problems where, when one is not enough, people just use more – parallel computers being a perfect example – we can just integrate multiple display tiles and graphics subsystems into a high-resolution display system.

A clear benefit of the tiling approach is limited burden that is placed on each graphics accelerator and the pixel bus. The other, and equally significant, benefit is that commodity

graphics accelerators can be used to construct the system. One can almost go out and buy the state-of-the-art components off the shell and build a giant display in a day!

But, there are still two critical pieces missing from this picture: the hardware and the software that "glues" the graphics accelerators and the display tiles together so that they function cooperatively as a seamless display system. For instance, with many graphics subsystems comes the issues of communication and data distribution. After all, primitives specifying images, texts, videos, polygons, and colors must be communicated from the processors and memory to the graphics subsystems. Very often, screen data may be moved between graphics subsystems, for example, when one drags a window from one display tile to next one.

A central question here is choosing the right computer architecture, both hardware and software, for gluing multiple graphics subsystems. There are three general approaches: the *naive*, the *ideal*, and the *cluster-based*. The hardware architecture entails tradeoffs between software complexity and performance. Below we examine each of the three architectures and provide a qualitative analysis by looking at two critical components in each system: the interconnection and the application computation layer.

The naive architecture In this simple solution, several graphics accelerator are plugged into the I/O bus of a computer that functions as the communication interconnect (Figure 1.2). The operating system presents a contiguous, virtual display space to applications. It is responsible for distributing graphics primitives to the graphics accelerators. Support for multiple graphics accelerators, or multi-monitor support, first became available on Apple computers in late 80's and was later incorporated into Microsoft's Windows 98 and Windows 2000.

The simplicity of the naive architecture is also its drawback. The bus is a potential bottleneck. It limits the amount of graphics primitives that can be distributed to the graph-

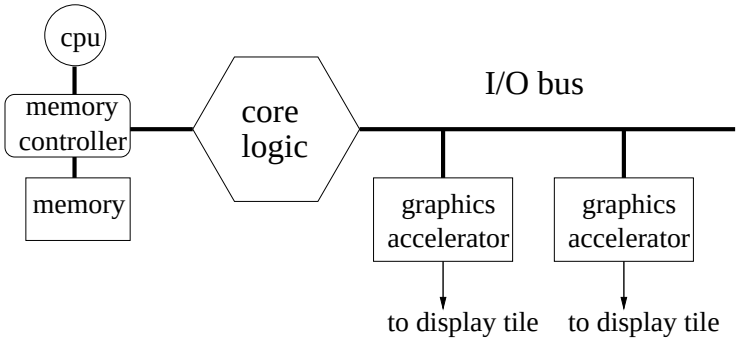


Figure 1.2: The naive display system architecture

ics accelerators. Furthermore, the main processor(s) and the system bus lack sufficient power to process raw data in order to generate highly-detailed graphics primitives in vast quantities.

To address these two bottlenecks, one can add more processors, replace the system bus with a multi-port switch, and replace the I/O bus with a switch fabric. Thus one arrives at the ideal display system architecture.

The ideal architecture Ideally, a parallel computer, with many processors, a powerful memory subsystem, and a large number of graphics ports to connect to graphics accelerators, has sufficient computation power for applications and enough capacity in the system interconnect to move graphics primitives around. Figure 1.3 shows one possible arrangement of processors, memory, and graphics accelerators in a parallel architecture. It is practically a modern cache-coherent distributed memory multi-processor [53, 54, 50]. On such a machine, a parallel application, written to take advantage of multiple processors, generates massive quantity of graphics primitives, which are distributed by the switch fabric to the graphics accelerators.

Commercially such an architecture have been already proven viable. SGI has built and marketed a scalable, cache-coherent, shared-memory multi-processor, Origin 2000, that scales easily above 256 processors. Multi-processor computers with 8 graphics pipelines,

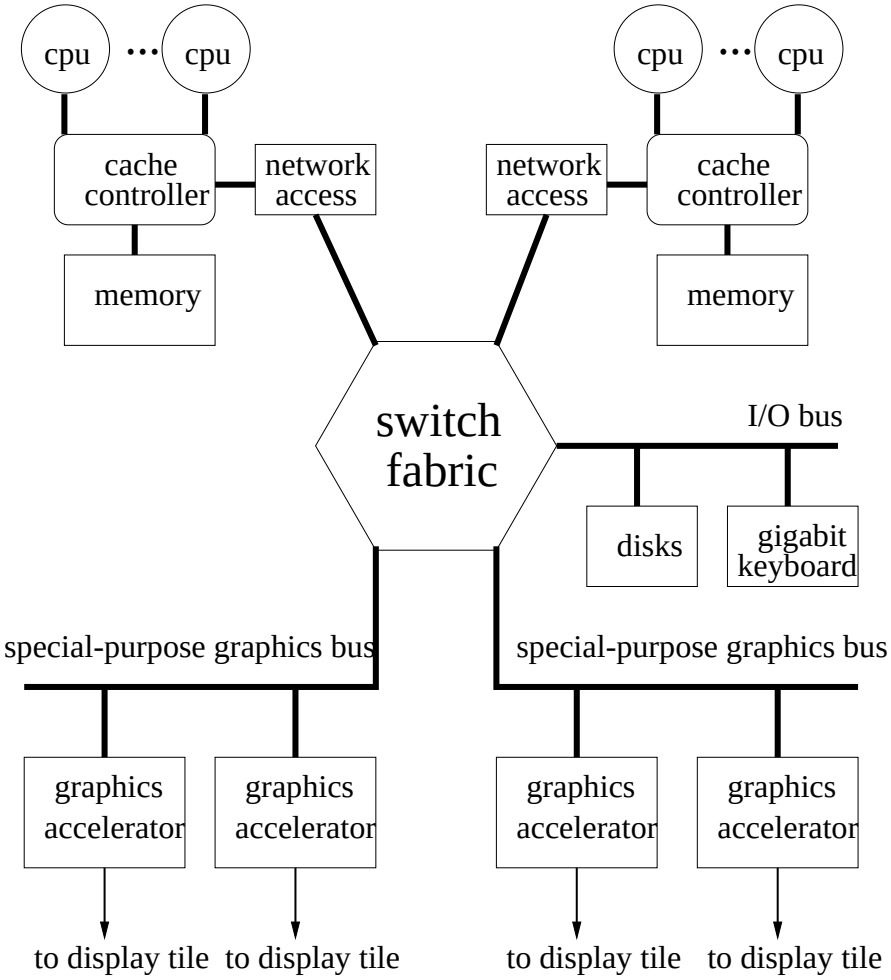


Figure 1.3: The ideal display system architecture

each capable of driving 2 displays, have also been built by SGI and enjoyed enjoyed limited success in the market [59].

As a research project, Wei *et al.* developed a distributed frame buffer, comprised of multiple frame buffer boards cooperating in interleaving fashion, for the Intel Paragon Multicomputer [85, 92]. A parallel rendering package, developed for message-passing architectures, uses the computation nodes in Paragon to rasterize 3D primitives and sends the pixel fragments to the frame buffer boards via high-speed Intel Paragon backplane. The frame buffer boards, though configured for interleaving a single display, can be easily modified to drive multiple displays.

A serious drawback of the ideal architecture its dependency on custom-made components. Both examples mentioned above relied on special-purpose parallel computers that were far more expensive to build and had a much longer time-to-market than commodity computers such as Intel-based symmetric-multi-processor (SMP) servers. Each also used custom-made graphics boards that suffered the same drawbacks as the computers. As the absolute performance, as well as the cost-performance ratio, of commodity processors and graphics accelerators improve according to the Moore's Law, it is more sensible to build an ideal architecture that integrates these commodity components using a high-bandwidth, low-latency interconnection network. A few recent commercial systems were built this way. But none supports multiple graphics accelerators. Due to the extremely high performance demanded from the interconnect, its design and engineering is highly complex. Besides, the market for scalable multi-processor servers is quite small; commercial vendors cannot yet leverage economy of scale for producing these systems. But the author remains hopeful that in time, machines built using the ideal architecture will become readily accessible at affordable prices. At the meantime, we can leverage commodity interconnection hardware to build something less ideal but still better than a bus-based naive architecture.

The clustering approach A middle ground can be found between the ideal and the naive architectures, by replacing the tight coupling in the ideal architecture with a loose one. Instead of a tightly-integrated multi-processor, we have a PC cluster, connected by an off-the-shelf system-area network. In this configuration, shown in Figure 1.4, each PC functions as a relatively independent unit; it has its own processors, memory, graphics accelerator(s), and perhaps disks; it runs applications on a standard operating system, for example, Windows 2001. A system-area network such as Myrinet [10] and Giganet [1] provides the communication link among the otherwise autonomous PCs.

The obvious advantage of a cluster-based architecture is the powerful graphics subsys-

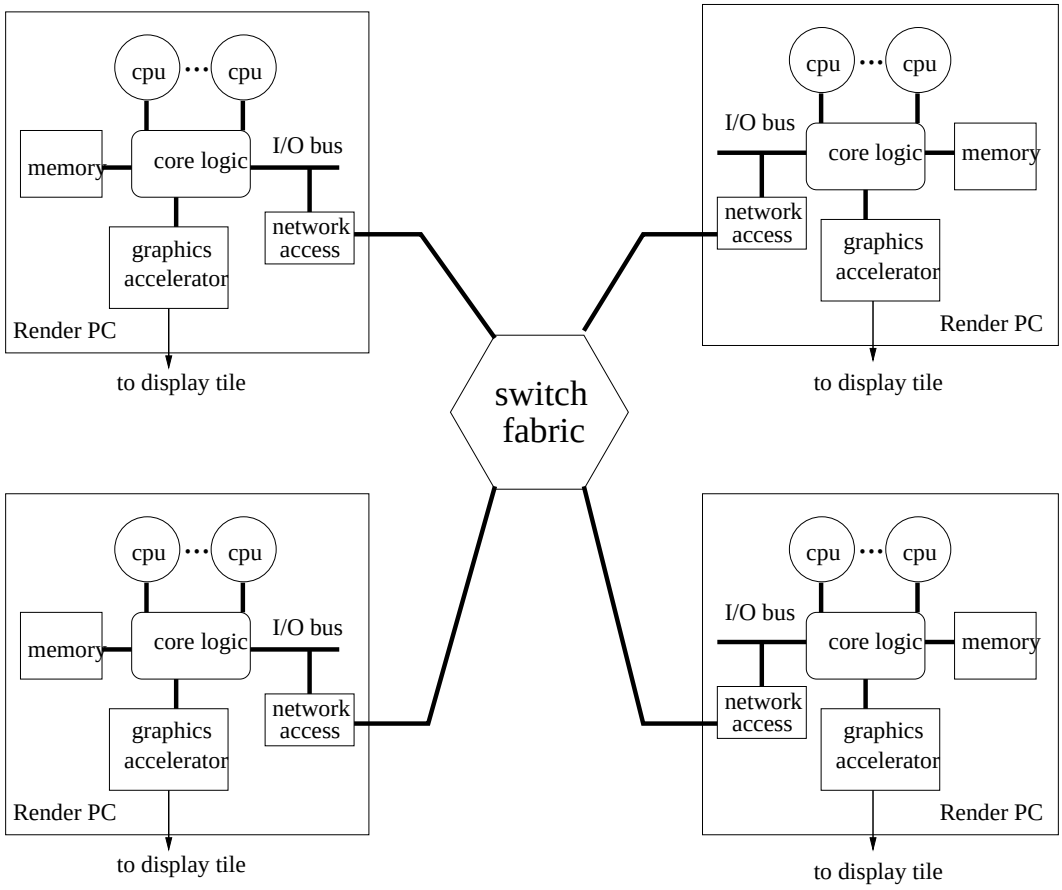


Figure 1.4: The cluster-based display system architecture

tem. At any time, one can simply replace the graphics accelerator in each PC with the best on the market. Each PC has enough computation power, memory, and graphics acceleration to drive one (or a couple of) display tiles(s). The entire system is so modularized that PCs themselves can be easily upgraded.

The drawback of this architecture lies in the interconnect which is weaker than that in the ideal architecture. An order of magnitude of difference in both communication bandwidth and latency exists between the interconnection network and the local memory subsystem of each PC. As a result, an operating system for running a single application across the cluster still remains an illusive though interesting goal on paper. Work-arounds, due to insufficient bandwidth and relatively high latencies, have to be devised. A cumber-

some programming model such as message passing, instead of the usual shared-memory paradigm, is adopted because it differentiates between local memory accesses and intra-cluster communication. Since local communication within each PC is far more efficient than that across the network, one has to devise clever partitioning of data and computation on the render nodes in order to avoid bottlenecks in the network.

1.4 A Prototype Display Wall Architecture

The Scalable Display Wall Project took the clustering approach to build a prototype immersive display system because clustering avoids the lengthy and costly process of engineering the ideal system; the components, being commodities, are all readily available. Furthermore, there is sufficient resemblance between a cluster-based display system and the ideal architecture, such that some research issues are common to both architectures. We hoped that insights gained from studying a cluster-based system will be applicable to the ideal architecture.

The design philosophy of our scalable display wall is to take commodity components as the basic building blocks and devise a clever "glue" to integrate them into a seamless and high-performance display system. In current implementation, the physical display is tiled by an array of portable projectors; driving the projectors is a cluster of PCs with commodity graphics accelerators; connecting the PCs together is an off-the-shelf system-area network, Myrinet [10].

To glue all these hardware pieces together, we developed the system and application software that computes complex scenes, distributes the graphics primitives to the PCs, and coordinates image refresh on all display tiles. This chapter provides a detailed look at each aspect of the Display Wall architecture and puts forth three research challenges.

1.4.1 Tiled projectors

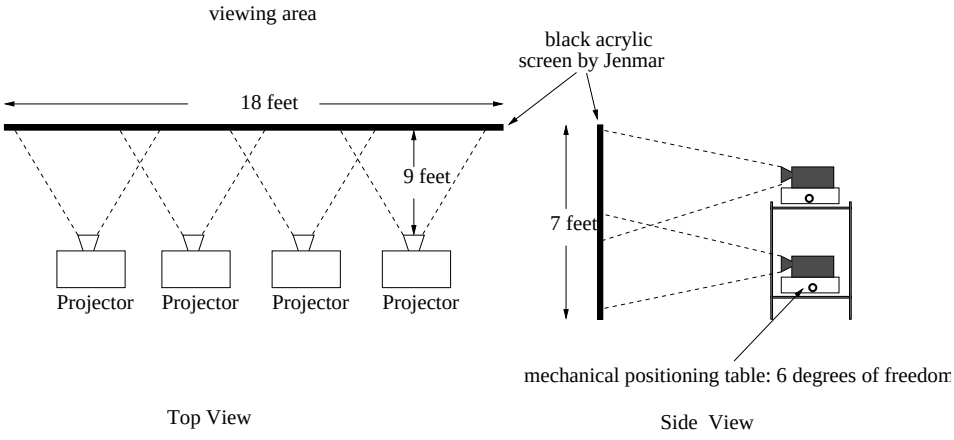


Figure 1.5: The projector arrangement

Figure 1.5 shows the actual arrangement of projectors in our display wall. A 2x4 array of 8 Proxima 9200 LCD projectors are placed behind a rear-projection screen. Two projectors are stacked vertically on a rack. A manual adjustment table, designed by us and manufactured by a division of Argonne National Lab, provides the fine position control required during manual alignment of the projectors.

The display surface is an acrylic screen made by Jenmar Visual Systems. The pitch-black screen offers higher contrast ratio than a grey or white screen. We opted for rear projection so that users can walk up close to, examine details on, and interact with the display without casting shadows. Standing close to an immersive, high-resolution display can create a personal touch that is missing in both traditional cinema environment and on a desktop monitor. This kind of personal feel can significantly enhance the quality of user-computer interaction.

Our of practical reasons, we used projection devices instead of flat panel displays. No viable technology currently exists to manufacture a single-piece, large-area, high-resolution flat panel display. In addition, existing panel displays cannot be seamlessly tiled. Present flat-panel technologies require the drive electronics to be placed on the sides of the panels,

leaving a gap between the viewable areas of adjacent flat panel displays.

At the present time, tiling portable presentation projectors is a viable and relatively economical way to build high-resolution displays. The projectors that we currently use employ transmissive LCD polysilicon as the imaging device. Newer technologies such as digital micro-mirror devices (DMD) from Texas Instrument and reflective LCD from several other companies promise to deliver brighter and crisper images. Brightness of these projectors are typically measured in excess of 800 ANSI lumen.

Tiling projectors, however, is not without its problems. Perfect alignment and color balance must be maintained to avoid seams between projectors. Manual alignment and color adjustment might be possible with just a handful of projectors. But it would be a daunting, if not impossible, task for a very high-resolution display that may have dozens of projectors, for many parameters can affect the image geometry and the colors of a projector. A research challenge here is to devise clever algorithms that use computer vision and computation to replace human labor – to perform automatic alignment and color balancing.

1.4.2 Render cluster

The Scalable Display Wall Project pioneered the clustering approach: multiple PCs, each driving one projector with a commodity graphics accelerator [2], are interconnected by a commodity, high-speed system-area network. Custom hardware, which in the past proved costly and delayed the time to market, is banished from the system. The clustering approach can easily leverage rapid cost-performance improvement in PCs and PC graphics accelerators. Scalability in both total pixel resolution and graphics performance can be achieved by just adding more render PCs, and in some cases more network switches.

Figure 1.6 shows details of our current display wall implementation. A cluster of 8

[2]The projector-to-PC ratio can be configured according to performance-cost consideration. Some graphics accelerators can drive multiple displays. A single PC can support more than one graphics accelerator.

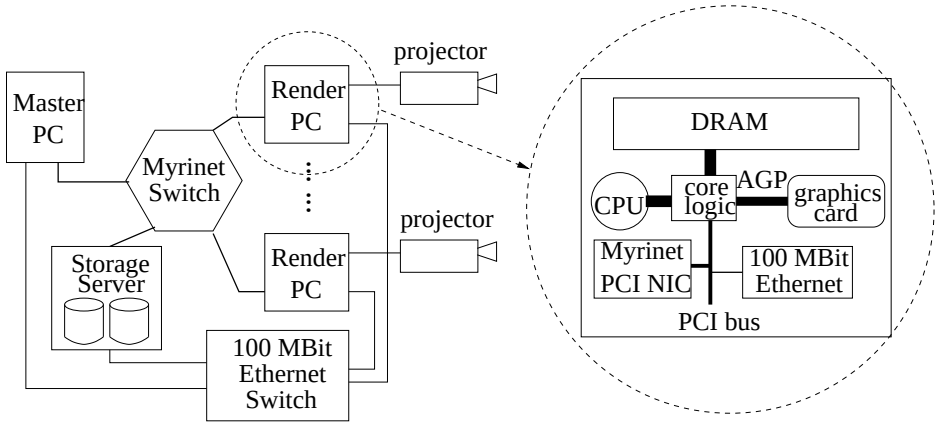


Figure 1.6: Details of the display wall render cluster

Render PCs and a *Master PC* form the nucleus of the display wall computer system. Each render PC is a 450 MHz Pentium II uniprocessor with 1 GB of DRAM and a workstation-class graphics accelerator. In a typical setting, an application runs on the Master PC. The graphics primitives that it generates, for example OpenGL primitives and Windows GDI commands, are intercepted and distributed to the Render PCs that render these primitives into pixels for display.

The PCs are interconnected by two networks, the Myrinet and the Ethernet. The Myrinet system-area network provides high throughput necessary for distributing the primitives. Its cross-bar switch can sustain up to 16 simultaneous data transfers with an aggregate cross-section bandwidth of 2.5 *gigabytes per second*. The PCs in our system currently use Myrinet PCI32 network interfaces to access the network. These NIC cards each have 1 MB SRAM and a 33 MHz RISC processor on board. The data link between a Myrinet NIC and the switch is capable of 160 MB/sec simultaneously in each direction. With next generation of Myrinet network interfaces and switches, the communication throughput available in the hardware will double.

In addition to high bandwidth, low latency in the network is also desirable. Small messages are often used to implement barriers and synchronized clocks in the system,

which are useful for synchronizing buffer swapping among multiple graphics accelerators.

The Ethernet serves as a convenient medium for communication tasks that do not require high bandwidth and low latency, for example, file sharing and computer management.

In order to make the best use of the performance provided by the hardware, an intelligent software layer must be devised that introduces as little overhead into the communication path as possible, and at the same time provides a convenient abstraction to developers. It is a non-trivial task to design a communication subsystem that meet both goals.

1.4.3 Runtime environments

Unlike a uni-processor with multiple graphics cards and a scalable multi-processor with multiple graphics channels, a cluster consists of loose-coupled and relatively independent PC units. Communication among them is usually message passing in nature. This kind of architecture poses an interesting challenge: how to bring a range of applications onto the display wall? Our experience shows that applications for a scalable display roughly fall into three categories: content playback, desktop applications, and custom applications.

Content playback A fair amount of graphical contents simply require play-back software. There are many such content types: static images, image-based animation, MPEG movies, digital TV broadcast, Macromedia FLASH animations, and VRML animations, to name a few. They usually have well-defined specifications. One can develop efficient play-back software parse and render the content in parallel within the display cluster. A good example is the parallel MPEG decoder that our project developed to decode live HDTV streams [55]. A splitter program running on the master PC splits the live MPEG content into sub-streams at the macro-block level, one for each display tile. The sub-streams are sent to respective render PCs to be decoded and rendered.

Desktop applications Many desktop applications such as CAD design and interactive 3D games do not render themselves to the play-back approach. These applications are typically interactive; the content changes every time depending on user interaction. Most of them are highly complex. Besides, source code for these applications is typical inaccessible. Developing play-back software requires tremendous amount of work and is often not worthwhile.

A feasible way to bring up these applications is to build middle-ware into the OS, either as a dynamically-linked library or as a device driver, that intercepts and distributes the graphics primitives generated by the application to the render PCs. We took this approach to bring OpenGL-based animation programs onto our display wall.

Custom applications One can develop applications specifically for the cluster environment of the display wall system. Performance is the primary objective for custom application: this allows maximum use of the processing power on the render nodes; only those graphics primitives that fall within the display tile are generated and sent to the accelerator via local graphics-memory bus. When running this kind of custom applications, a cluster-based display system closely resembles the ideal architecture.

The choice of a software model depends not only on the nature of the application, but also on the balance between the communication throughput in the network and computation resources available on the render nodes.

1.5 Thesis Outline

Research issues in a scalable display architecture generally fall into two categories: those that are common to all three architectures and those that are specific to a particular architecture. In many cases, the research issues have been studied in other contexts. But

the scalable display system, especially the one based on clustering, offers new tradeoffs between various components in the system, and makes old research questions interesting and relevant. Two examples of common research issues are primitive distribution and load balancing. The graphics primitives, once generated, must be distributed to graphics accelerators. If sorting and culling can be done prior to distribution, less burden will be put on the distribution network. However, sorting the primitives with processors may become a bottleneck. Achieving proper balance between sorting and distribution is an interesting research topic.

Since multiple graphics accelerators are used to render tiled images, a natural question comes up regarding load balance among the accelerators. Optimal partition of the scene data to achieve evenly balanced loads is one of the many research questions. Another is pixel redistribution amongst the graphics accelerators, since the data partitioning often does not follow the display tile boundaries.

Several research issues are specific for the clustering environment. A unique aspect of a cluster system is the wide performance disparity between local memory access and intra-cluster communication. The communication is often necessary to hold the render nodes together. The question is when the communication becomes a bottleneck in the system, and if it does, whether clever data partition algorithms can be developed to mitigate its adverse effects.

This dissertation will explore the follow three specific issues in building a cluster-based scalable displaywall system.

Seamless tiling In a multi-projector display, physical imperfections such as misalignment, color imbalance, and distortions produce annoying seams on the display surface. In some cases such as color imbalance the seams are generally cosmetic; but in the cases like misalignment, the seams mis-represent information. One goal of my research is to develop

computational methods to avoid seams on a multi-projector display. The results of this research will be described in Chapter 2.

Scalable networking Since the interconnection network is a key to the performance of a rendering cluster, it is of importance to achieve the best possible performance provided by the networking hardware and at the same time make network programming as easy as possible. Direct data transfers between virtual memory address spaces is a lean and convenient communication paradigm. An implementation of VMMC is used throughout our displaywall system. Chapter 3 describes the VMMC concept and a key research issue in VMMC: address translation.

Runtime environments Clustering has a potential for high performance and scalability. However, it also poses an interesting question: how to run applications with high resolution on a cluster? Chapter 4 surveys techniques for running desktop applications on a cluster displaywall system, and introduces a novel mechanism, synchronized programming execution, that can eliminate network bottlenecks in a cluster environment.

Chapter 2

Seamless Tiling

The two key challenging problems in building a seamless, multi-projector display are (1) achieving pixel-level image alignment across overlapped regions and (2) achieving color uniformity across projectors. In this chapter, I will describe a novel method to automatically align a multi-projector display and an automatic method to balance the colors among the projectors.

2.1 Introduction

Tiling multiple projectors together is a viable way to build a bright, high-resolution, seamless display. But, as such a display system scales beyond three or four projectors, aligning the projectors becomes a challenging issue. A multi-projector display might in principle be perfectly aligned by manual means just once; but in practice physical realities (vibration, lamp-changing, and so on) mean that re-alignment is frequently needed. Aligning the projectors by hand is a time-consuming task that requires both skill and experience. It also requires the use of either sophisticated mechanical devices or electronic beam adjustment found only on expensive CRT projectors.

Alternatively one can employ computer vision and image processing techniques to digitally align the projectors. By warping the images, the computer can correct the projected imagery to account for the physical realities of misalignment [33, 86, 87]. In order to apply the correct amount of digital compensation, the computer has to first measure accurately the actual misalignment.

The research challenge here is to devise efficient algorithms without using expensive cameras and instruments. One could in theory use a camera with extremely high resolution to measure the misalignment. Unfortunately, such a device is hard to get and can cost even more than the display itself. Commodity video cameras are inexpensive and available everywhere, but they suffer from low resolution. Although calibrating the cameras overcomes this problem, the calibration process itself involves human labor and could become time-consuming.

We here describe a novel algorithm that uses an inexpensive and uncalibrated camera to align the projectors. Our method avoids camera calibration by taking only relative measurements — for instance, matching a few points and lines between a pair of projectors. Since the camera only has to make "binary" decisions regarding these measurements, it is free to zoom and pan arbitrarily close to the target spot to obtain highly accurate observations. Our alignment algorithm employs the simulated annealing technique to "stitch" the local observations together into a self-consistent global picture, and find a set of projection mappings that are consistent with the observations.

2.2 Background and Previous Work

Multi-projector displays have been around for more than two decades [59]. Previous systems employed expensive CRT projectors which have sophisticated mechanisms for adjusting image distortion and color balance. Aligning the projectors was typically done

manually by trained technicians; the process often took hours. In recent years, portable presentation projectors based on LCD or Digital Micro-mirror Devices (DMD) have become increasingly cheaper, brighter, and more compact. There is new interest in building high-resolution displays out of these commodity projectors for office and research environments and for entertainment [74, 55, 79, 93, 94]. This has spurred several research projects that study seamless tiling of inexpensive LCD and DMD projectors [86, 33, 58, 47].

Existing alignment algorithms usually consist of two stages, camera calibration and geometric registration. In the first stage, one or more cameras are calibrated according to a fixed global coordinate system (either 2-dimensional or 3-dimensional). In the second stage, the calibrated cameras serve as measurement instruments to map pixels from each projector to the points in the global coordinate system.

Surati and Knight developed an algorithm that uses a camera to map the pixels from each projector to the points in a pre-established global screen coordinate system [86]. During the calibration stage, a camera is calibrated against a fine grid affixed to the display surface. The grid is physically drawn by a high-precision plotter. A mapping is established between pixels in the camera field and the physical points on the display surface. In the registration stage, each projector projects a regular grid onto the display surface. A computer vision algorithm accurately locates each projected grid point in the camera's field of view. Using the camera-to-display-surface mapping established previously, the projected grid point (or a pixel in the projector) is mapped to a physical point on the display surface with high precision. This method works well for a small-scale display wall. It can deal with arbitrary distortions of the projectors. However, using an absolute measurement grid to calibrate the camera prevents this method from scaling for a large display wall; it is problematic whether one can generate a physical or project a virtual measurement grid that is large enough but still has fine precision.

Raskar *et al.* attempted to solve a general case in which the display surface can be

arbitrarily complex, for example, the corner of a wall, or a curved screen [33]. This requires registration of the 3D surface geometry of the screen surface as well as registration of the projected pixels on the display surface. Their algorithm uses known 3D objects such as painted boxes to calibrate the extrinsic and intrinsic parameters of a few fixed cameras. Two calibrated cameras that overlap in their fields of view can observe the same mesh pattern displayed by a projector. The observations from both cameras are correlated using the stereo vision technique; from this correlation the exact location of a projected pixel on the display surface is derived. The location information is in the 3D space and therefore also reveals the surface contour of the screen. Again requiring camera calibration is the drawback of this approach.

Our work differs from previous work by the use of an uncalibrated camera to observe misalignment among the projectors. The use of camera calibration implies that the cameras themselves must be fixed and cannot pan and zoom during measurement, for otherwise camera parameters will have to calibrated continuously. It also requires setting up known objects such as as fine-plotted grid and regular 3D objects. Avoiding camera calibration can greatly minimize the amount of human involvement and equipment required in multi-projector alignment.

2.3 Projection function

Before getting into the details of our alignment algorithm, let us first review the mathematical representation for a multi-projector system. Projection can be thought of as a mapping between pixels in projector space (x, y) and the illuminated dots (u, v) on the global display space. This mapping, or the *projection function*, is normally accomplished with a lens system.[1] Figure 2.1 shows a conceptual diagram of a typical lens system. The projec-

[1]The algorithm presented here could in principle be applied to curved display surfaces as well, in which case, a 2D parametric coordinate system can be used o the display surface.

tion function can be decomposed into two parts, the projective transformation P and the non-linear distortion D:[2]

$$(u, v) = (D_u(P_u(x, y), P_v(x, y)), D_v(P_u(x, y), P_v(x, y)))$$

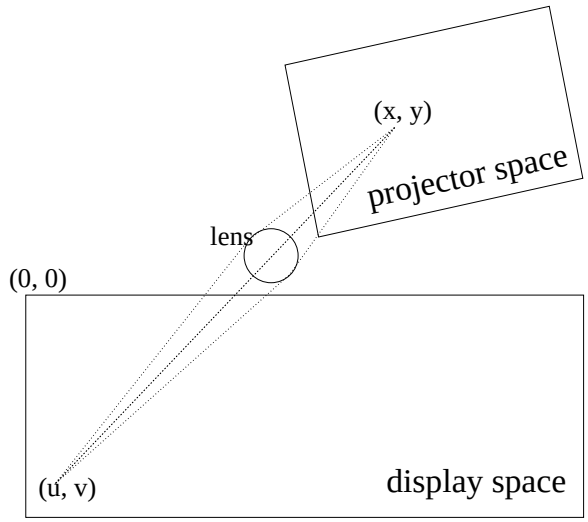


Figure 2.1: Conceptual diagram of a projection system

The projective transformation can be expressed by a 3x3 projection transformation in homogeneous coordinates using 8 free parameters $(m_{ij}),\ i = 1..3,\ j = 1..3$

$$\begin{aligned} P_u(x, y) &= \frac{m_{11} \cdot x + m_{12} \cdot y + m_{13}}{m_{31} \cdot x + m_{32} \cdot y + 1} \\ P_v(x, y) &= \frac{m_{21} \cdot x + m_{22} \cdot y + m_{23}}{m_{31} \cdot x + m_{32} \cdot y + 1} \end{aligned} \tag{2.1}$$

The proof can be found at the end of this section.

The radial distortion with respect to an optical center (c_x, c_y) and a distortion parameter

[2]This is only an approximation to an actual projection device, as it ignores several distortion effects such as color dispersion.

ρ is given as

$$\begin{aligned} D_u(u,v) &= u + \rho \cdot (u - c_u) \cdot d^2 \\ D_v(u,v) &= v + \rho \cdot (v - c_v) \cdot d^2 \end{aligned}$$

$$\begin{aligned} d &= \sqrt{(u - c_u)^2 + (v - c_v)^2} \\ (u,v) &= (P_u(x,y), P_v(x,y)) \\ (c_u, c_v) &= (P_u(c_x, c_y), P_v(c_x, c_y)) \end{aligned} \tag{2.2}$$

The implementation described in the paper only considers the projective component. However, the algorithm itself can in principle deal with non-linear distortions as well.

2.3.1 Proof of the 3x3 projective representation

We establish a global 3-dimensional coordinate system with its X-Y plane coincide with the screen surface ($SPACE_s$), so that a point on the screen is $(x_s, y_s, 0)$. A pixel (x_p, y_p) in $SPACE_p$ is converted to the global coordinate (x', y', z') using following formula for coordinate system transformation:

$$p \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = M' \cdot \begin{bmatrix} x_p \\ y_p \\ 0 \end{bmatrix} + \begin{bmatrix} c_x \\ c_y \\ c_z \end{bmatrix} \tag{2.3}$$

$$where, M' = R_x \cdot R_y \cdot R_z \cdot S_x \cdot S_y \tag{2.4}$$

R_x, R_y, and R_z are standard rotation matrices. S_x and S_y are standard scaling matrices that convert the pixel units on the image plane into proper units in the global coordinate system $SPACE_s$. We have separate scaling factors for X and Y axes because the pixels on the

image plane need not be perfect squares. The vector $[c_x, c_y, c_z]^T$ is the translation from the origin of the global coordinate system (or, original on the screen) to the origin on the image plane which is pixel (0, 0).

The definition of an ideal lens system defines an optical center which acts just like the pinhole in a pinhole camera. Therefore, the three points, (x', y', z'), its image on the screen $(x_s, y_s, 0)$, and the optical center (x_o, y_o, z_o) are collinear and satisfy the following parametric line equation

$$\begin{aligned} x_s &= x_o + (x' - x_o) \cdot \lambda \\ y_s &= y_o + (y' - y_o) \cdot \lambda \\ z_s = 0 &= z_o + (z' - z_o) \cdot \lambda \end{aligned}$$

Substituting λ in the first two equations we get

$$\begin{aligned} x_s &= \frac{x_o \cdot z' - x \cdot z_o}{z' - z_o} \\ y_s &= \frac{y_o \cdot z' - y \cdot z_o}{z' - z_o} \end{aligned} \tag{2.5}$$

According Equation 2.3, x', y', and z' are each represented by an expression linear in x_p and y_p. Substituting x', y', and z' result in something like

$$\begin{aligned} x_s &= \frac{m_{11} \cdot x_p + m_{12} \cdot y_p + m_{13}}{Z} \\ y_s &= \frac{m_{21} \cdot x_p + m_{22} \cdot y_p + m_{23}}{Z} \\ Z &= m_{31} \cdot x_p + m_{32} \cdot y_p + m_{33} \end{aligned} \tag{2.6}$$

One can trivially show that m_{33} is non-zero, because otherwise a solution does not exist for pixel (0, 0). Therefore, we can set m_{33} to 1 without affecting the values of x_s and y_s. The

system of equations 2.6 exactly expresses a perspective transformation.

Note, that Equation 2.3 are general. They describe an image plane in the projector that can tilt at an arbitrary angle to the optical axis of the lens system. Perfect alignment of the image plane such as LCD to the optical axis is not possible due both to manufacturing difficulties and to the zoom capabilities of the lens system.The aforementioned perspective transformation captures even this non-perfection.

2.4 The alignment algorithm

Our automatic alignment algorithm consists of two stages. In the *misalignment measurement* stage, the camera observes geometric relationships – point matches and line matches – between adjacent projectors. In the *alignment computation* stage, we set up a multi-dimensional global optimization problem whose constraints are those observations. The optimization process produces a set of alignment correction functions, one for each projector, that maintain G^0 and C^1 *continuities* across the projectors. G^0 geometric continuity means point-wise continuity. C^1 implies that two curves from adjacent screens match in their tangent vectors [34]. Figure 2.2 shows a schematic of our alignment system. Our algorithm assumes that projectors are already roughly aligned. This is a reasonable assumption, because coarse alignment can be easily accomplished with an inexpensive projector rack and some manual adjustment. Essentially we assume that the projectors are not so badly misaligned that computational alignment is impossible.

2.4.1 Alignment measurement

The alignment algorithm makes use of geometric relationships between adjacent projectors to figure out their projection functions. The hope is that by gathering local information regarding how each projector misaligns with respect to its neighbors, we can deduce the

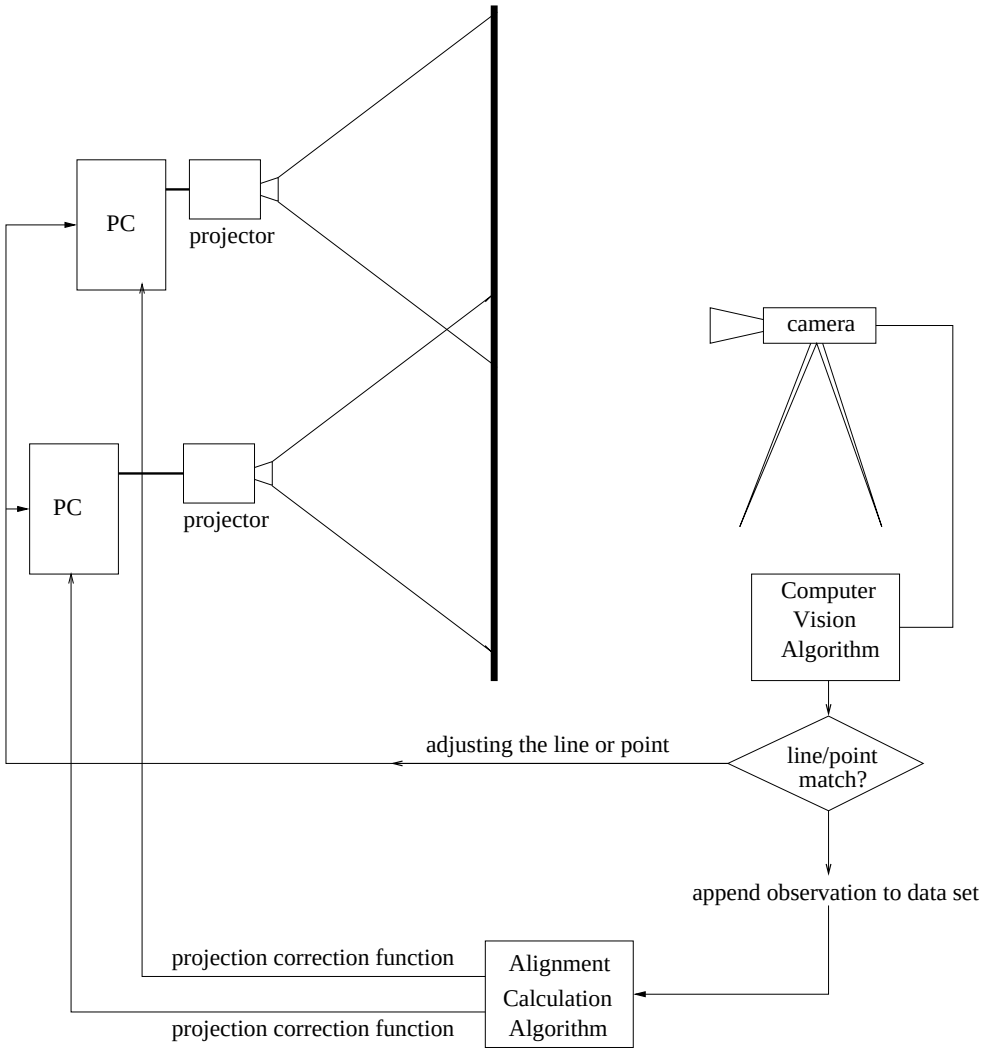


Figure 2.2: Camera-based Alignment Data Collection

actual projection functions, one for each projector, that are globally consistent with each other.

Our algorithm relies on two types of inter-projector relationships: point matches and line matches. A point match simply states that a pixel (x, y) from one projector locates at the same spot on the display surface as another pixel (x', y') from an adjacent projector.[3] A line match means that a projected line from one projector is collinear with another line from the neighboring projector.

[3]Note here that we use the fact that adjacent projectors overlap by a small amount.

The rationale for using point matches is simple: if there are a lot of point matches between any pair of adjacent projectors, the result of alignment computation will yield a set of projection functions that maintain G^0 continuity across projectors. However, the point matches themselves are insufficient to constrain the system. This is particularly so when the projectors overlap only a small portion of their screens, as in a typical display wall system. The line matches provide further shape constraints. The point and line matches together guarantee C^1 continuity across the projectors. Given a sufficient number of point and line matches, there is enough information to figure out the relative position and the orientation of the projectors.

The line and point matches can be obtained automatically with a camera attached to a computer. Figure 2.3 contains a brief sketch of the measurement algorithm that obtains a point match between point P from projector A and a point Q from projector B. It is essentially a negative feedback loop. The feedback parameter ρ in the algorithm is determined before hand either manually, or automatically by measuring the distance between two pixels in the camera. The algorithm to obtain line matches works in a similar fashion. Each line match consists of a point match between the inner ends of the two line segments and a match between the slopes of the two line segments. More specifically, the first line segment is displayed from pixel P_{1A} to pixel P_{2A} by projector A; the second line segment from P_{1B} to P_{2B} by projector B. The two line segments are said to be matched when (a) the have the same slope in the camera's field of view, and (b) (P_{2A}, P_{1B}) is a point match.

2.4.2 Alignment computation

Given the point and line matches that we gather from the measurement process, our next task is to figure out a set of projection functions that fit the observed matches. Symbolically speaking, we want to find a set of projection functions $P^i = (P^i_u, P^i_v)$, where i represents the ith projector, such that all observed point matches are line matches are satisfied with

```
1. pan the camera to roughly center its FOV on pixel P
2. display a cross centered at P on projector A
3. measure P's location L in the camera
4. take a guess of a pixel Q in projector B
5. loop
6.      display a cross centered at Q on projector B
7.      measure Q's location L' in the camera
8.      if ||L, L'|| > ε
9.         Q = Q + ρ · (L − L')
10.     else
11.        return (P, Q)
```

Figure 2.3: pseudo-code for obtaining a point match

respect to this set of functions.

More specifically, for a point match between pixel (x_i, y_i) from the ith projector and pixel (x_j, y_j) from the jth projector, let (u_i, v_i) represent the projected image of pixel (x_i, y_i), or $(u_i, v_i) = (P_u^i(x_i, y_i), P_v^i(x_i, y_i))$, we wish to maintain the following equality

$$\begin{aligned} u_i &= u_j \\ v_j &= v_j \end{aligned}$$

In reality, due to measurement errors and inaccuracy of our model, equality is difficult to achieve for all point matches. Therefore, instead of equalities, we can measure the goodness of our projection functions by errors. For a point match, the error can be expressed in terms of the Euclidean distance between the two *imagined* pixels on the display surface

$$E^p(p_i, p_j) = (u_i - u_j)^2 + (v_i - v_j)^2 \tag{2.7}$$

Similarly, each line match between two line segments $l_i = \overline{p_{11}, p_{12}}$ and $l_i = \overline{p_{21}, p_{22}}$ pro-

duces a point-match error and an error based on the angle between two line segments:

$$E^l(l_1, l_2) \quad = \quad \max(E^p(p_{12}, p_{21}), (\angle\overline{p_{11}p_{12}}, \overline{p_{21}p_{22}})^2)$$

Note that the error term $(\angle\overline{p_{11}p_{12}}, \overline{p_{21}p_{22}})^2)$ is also computed in the global display space.

With above formulation, we have turned the problem of figuring out a set of projection functions into a global optimization problem – finding a set of projection functions that minimize the errors computed from the point and line matches. The goal is to minimize the maximum of the errors over all line and point matches. Note that it is straightforward to obtain an initial guess for each projector's position shifts, horizontal and vertical, based on the point match data. Since the projectors are already roughly aligned, this guess produces a good starting point that is close to the globally optimal solution for our problem.

Global optimization over a large number of continuous variables remains a tough problem. However, several effective methods do exist. We chose Simulated Annealing [60, 48, 70] as the optimization method. It has been used successfully in many scientific computations with hundreds and even thousands of continuous variables. This technique is a generalization of a Monte Carlo method for examining the equations of state and frozen states of n-body systems [60]. It mimics the manner in which metals recrystallize in the process of annealing. Among several publicly available implementations, we chose the one provided by *Numerical Recipes in C* [70]. The state evaluation function required by the annealing method is the error function that we just described.

The simulated-annealing method requires representing each projection functions using a vector of continuous variables. During our initial trials, we found it was difficult to constrain relative scales among the matrix parameters in order to produce a realistic projection geometry. In particular, shear deformation, which only occurs when the optical axis is far off from the center of the imaging (LCD) plane, was often produced by simulated anneal-

ing. Therefore, instead of trying to compute the 8 free parameters in the projection matrix (Equation 2.1), our optimization routines computes the extrinsic and intrinsic parameters of each projector, because their relative scales can be easily set *a priori*. A projector can be modeled with 9 parameters, the rotations, (r_x, r_y, r_z), of the projector around its optical center, translations of the optical center, (t_x, t_y, t_z), in the global coordinate system, the focal length f, and the center of the projector space (c_x, c_y). Given these parameters, one can write down a projective matrix as follows:

$$
\begin{aligned}
m'_{11} &= t_x \cdot r_{31} - t_z \cdot r_{11} \ , \quad m'_{12} = t_x \cdot r_{32} - t_z \cdot r_{12} \\
m'_{13} &= t_x \cdot r_{33} - t_z \cdot r_{13}) \cdot f - \ (t_x \cdot r_{31} - t_z \cdot r_{11}) \cdot c_x \\
&\quad - (t_x \cdot r_{32} - t_z \cdot r_{12}) \cdot c_y \\
m'_{21} &= (t_y \cdot r_{31} - t_z \cdot r_{21} \ , \quad m'_{22} = t_y \cdot r_{32} - t_z \cdot r_{22} \\
m'_{23} &= (t_y \cdot r_{33} - t_z \cdot r_{23}) \cdot f - \ (t_y \cdot r_{31} - t_z \cdot r_{21}) \cdot c_x \\
&\quad - (t_y \cdot r_{32} - t_z \cdot r_{22}) \cdot c_y \\
m'_{31} &= r_{31} \ , \quad m'_{32} = r_{32} \\
m'_{33} &= r_{33} \cdot f - r_{31} \cdot c_x - r_{32} \cdot c_y \\
\\
m_{ij} &= m'_{ij} / m'_{33}
\end{aligned}
\tag{2.8}
$$

Where, r_{ij} is the element of a 3x3 rotational matrix R that represents rotations r_x, r_y, and r_z.

Given N projectors in the display wall system, alignment computation amounts to minimizing the error function over $9N$ continuous variables (or $10N$ if radial distortion is also included). The total degree of freedom in this problem is quite reasonable for the simulated annealing method, as our experiments will shortly show.

2.4.3 Computational re-alignment

Having the mapping function for a projector, we can apply it to correct the imagery displayed by that projector, simply by re-sampling the image. Given a projector's mapping function (P_u, P_v), and an image source $I_s(u, v) = (r, g, b)$, we obtain the intensity value I_p for a particular pixel (x, y) using the formula

$$I_p(x, y) = I_s(P_u(x, y), P_v(x, y)) \tag{2.9}$$

Many studies have been done on efficient re-sampling of an image. One can use the MMX instructions on a Pentium processor to sample multiple pixels at once. Another interesting approach is to leverage the capability of graphics accelerators. Raskar *et al* described a method using the texture-mapping hardware found on most graphics accelerators [73, 72]. Recently ComView Visual Systems has introduced an ASIC solution that provides both geometric correction and color balancing for multi-projector display systems [87].

2.4.4 Discussion

A salient feature of the alignment algorithm just described is that it avoids camera calibration. No human involvement is required to take misalignment measurement other than placing the camera(s) in front of the display wall. This feature is made possible by taking only relative measurements, *i.e.*, point and line matches. Such observations require only local and "binary" decisions that any inexpensive camera will do.

Measuring only the point and line matches also makes it easy to overcome the resolution limitation of an off-the-shelf camera. One can simply pan and zoom the camera arbitrarily close to the display surface, or place multiple fifty-dollar cameras close to the display surface. Highly accurate measurements, finer than a pixel, are easily obtained this way. Unlike methods based on camera calibration, this new method is insensitive to change of

camera parameters during the zoom and pan motions, and can easily employ many cameras at no additional complexity.

Our alignment algorithm is much less sensitive to camera distortions than previous methods. For example, it can tolerate any kind of camera distortion while taking point matches, provided that the camera remains steady during measurement. The only additional requirement for obtaining a line match is that camera's field of view (FOV) is free from non-linear distortions – as long as a straight line on the display surface shows up straight in the camera's FOV.[4] A naive way to meet this requirement is to use the central area of the FOV. A sophisticated solution is to center the FOV on the adjoining ends of the two matching line segments, such that the line segments pass the optical center of the camera's FOV. This makes the entire camera's FOV usable even in the presence of radial distortion, because a straight line passing through the center of a camera FOV is not bent by radial distortion.

The drawback of our algorithm is that it relies on a global optimization technique that gives no convergence guarantee. Although the non-convergence situation has not occurred in our experiments, we remain interested in finding a deterministic and more efficient method to calculate the projection functions.

2.5 Implementation and Results

In this section we evaluate the effectiveness of our alignment algorithm with empirical results as well as a simulation study.

Experimental platform

We conducted experiments on our 2x4 prototype display wall. It consists of 8 Proxima 9200 LCD portable projectors in a 2x4 arrangement [55]. Each projector is capable of

[4] This is less stringent than requiring the camera FOV be a linear field.

displaying a true resolution of 1024x768. Adjacent projectors overlap between 40 and 70 pixels. The effective resolution on the display wall comes out to about 3800 pixels wide and 1500 pixels high.

The projectors are mounted on mechanical positioning tables that have 6 degrees of freedom. These tables are normally used for time-consuming manual alignment of the projectors. But in our experiments, they provide us with an easy means to "mis-align" the projectors with arbitrary rotations and translations.

We place a Canon VC-3 video conferencing camera at an 8-feet distance away in front of the display wall. This camera has motorized pan, tilt, zoom and focus, all controllable through the serial port. We wrote our software in Python and C that controls the camera to gather misalignment observations. Video digitization is done by an Integral Video Grabber card.

2.5.1 Empirical results

Misalignment measurement: On our 8-projector display wall, it takes 33 minutes to collect the point and line matches over a total of 10 overlapped regions. For each pair of adjacent projectors, 10 point matches and 6 line matches are observed. A large amount of time is spent in panning and tilting the camera to zoom onto a spot. Using multiple cameras, each responsible for a sub-area of the display wall, can reduce the measurement time proportionally. Besides, there is no need to correlate observations from different cameras.

The quality of alignment computation depends critically on the accuracy of the point and line matches. In our experimental setup, the camera can easily distinguish two adjacent pixels from a projector. We use nearest-neighbor fit to match two pixels and two lines. This implies that the worse measurement error for a point match is a half pixel. A more sophisticated algorithm such as weighted average could be used to increase the measurement accuracy.

configuration	annealing steps									
	1,000		2,000		5,000		10,000		20,000	
	error (pixels)	time (min)	error (pixels)	time (min)	error (pixels)	time (min)	error (pixels)	time (min)	error (pixels)	time (min)
1x4	0.96	1.2	0.76	2.4	0.77	6.1	0.89	11.9	1.19	24.8
2x2	1.25	1.5	1.38	3.0	1.32	7.8	1.14	15.4	1.10	30.9
2x3	1.27	1.7	1.27	3.5	1.27	8.7	1.42	17.3	1.12	34.6
2x4	1.49	1.8	1.44	3.5	1.32	8.8	1.50	18.1	1.35	35.7

Table 2.1: Alignment accuracy and time for various configurations

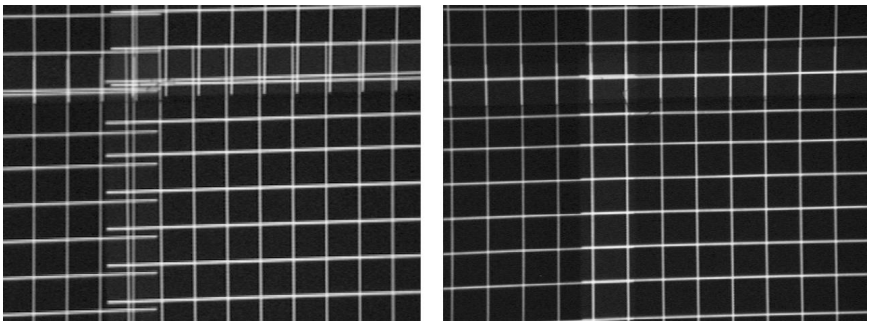


Figure 2.4: Aligning a grid: before and after pictures

Alignment computation: We run the alignment computation algorithm on an 833 MHz Pentium III PC with Rambus memory. Table 2.1 shows the time and quality in the alignment computation, as the number of annealing steps increases, for various projector configurations on our display wall. The quality is expressed in terms of the maximum error between two points in a point match (in pixels). The pixel-level error can be largely attributed to the error in the measurement. Figure 2.4 shows the zoomed-in view of actual alignment result of a grid pattern on our 2x4 multi-projector configuration. The readers are referred to our paper on automatic alignment [18] for more detailed results.

The quality of simulated annealing depends on the number of steps in the annealing process. The more steps taken, the more gradual the annealing process is and usually the better the alignment result. Figure 2.5 plots the alignment accuracy as a function of total annealing time for a few projector configurations on our display wall. The improvement of accuracy is very gradual, as the number of annealing steps increases; in a few cases, the

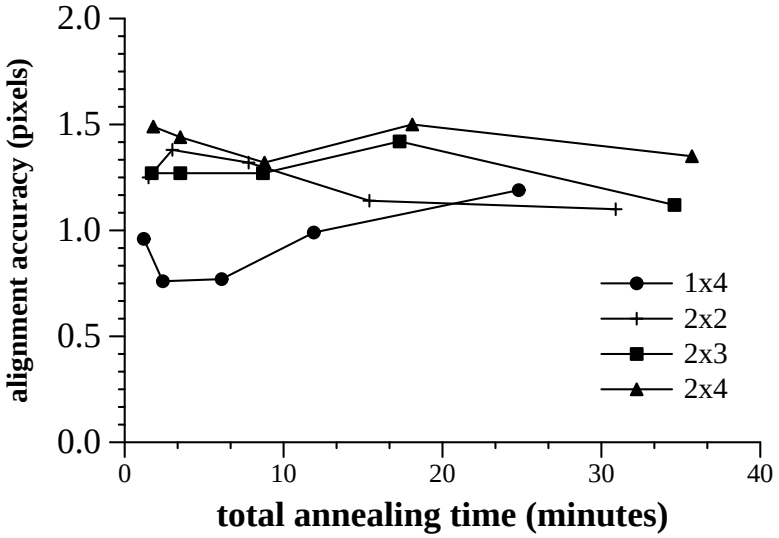


Figure 2.5: Alignment computation accuracy vs. total annealing time

accuracy actually worsens. The very slow improvement could very well be the nature of the simulated annealing technique. The measurement errors and non-linear distortions in the projectors also contribute to the final errors in the alignment computation. Section 2.5.2 confirms this with error-free measurements from simulated display wall configurations.

2.5.2 Simulation results

In order to evaluate the scalability of our alignment algorithm, we wrote a simple simulator that generates misalignment observations for an arbitrarily configured multi-projector display. The simulation assumes that after rough adjustment of the projectors, imperfect position and orientation of a projector (total 6 degrees of freedom) contribute 10 pixels of misalignment, independently. The variation in zoom distance and focal length is 5 %. In other words, our simulation assumes that the projectors are roughly aligned – a quite realistic assumption based on our experience. It is these 10-pixel variations that our automatic alignment algorithm tries to eliminate. Table 2.2 shows the quality of alignment computation for various simulated configurations.

The simulation results confirm that our algorithm can deal with a variety of projector

configuration	annealing steps											
	1,000		2,000		5,000		10,000		20,000		50,000	
	error (pixels)	time (min)	error (pixels)	time (min)	error (pixels)	time (min)	error (pixels)	time (min)	error (pixels)	time (min)	error (pixels)	time (min)
1x4	0.47	1.2	0.63	2.3	0.88	5.8	1.23	11.4	0.38	23.3	0.78	59.6
1x8	0.38	1.2	0.43	2.5	0.53	6.2	0.57	12.6	0.74	25.2	0.62	63.3
2x2	0.59	1.5	0.74	2.9	1.07	7.4	0.73	15.2	1.11	32.1	0.95	78.2
2x4	0.55	1.7	0.42	3.4	0.60	8.5	0.52	17.2	0.35	34.5	0.61	86.7
3x3	1.53	1.8	0.52	3.6	0.50	8.9	1.29	17.9	0.43	36.1	0.66	91.2
3x6	3.98	1.8	2.32	3.7	1.35	9.0	1.52	18.5	0.96	37.3	0.99	94.9
4x6	11.95	1.6	11.83	3.6	6.85	9.3	2.91	18.6	1.00	38.0	0.83	95.5

Table 2.2: Alignment computation results on simulation data

configurations and misalignment situations, for up to 24 projectors, with satisfactory results. Unlike in the experiments, the measurements in the simulation study are precise. The effect of error-free measurements is manifested as generally higher alignment accuracy for the same projector configuration than in the actual experiments.

The alignment computation time that is required to achieve certain quality, i.e., sub-pixel alignment, generally increases with the number of projectors in the system. Half an hour is sufficient for getting a sub-pixel alignment result on configurations varying from 1x4 to 4x6. For a very large system with 50 or 100+ projectors, the annealing will certainly take hours to achieve alignment at the single pixel level. This is the drawback of our algorithm. A possible solution is to parallelize the computation using the PCs that drive the projectors and the fast network that connects these PCs together. The computational resource in our system scales with the number of projectors. But the challenge is to parallelize the annealing algorithm. We are currently investigating this approach.

2.6 Color Balance

This section addresses the problem of balancing the color temperatures of the projectors. The color characteristics of a projector is influenced by many factors; the most significant of them is the color temperature of the lamp. As the lamp ages, its energy distribution over

the spectrum slowly changes, and it is very difficult to bring the lamp temperatures among many projectors into agreement. Thus, when we display a single image tiled across many projectors, the colors of different regions of the image don't match.

The color balance problem can be broken down into two sub-problems: color uniformity and color conformity. Color uniformity means that all projectors have nearly the same color characteristics. Color conformity means that the projector's color spectrum conforms to a standard color spectrum such as specified by the international standards $CIE - XYZ$. We tackle the first sub-problem, color uniformity, because lack of color uniformity among the projectors has the most significant visual impact on the quality of the image.

To achieve color uniformity, we perform *digital color correction* on the image source, for example, lowering the red component of a particular pixel, before it is sent to the projectors for display. In the most general case, this means mapping every (r, g, b) triple in an image to another triple (r', g', b'), for each projector. Given 8 bits for each color channel, the mapping requires a total of 2^{24} entries. However, we can decouple the problem into the separate color channels, so we treat each channel as a one-dimensional (luminance) matching problem. This lookup can be done efficiently if the projector hardware or graphics card supports a separate lookup table (or gamma correction table) for each color channel.

Like the automatic alignment algorithm just described, our color balancing algorithm consists of two steps: luminance measurement and luminance matching.

2.6.1 Luminance measurement

In the first step, we measure the response curve for each color, on each projector. The curve maps a color index value, x to a luminance reading L. The color index x is in the range $[0..2^n]$, where n is the number of bits for each color channel, typically 8 for portable projectors and commodity graphics accelerators.

Precise measurement of a projector's color response curves can be obtained with an

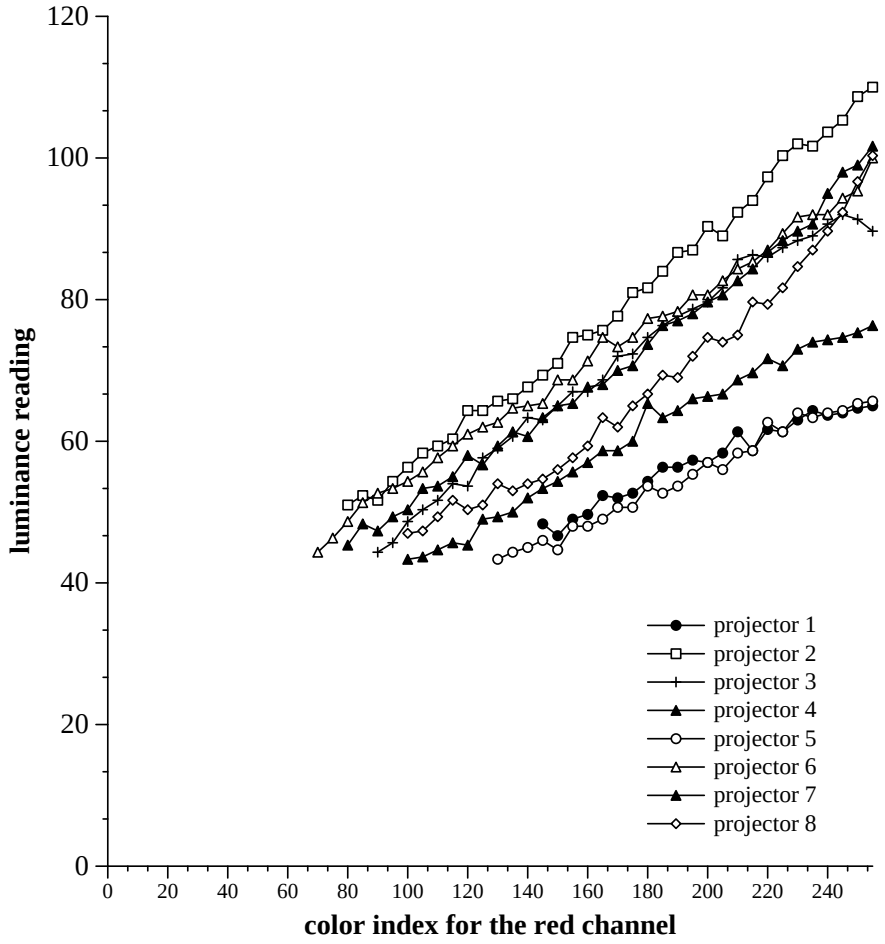


Figure 2.6: Measured color responses for 8 projectors (red channel)

expensive color spectrometer [58]. A challenge that we set upon ourselves is to see whether a video camera can do the job with reasonable accuracy but faster. For this experiment, we used a video-conferencing camera, Canon VC-C3, and fixed its shutter speed and exposure time so that the same luminance on different projectors induces the same reading in the camera.

Figure 2.6 shows the actual red channel responses from 8 projectors as seen by the Canon VC-C3 camera. Responses for the other two channels, green and blue, are similar. Two phenomena in this graph are worth noting. First, the curves are jaggy instead of being smooth. This is due partly to the background fluctuation in the projectors, and partly to

deviations	projector 1	projector 2	projector 3	projector 4	projector 5	projector 6	projector 7	projector 8
red	1.0	1.5	1.9	1.3	1.0	1.2	1.8	3.1
green	3.3	8.3	8.0	5.2	1.5	9.6	11.8	4.5
blue	4.3	7.2	13.1	7.5	3.3	7.6	7.8	4.8

Table 2.3: Accuracy of color-response interpolation: standard deviations

the measurement noise in the camera itself, even though special caution has been taken to smooth out the fluctuation with repeated measurements. Second, due to high background light emission in the projectors and stray ambient light in the room, the camera cannot obtain precise measurement for low color indices.

We apply interpolation to smooth out sampling noises and also to extend the measured curves to entire color index range. The exact method for interpolation depends on the physical properties of the projection devices. However, since most curves obtained in practice are fairly close to being a straight line, we apply linear interpolation on measured data. Table 2.3 gives the standard deviation on color response interpolation.

2.6.2 Luminance matching

Given the measured luminance curves for a single color channel on all projectors, we can find a reference curve, L^c, for each color channel c, that is within the luminance range on all projectors, if possible, so that for a given color index, x, there is a corresponding, adjusted value, x_i, for the i-th projector, such that $L_i^c(x_i) = L^c(x)$. Note that at the lowest luminance end, it is possible to adjust all projectors to match the brightest projector, and at the highest end, adjust all projectors to match the dimmest projector.

There are two conflicting goals here. On the one hand, the reference curve should be calculated so that the adjusted values can be found for all projectors throughout color index range. On the other hand, if we simply take the intersection of projectors' response curves, the reference curve would have smaller dynamic range, and as a result, the corrected images

would appear washed out.

In our current approach, color uniformity takes precedence over wide dynamic range. The algorithm that we use to find the reference curve is based on the criterion that color index adjustment for each projector should be kept at minimum. In other words, we wish to minimize the following expression:

$$\sum_{p \,\epsilon\, all\ projectors} \sum_{i=0}^{255} (x'_p - i)^2$$

Since the problem space is quite small, we used an exhaustive search to find the reference curve.

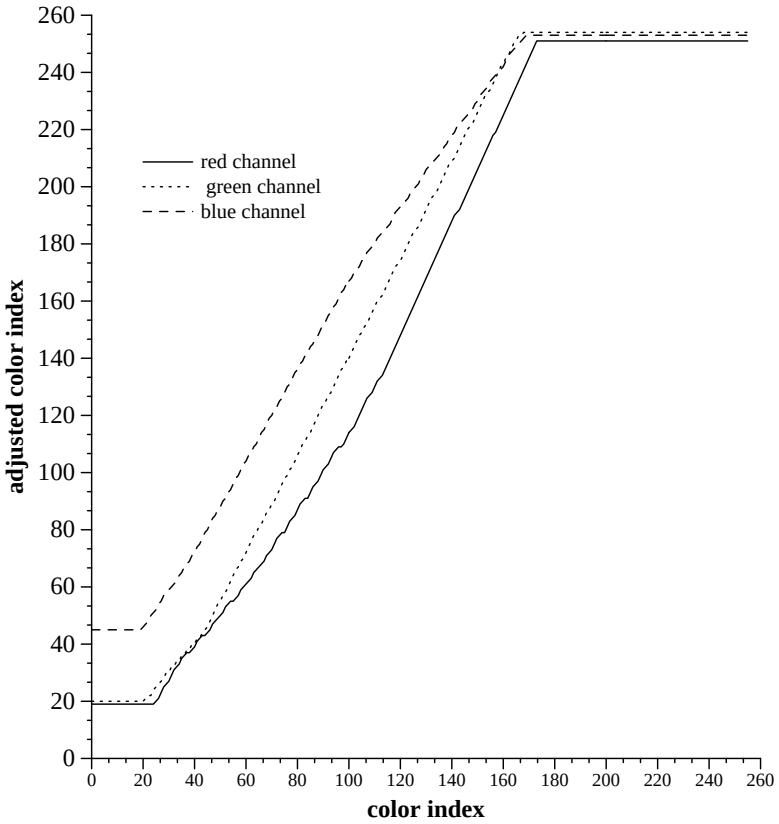


Figure 2.7: An example correction curve for one projector

After deriving a reference luminance curve, we establish a correction function for each color channel on each projector. The correction function is essentially a lookup table T

with 256 entries:

$$T_r(x) = L_r^{-1}(LL_r(x))$$
$$T_g(x) = L_g^{-1}(LL_g(x))$$
$$T_b(x) = L_b^{-1}(LL_b(x)) \tag{2.10}$$

For a given (r, g, b) triple, we will get the same color on all projectors if they each display instead the translated triple $(T_r(r), T_r(g), T_r(b))$.

Figure 2.7 shows the three adjustment curves, one for each color channel, for a projector. Note the flattened head and trail, as a result of shortened dynamic range. The image plate Figure 2.8 shows the final result of color correction.

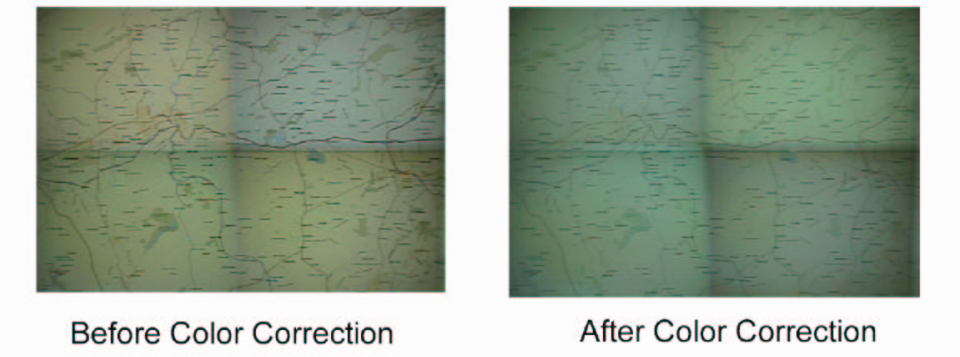


Figure 2.8: Color correction of a map image on four adjacent projectors

2.6.3 Discussion

The result of color balancing is somewhat encouraging. We were able to obtain reasonable luminance measurements for each color channels. Our algorithm to compute digital color compensation seems to work for some images. But our experiment also reveals two problems.

First, we had trouble obtaining smooth color response curves for the entire color index range. We suspect that the major causes for this problem are (1) inherent fluctuations in the projectors and (2) background emission noises in the LCD. We hope both can be solved with better, new projectors.

Second, taking the intersection of color responses produces a reference response curve that has much smaller dynamic ranges in some cases. This problem seems unavoidable, unless we can adjust each projector before hand, to bring their color response range to the same level. This is possible with newer projectors which have digital "knobs" for just such adjustment.

2.7 Conclusions

I have just described an automatic algorithm to align a multi-projector display and an algorithm to color balance the projectors.

The alignment algorithm uses an uncalibrated camera to obtain the inter-projector misalignment information. It then employs a global minimization technique, simulated annealing, to figure out good correction functions to counter the effect of physical misalignment. I implemented and experimented our automatic alignment algorithm for an 8-projector display wall. The experimental results show that our algorithm works well in the real setting. We also evaluated our algorithm using data from simulated multi-projector display systems. The simulation results show that our method works for a system with up to 24 projectors. But for systems with more projectors, the simulated annealing process takes a very long time to converge.

The alignment algorithm takes only relative measurements that require no camera calibration. It solves the inherent tension between the relatively low resolution of a camera and the very high resolution of a scalable display wall, by allowing the camera to freely zoom

in on a measurement target. The result is a highly accurate computational alignment among projectors. In addition, since no camera calibration is required, this algorithm can be fully automated. Although the algorithm currently ignores the radial distortion of a projector, the global minimization technique can be adapted to solve for the radial distortion parameters.

There is some room left to improve the speed of alignment data collection and alignment computation, as we have not yet tried to optimize these processes. Two possible ways to improve our algorithm are (1) using multiple uncalibrated cameras to speed up the data collection process, and (2) parallelizing the annealing computation over the PC cluster and the ultra-fast network that together drive the display wall.

The color-balancing algorithm uses a video camera to automatically measure color imbalances among the projectors and calculate digital color corrections. It works for some images, but suffers from reduced dynamic ranges. Further work is needed to test color measurement accuracy on newer projectors.

Chapter 3

Cluster Communication

3.1 Introduction

Efficient communication is at the heart of a cluster-based display system. High throughput is needed for efficient distribution of graphics primitives throughout the cluster. It is often the case that the underlying hardware offers excellent performance, yet layers of software add much overhead and cause excessive loss in performance. Most software overhead can be attributed to multiple data copying, which cuts down the throughput, and OS invocations, which increase end-to-end latencies.

To address this problem, an intelligent software layer must be developed that introduces minimum overhead into the communication path, and at the same time provides a convenient abstraction to the application layer. Virtual Memory Mapped Communication, or VMMC for short, is a communication model designed to meet both goals. Proposed and first implemented by Blumrich *el al* [9, 7], VMMC provides protected, direct data transfer between the virtual address spaces of two applications. Two kinds of virtual memory buffers are used for data transfer: the *receive buffer* and the *send buffer*. The receive buffer is made visible to remote applications through an *export* system call. A remote application

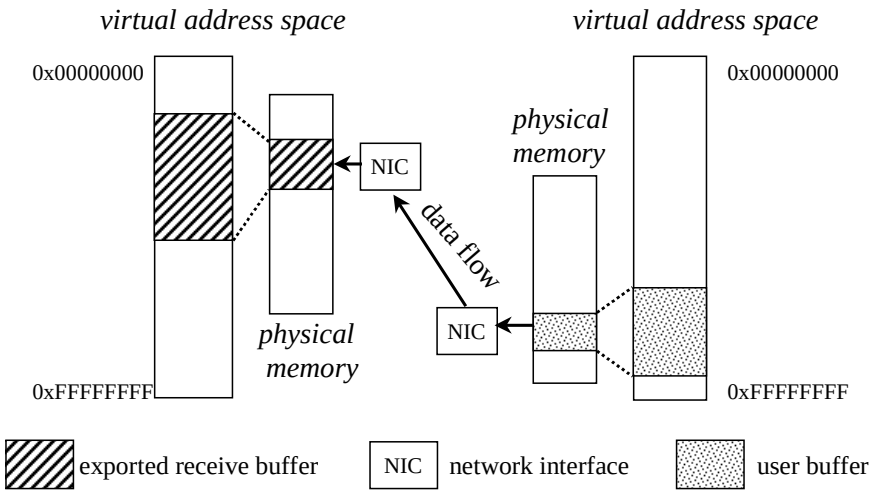


Figure 3.1: The VMMC Communication Model

gains access rights to an exported receive buffer by *importing* it. The basic VMMC model supports *remote store*, which allows an application to send data from a local buffer directly into another application's receive buffer via the network (Figure 3.1).

A user process issues VMMC requests *directly* to the network interface, bypassing the operating system, whose invocations often take longer than sending the actual message. In each request are specified local and remote virtual addresses, as well as the number of bytes to transfer. The communication subsystem takes care of moving data directly between the two virtual address spaces; by doing so, it avoids redundant data copying.

In order for VMMC to work, the network interface must be able to *safely* and *correctly* translate the virtual memory addresses, specified in a user request, to corresponding physical memory addresses. Two OS aspects make this task highly non-trivial. First, the virtual-to-physical mappings are kept internally by the operating system and are inaccessible to both the user applications and to the network interface. Second, the communication subsystem must guarantee that the application buffer remain resident in physical memory until the data transfer is complete.

Several schemes for communication-related address translation do exist. None of them,

however, provides an efficient solution for VMMC-like data transfer between virtual address spaces. Some require OS invocations to initiate communication in kernel mode. Others restrict where in the application's virtual address space data communication takes place, often making it difficult to achieve zero copy.

This chapter describes a novel address-translation mechanism, called User-managed TLB (UTLB), for network interfaces. UTLB eliminates system calls and device interrupts in the common case; the OS is invoked to pin a user buffer only when it is first used in communication. Since UTLB relies neither on OS modifications nor on esoteric OS features, it is portable across a wide variety of OS platforms and network interfaces.

3.2 Background and Previous Work

There is a large body of literature on communication subsystems. Most related work is on how and where address translation is performed in a communication subsystem. The four key issues are as follows. First, how to initiate data transfer requests from an application. Second, how to maintain consistency between the translations on the host and those on the network interface. Third, how to replace translation entries on the network interface. And fourth, how to deal with protection in a multiprogramming environment.

Early implementations of communication subsystems typically used dedicated, pinned, system-wide send and receive buffers [63, 66]. Each buffer is laid out in contiguous physical memory so that the network interface needs to know the starting physical address and the number of bytes for a data transfer. When sending a message, the application process traps into the OS to initiate the send. The OS copies the application data into the system send buffer. The network interface then transmits the message from the system buffer. Upon message arrival, the network interface DMAs the data into the system buffer. From there, the OS copies the data into an application process' buffer. Address translation is car-

ried out in two places: one, in the kernel when copying the data to and from the application buffer, and two, on the network interface when accessing the kernel buffer.

A method to lift the constraint of using a large contiguous piece of physical memory for a system buffer is to use a table or a chain of descriptors. Each descriptor contains the address translation for a small contiguous piece of the kernel buffer. The descriptor table or list is stored in a linked list on the network interface. This approach is also called a scatter/gather table. It allows the OS to initiate network data transfers at any place in physical memory. Furthermore, to avoid copying data to and from an application buffer, the OS can pin the buffer and store its address translations in the descriptor table or list. System calls are required to build the descriptors inside the OS. Autonet [81], for example, uses a chained list of descriptors and VAXClusters [49] uses a descriptor table.

Page re-mapping is a method to avoid copying [52, 22, 28, 46, 14]. When transfers are properly aligned and the "right" length (i.e. a multiple of the physical page size), the OS swaps the virtual-to-physical mappings between the pages of the kernel buffer with those of the application buffer. This technique can achieve *zero copy* with buffer restrictions. Furthermore, page re-mapping incurs significant overhead due to OS involvement, interrupt processing, and context switching.

Some systems use a communication processor that runs as a part of the operating system to either access or cache the OS page table in order to translate the application buffer's virtual addresses and initiate DMA transactions on the network interface. The Intel Paragon [68] dedicates an SMP microprocessor on a cache-coherent memory bus for this purpose. The Intel Paragon communication processor also has the ability to control page pinning and swapping, which eliminates the need for system calls and interrupts at the expense of taking a microprocessor away from doing useful computation.

Another approach is to use a protocol processor to deal with address translation and data transfer. Meiko CS-2 [39] and Typhoon [75] use a protocol processor. Stanford FLASH

multiprocessor [50] uses a programmable processor to integrate a memory controller, an I/O controller, a network interface, and a programmable protocol processor.

Several communication subsystems transfer data directly between application buffers and network interface [25, 83, 65]. With this approach, the application is responsible for translating addresses or performing programmed I/O operations to access the network. But this approach is not designed for multiprogramming environments.

An improvement to this approach is to virtualize the network interface. The basic method is to provide a virtual communication port abstraction through which an application process can directly issue requests to the network interface, bypassing the OS. Examples of such systems include Application Device Channels (ADC) [27], Hamlyn [15], U-Net [90], and Virtual Interface Architecture [23]. They all require that applications explicitly pin buffers and install descriptors on the network interface. The descriptors contain address translations for both send and receive sides. Hamlyn calls such a unit a *slot*, U-Net calls it a *communication segment*, and VIA calls it a *memory region*. They typically deal with protection by using a permission key.

Memory-mapped communication takes a direct approach. PRAM [57], SHRIMP [7] and Memory Channel [35] implement memory-mapped communication models [84] that allow applications to send messages to remote memory. PRAM implements a physical memory-mapped model that allows an application to map network interface DRAM into its address space. Writes to this memory are propagated to remote network interface memory. It is the application's responsibility to move data from a send buffer to the sender's network interface memory and move data from the receiver's network interface memory into a receive buffer. The SHRIMP approach [7, 9, 8] implements protected user-level communication using the Virtual Memory-Mapped Communication (VMMC) model. This approach allows an application to send data directly from its virtual memory to a remote process' virtual memory in a multiprogramming environment. This approach requires re-

ceivers to pin and export receive buffers before the data is transfered. The OS translates the virtual addresses and stores the physical addresses on the network interface. The SHRIMP implementation uses a modified OS to automatically pin application buffers used for sending data. A User-level DMA (UDMA) mechanism [9] is used to allow the network interface hardware to obtain virtual-to-physical translations without OS intervention. SHRIMP also provides *automatic update* which automatically propagates application buffer updates to remote virtual memory buffers. Digital's Memory Channel [35] uses an approach that is similar to PRAM on the sending side and to SHRIMP's automatic update on the receiving side.

Another direct approach allows applications to compose and retrieve messages using network interface registers [37, 82]. In addition to network interface registers, the Cray T3E [82] supports remote memory accesses with an approach similar to UDMA. It uses complete page tables to describe global communication segments and all communication pages are pinned in memory.

A network interface typically can hold only a limited number of translation entries, which poses questions about how to maintain consistency between the translations on the host and those on the network interface and how to deal with misses on the network interface. One approach is to let network interface interrupt the host processor on a translation entry miss, and the host processor installs the translation entry on the network interface. The VMMC [31] (the same communication API as that on SHRIMP [7]) for the Myrinet PC cluster employs this approach. It uses a per-process translation table on the network interface.

UNet-MM [5], an extension U-Net, stores address translations in a translation cache on the network interface. Misses in the translation cache are handled by the host OS which pins virtual pages and installs their translations on the network interface.

The UTLB approach described here was presented in Hot Interconnect '97 [30] and in

a project update note [29]. A recent paper by Schoinas and Hill [80] describes a similar approach. None of these papers deal with the issues of a shared translation cache in a multiprogramming environment. Further, they do not study translation replacement, nor the effects of prefetching translation entries.

3.3 Design of UTLB

The User-managed TLB (UTLB) is an address translation mechanism for user-level communication. The main ideas in UTLB are demand-driven page-pinning, protected translation table, and user-level lookup.

The first idea is *demand-driven page-pinning*: pin the local buffer when it is used in communication for the first time, at the same time supplying the address translations to the network interface. The buffer remains pinned in physical memory so that subsequent data transfers using this buffer can be initiated directly at the user level. For applications that display spatial locality in their communication patterns, the cost to pin the virtual pages is amortized over multiple communication requests.

The second idea is to establish a *protected* translation table for pinned virtual pages. UTLB allocates a translation table for each process on the network interface. A translation table contains physical addresses for a process' virtual pages that have been pinned in physical memory. The translation table is invisible to the user process. However, the user process can specify to the operating system where in the table to store the physical translations for a given virtual buffer, hence the name "User-managed" TLB. To transfer data on a virtual page, the user process specifies to the network interface the index in the translation table where the page's physical address stored. Using this index, the network interface reads the physical address directly from the translation table.

The third idea is to construct a fast *user-level lookup* data structure. The user process has

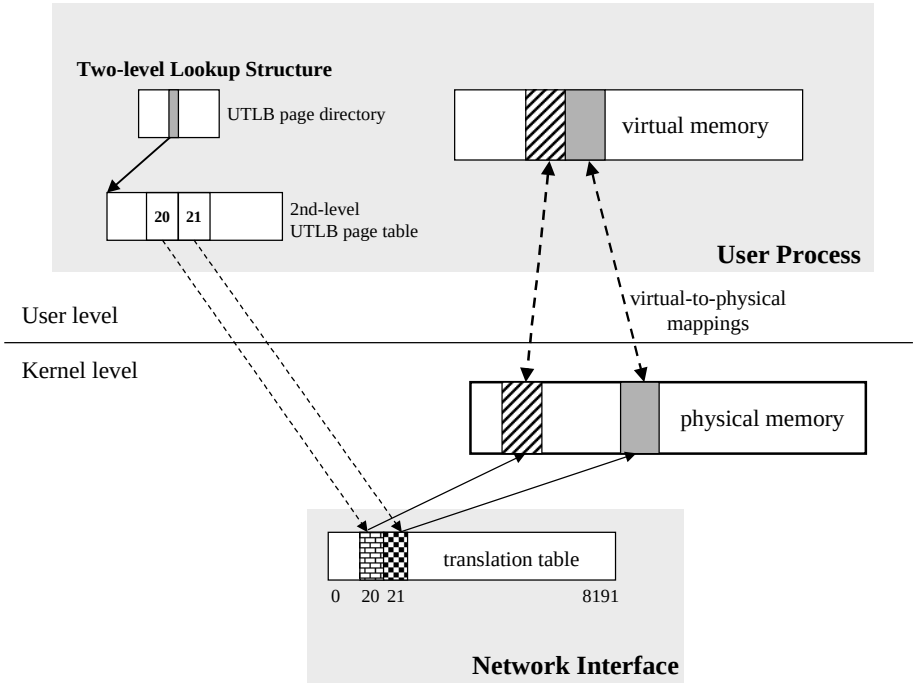


Figure 3.2: Structure of UTLB on VMMC

to keep track of the mapping between the translation table indices and the pinned virtual pages. The lookup table uses a standard two-level page table architecture [36, 45]. It contains one entry for each virtual page. An entry can either be invalid or contain the index in the translation table where the physical address for this virtual page is stored. Only two memory references are required to obtain the UTLB index for a given virtual page address.

3.3.1 Per-process UTLB

Combining the above three ideas results in a *Per-process* UTLB shown in Figure 3.2. The communication subsystem allocates a fixed-sized translation table for each process. The translation tables are allocated directly in the network interface memory. They are protected from user processes. The user process asks the OS to pin certain virtual pages and “install” their physical translations at specified locations in its translation table. This can be done

with a device driver call. A user-level library maintains the mapping between the translation table indices and the virtual page addresses in a two-level lookup tree.

It is possible that the translation table is filled up while more virtual pages need to be pinned. The user-level library can detect such *capacity misses* and *evict* some translations already in the UTLB translation table. Eviction of an entry results in unpinning of the virtual pages. The UTLB library decides which translation table entries to evict and asks the OS to unpin corresponding virtual pages and invalidate the entries. To reduce the frequency of capacity misses, the user-level library monitors the virtual page usage and use a replacement policy such as *LRU* to select the victim entries for eviction.

To ensure correctness, the user-level library must only select virtual pages that will not be involved in any outstanding send requests. Otherwise, the network interface must be able to check for possible unpinned pages, and interrupt the host to pin pages before executing the requests.

3.3.2 Shared UTLB-Cache

A drawback of the per-process UTLB is that it statically allocates translation tables from the network interface memory. This results in a fairly small translation table for each process. On a workstation with large physical memory, a significant portion of a user process' virtual address space can be involved in data communication. The size of the UTLB translation table must be increased to reduce capacity misses.

To overcome the size limitation of the per-process UTLB translation table, we place a Shared UTLB-Cache on the network interface and move the entire translation tables to host physical memory (DRAM), as shown in Figure 3.4. In this scheme, the Shared UTLB-Cache caches the entries from the translation tables. Each Shared UTLB-Cache entry contains the process ID and part of the translation table index to uniquely identify an entry from a particular translation table. When a miss occurs in the Shared UTLB-

```
User Program:
send_message(vaddr, nbytes)
    1) look up (vaddr, npages) in the user-level
       table
    2) if (some pages in the range are not pinned)
           find free UTLB translation entries
           invoke the ioctl call to
                   a) lock the pages
                   b) fill the translation entries
    3) submit the request to NI using the indices


Network Interface:
1) receive user request
2) obtain physical addresses by directly
   indexing the translation table
3) perform DMA
```

Figure 3.3: Pseudo-code that illustrate the steps taken by the user process to send data from its virtual buffer

Cache, the network interface simply reads the entry from the translation table in physical memory. The cost of reading an entry over the I/O bus is only a couple of microseconds. Therefore, in the worst case, when every translation lookup misses in the Shared UTLB-Cache, the lookup time is a few microseconds. This is typically better than interrupting the host OS and letting the host install the required translation on the network interface on every translation miss.

A miss in the Shared UTLB-Cache requires that the network interface read translation entries over I/O bus. The miss penalty is therefore several times the hit cost which is simply a memory reference on the network interface. A miss in the Shared UTLB-Cache will have a perceivable impact on small-message latency.

We apply existing techniques in processor cache design [36, 71] to reduce the miss rates in the Shared UTLB-Cache. Misses fall into three categories: capacity misses, conflict

misses, and compulsory misses [38]. When only one process is using the network interface, both capacity misses and conflict misses may occur in the Shared UTLB-Cache. Multi-programming may further increase conflict misses.

Capacity misses can be reduced by enlarging the size of Shared UTLB-Cache. Conflict misses can be reduced by making the cache set-associative. Associativity can also help reduce conflict misses that are caused by multi-programming. A simple scheme to reduce the conflict misses is to *offset* a translation table index by a process-dependent constant. The same index from different translation tables will be hashed into different locations in the Shared UTLB-Cache. Prefetching translation entries in the Shared UTLB-Cache reduces the miss rates when applications display spatial locality. But, it also incurs additional cost for fetching more entries. In Section 3.6, we evaluate the effect of cache size, associativity, and prefetching.

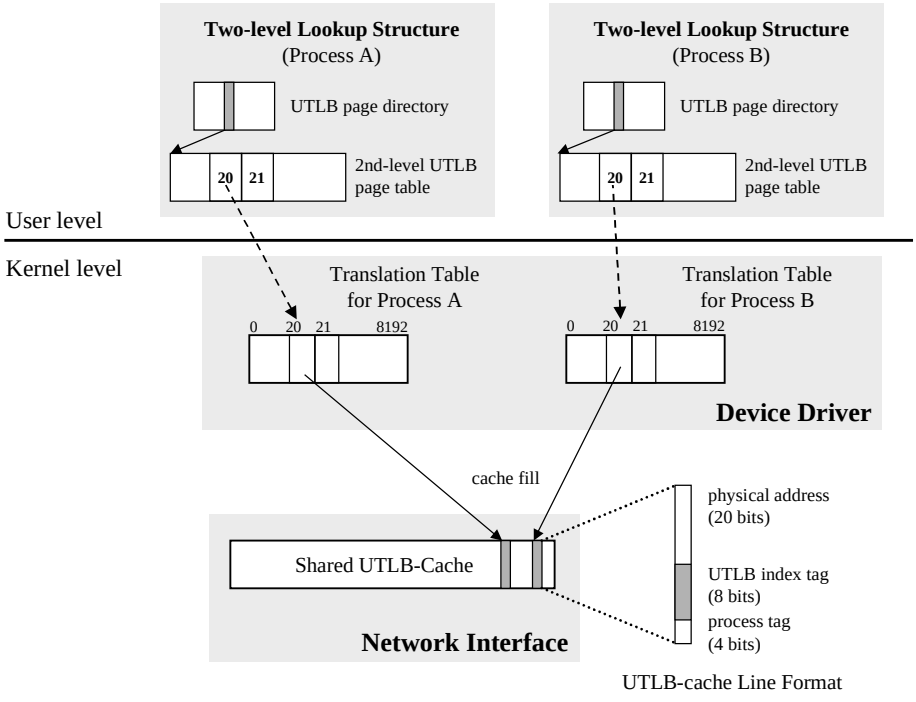


Figure 3.4: Structure of Shared UTLB-Cache

3.3.3 Hierarchical-UTLB

The Hierarchical-UTLB is a simplification of the UTLB. Instead of letting the user process search its lookup data structure for UTLB indices, Hierarchical-UTLB directly uses the protected translation table as the lookup data structure. Under Hierarchical-UTLB, the user process submits virtual addresses to the network interface. The network interface directly searches Hierarchical-UTLB translation table for physical address translation (Figure 3.5).

The translation table in Hierarchical-UTLB resembles a typical two-level page table structure. The first-level directory points to second-level page tables in physical memory. Each page table entry stores the physical address of a pinned virtual page. The Hierarchical-UTLB translation table differs from a real page table in one important aspect: the entries in the second-level Hierarchical-UTLB translation table are the physical addresses of virtual pages that have been explicitly pinned by the user process.

The top-level directory of a Hierarchical-UTLB translation table is always stored in the network interface so that when there is a miss in the Shared UTLB-Cache it takes one memory reference in the SRAM to access the page directory and one DMA to access the second-level page table.

The user-level library only needs a bit array to maintain the memory-pinning status of virtual pages. In addition, a process can directly use the virtual address to represent its buffers. Instead of indices, the network interface simply uses the virtual address to look up the physical address in the Shared UTLB-Cache or query the UTLB page table on a miss.

The Hierarchical-UTLB eliminates the need to handle UTLB fragmentation: after complex data accesses, a user buffer's translations may be scattered in the translation table. Integrating Hierarchical-UTLB with the translation table used for *receive buffers* is also straightforward. Virtual addresses are uniformly used to represent both remote and local buffers.

In rare situations, the second-level translation tables in the Hierarchical-UTLB occupy

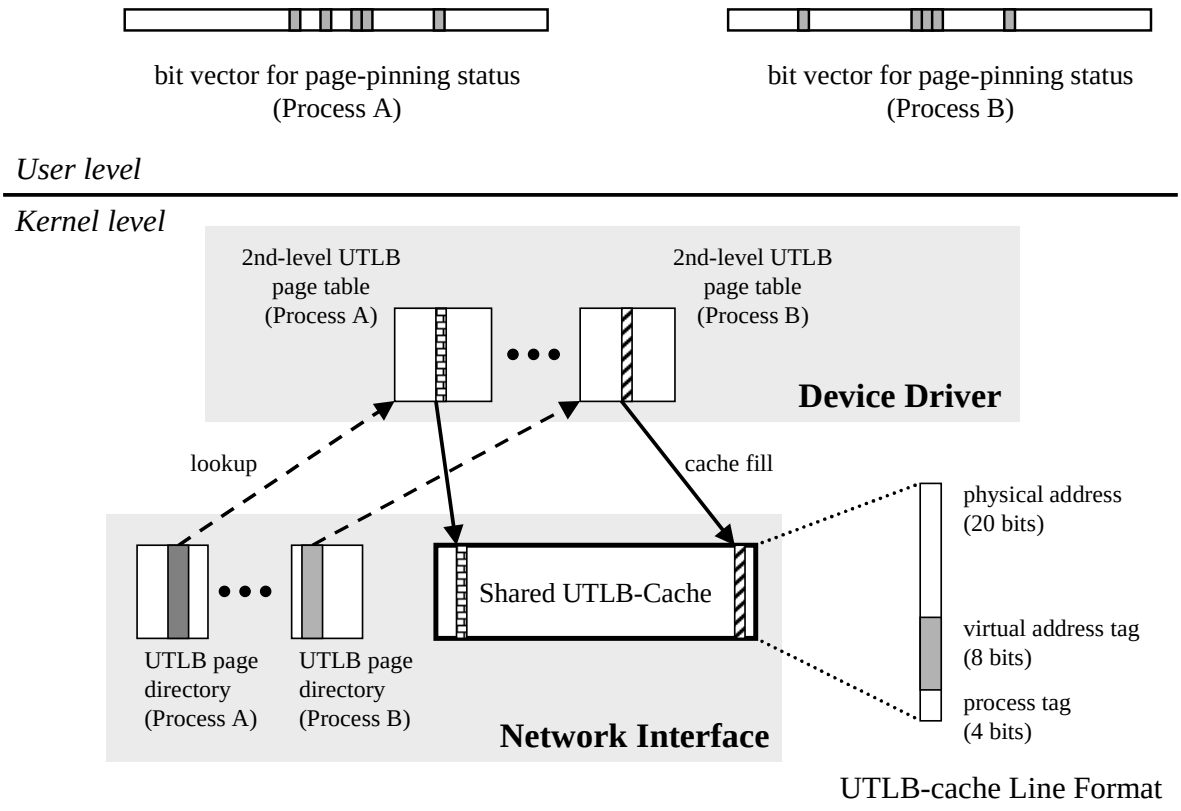


Figure 3.5: Structure of Hierarchical-UTLB

too much physical memory A solution to this problem is to manage the second-level translation tables in the same manner as virtual memory paging. One bit of information is added to each entry in the top-level directory which indicates whether the second-level table is in physical memory or on the disk. If the second-level table is swapped out, the directory entry contains the disk block number instead of the physical address of the second-level table. When the network interface detects that a page of the second-level table has been swapped out, it can interrupt the host OS to bring in the page.

To summarize, at the cost of one extra SRAM reference to process a UTLB cache miss, Hierarchical-UTLB simplifies the construction of UTLB and eliminates the fragmentation problem. In the rest of the chapter, UTLB refers to Hierarchical-UTLB.

3.3.4 User-level replacement policies

An important feature of UTLB is that it allows an application to decide which virtual pages to unpin when the system runs out of available physical memory. Because the application process often has knowledge about its virtual memory access, it can use a custom replacement policy to minimize the number of page pinning and unpinning operations. UTLB predefines five replacement policies for applications to choose: LRU, MRU, LFU, MFU, and RANDOM.

An important issue related to the replacement policies is how to manage the amount of physical memory that a user process can pin. This is a complex issue especially in a multiprogramming environment where physical memory pages can be shared. Enforcing a static limit on the number of pages a process can pin is straightforward, But, implementing a dynamic limit requires that the OS synchronize with the user-level UTLB data structures when reclaiming pinned physical pages.

The combination of the two issues is similar to the application-controlled file caching problem [16], where multiple applications share a set of file cache blocks in the kernel and each application can choose its own replacement policy. So, related theoretical results of the application-controlled file caching problem [17] apply to the application-controlled memory pinning/unpinning problem. On the other hand, this requires experimental studies to understand the solutions to the combined problem.

3.4 An Implementation of UTLB

We developed an implementation of the Hierarchical-UTLB for our custom protected user-level communication model called Virtual Memory-Mapped Communication (VMMC).

Our UTLB implementation is a part of the VMMC communication mechanism for Myrinet PC clusters. Myrinet [10] is a switched point-to-point network capable of trans-

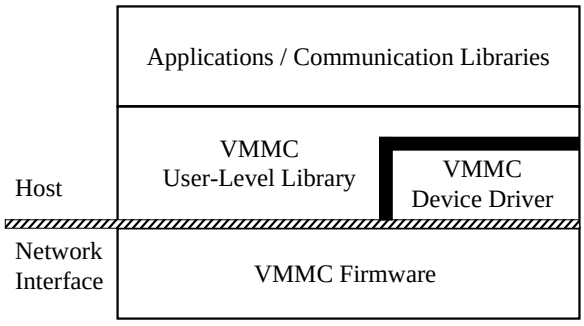


Figure 3.6: The VMMC System Architecture

ferring data at 160 MB/sec on each link. The Myrinet PCI network interface has a 33 MHz RISC microprocessor (LANai 4.2) and 1 MB Static RAM (SRAM). A cluster of 300 MHz Pentium-II PC workstations are connected to the Myrinet network. The host operating system is Windows NT 4.0.

The communication subsystem of VMMC consists of three components: the VMMC Myrinet firmware, the device driver, and the user-level library [31] (see Figure 3.6). The VMMC device driver initializes the network interface and downloads the firmware, called the Myrinet Control Program (MCP), into the network interface SRAM. The driver also allocates a special command post buffer from the Myrinet SRAM and maps it into the application's address space. The user-level VMMC library posts communication requests to the command buffer. The address of a command buffer is used to identify the user process. The MCP polls user requests from each command buffer and process them in the order that they are received.

Our implementation of Hierarchical-UTLB follows directly from the layout in Section 3.3.3; we implemented a Shared UTLB-Cache. The size of Shared UTLB-Cache that we chose is 32 KB (or 8 K entries). The device driver allocates the two-level translation table dynamically for each application that uses the VMMC system. An `ioctl()` call is added to the VMMC device driver for pinning virtual pages and storing physical addresses in the translation table. The implementation did not require any modifications to

the Windows NT operating system. An earlier implementation of UTLB on Linux was also done without OS modifications. The device driver allocates and pins a "garbage" page. All UTLB translation table entries are initialized with the physical address of the garbage page. This scheme saves the network interface from checking the validity of user-submitted indices. At worst, the network interface transfers data to and from an unused garbage page; no harm is done to the system or other applications.

3.5 Performance of UTLB

The overall cost for translating a virtual page in the UTLB includes the overhead on the host processor and the overhead on the network interface. Each overhead varies depending on whether the virtual page is pinned and whether the physical address is in the UTLB network interface cache. The fastest path to translate a virtual address on the network interface is taken when the virtual page is pinned and its physical address is present in the UTLB network interface cache (a hit). The total overhead for this path is only 0.9 μs (0.4 μs on the host and 0.5 μs on the network interface).

The time on the network interface is measured with LANai's real-time clock register, with an accuracy of 0.5 μs. Because the LANai 4.x processor has *no* instruction or data caches, averaging the total time of repeated operations gives the exact timing for an individual operation. On the host, the time is measured with the Pentium processor's cycle counter with an accuracy of a CPU clock cycle. Reading the cycle counter has an overhead of 39 cycles.

3.5.1 Host-side performance

Table 3.1 lists the overhead of UTLB host-side operations: user-level lookup (as *check*), page-pinning (as *pin*), and page-unpinning (as *unpin*). The lookup procedure checks a bit

num_pages	1	2	4	8	16	32
check min	0.2	0.2	0.2	0.2	0.2	0.2
(μs) max	0.4	0.6	0.6	0.6	0.6	0.7
pin (μs)	27	30	36	47	70	115
unpin (μs)	25	30	36	50	80	139

Table 3.1: UTLB overhead on the host processor.

map to see if the virtual pages of a buffer are already pinned in the physical memory. The cost of checking the bit map varies with the first bit's position in the bit map. The table reports both the minimum and maximum costs from all possible bit positions.

If some virtual pages are not pinned, the lookup procedure invokes a device driver `ioctl()` call to pin these pages and store their physical addresses in the UTLB translation table. As the numbers suggest, page-pinning is an expensive operation. They validate the UTLB design strategy of on-demand pinning and translation caching.

3.5.2 Network interface performance

Table 3.2 shows the cost of UTLB operations on the network interface: hit cost, DMA cost and miss handling cost (as *miss cost*). The number given here is for a direct-mapped cache with 8K entries. The hit cost is the time it takes the network interface to look up a virtual page's physical address in the UTLB network interface cache [1].

num_entries	1	2	4	8	16	32
DMA cost (μs)	1.5	1.6	1.6	1.9	2.1	2.5
total miss cost (μs)	1.8	1.9	1.9	2.3	2.8	3.2

Table 3.2: UTLB overhead on the network interface (The hit cost is a constant 0.8 μs.)

The miss cost includes the time to obtain the physical address for the second-level page table (see Section 3.3.3) and the time to DMA the entries into the UTLB network interface

[1] The Myrinet VMMC firmware breaks down data transfer at 4KB page boundaries. Translation lookups are performed one page at a time. Therefore, we list the hit cost for only one entry here.

cache. Multiple entries are prefetched at once during a miss to exploit the spatial locality of a program's data access. The prefetching cost remains relatively constant with respect to the number of entries fetched because DMA setup dominates the total fetch time for a small number of words (see Table 3.2).

3.6 Application-driven Analysis of UTLB

We would like to answer three questions:

- How does the UTLB compare with the approach where the network interface interrupts the host to handle translation misses?
- What are the appropriate values for the size and the associativity of the Shared UTLB-Cache?
- What are effects of prefetching translation entries?

We used a trace-driven simulation approach to evaluate these issues. A trace-driven approach allows us to determine the best values for parameters and to compare the UTLB with other address translation mechanisms. We chose communication traces from a Myrinet cluster of SMP workstations because they allow us to study the behavior of UTLB under a high degree of multi-processing. The cluster consists of four 4-way Intel SMP workstations connected with a Myrinet network. Each SMP has four 200 MHz PentiumPro microprocessors and 256 MB of DRAM.

We ran a number of applications from the SPLASH2 [95] Application Suite with the Home-based Release Consistency SVM Protocol [96, 76]. On each SMP, there are four application processes and a protocol process, all of which use Myrinet for sending and receiving messages. We instrument the VMMC software to trace each send and remote read

request along with a globally-synchronized clock [56]. Time stamps are used to serialize the traces from the five processes on each SMP. The traces are then fed to a UTLB simulator.

The simulator mimics the behavior of a network interface translation cache, the host-side UTLB driver, and user-level library. The simulator reads traces, serializes the communication requests using the time stamps in the trace, and derives detailed statistics on translation misses, and the number of page pinnings and unpinnings. The network interface translation cache is simulated with direct-mapping, 2-way, and 4-way associativity. The simulator implements an LRU replacement policy to manage pinned virtual pages given a fixed physical memory constraint. We also developed a simulator for the interrupt-based approach where the network interface interface interrupts its host CPU on a translation miss, and the CPU handles page pinning, unpinning, and installing new translation entries.

3.6.1 Applications

Seven applications from the SPLASH-2 suite are used:

- **Barnes** implements the original Barnes-Hut algorithm for NBody simulation. Each process gets a partition of the particles and calculates their new positions during one time step. Communication in this application is moderate as the particle partition exhibits spatial locality.

- **FFT** implements a parallel 2D Fast Fourier Transform algorithm. This program exhibits high degree of data communication.

- **LU** is a parallel LU matrix decomposition program.

- **Raytrace** uses a task-farm model to raytrace a scene. Communication in Raytrace revolves around the task queues.

- **Radix** sorts an array of integer keys in parallel. The algorithm consists of a number of radix-sort phases. During a phase, each process sorts a contiguous sequence of the keys according to part of the keys. At the end of the phase, the results from each processes are combined to form a new array for subsequent radix-sort phases.

- **Volrend** uses a task-farm model to render a 3-D volume. Communication in this application also centers on the task queues.

- **Water** calculates movements of molecules using a spatialized algorithm to exploit data locality.

Applications	Problem Size	Footprint (4 KB pages)	# translation lookups
FFT	4M elements	10,803	43,132
LU	4K x 4K matrix	12,507	25,198
Barnes	32K particles	2,235	35,904
Radix	4M keys	6,393	11,775
Raytrace	256 x 256 car	6,319	14,594
Volrend	256^3 CST head	2,371	9,438
Water-spatial	15,625 molecules	1,890	8,488

Table 3.3: Application problem size, communication memory footprint, communication translation lookup frequency.

Table 3.3 shows the application problem size, communication memory footprint, and translation lookup frequency. The communication memory footprint indicates the average number of distinct virtual pages used for communication on each node. The number of translation lookups is the average number of communication operations performed on each node.

3.6.2 Comparing UTLB with an interrupt-based mechanism

To compare the two approaches, we assume that the cache structures are the same for both cases. We varied a set of parameters to study the behavior of the UTLB mechanism and the interrupt-based approach. The parameters include the amount of physical memory available to each process, the size and the associativity of the network interface translation cache. For each application and a particular set of parameters, the UTLB simulator reports the number of user-level check misses, the number of network interface translation misses (caused by capacity or conflict), and the number of pages unpinned. The simulator for the interrupt-based approach reports only the number of network interface translation misses and the number of unpin operations.

Table 3.8 shows the number of check misses per lookup, the number of network interface translation misses, and the number of unpinned pages per lookup. All the numbers are averaged over the total number of lookups. The numbers for the interrupt-based approach are under the *Intr* label. Our trace-driven simulations show that UTLB requires fewer page pinning and unpinning operations than the interrupt-driven approach for all cache sizes. The cache simulated here is a direct-mapped cache with index offsetting (see Section 3.6.4). The host physical memory is unlimited, therefore, UTLB does not unpin application pages. However, the interrupt-based approach always unpins a page that is evicted from the network interface translation cache [5]. This is a major difference between UTLB and the interrupt-based approach.

Note that network interface misses are handled differently by the two approaches. With UTLB, a network interface interrupt moves the missed entry from host memory into the network interface translation table directly, whereas the interrupt-based approach has to interrupt the CPU for every translation miss in the network interface. On most computer systems, interrupts are an order of magnitude more expensive than memory references over the I/O bus. The UTLB mechanism can offer significantly faster miss handling than an

interrupt-based approach. On the other hand, once in the interrupt handler, pin or unpin requires no protection domain crossing, whereas the UTLB approach requires paying the overhead of a system call.

The real cost of a translation lookup depends on the components shown in the table. In particular, the cost function for each mechanism is:

$$\begin{aligned} lookup_{utlb} &= user\_check\_hit \\ &+ user\_pin\_cost \cdot check\_miss\_rate \\ &+ ni\_check\_hit \\ &+ ni\_miss\_cost \cdot ni\_miss\_rate \\ &+ user\_unpin\_cost \cdot unpin\_rate \end{aligned}$$

$$\begin{aligned} lookup_{intr} &= ni\_check \\ &+ (intr\_cost + kernel\_pin\_cost) \cdot ni\_miss\_rate \\ &+ unpin\_kernel\_cost \cdot unpin\_rate \end{aligned}$$

In the above equations,

- *ni_miss_cost* is the average cost for the network interface to fetch entries from the UTLB translation table in host memory.
- *user_check_hit* and *ni_check_hit* are the costs that every UTLB translation lookup incurs. The interrupt-based approach incurs the *ni_check_hit* cost every time.

For example, on our Myrinet implementation of UTLB, the *ni_check* is measured at 0.8 μs per lookup, the *user_check* at 0.5 μs, and 10 μs for invoking the system interrupt handler by the network interface. Per-page cost for pinning and unpinning the application

buffer depends on how many pages are involved in one call. The SVM applications, whose traces we use to drive our simulation, typically transfer one page of data at a time. On a 300 MHz Pentium-II PC running Windows NT 4.0 pinning one page takes 27 μs and unpinning take 25 μs. On Linux, the pinning and unpinning costs are similar to those on NT. When computing the lookup cost for the interrupt-based approach, the pinning and unpinning costs must be adjusted to factor out context switches. Using these numbers, we can calculate the average translation lookup cost for given applications, as shown in Table 3.4. The reason why the lookup cost for FFT is higher that for Barnes is that FFT accesses a large amount of virtual memory and hence incurs high page pinning overhead. In both applications, UTLB offers faster translation lookup than the interrupt-based approach.

Cache Entries	Barnes		FFT	
	UTLB	Intr	UTLB	Intr
1 K	2.6 μs	4.9 μs	9.0 μs	21.7 μs
4 K	2.5 μs	2.5 μs	8.9 μs	20.9 μs
16 K	2.5 μs	1.9 μs	8.7 μs	14.8 μs

Table 3.4: Average lookup cost comparison: UTLB vs. Intr. (infinite host memory, no prefetch, with cache index offsetting)

We also ran the simulation with 4 MB memory restriction on each process to study how each address translation mechanism behaves under limited memory constraints. The results are shown in Table 3.9. The number of page pinnings and unpinnings increases for most applications. UTLB stills achieves lower overhead than the interrupt-based approach with the physical memory constraint.

Our trace-driven simulation results show that UTLB has fewer page pinnings and unpinnings than the interrupt-based approach. UTLB does not suffer from a large number of interrupts as does the interrupt-based approach. In addition, the unique design of the host-side translation table permits UTLB to keep translations “alive” even after they are evicted from the network interface translation cache. This results in fewer unpinned pages than the

interrupt-based approach. Our results also show that the cost for pinning and unpinning pages can sometimes dominate the cost of address translation (e.g. FFT). It is therefore important to reduce the cost to pin and unpin application pages.

3.6.3 UTLB overhead in Display Wall applications

We also instrumented 7 applications to measure UTLB overhead in actual settings. Four applications are from the SPLASH-2 benchmark and described in previous sections. Three are Display Wall applications. They are *vis*, *glaze*, and *vdd*. These applications use VMMC Winsock as their underlying communication mechanism.

Vis is a visualization program, implemented by our colleagues that uses a cluster of 13 PCs to visualize isosurfaces of scientific data. The program has three components: client control, isosurface extraction, and rendering. The client control runs on a single PC to implement the user interface for users to steer the entire visualization process. The isosurface extraction uses 4 PCs in the cluster to extract isosurfaces in parallel, and sends them to the rendering program which runs on 8 PCs. Each renderer PC simply receives isosurfaces, renders and displays the rendered images on a tiled, scalable display wall. VMMC Winsock is used for all interprocess communication among the PCs running the client control, isosurface extraction, and rendering programs.

Glaze is a commercial OpenGL evaluation program from Evans & Sutherlands. An OpenGL interception layer, developed by our colleagues, allows us to use one PC to drive OpenGL animation of any commercial software on a tiled display. We ran the glaze program on a PC; a custom wrapper DLL (opengl32.dll) intercepts the OpenGL calls made by the program and distributes the commands to 8 separate PCs which together drive a 2x4 tiled display. The wrapper DLL transmits rendering commands over the VMMC Winsock layter to the renderers. The renders behave just like those in *vis*: they simply receive the commands and render them for its portion of the tiled display.

Applications	fft	barnes	radix	water
memory footprint	57.4 MB	16.6 MB	54.4 MB	8.9 MB
memory constraint	50.0 MB	16.0 MB	30.0 MB	6.0 MB
host lookup hit rate	64.46%	96.67%	54.99%	97.63%
host pin-page rate	35.54%	3.33%	45.01%	2.37%
host unpin-page rate	23.36%	2.13%	36.10%	2.27%
avg lookup hit cost	0.36 μs	0.50 μs	0.35 μs	0.40 μs
avg pin overhead on NT	39.1 μs	40.0 μs	42.1 μs	29.6 μs
avg pages per pin on NT	1.00	1.07	1.00	1.07
avg unpin overhead on NT	33.2 μs	36.7 μs	35.2 μs	35.0 μs
avg miss overhead on NT	42.2 μs	49.6 μs	68.7 μs	32.8 μs
predicted ovhd on Linux	21.6 μs	29.3 μs	46.4 μs	8.6 μs
avg host overhead on NT	15.23 μs	2.13 μs	31.1 μs	1.27 μs
predicted ovhd on Linux	7.91 μs	1.45 μs	21.1 μs	0.59 μs

Table 3.5: UTLB performance with memory constraint

Vdd is a virtual display driver that implements a large desktop that has the same resolution (3850x1500) as the Display Wall. It is installed on a 450 MHz Pentium II PC running NT 5B2 OS. A user process is responsible for packetizing the updates made to the vdd's memory-resident framebuffer and distributing them to the 8 PCs that drive the wall. We used NT VMMC Sockets with the pointer-based extensions to distribute framebuffer updates efficiently [26].

Results Table 3.10 presents OS-dependent components of UTLB overhead, without any memory constraint. The miss rates and overheads for the NT platform are measured directly by running the three applications on our NT VMMC cluster. We use these numbers, as well as the micro-benchmark results, to predict the overheads on Linux. In the table, *host pin rate* is the ratio between the number of pin operations and the number of UTLB lookups; and similarly for *host unpin rate*. *Memory footprint* is the total amount of distinct virtual memory that is involved in data transfer for each application. Lower page-pin rate (or lookup miss rate) translates into lower average UTLB overhead. Since the host UTLB miss rates are quite low, the OS overhead for pinning and unpinning user buffers has little impact

on the average UTLB host overhead.

Under tight memory constraints, such as in a multi-programming environment or a large-memory applications, there may not be enough physical memory to absorb the entire memory footprint of applications. UTLB deals with such low-memory situations by unpinning virtual pages that are not currently involved in data transfer. This is a critical feature for UTLB-based VMMC to be useful in a multi-programming environment. Table 3.5 presents the the same experiments with additional memory constraints such that the total amount of application-pinnable physical memory is less than the memory footprint. Due to varying nature of the applications, the memory constraint is set differently on a per-application basis. And also, it turned out that our Display Wall applications require most of their working sets to be pinned as receive buffers. This is because each Display Wall application uses a small buffer (often less than 1 MB) to gather commands and send the whole chunk to the receivers. Therefore, we only present the memory-constrained results for the 4 SVM applications. Again, the measurements are all taken by running the applications on the NT cluster, and predictions are made for the Linux platform.

With low memory constraint, we see higher UTLB miss rates and page unpin rates. Note that our prediction for UTLB performance on Linux suggests that faster page-pin and page-unpin OS calls further reduce the average UTLB overhead by as much as 50%. We conclude that improving system calls such as page-pin and page-unpin benefits user-level communication.

3.6.4 The effect of cache size and associativity

A miss in the UTLB network interface cache costs several times more than a hit. The average translation lookup cost in the network interface depends on the hit/miss ratio in the UTLB cache. Misses in the network interface cache include capacity, conflict, and compulsory misses. Cache size and associativity directly affect the capacity and conflict

miss rates.

Cache Entries	Associativity	Applications						
		Barnes	FFT	LU	Raytrace	Radix	Volrend	Water
1 K	direct	0.10	0.31	0.35	0.48	0.50	0.50	0.62
	2-way	0.12	0.30	0.32	0.48	0.49	0.50	0.63
	4-way	0.13	0.30	0.30	0.49	0.50	0.51	0.63
	direct-nohash	0.36	0.50	0.51	0.57	0.60	0.78	0.90
2 K	direct	0.07	0.27	0.29	0.46	0.49	0.50	0.60
	2-way	0.06	0.26	0.27	0.46	0.48	0.50	0.60
	4-way	0.07	0.22	0.26	0.47	0.48	0.50	0.60
	direct-nohash	0.35	0.42	0.48	0.57	0.60	0.74	0.90
4 K	direct	0.05	0.12	0.27	0.45	0.49	0.49	0.57
	2-way	0.05	0.11	0.25	0.45	0.47	0.49	0.57
	4-way	0.04	0.10	0.25	0.44	0.46	0.49	0.57
	direct-nohash	0.27	0.35	0.47	0.56	0.60	0.71	0.90
8 K	direct	0.04	0.11	0.25	0.44	0.46	0.49	0.55
	2-way	0.04	0.10	0.25	0.44	0.44	0.49	0.55
	4-way	0.04	0.10	0.25	0.41	0.43	0.49	0.55
	direct-nohash	0.27	0.35	0.46	0.56	0.57	0.71	0.90
16 K	direct	0.04	0.10	0.25	0.38	0.43	0.49	0.54
	2-way	0.04	0.10	0.25	0.37	0.43	0.49	0.54
	4-way	0.04	0.10	0.25	0.34	0.43	0.49	0.54
	direct-nohash	0.27	0.35	0.46	0.50	0.55	0.71	0.90

Table 3.6: Overall miss rates in Shared UTLB-Cache vs. cache size (infinite host memory, no prefetch, with cache index offsetting for direct, 2 and 4 way)

In Table 3.6, we show the overall miss rates in the network interface cache for various cache sizes and associativities. In the table, the rows marked with “direct-nohash” represent a simple direct-mapped cache. The rows marked with “direct” represent a direct-mapped cache that offsets each virtual address with a process-dependent constant in the network interface. This address offsetting technique is used to reduce conflict misses resulting from simultaneous accesses to the network interface by multiple processes. Our simulation results show that this technique works very well. The overall miss rates in a direct-mapped cache are close to, and frequently lower than, those of two-way and four-way set-associative caches. The same offsetting technique is also used for set-associative

caches. Cache miss rates would be higher without offsetting.

However, offsetting the cache index may interfere with set-associativity. This may explain the fact that miss rates in the set-associative cache (with offsetting) are higher than those in the direct-mapped cache (with offsetting).

When the actual cost of lookup is considered, the set-associative caches lose to the direct-map cache. The reason is that a set-associative lookup needs to check more entries per cache line than a direct-mapped cache. In a hardware cache, checking of multiple entries on a line can be done in parallel. Since the Shared UTLB-Cache is implemented in Myrinet firmware, the network interface processor can only check one cache entry at a time. Therefore, the cost per translation lookup is higher in a set-associative UTLB cache than a direct-mapped cache. For this particular implementation of UTLB, the trace-driven analysis justifies our choice to use direct-mapping (with offset) for the cache.

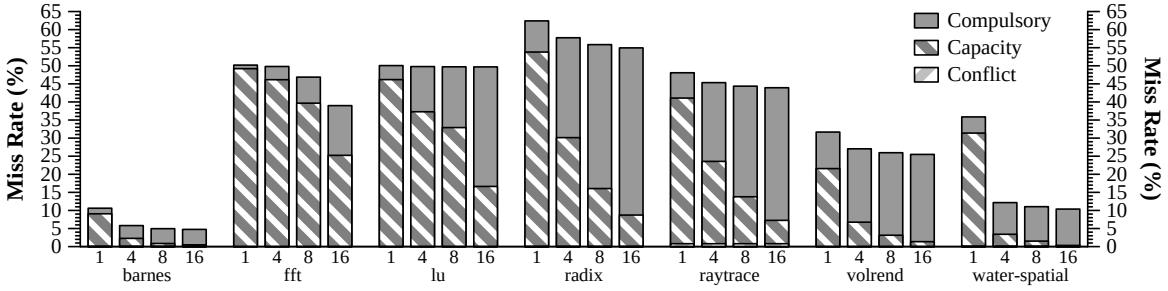
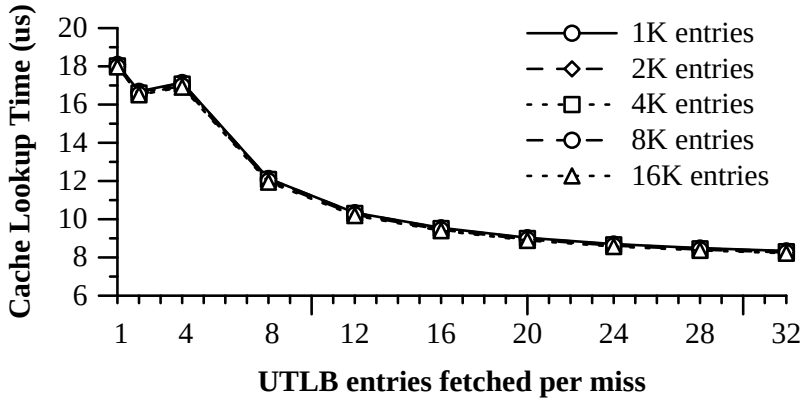


Figure 3.7: Breakdown of translation cache miss rates for 1K-16K cache entries (with infinite host memory and no prefetch)

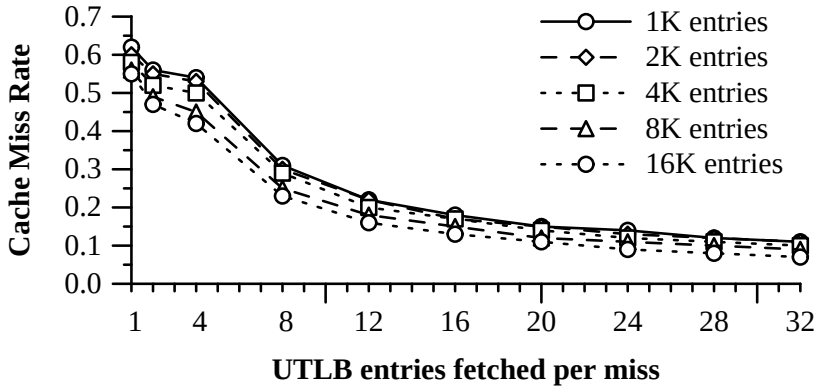
3.6.5 The effect of prefetching

Enlarging the translation cache and offsetting the translation indices can reduce the capacity and conflict misses in the UTLB network interface translation cache. Figure 3.7 shows the breakdown of translation misses on the network interface for all seven applications, using infinite host memory and a direct-mapped network interface cache without any prefetching.

As expected, the number of conflict misses and capacity misses decrease as the cache size increases. But more importantly, this breakdown shows that compulsory misses still constitute the majority of translation misses.



Average cache lookup cost vs. prefetch size



Cache miss rates vs. prefetch size

Figure 3.8: Prefetching effect in the translation cache (RADIX with infinite host memory and a direct-mapped cache)

To reduce compulsory misses, we let the Shared UTLB-Cache prefetch multiple entries when handling a translation miss in the network interface cache. We plot the miss rates and lookup cost from RADIX as a function of prefetching size in the two graphs shown in Figure 3.8. As expected, the overall miss rates decrease as prefetching becomes more aggressive. Prefetching can effectively reduce overall miss rate. This happens for two

reasons. First, an application usually displays spatial locality. Prefetching consecutive translation entries takes advantage of such locality. Second, the cost of fetching multiple translation entries increases at a much slower rate than the rate at which overall miss rate drops. As a result, average lookup cost decreases as fetching becomes more aggressive. However, in order for prefetching to work well, translations for contiguous application pages must be available during a miss.

3.6.6 User-level page-pinning

One way to ensure the availability of translations for contiguous pages is to sequentially *pre-pin* application pages on a check miss in the UTLB user-level library. If the communication's data access pattern displays spatial locality, prepinning reduces the page-pinning overhead for each page pinned, because on most computer systems, pinning a user buffer one page at a time is significantly more expensive than pinning the entire buffer all at once.

Currently the UTLB uses a sequential pre-pinning policy, where if a virtual page needs to be pinned, the user library tries to pin a number of contiguous pages starting with that page. On the other hand, unpinning is still done one page at a time.

Cost	pages	barnes	radix	raytrace	water	FFT	LU
pin	1	1.0	13.0	10.5	2.5	6.1	12.0
	16	0.8	7.3	5.0	1.5	15.8	2.3
unpin	1	0.1	0.1	0.8	0.1	0.1	0.1
	16	0.1	10.8	3.5	0.1	93.0	0.1

Table 3.7: Amortized pinning and unpinning for different page-pinnng strategy.

In Table 3.7, we compare the translation lookup performance of UTLB and that of UTLB with 16-page prepinning. The physical memory limit in both cases is 16 MB. In both cases, the misses in the network interface are the same. The performance difference between the two approaches are from the amortized cost of page-pinnings and unpinnings. We show the amortized cost (averaged over total translation lookups) because page-pinning

cost is not a linear function with respect to the number of pages pinned in a system call.

The applications fall in two categories based on their communication patterns: *regular*, which include FFT and LU, and *irregular*, which include the rest [43, 61]. Even the simple sequential policy is very effective for most applications. The only exception is FFT which performs a lot of unnecessary pinning/unpinning with 16-page prepinning. FFT is a regular application with a strided access pattern such that it does not access most of the pages that are pre-pinned. UTLB is forced to unpin these unused pages when physical memory limit is reached.

3.7 Conclusions

In this chapter I described the design and implementation of UTLB, a user-managed address translation mechanism for network interfaces. By maintaining a user-level lookup data structure and a protected translation table, UTLB avoids system calls in the common path of communication and completely eliminates device interrupts. UTLB allows large communication memory footprints, and requires no special operating system support.

UTLB provides a convenient programming interface to system software developers. Only virtual addresses are used; pinning and unpinning virtual pages are hidden from the users. Direct measurements of three Display Wall applications show that UTLB amortizes page-pinning cost over many communication requests, hence results in low average translation overhead.

We further conducted trace-driven simulations to show

- that UTLB can detect most translation misses at user level and thus avoid interrupts, whereas the interrupt-based approach requires an interrupt on every translation miss,
- that UTLB is less sensitive to the translation table sizes than the interrupt-based approach.

- and that the direct-mapped approach is adequate for implementing the translation table.

Our study has limitations. In particular, the traces are from shared memory parallel programs though they ran in a multiprogramming environment. The memory accesses are quite balanced on all processors. Thus, they may not reveal certain behaviors that a true multi-programming environment displays.

Fast address translation alone is not sufficient for good end-to-end communication performance. The design of the translation caching mechanism and its integration with the communication subsystem must allow applications to send and receive data, without copying, using arbitrary buffers. This goal drove the design of the UTLB.

Cache Entries	Characteristic (per lookup)	Application													
		Barnes		FFT		LU		Radix		Raytrace		Volrend		Water	
		UTLB	Intr	UTLB	Intr	UTLB	Intr	UTLB	Intr	UTLB	Intr	UTLB	Intr	UTLB	Intr
1 K	check misses	0.04	-	0.25	-	0.49	-	0.54	-	0.43	-	0.25	-	0.10	-
	NI misses	0.10	0.10	0.50	0.50	0.50	0.50	0.62	0.62	0.48	0.48	0.31	0.31	0.35	0.35
	unpins	0.00	0.09	0.00	0.49	0.00	0.46	0.00	0.54	0.00	0.41	0.00	0.22	0.00	0.31
2 K	check misses	0.04	-	0.25	-	0.49	-	0.54	-	0.43	-	0.25	-	0.10	-
	NI misses	0.07	0.07	0.50	0.50	0.49	0.49	0.60	0.60	0.46	0.46	0.29	0.29	0.27	0.27
	unpins	0.00	0.04	0.00	0.48	0.00	0.43	0.00	0.44	0.00	0.33	0.00	0.13	0.00	0.21
4 K	check misses	0.04	-	0.25	-	0.49	-	0.54	-	0.43	-	0.25	-	0.10	-
	NI misses	0.05	0.05	0.49	0.49	0.49	0.49	0.57	0.57	0.45	0.45	0.27	0.27	0.12	0.12
	unpins	0.00	0.02	0.00	0.46	0.00	0.37	0.00	0.30	0.00	0.24	0.00	0.07	0.00	0.03
8 K	check misses	0.04	-	0.25	-	0.49	-	0.54	-	0.43	-	0.25	-	0.10	-
	NI misses	0.04	0.04	0.46	0.46	0.49	0.49	0.55	0.55	0.44	0.44	0.25	0.25	0.11	0.11
	unpins	0.00	0.01	0.00	0.40	0.00	0.33	0.00	0.16	0.00	0.14	0.00	0.03	0.00	0.02
16 K	check misses	0.04	-	0.25	-	0.49	-	0.54	-	0.43	-	0.25	-	0.10	-
	NI misses	0.04	0.04	0.38	0.38	0.49	0.49	0.54	0.54	0.43	0.43	0.25	0.25	0.10	0.10
	unpins	0.00	0.00	0.00	0.25	0.00	0.17	0.00	0.09	0.00	0.07	0.00	0.01	0.00	0.00

Table 3.8: Average translation overhead breakdown: UTLB vs. Intr. (infinite host memory, direct-mapped translation cache with cache index offsetting, and no prefetch)

Cache Entries	Characteristic (per lookup)	Application													
		Barnes		FFT		LU		Radix		Raytrace		Volrend		Water	
		UTLB	Intr	UTLB	Intr	UTLB	Intr	UTLB	Intr	UTLB	Intr	UTLB	Intr	UTLB	Intr
1 K	check misses	0.04	-	0.49	-	0.49	-	0.55	-	0.43	-	0.25	-	0.10	-
	NI misses	0.10	0.10	0.50	0.50	0.50	0.50	0.62	0.62	0.48	0.48	0.31	0.31	0.35	0.35
	unpins	0.00	0.09	0.40	0.49	0.33	0.46	0.21	0.54	0.13	0.41	0.00	0.22	0.00	0.31
2 K	check misses	0.04	-	0.49	-	0.49	-	0.55	-	0.43	-	0.25	-	0.10	-
	NI misses	0.07	0.07	0.50	0.50	0.49	0.49	0.60	0.60	0.46	0.46	0.29	0.29	0.27	0.27
	unpins	0.00	0.04	0.40	0.48	0.33	0.43	0.21	0.44	0.13	0.35	0.00	0.13	0.00	0.21
4 K	check misses	0.04	-	0.49	-	0.49	-	0.55	-	0.43	-	0.25	-	0.10	-
	NI misses	0.05	0.05	0.50	0.49	0.49	0.49	0.58	0.57	0.45	0.45	0.27	0.27	0.12	0.12
	unpins	0.00	0.02	0.40	0.46	0.33	0.37	0.21	0.31	0.13	0.28	0.00	0.07	0.00	0.03
8 K	check misses	0.04	-	0.49	-	0.49	-	0.55	-	0.43	-	0.25	-	0.10	-
	NI misses	0.04	0.04	0.50	0.48	0.49	0.49	0.56	0.56	0.44	0.44	0.25	0.25	0.11	0.11
	unpins	0.00	0.01	0.40	0.42	0.33	0.34	0.21	0.23	0.13	0.21	0.00	0.03	0.00	0.02
16 K	check misses	0.04	-	0.49	-	0.49	-	0.55	-	0.43	-	0.25	-	0.10	-
	NI misses	0.04	0.04	0.49	0.47	0.49	0.49	0.55	0.55	0.44	0.44	0.25	0.25	0.10	0.10
	unpins	0.00	0.00	0.40	0.39	0.33	0.33	0.21	0.21	0.13	0.17	0.00	0.01	0.00	0.00

Table 3.9: Average translation overhead breakdown: UTLB vs. Intr. (4 MB host memory, direct-mapped translation cache with cache index offsetting, and no prefetch)

Applications	vis	glaze	vdd	fft	barnes	radix	water
memory footprint	21.4 MB	39.9 MB	78.9 MB	57.4 MB	16.6 MB	54.4 MB	8.9 MB
host lookup hit	99.93%	99.99%	99.99%	79.38%	98.62%	67.36%	99.35%
lookup hit cost	0.65 μs	0.63 μs	2.41 μs	0.32 μs	0.46 μs	0.32 μs	0.40 μs
avg lookup pages	1.00	1.94	54.8	1.00	1.00	1.04	1.00
avg pin ovhd (NT)	54.3 μs	401.6 μs	939.0 μs	47.3 μs	53.1 μs	50.9 μs	55.6 μs
avg pages pinned (NT)	1.00	97.0	289.0	1.00	1.13	1.00	1.25
avg miss ovhd (NT)	63.4 μs	557.1 μs	1294 μs	50.7 μs	57.5 μs	54.6 μs	60.0 μs
predicted ovhd (Linux)	41.7 μs	410.1 μs	882 μs	34.7 μs	40.5 μs	42.0 μs	47.4 μs
avg host ovhd (NT)	0.69 μs	0.69 μs	2.54 μs	10.71μs	1.25 μs	18.04μs	0.79 μs
predicted ovhd (Linux)	0.68 μs	0.67 μs	2.50 μs	7.41μs	1.01 μs	13.92μs	0.71 μs

Table 3.10: UTLB performance without memory constraint

Chapter 4

Runtime Environments

A unique aspect of our scalable display wall system is the use of a PC cluster to drive tiled displays. Its obvious advantage is low cost and close tracking of technology, because high-volume commodity components typically have better price/performance ratios and improve at faster rates than special-purpose hardware. On the other hand, the clustering approach entails complex system software. Unlike a tightly-coupled multi-processor and a uniprocessor system with multiple graphics cards which have mature operating systems to transparently support all kinds of applications, a cluster of PCs is essentially a conglomerate of relatively autonomous computers without a global and unified OS environment. How to bring up a variety of applications on such a system is the challenge addressed in this chapter.

4.1 Introduction

Many applications either play back multi-media content, for example, MPEG movies, HDTV streams, images, and VRML models, or allow users to interactively navigate content such as web pages and Macromedia Flash movies. The content usually conforms to publicly-documented standards. Therefore, it is feasible, though sometimes tedious, to

write special playback software to parse and play back these kinds of content for a cluster environment.

Yet for many other applications, it becomes extremely difficult, if not impossible, to apply the play-back approach. One reason is that the interactive behaviors of these applications are often encoded in the software itself rather than in the raw data, as in a Macromedia movie. An example is a typical 3D game for personal computers. The scenes in the game are described using standard file formats. Manipulation of the scenes and interpretation of user inputs, on the other hand, is often a trade secret and hidden in millions of lines of machine instructions. Another obstacle is the sheer amount of work required to write the playback software that mimics the behavior of the original application. Consider Macromedia Photoshop, a high-quality image editing tool. Many of its functionalities and features are well known and described in textbooks and technical papers. However, to produce a new software package that has similar "look and feel" would require many months of work.

One important objective of the Scalable Display Wall Project is to design a a runtime environment for running and developing applications that we daily use on regular desktop computers, but with much higher intrinsic resolution. This chapter focuses on the issue of developing software tools to bring off-the-shelf, sequential applications to a scalable display wall and run them efficiently at its intrinsic resolution.

Two models of program execution are studied: the *master-slave* model and the *synchronized program execution* model. With the master-slave model, a master node executes an application, intercepts all graphics outputs such as 2-D and 3-D primitives (polygons, lines, points, etc), and sends them to the nodes that drive the tiled projectors for execution. With the synchronized execution model, an instance of the application runs on each of the nodes that drive the tiled projectors. Multiple instances are synchronized and coordinated so that they act as if they are a single application designed for the scalable-resolution display system. The synchronization (or coordination) can be done either at the application level via a

programming API or by the runtime system transparently.

We have designed and implemented four software tools to support these two execution models. They are

- Virtual Display Driver (VDD), which supports 2-D Windows applications using the master-slave model,
- A Distributed GL tool that supports 3-D applications using the master-slave model,
- A runtime system that supports applications using the synchronized program execution model, and
- A system-level tool that supports applications using the synchronized program execution model.

In order to understand the performance implications and resource requirements of each method, I measured several 3-D applications with our software tools on the scalable display wall system. The results show that the master-slave model works reasonably well for applications that send few or no graphics primitives during each frame. The runtime support for the synchronized program execution model achieves similar or sometimes better performance than the master-slave model. The application-level synchronization tool, when source code modification is possible, can achieve much better performance and improve the application response time. Of the two applications that I measured, a factor of two to five of improvement is achieved through application-level synchronization.

4.2 Previous Work

The common way to run digital applications on a video wall is to use the video content scaling hardware (video processor) to drive an application's output on a tiled display sys-

tem. This approach does not allow applications to use the intrinsic resolution of the tiled displays.

Various kinds of tiled display systems have been constructed for data visualization during the past few years. Examples include the Power Wall at the University of Minnesota, the Infinite Wall at the University of Illinois at Chicago [24], the Office of the Future at UNC [74], the Information Mural at Stanford [40], and various immersive products from several vendors. In most cases, an SGI Onyx2 or equivalent high-end machine with multiple graphics pipelines is used to drive multiple projectors. Since these systems are not using a PC-cluster architecture, they do not address the issue of software support for running sequential, off-the-shelf applications on a scalable resolution display wall.

The idea of executing graphics primitives remotely can be found in X windows [78]. The tools such as VDD and GRL described in this chapter leverage and extend these ideas for remote graphics primitive executions for scalable resolution displays. VDD intercepts primitives in a device driver and executes them remotely at the user level whereas GLR substitutes primitives in a local OpenGL library. The mechanisms for remote procedure call [62, 6] and remote execution model [84] have been investigated in the context of programming languages.

The parallel graphics interface designed at Stanford [44] proposes a parallel API that allows parallel traversal of an explicitly ordered scene via a set of predefined synchronization primitives. The goal is to expose parallelism while retaining many of the desirable features of serial programming. The parallel API was not designed to execute multiple instances of a sequential program on a scalable display system built with a PC cluster.

4.3 Existing Desktop Applications

Our first approach to bringing desktop applications onto the display wall is to run the application on a *master* node, intercept the graphics primitives that the application generates, and send them over the network to the render nodes (or the *slaves*). Figure 4.1 shows a conceptual diagram of this master-slave approach. The primitives can be broadcast to all render nodes or, as an optimization, be sent only to the nodes with whose screen tiles they overlap. In either case, a slave node performs view-frustum clipping to render those primitives within its screen tile.

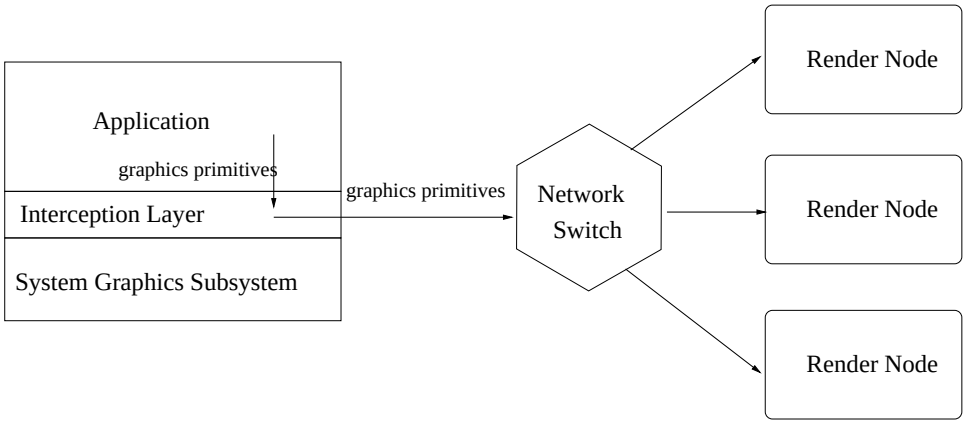


Figure 4.1: Conceptual view of the master-slave approach

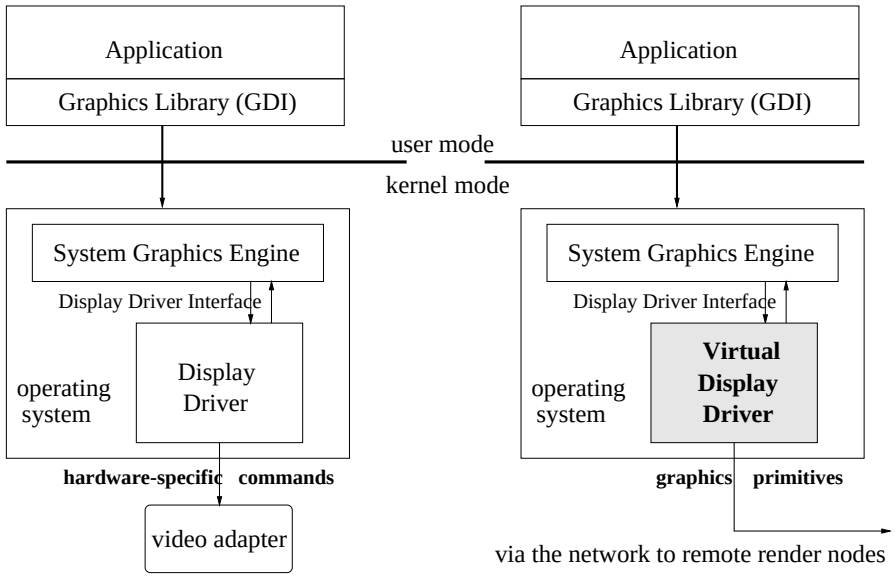
Clipping 2D primitives is straightforward. For 2D vector graphics, this means translation of the 2D coordinates in the global display space to the local coordinates on each render node. The translation is simply an addition of offsets in X and Y directions. The task is even simpler on the Windows platform. One can define a viewport transform to for the graphics device that does exactly this. Windows automatically applies the viewport transform to all subsequent drawings. For bitmaps, the render node selects a portion from the bitmap that falls within its screen tile. Once again, this can be trivially accomplished by specifying the appropriate rectangle to a `bitblt` operation.

To clip 3D primitives, the render node can simply specify a sub-volume of the global

viewing frustum as its viewing frustum. This sub-frustum can be trivially calculated using the screen tile's relative position in the global display space.

Ideally, primitive interception should be transparent to the applications, so that we can run any application binaries and have their graphics shown on the display wall without source modification or re-linking. This means that we inject the interception mechanism in the master node's graphics subsystem that sits beneath the running application. We found two ways to implement the transparent interception mechanism: via a *virtual display driver* and by replacing a dynamically-linked library (DLL).

4.3.1 Virtual display driver



(1) A typical Display Driver architecture (2) The Virtual Display Driver architecture

Figure 4.2: Architectural diagrams of a typical display driver and a virtual display driver (VDD)

In a typical PC operating system such as Microsoft Windows NT and Linux, the graphics subsystem is implemented by a display driver, which speaks a standard protocol to the application layer on one side, and translates the application's graphics commands into

device-specific commands on the other side. The interface between the application layer and the display driver is standardized on a given operating system, so as to allow any graphics accelerator vendors to implement their own vendor-specific drivers. Therefore, we can implement a *virtual* display driver that acts as if it operated a real piece of graphics hardware, but actually takes the graphics commands (or primitives) from the application layer and sends them over the network to the render nodes. Figure 4.2 illustrates the common block structure of a typical display driver and that of a virtual display driver.

An advantage of using such a virtual display driver is that it presents to the applications a large display with intrinsically high resolution. The applications and the operating system normally query the display driver to obtain its screen resolution and adjust the windows, menus, and drawing parameters accordingly. Unconstrained by the actual graphics hardware, the virtual display driver is free to fake a display with an arbitrarily large number of pixels, causing most applications to adapt their drawing resolutions to the high resolution. As a demonstration, we ran Microsoft PowerPoint on our 8-node Display Wall. In its "slide sorter view" mode, PowerPoint placed many slides on the display with clear definition for even the smallest fonts.

Due to the compact nature of the display driver protocol, implementing a virtual display driver is usually not a daunting engineering task. For example, on both the Windows NT 4.0 and Windows 2000 operating systems, the protocol between the 2D drawing layer in the application and the display driver, the *Display Driver Interface* (DDI), consists of no more than 25 commonly used graphics commands for drawing bitmaps, lines, polygons, etc. However, there are still several non-trivial issues in implementing a virtual display driver. The first issue is to translate the graphics objects that are meaningful in the master node's kernel environment to those that can be used by the render nodes. An example of this translation is the drawing surface object that is passed as an argument in most DDI commands to the virtual display driver. The surface object can either identify a bitmap

stored internally in the kernel or the drawing surface.

The second issue is performing *bitblt* across the render nodes. Unlike the bitblt operations on a single PC, the bitblt operations among render nodes transfer pixels across the network. Similar to the operations on a single PC, the order of operations is important. Another method is to refresh from the master node, which is inefficient.

The third issue is how to distribute the graphics primitives efficiently among the render nodes. This issue is common to both virtual display and DLL replacement approaches. We postpone its discussion until Section 4.3.3

One can use the virtual display driver to implement most drawing protocols on the Windows operating system: DDI and DirectDraw for 2D drawing, and Direct3D for 3D rendering. Similarly, we can also implement the X Windows Protocol [78] The X Windows server already embodies the concept of a virtual and networked display. In this case, the virtual X Windows server is situated on the master node, and speaks the X protocol to local applications. It takes and translates the application's X Windows commands and forwards them to the render nodes.

As a final note, the applications running on the master node still receive user inputs through conventional channels such as from a real keyboard and a real mouse. We are experimenting with virtualizing the user inputs as well, so that user interactions with the applications (and hence with the display wall) are no longer restricted to the keyboard and the mouse attached to the master node.

4.3.2 DLL replacement

On modern operating systems, many services no longer reside inside the OS kernel. Instead, they are provided as dynamically-linked libraries (DLL) and are linked into the applications by the OS runtime. An example of this is the `opengl32.dll` on Windows NT/2000 operating systems. This dynamically-linked library contains a default implemen-

tation for OpenGL renderings. OpenGL is an industrial standard for high-performance 3D rendering. It is based upon a stateful client-server interaction model. Major OpenGL commands include those that change and retrieve the states on the server side, for example, setting the model transformation matrix, and those that specify the actual rendering primitives, for example, 3D vertex and color specifications. The default OpenGL implementation in `opengl32.dll` performs most rendering stages on the CPU and then calls Windows' 2D drawing API, GDI, to carry out the final rasterization, i.e., converting a 2D line or 2D polygon to actual pixels on the screen. If the graphics accelerator vendor implements OpenGL in hardware, a vendor-supplied OpenGL DLL is also linked into the application; all OpenGL calls made by the application are forwarded to the vendor-supplied DLL by `opengl32.dll`.

To intercept the OpenGL commands from an application, we can either write a replacement DLL that has a compatible interface with the native `opengl32.dll` or one that is compliant with the OpenGL vendor DLL interface. This latter method is less of a hack than former, and can be well integrated with the virtual display driver.

An advantage of using DLLs to intercept graphics primitives is that communication between the master node and the render nodes occurs at the user level. Not only is it easier to debug this approach, it can also leverage several efficient user-level communication implementations, including the one developed by us [21, 90, 32].

The drawback of this approach is that the API exported by the DLL is typically large, and highly complex in some cases. This is why writing a virtual display driver for 2D Windows applications is a less painful job than writing a corresponding replacement DLL. One exception is the OpenGL API. It is a fairly regular and well thought-out interface. Even though there are between 200 and 300 calls in the API, we were quite successful at automatically generating the bulk of the replacement DLL using a simple parser.

4.3.3 Discussion

A remote display protocol is a general solution. It does, however, have a potential problem in efficient distribution of graphics primitives. An obvious solution is to broadcast the graphics primitives to all render nodes. System-area networks today typically implement point-to-point communication via a switch. Few networks except Ethernet implement broadcast or multicast in hardware. We currently employ a ring topology to broadcast graphics primitives in 64 KB chunks over Myrinet. This method may incur high latency as the number of projectors scales up. Another common broadcast topology is a binary balanced tree. Although its latency is $O(log(N))$, this topology may suffer from poor bandwidth because each network endpoint has to send two copies of each packet out to its children. Efficient broadcast over a point-to-point network still remains an open problem.

Aside from lacking an efficient broadcast mechanism, the primitive distribution speed in the master-slave approach is essentially constrained by the speed of the outgoing network link on the master node. The network interface is typically connected to an I/O bus. Its bandwidth is often an order of magnitude smaller than that of the internal memory bus. For immediate-mode applications that generate 3D primitives for each frame, this means they may run slower on the display wall than on a single machine. The slow-down factor may not be an order of magnitude, because the primitive rendering task is now partitioned among many render nodes according to the screen tiles. But when these immediate-mode applications, typically 3D games, are written for a balanced system where the graphics accelerator's performance matches the bandwidth of the local graphics/memory bus, the relatively slow network link will definitely hamper the rendering rates of the display wall system.

Many retained-mode applications, however, will not suffer from the network bottleneck. These applications largely deal with static scenes whose rendering can be compiled into OpenGL display lists or the like. The master node only pays an up-front cost to send

the rendering commands for each display list to the render nodes. It then can simply issue rendering commands by referencing the display lists. As a result, very little data, other than changes in the viewing position and the movements of the objects, must be sent over the network.

4.4 Synchronized Program Execution

The basic idea in synchronized program execution is to run multiple instances of a program on the display wall render nodes. The execution of these program instances are synchronized at some level, with respect to a synchronization boundary, so that within this boundary these instances assume identical behaviors. As a result of synchronization, the program instances generate identical scene descriptions, which can simply be identical OpenGL 3D primitives across all render nodes or some higher-level scene description that each node instantiates in a tile-specific fashion.

In the first scenario, as depicted in Figure 4.3, the graphics accelerator performs tile-specific culling and only renders those primitives that fall within its respective screen tile. This can be enabled by setting the appropriate view frustum matrix for each render node [34].

In the second scenario, as illustrated in Figure 4.4, the higher-level scene description can be fed into the view-dependent software layer that generates tile-specific primitives. An example of the second scenario is a scene graph render program that organizes the scene data in a hierarchy of objects. Given a tile-specific view frustum, the program can remove the objects that fall completely outside the frustum.

To better illustrate our idea, we partition a program conceptually into two components: scene management and scene rendering. The scene management component interacts with the program's operating environment, for example, reading files and getting the keyboard

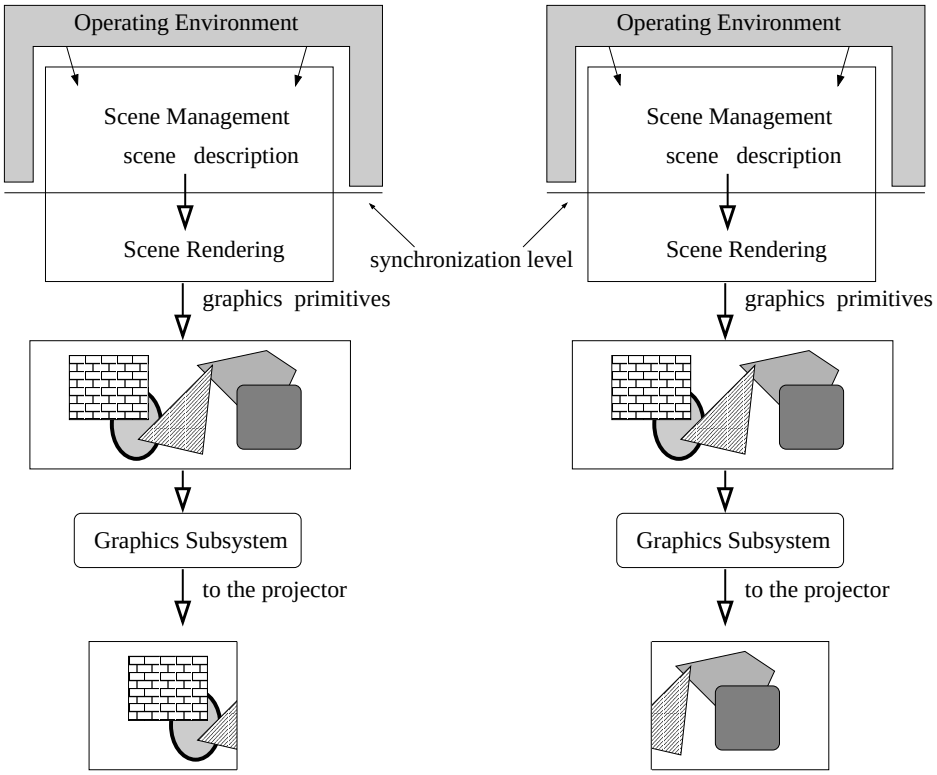


Figure 4.3: Full replication of multiple program instances

and the mouse inputs, and changes its internal behaviors accordingly. In other words, the scene management *responds* to the events in the environment; its behaviors are completely determined by its interaction with the environment. The scene rendering component takes from the scene management layer the scene description, from which it generates the graphics primitives. The picture is even clearer if we take a pipeline view of a program, as illustrated in Figure 4.5. The upper stage of the pipeline, the scene management, is synchronized across all render nodes. Beneath the synchronization level, the scene rendering layer is free to perform any tile-specific tasks, provided its actions *do not alter* the behavior of the scene management layer.

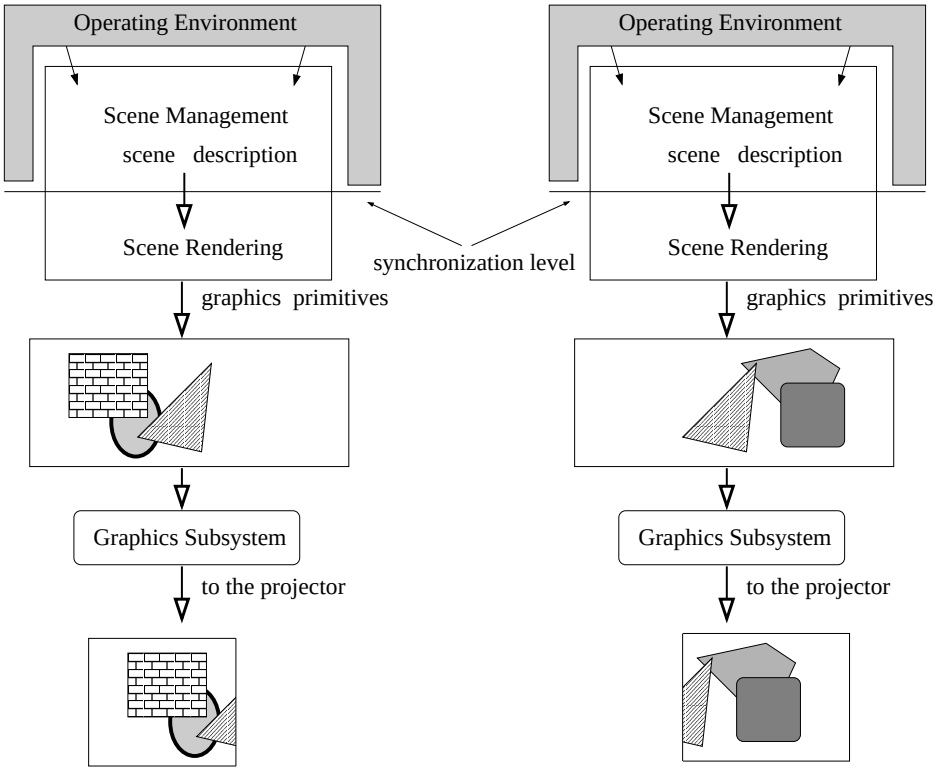


Figure 4.4: Tile-specific primitive generation in application-level synchronization

4.4.1 Synchronization infrastructure

Our program synchronization framework consists of a thin synchronization layer on each render node and a coordinator. The synchronization layer intercepts certain function calls made by the scene management component of the program to interact with the environment. For each intercepted function call, the results from one render node are picked by the coordinator and sent to the other nodes.

Each synchronization of a function call acts like a barrier synchronization. Since the result from only one node is sent back to other nodes, one can use an efficient broadcast topology such as a multi-cast tree to implement a call synchronization. Furthermore, only those function calls that can potentially alter the program behaviors need to be synchronized. They typically include calls to query keyboard and mouse inputs, calls to read the

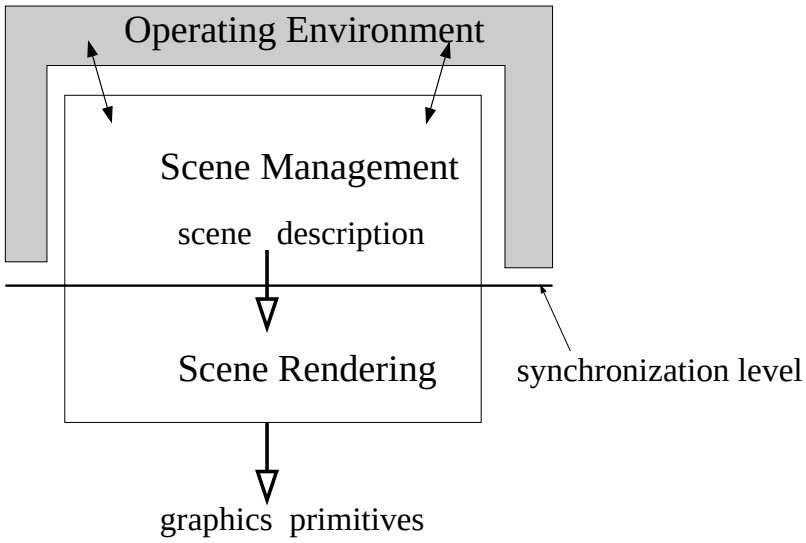


Figure 4.5: The pipeline view of the two program components

system timers, and file I/O operations. Most of these calls have results that are small in size. Hence synchronizing these calls require little communication bandwidth, though low communication latency is vital to keeping the synchronization overhead down. Large file reads can be synchronized with an efficient multicast mechanism such as the UDP-based broadcast mechanism that we describe later in the paper.

The actual placement and implementation of the synchronization layer depends on whether we employ system-level or program-level synchronization.

4.4.2 System-level program synchronization (SSE)

The goal of system-level program synchronization (SSE) is to replicate a program on multiple nodes in a transparent fashion, i.e., without modifying and re-linking the program. We also require that SSE incur very low overhead, even as the number of nodes in the system scales. System-level program replication has been studied in the context of fault-tolerant computing. In Hypervisor, Bressoud et al proposed a method that treats an actual software system as running on a virtual machine [12, 13]. In their framework, two or more such sys-

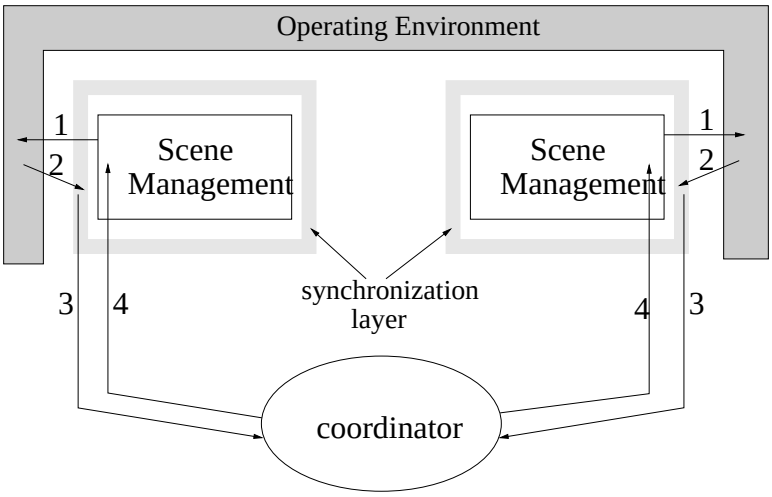


Figure 4.6: The synchronization framework

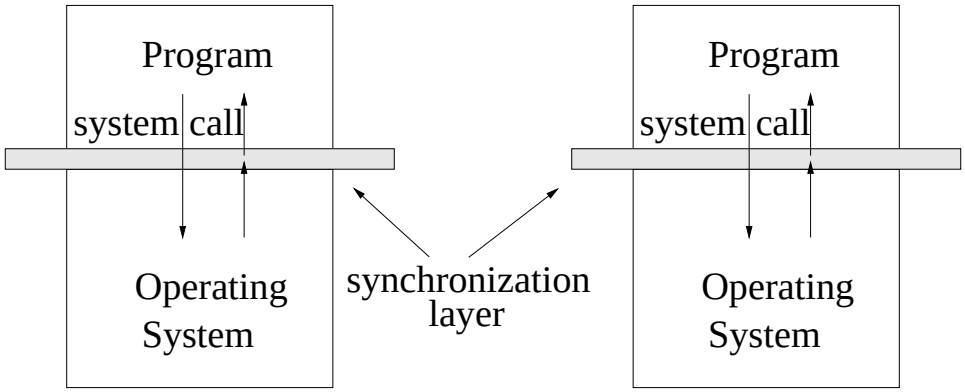


Figure 4.7: Program synchronization at the system-call level

tems can be replicated by synchronizing the run-time semantics of the virtual machine I/O instructions. However, the virtual machine defined in Hypervisor is too close to the actual microprocessor architecture. The fine-grain synchronization causes excessive overhead and slows down the program by as much as a factor of 2.

In order to avoid the high overhead incurred by the Hypervisor approach, we define a virtual machine as the operating system. The macro instruction set of this virtual machine is the system-call API defined on the OS, as shown in Figure 4.7. For a single-threaded program, one can prove that synchronizing at the system call level leads to synchronized program execution. The reason is that code execution is deterministic from the micropro-

cessor architecture's point of view. A single-threaded program's execution path is only influenced by external events. The way these external events affect the program behavior is through the system-call interface and signal handling on UNIX. It follows that if we make the interaction between the program and the OS environment identical on all nodes, the program instances will all follow the same execution path and exhibit the same behavior, and as a result, produce the same graphics primitives.

We have made a few simplifying assumptions in our SSE approach. First, we ignore those programs that interact with the rest of the system via shared-memory segments, because in such cases we have to intercept all reads and writes to the shared-memory segments in order to achieve exact program replication. Second, we assume most programs access the CPU cycle counter via a well-defined API, such as the `QueryPerformanceCounter()` on the Windows platform, so that we can intercept these accesses as well. For programs that use assembly instructions to read the cycle counter, as allowed by the Intel architecture, we could certainly edit the program binary and replace offending instructions with calls to a special handler. In practice, however, this need has not arisen.

Our SSE mechanism does have one limitation: it is not guaranteed to work with arbitrary multi-threaded applications. The problem lies in the fact that it is hard to ensure identical interleaving of threads among multiple program instances. This problem becomes even harder for multi-processor nodes on which several program threads can be running at the same time, because in this case the threads can interact with each other via shared physical memory. To capture such interactions we would have to intercept the load and store instructions, basically going back to the Hypervisor approach.

Among numerous system calls a program makes, only a handful of them can alter program states. They include the calls to query Window messages and the system timer. Therefore, our system-level synchronization layer only performs synchronization for these selected system calls. Our system call synchronization technique works primarily for pro-

grams with a single thread. In theory, it should also work for those multi-threaded programs, in which a single thread interacts with the environment while the other threads simply compute the scenes without affecting the internal states of the program.

4.4.3 Application-level program synchronization (APE)

Program replication at the system call level can transparently synchronize multiple program instances. However, it is not guaranteed to work for multi-threaded programs. In addition, since the system-level approach treats the entire program as a whole, it cannot separate the program into scene management and scene rendering components and let the latter perform tile-specific primitive generation and rendering. We can solve these two problems by moving the synchronization boundary into the application itself.

The same mechanism for system-level program synchronization can be used to synchronize program instances at the application level. Instead of system calls, we synchronize function calls within the application. We currently provide a simple API call, `SynchronizeResult()` to let all render nodes get consistent results. Figure 4.8 illustrates the semantics of a function call synchronization. One can also imagine building a sophisticated interface description language (IDL) that automates the synchronization of user functions.

synchronized fun $F(a_1, a_2, ...) \Longrightarrow$ result :
$tmp \Longleftarrow F(a_1, a_2, ...)$
$SynchronizeResult(tmp)$
$result \Longleftarrow tmp$

Figure 4.8: The semantics of a function call synchronization

To facilitate calculation of the tile-specific view frustum, we also provide two function calls, `QueryDisplayInfo()` and `GetLocalViewFrustum()`. The first call returns a render node's screen position and its node ID in the tuple `(GlobalScreenRectangle,`

`MyScreenRectangle, MyNodeId)`. The second call takes a global view frustum, as defined in a single-node version of the program, and calculates the local view frustum based on the node's tile position in the display wall.

An application can take advantage of application-level synchronization by simply performing high-level object culling based on its local view frustum. Given N projectors in a display wall, each screen tile has only $1/N$ of the the global frustum. Generally this results in a small fraction of objects for each render node to render. Instead of generating all the primitives and letting the graphics accelerator throw out primitives that fall outside the local frustum, the scene rendering component of each program instance can reject a group of graphics primitives or avoid generating them, by comparing their bounding box with its view frustum. Such high-level culling needs much less computation time and consumes far less local bus bandwidth. And it is generally easy to implement. Note that this kind of high-level culling is not possible in the master-slave approach, because the master has to generate primitives for the entire global view frustum.

With more programming effort, one can further reduce the amount of computation required to generate graphics primitives. Many data visualization applications that deal with large data sets belong to this category. One example is an isosurface extraction program developed in our department. An isosurface of a 3D scalar field is made of points, the isopoints, that have the scalar value of a given constant. Extracting isosurfaces is a very useful technique to understand complex volume data that are generated by many medical applications like CT and scientific computations such as astrophysics simulations. These applications often generate scalar fields that are sampled at discrete points in 3D space. The isosurface in these cases is a smooth interpolation of discrete isopoints. Traditional isosurface extraction algorithms extract the whole isosurface. However, the size and complexity of datasets is constantly growing. Due to the high depth complexity and the large dimensions of the dataset, typically only part of the isosurface is visible. *Isoview* is an

isosurface extraction program that tries to extract only the visible portion of the isosurface. The algorithm casts a certain number of rays from the eye through the screen into the dataset and calculates the intersection of the ray and the isosurface. From the intersection point, by exploiting isosurface's continuity property, part of the isosurface can be extracted efficiently.

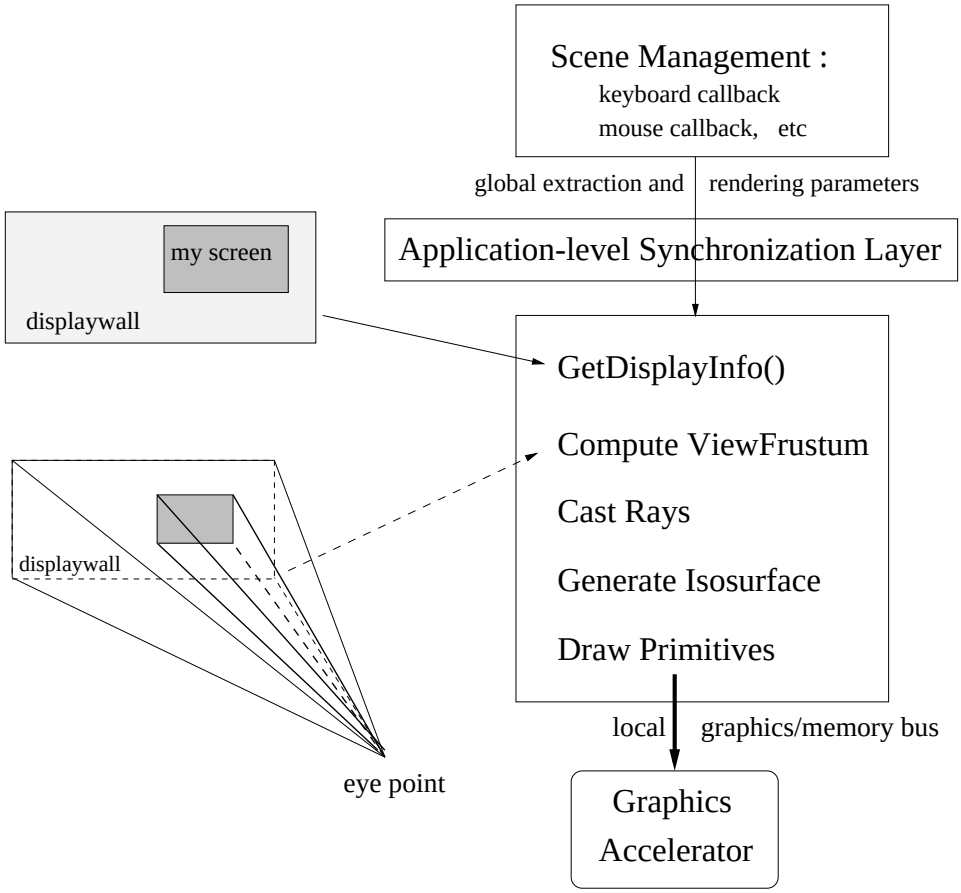


Figure 4.9: The flow diagram of the APE version of isoview

Running IsoView on the Display Wall can give the user more details and information about the data. Since this algorithm is screen-space based, it can be easily parallelized on the Display Wall. We partitioned the program into two components, the management and the extraction. The management component obtains user directives such as mouse and keyboard inputs. It sets the meta state of the program, which is synchronized by a function

call that every program instance makes prior to performing the actual isosurface-extraction algorithm. The extraction component obtains the meta state and a node-specific screen tile position and calculates the perspective projection matrix that is used for both rendering and ray-casting. Data partitioning is automatically obtained through partitioned ray-casting. Figure 4.9 depicts the conceptual flow of the isoview program using application-level synchronization. Very little extra work is required to make the single-node isoview program into a display wall-aware version employing application-level synchronization. It took us only half an hour.

4.4.4 Past work on program replication

Synchronizing multiple instances of a same program has been studied by the fault-tolerant computing community. An early technique relied on a specially-built processor pair that executed instructions in lock steps. The memory accesses by the processors are checked by a special logic located between the processors and the memory subsystem. A third microprocessor was typically included in the mirror pair to provide what is known as triple modular redundancy (TMR). Some commercial vendors extended the hardware mirroring concept throughout the entire computer system to provide very high degree of availability in face of hardware component failures [4, 91]. Synchronizing processor pairs at the instruction level provides a transparent means to replicate program instances; the program need not be rewritten in order to run such a processor pair. However, this technique not only requires expensive engineering efforts, but also faces a tremendous difficulty to keep up with the rapid increase of CPU clocks.

As an alternative, several commercial vendors of fault-tolerant computer systems provide special programming API to synchronize program instances in software. For example, on a Tandem computer system, an application uses messages to communicate with the operating system and other system services. Multiple instances of the application can be

brought to be in sync as long as the messages they send and receive are synchronized, provided that between any two messaging events the application follows a deterministic execution path. A traditional architecture that only supports single-threaded execution guarantees this deterministic property. For a multi-processor multi-thread environment, program synchronization requires a special API at a level higher than messaging and application-specific programming.

Bressoud et al investigated a system-level approach to automatically and transparently replicate programs [12, 13]. Their key insight is that by virtualizing the hardware on which the program and the operating system runs, one can implement the traditional processor-pair approach as mirroring a virtual machine pair. The study by Bressoud et al further demonstrates that only a few I/O instructions in the virtual machine instruction set require synchronization. Their results show that two synchronized computer system using the virtual-machine-pair (or Hypervisor) approach runs at worse twice as slow as one system without synchronization. A significant benefit of the Hypervisor approach is that virtual machine synchronization (or replication) can be done entirely in software, thus can keep close track of the processor technology.

Unlike Hypervisor, our purpose is to synchronize a specific application program instead of the entire system. A Hypervisor-like virtual machine that mimics the actual processor is an overkill and incur too much synchronization overhead, as synchronization is performed on each I/O instruction. Instead, we lift the virtual machine abstraction even higher, and consider the interface between the application and the operating system as the virtual machine abstraction [1]. Synchronizing program instances at such a macro level reduces the synchronization frequency. In our case, it makes synchronization cost almost negligible.

[1] Although we came up with this technique independently, we did later find a U.S. patent about exactly the same technique [11].

4.4.5 A unified view

The two seemingly disparate approaches that I have just described, the master-slave model and the synchronized program execution model, can be really thought of as the two extreme cases of a single approach, that of separating a program into two communicating components. In the former model, the bottom layer that interprets and executes OpenGL or Display Driver commands is yanked out of the host machine, on which the application is running, and placed on remote render nodes; whereas in the latter, the entire program minus the part feeding user inputs, is put on remote render nodes. In either case, some form of synchronization is necessary to keep the components on render nodes running in sync with each others. For the master-slave model, we employed synchronized swapping of frame buffers; and to synchronize multiple program instances, we intercepted and synchronized a few key system calls.

Since as I just said that the two cases represented two extremes, there ought to be other approaches that are amidst the spectrum. The general methodology would be to partition a program into two components such that either communication across the component boundaries is at minimum or (sometimes and) computation loads in both components are balanced. Automatic decomposition of a program is a very difficult task. With a proper representation of the software, for example, a collection of COM objects, each of which has well-defined interfaces, this task might just be achievable. In a different context, Hunt and Scott devised a clever technique to distribute a program, composed of several hundred COM objects, over a network of workstations [42]. Their graph-theory-based algorithm minimizes communication among collections of components. One can imagine that the same technique can be borrowed to partition interactive programs to run on our cluster-based display wall. I believe this is a very promising and interesting research direction.

4.5 Experimental Results

We have implemented all four approaches described in the previous sections: VDD (Virtual Display Driver), GLR (GL-DLL Replacement), SSE (System-level Synchronized Execution), and ASE (Application-level Synchronized Execution). These tools all run on the scalable display wall prototype system described in Section 2. In writing the SSE tool, we used the Detours package from Microsoft Research [41]. It is a binary re-write tool for intercepting any functions in a program or a DLL.

The display wall system has two communication mechanisms available to applications: the Microsoft Winsock over the 100 Mbit/sec Ethernet, and VMMC over the Myrinet. VDD, SSE and ASE all use the Winsock protocol while GLR uses VMMC. The main reason for the difference is that GLR requires a fast communication mechanism to run well. If a 3D application generates a large number of polygons at run time or so called immediate mode in GL, GLR needs high-performance communication paths to the render nodes to send polygons efficiently. The other three tools do not need to transfer polygons among nodes, so they work well with the Ethernet.

We used these tools to run two sets of applications in our experiments. The goal is to understand the tradeoffs of these tools in terms of speed, memory requirements, communication requirements, I/O requirements, and software development efforts. The first set of experiments use VDD, SSE and ASE to run with a set of 2D applications. The second set use GLR, SSE, and ASE to run with a set of 3D applications. The rest of this section reports our experimental results.

4.5.1 2D applications

VDD, SSE and ASE environments support 2D applications. Our experience with the VDD environment is quite positive. Although VDD uses Microsoft's Winsock over the Ethernet

as its communication mechanism, it runs most 2D-primitive based desktop applications on the display wall reasonably well. In the laboratory, we typically use VDD to run applications such as Adobe Photoshop, MS Powerpoint and MS Internet Explorer in the intrinsic resolution of the display wall.

Because of the high resolution and the scale of the display wall, the usage patterns of these 2D applications are different from using a desktop. For example, we noticed that users would like to use the slide sorter of the Microsoft's Powerpoint to look at the entire presentation instead of using the slide show.

These popular applications do not work with SSE and ASE. SSE works only with applications that use a single thread to communicate with the external environment and ASE requires modifying their source codes by adding a line to invoke the ASE environment. These popular programs satisfy neither of the conditions.

We found that the communication requirements vary, but four GDI functions that makes up more than 99% of the calls. The following table shows when Swirl, Powerpoint and Internet Explorer applications run on the wall, for 5, 2 and 2 minutes respective, how many times each of these functions were called:

	BitBlt	CopyBits	TextOut	LineTo
Swirl (Flash app)	30	2,878	8	0
Powerpoint	648	702	35	85
Internet Explorer	9,377	547	2,349	846

The next table shows the average size of data in bytes each call sent over the network:

	BitBlt	CopyBits	TextOut	LineTo
Swirl (Flash app)	488	1,116,470	825	0
Powerpoint	124	44,483	1,571	30
Internet Explorer	218	314,273	1,727	13,284

Whether the VDD approach is a practical approach for running 2D desktop applications on the display wall depends on the communication requirements of applications. For bitmap based applications, the communication requirements for VDD may be too high. For 2D-primitive based applications, the VDD approach works well.

4.5.2 3D Applications and Results

Performance Metrics	Applications/Methods											
	Cars			GLQuake			Atlantis			Isoview		
	GLR	SSE	ASE	GLR	SSE	ASE	GLR	SSE	ASE	GLR	SSE	ASE
frame time (ms)	370	325	-	80.6	65.7	-	147.6	86.2	57.9	20163	16825	4798
sync cost/frame (ms)	-	4.9	-	-	20.0	-	-	2.1	1.8	-	2.0	3.6
sync msgs/frame (count)	-	4	-	-	8	-	-	2	2	-	2	3
data size/frame (KB)	99.2	0.47	-	174	0.82	-	3300	0.24	0.19	46500	0.24	0.48

Table 4.1: The performances of three methods: GL-DLL Replacement (GLR), System-level Synchronized Execution (SSE), and Application-level Synchronized Execution (ASE)

We have selected 4 representative applications: Cars, GLQuake, Atlantis, and Isoview. We chose these applications to cover a wide range of 3D animation characteristics.

- **Cars**: This is a demo program of the WorldUp Toolkit from EAI, Inc. It renders a virtual-reality scene that includes two highly sophisticated car models in an exhibition hall. There are a lot of texture details and fancy lighting in the scene. The cars themselves are made up of 200,000+ polygons each. This makes rendering time the dominant cost. All the objects in the scene are made into OpenGL display lists. The GLR tool needs to send the display lists once at the beginning of the program execution.

- **GLQuake**: This is an OpenGL program to view Quake scenes, downloaded from the Internet. The Quake scene description is in BSP-tree format. By parsing the BSP tree, the program generates polygons in real time. The scene that we use is fairly simple, containing only 12,722 polygons.

- **Atlantis**: This is a demo program from Silicon Graphics, Inc. It simulates a pool of swimming sharks, whales, and a dolphin. The body poses for each object are computed in real time, so it is not possible to use OpenGL display lists to avoid generating polygons for each frame. In our experiment, we increased the number of sharks to 300 to simulate a lively scene with many fish. This results in a large number of graphics primitives for each frame. Furthermore, in the application-level synchronization version of the program, we perform object culling based a coarse bounding sphere for each shark.

- **Isoview**: This is a program to visualize the isosurfaces of volumetric datasets. The program extracts a 3D surface from the data where $F(x, y, z) = v$ for a given threshold v. The program uses a combination of ray casting and isosurface propagation techniques to extract only the visible portions of the isosurface. Since the program allows users to change the view of the isosurface dynamically, it has to generate polygons for each frame at run time. Our experiment uses Isoview to view $256 \times 256 \times 209$, a down-sampled version of the dataset of the head of a visible woman [64]. The visualization of the data typically generates about hundreds of thousands of polygons per frame. With the ASE tool, this program performs high-level object culling.

The following table summarizes the features of the four 3D applications:

Applications	GL features	# Polygons/frame	Source
Cars	display list	200,000	no
GLQuake	immediate	12,722	no
Atlantis	immediate	94,549	yes
Isoview	immediate	645,237	yes

Since the source codes of Cars and GLQuake are not available, we are not able to run them with the ASE tool.

Table 4.5.2 shows the results we gathered from running the four applications with GLR, SSE and ASE on the display wall system. The performance metrics include the average time for each frame, the number of synchronization messages and synchronization overhead (when appropriate), the amount of data sent over the network, and the maximum amount of memory consumed on each render node.

The frame times for each application running with the three tools are also plotted in Figure 4.10.

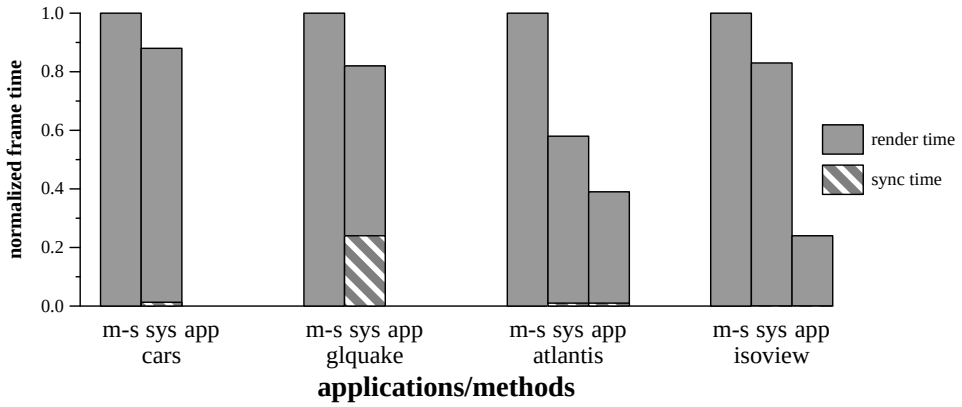


Figure 4.10: Frame time comparison of three methods

Our results show that the application-level synchronized execution approach using the ASE tool is the most efficient approach. For *Atlantis*, ASE is 2.5 and 1.48 times faster than GLR and SSE respectively. For *Isoview*, ASE is 4.2 and 3.5 times faster than GLR and SSE respectively. The second fastest is the system-level synchronized execution approach with the SSE tool. The master-slave approach with the GLR tool is the least efficient among the three.

There are two main reasons why ASE is the most efficient. The first is that its communication requirements are very small, less than 500 bytes per frame for either application. The second is that ASE provides each render node with opportunities to avoid doing duplicated work for other render nodes. To take advantage of this approach, one needs to modify

the application to perform high-level object culling according to the knowledge about the tiled screen space. Our experience with both *Atlantis* and *Isoview* demonstrates that with a convenient mechanism such as application-level synchronization, one can easily modify the application codes. In the *Atlantis* case, the resulting reduction of shark pose computation and graphics primitives averages 50% for each render node. On the other hand, this approach requires accessing application source codes.

The system-level synchronized execution approach with the SSE tool works quite well without having to access application source codes. Similar to the application-level synchronized execution approach, it imposes very little communication overhead in all applications.

The master-slave approach using the GLR tool works reasonably well with applications that use display lists, because polygon data need to be transferred only at the beginning of the program and very little data during each frame. For example, the master node running the *Cars* program sends roughly 99 KBytes worth of data to the render nodes during each frame. Its performance is comparable to that of the system-level synchronization which requires very little communication. We do notice the frame time difference of 45 milliseconds between the master-slave approach and the SSE approach for the Cars program. Its cause is still unclear to us. We are investigating this issue using detailed performance monitor.

The master-slave approach is sensitive to the amount of immediate-mode graphics primitives. With the *GLQuake* program, the GRL tool has about 15ms overhead. This is because the program creates only a relatively small number of polygons per frame which translates to about 174 Kbytes per frame. With *Atlantis* and *Isoview*, the overheads are 61.4ms and 3338.7ms respectively because these two programs require the master node to transfer 3.3Mbytes and 46.5Mbytes per frame respectively. However, the percentage difference in the case of *isoview* is relatively small, because the computation time dominates.

Therefore, our conclusion is that for immediate-mode applications with low computation time per primitive, synchronizing multiple instances at the system level has a clear advantage over the master-slave approach. When per-primitive computation is high, communication time becomes either less significant or completely negligible due to its overlapping with computation.

On the other hand, the master-slave approach is the best in terms of memory requirements. The master node runs a single copy of the application on the master node, whereas SSE and ASE runs an instance of the application on every render node. The application-level synchronized execution approach can have smaller working sets than the system-level synchronized execution approach when it performs high-level object culling.

4.6 Conclusions

In this chapter I described four software tools for bringing off-the-shelf, sequential applications onto a scalable display wall system. We have used these tools to experiment with several 2D and 3D applications on our display wall system to study the scalability issue. What we have learned is that these methods have different tradeoffs in terms of speed, memory requirements and communication requirements.

The master-slave approach does not scale well in performance for applications that generate a large number of primitives dynamically, but it scales well in terms of memory requirements. Our experience with the GLR tool, for example, shows that the approach works reasonably well with 3D applications that use display lists whereas it performs poorly with applications using immediate-mode 3D primitives. Despite of its performance drawbacks, it is quite convenient to use.

The synchronized execution approaches trade memory space for less communication requirements. The system-level synchronized execution approach scales well in perfor-

mance because its communication requirements are minimal. Our experiment with the SSE tool, for example, shows that it performs better than the master-slave approach with all applications. Like the master-slave approach, it requires no source code modifications to applications. Unlike the master-slave approach, the SSE tool works only with applications that use a single thread to communicate with the external environment.

The application-level synchronized execution approach is the most efficient method when one performs high-level object culling. For *Atlantis*, ASE is 2.5 and 1.48 times faster than GLR and SSE respectively. For *Isoview*, ASE is 4.2 and 3.5 times faster than GLR and SSE respectively. By default, this approach takes the same amount of memory as the system-level synchronized execution approach, but by high-level object culling, it can reduce memory requirements. Unlike the system-level synchronized execution approach, this method works with all applications though it does require application source code modifications.

The software tools development described in this chapter is our first step towards understanding how to build scalable display wall software systems. Our studies have several limitations. First, our GLR tool was implemented using the VMMC on Myrinet. Although the comparisons are conservative from the point of view of the synchronized execution model, it would be better if we can port the implementation to Winsock on the same Ethernet.

Chapter 5

Conclusions and Future Work

In preceding chapters, I have described the architecture of a scalable, high-resolution display system and three research issues involved. The philosophy of this architecture is to use commodity components as basic building blocks, and design a software glue layer to integrate them into a single-image, seamless display system. Our prototype display system consists of an array of tiled projectors and a cluster of PCs inter-connected by a high-speed system-area network. The three research issues that were explored in this dissertation are (1) automatic alignment and color balancing of a multi-projector display, (2) efficient address translation for a user-level communication system, and (3) runtime environments for a cluster-based scalable display system.

Seamless tiling In order to make it easy to build a multi-projector display, we developed methods to automatic align and color balance projectors. The automatic alignment algorithm employs an inexpensive video camera to obtain precise, relative measurements and a global optimization technique, simulated annealing, to derive digital corrections. Pixel-level alignment accuracies were achieved on a real, 8-projector display wall. Through simulation, we also showed that subpixel-level accuracies are possible in larger-scale set-

tings. A drawback of our approach is that the computation time for simulated annealing grows rapidly with the number of projectors in the system. A possible remedy, which remains to be experimented, is to parallelize the computation on the PC cluster that drives the projectors.

There is one more promising approach to automatic alignment that this author is interested in exploring as future work: pan a camera across a mis-aligned multi-projector display and a build a panoramic image. There is a body of literature on building panoramas with a commodity, hand-held video camera [88]. Surprisingly, little has been done on applying this technique to digitally aligning the projectors. One advantage this method has over the method described in the dissertation is that it guarantees building a detailed mis-alignment map, which serves as a starting point for calculating digital corrections. There are still challenges, though, in stitching many video images into a highly accurate panorama, for the requirement of precision here is much higher than conventional applications of panoramic imaging. It remains to be seen whether the method proposed will work out.

The result of color balancing is encouraging, to some degree. The video camera is quite capable of measuring the color imbalance among the projectors. My method seems to work fine for some images. Through this work, we also discovered two problems. First, it is hard to obtain precise measurement of its color responses from a projector. We suspect two inherent problems in the project, thermal fluctuations and background noise, are primary causes. With better projection devices such as ones based on DMD, we hope to get "cleaner" sample data. The second problem is that color balancing tends to reduce the dynamic range of each color channel, because of the comprise between dimmest projectors and the brightest projectors. Some projectors have digital "knobs" to adjust their brightness and black levels for red, green, and blue separately. We can adapt our algorithm to first bring the luminance range of all the projectors to the same levels and then apply color

corrections. This approach will have minimal impact on the resulting dynamic ranges.

Fast communication Efficient communication is at the heart of a cluster-based display system. In the past, software overhead caused excessive loss in performance on a piece of communication hardware that was an order of magnitude more efficient than the software. User-level communication is one way eliminate excessive software overhead from the common path of communication.

User-managed Translation Look-aside Buffer (UTLB) provides a portable, efficient solution for translating virtual addresses in a user-level communication paradigm. Direct data transfer between virtual memory address spaces is a powerful and convenient communication model. Fast, user-level address translation is necessary to make this model work well. My work on UTLB demonstrates that such a scheme can be designed and implemented for a modern-day, commodity network interface such as Myrinet. I also gave thorough evaluation of UTLB in a multi-programming environment and with memory constraints, using both empirical data and simulation study.

The study on address translation is only one part of improving communication in the rendering cluster. Efficient multi-cast and reliable communication are just two more examples of desirable features in a cluster environment. Efficient multi-cast is critical for distributing graphics primitives within the cluster; while reliable communication allows the system to tolerate temporary failures of some nodes. At the same time, higher-level protocols must be so designed to take advantage of reliable communication that they can reconfigure the system to bypass failed nodes. All these remain interesting research issues and are relevant for bringing scalable display systems to the market.

Runtime environments The chapter on scalable runtime environments provides a taxonomy of execution models for running applications on a cluster-based display system. In the master-slave model, a master PC intercepts and distributes graphics primitives to the PCs

driving the physical displays. This model can be integrated seamlessly into the runtime system of an off-the-shelf operating system. It can transparently support a wide variety of applications without requiring source code modification and re-compilation. However, the master-slave model suffers from the network bottleneck in some cases. My answer to this problem is the synchronized program execution model, which runs multiple program instances synchronously on the render PCs. In this model, only very little communication occurs, except for small messages to synchronize a few key system calls. This model was implemented for the Windows NT operating system and on top of fast Ethernet. It improved the performance of several 3D applications by 5% to 50%. Further evaluation is needed to compare this model with the master-slave model for 2D applications.

Process synchronization at the system call level only guarantees correctness for single-threaded applications. For multi-threaded applications, thread scheduling and inter-thread communication must also be synchronized in order for multiple applications instances to exhibit the same behavior. One can first study how to synchronize multi-threaded applications on a uni-processor as the first step, because in this case only thread scheduling needs to be considered.

With maturing technologies for producing low-cost, large-area, thin film display materials, immersive and high-resolution displays will become affordable and in wide use. Recognizing this inevitable trend, we can pursue research in two directions. The first direction is to make high-resolution displays more accessible. This entails work not only in material science, for example, improving the life-time, brightness, color quality, and manufacturing process of several promising thin-film display technologies such as Organic LED, but also in making the best out of current technology, for instance, multi-projector displays.

The work described in this dissertation focused primarily on issues in the second re-

search direction. Using portable presentation projectors and a cluster of PCs, one can construct a high-resolution display system with a reasonable cost, for example, under $200,000 for our 8-projector display wall, that can fit nicely within budget plans of many scientific research projects.

The second direction involves studying content creation issues for a high-resolution immersive displays, because such a display offers a brand-new medium for presenting information. It is visually pleasing merely to increase the number of 3D polygons in a scene or render models at super-fine quality. The personal and intuitive feel that a large-size, high-resolution display offers opens up a much wider avenue for exploration: intuitive computer-user interaction, immersive data exploration, and collaborative work space, to name just a few. One should also keep in mind that there is a practical limit to human visual acuity, that "zooming-in" is a necessity and has seen enormous success ever since the invention of the telescope (and later microscope). It is very much an open question how to integrate seamlessly zoomed-in details with immersive views. Strategic and aesthetic use of the large space available is a very interesting challenge that has far-reaching ramifications in a decade when large displays become popular.

Bibliography

[1] Giganet network interfaces and switches. http://www.giganet.com.

[2] Universal display intellectual property. http://www.universaldisplay.com/intell.html.

[3] R. Azuma and G. Bishop. Improving static and dynamic registration in an optical see-through hmd. In *Proceedings of SIGGRAPH 94*, pages 197–204, July 1994.

[4] J. F. Bartlett. A nonstop operating system. In *Proceedings of the 11th Hawaii International Conference on System Sciences*, volume 3, 1978.

[5] A. Basu, M. Welsh, and T. von Eicken. Incorporating memory management into user-level network interfaces. In *Presentation at IEEE Hot Interconnects V*, Aug. 1997. Also available as Tech Report TR97-1620, Computer Science Department, Cornell University.

[6] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, Feb. 1984.

[7] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. A virtual memory mapped network interface for the SHRIMP multicomputer. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 142–153, Apr. 1994.

[8] M. A. Blumrich, R. D. Alpert, Y. Chen, D. W. Clark, S. N. Damianakis, C. Dubnicki, E. W. Felten, L. Iftode, K. Li, M. Martonosi, and R. A. Shillner. Design choices in the

shrimp system: An empirical study. In *Proceedings of the 25th Annual Symposium on Computer Architecture*, June 1998.

[9] M. A. Blumrich, C. Dubnicki, E. Felten, and K. Li. Protected, user-level dma for the shrimp network interface. In *Proceedings of the Second International Symposium on High Performance Computer Architecture*, Feb. 1996.

[10] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, Feb. 1995.

[11] T. Bressoud, J. Abhern, K. Birman, R. Cooper, B. Glade, F. Schneider, and J. Service. Transparent fault tolerant computer system. United States Patent 5,968,185, Oct. 1999.

[12] T. C. Bressoud and B. B. Schneider. Hypervisor-based Fault-tolerance. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles (ASPLOS V)*, pages 1–11, Copper Mountain Resort, Colorado, Dec. 1995. ACM.

[13] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 14(1):80–107, Feb. 1996.

[14] J. Brustoloni and P. Steenkiste. Effects of buffering semantics on i/o performance. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 1996.

[15] G. Buzzard, D. Jacobson, M. Mackey, S. Marovich, and J. Wilkes. An implementation of the hamlyn sender managed interface architecture. In *Proceedings of the Operating Systems Design and Implementation Symposium*, Oct. 1996.

[16] P. Cao, E. Felten, and K. Li. Application-controlled file caching policies. In *USENIX Summer 1994 Technical Conference*, pages 171–182, June 1994.

[17] P. Cao, E. W. Felten, A. Karlin, and K. Li. A study of integrated prefetching and caching strategies. In *Proceedings of the ACM SIGMETRICS*, May 1995.

[18] Y. Chen, D. Clark, A. Finkelstein, T. Housel, and K. Li. Automatic alignment of high-resolutino multi-projector displays using an un-calibrated camera. In *Proceedings of IEEE Visualization 2000*, Salt Lake City, Utah, Oct. 2000.

[19] Y. Chen, D. Clark, A. Finkelstein, and K. Li. A method to align high-resolution multi-projector displays using an uncalibrated camera. Technical report, Princeton Unversity, Computer Science Department, Mar. 2000.

[20] Y. Chen, S. N. Damianakis, S. Kumar, X. Yu, and K. Li. Porting a user-level communication architecture to nt: Experience and performance. In *3rd Usenix Windows NT Symposium*, July 1999.

[21] Y. Chen, C. Dubnicki, S. N. Damianakis, A. Bilas, and K. Li. Utlb: A mechanism for translations on network interface. In *The 8th International Conference on Architectural Support for Porgramming Languages and Operating Systems (ASPLOS-VIII)*, Oct. 1998.

[22] D. R. Cheriton. The unified management of memory in the v distributed system. Draft, 1988.

[23] Compaq/Intel/Microsoft. *Virtual Interface Architecture Specification, Version 1.0*, Dec. 1997.

[24] C. Cruz-Neira, D. J. Sandin, and T. A. DeFanti. Surround-screen projection-based virtual reality: The design and implementation of the cave. In J. T. Kajiya, editor,

Computer Graphics (SIGGRAPH '93 Proceedings), volume 27, pages 135–142, Aug. 1993.

[25] C. Dalton, G.Watson, D. Banks, C. Calamvokis, A. Edwards, and J. Lumley. Afterburner. *IEEE Network*, 7(4):36–43, 1995.

[26] S. N. Damianakis. *Efficient Connection-Oriented Communication on High-Performance Networks*. PhD thesis, Dept. of Computer Science, Princeton University, May 1998. Available as technical report TR-582-98.

[27] P. Druschel, B. S. Davie, and L. L. Peterson. Experiences with a high-speed network adapter: A software perspective. In *Proceedings of SIGCOMM '94*, pages 2–13, September 1994.

[28] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 189–202, December 1993.

[29] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. Shrimp project update: Myrinet communication. *IEEE Micro*, 18(1):50–52, Jan. 1998.

[30] C. Dubnicki, A. Bilas, Y. Chen, S. N. Damianakis, and K. Li. Vmmc-2: Efficient support for reliable, connnection-oriented communication. In *IEEE Hot Interconnects V*, Aug. 1997.

[31] C. Dubnicki, A. Bilas, K. Li, and J. Philbin. Design and implementation of virtual memory-mapped communication on myrinet. In *Proceedings of the 1997 International Parallel Processing Symposium*, 1997.

[32] T. Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 256–266, May 1992.

[33] R. et al. Multi-projector displays using camera-based registration. In *IEEE Visualization '99*, 10 1999.

[34] J. Foley, A. van Dam, S. K. Feiner, and J. Hughs. *Simulated Annealing Methods*, chapter 0, pages ???–??? Addison-Wesley, 2 edition, 1996.

[35] R. Gillet, M. Collins, and D. Pimm. Overview of network memory channel for pci. In *Proceedings of the IEEE COMPCON '96*, pages 244–249, 1996.

[36] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, 2nd Ed*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1996.

[37] D. S. Henry and C. F. Joerg. A tightly-coupled processor-network interface. In *Proceedings of 5th International Conference on Architectur al Support for Programming Languages and Operating Systems*, pages 111–122, Oct. 1992.

[38] M. D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance*. PhD thesis, Unversity of Berkeley, 1987.

[39] M. Homewood and M. McLaren. Meiko CS-2 interconnect elan – elite design. In *Proceedings of Hot Interconnects '93 Symposium*, Aug. 1993.

[40] G. Humphrey and P. Hanrahan. A distributed graphics system for large tiled displays. In *IEEE Visualization '99*, 1999.

[41] G. Hunt and D. Brubacher. Detours: Binary interception of win32 functions. In *The 3rd USENIX Windows NT Symposium*, pages 135–143, Seattle, Washington, July 1999.

[42] G. Hunt and M. Scott. The coign automatic distributed partitioning system. In *Third Symposium on Operating Systems Design and Implementation*, pages 187–200, New Orleans, Lousiana, Feb. 1999.

[43] L. Iftode, J. P. Singh, and K. Li. Understanding application performance on shared virtual memory. In *Proceedings of the 23rd Annual Symposium on Computer Architecture*, May 1996.

[44] H. Igehy, G. Stoll, and P. Hanrahan. The design of a parallel graphics interface. In *ACM 1998 SIGGRAPH*, 1998.

[45] Intel Corporation. *Pentium Processor Data Book*, 1993.

[46] D. Johnson and W. Zaenepoel. The peregrine high-performance rpc system. *Software: Practice and Experience*, 23(2):201–221, Feb. 1993.

[47] V. Kindratenko. Compute vision guided cross-projector color alignment on multi-projector displays. In *The Fourth International Immersive Projection Technology Workshop*, June 2000.

[48] S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[49] N. Kronenberg, H. Levy, and W. Strecker. Vaxclusters: A closely-coupled distributed system. *ACM Trans. Comput. Syst.*, 4(2):130–146, May 1986.

[50] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The stanford flash multiprocessor. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 302–313, Apr. 1994.

[51] E. Lantz. Future directions in visual display systems. *IEEE Computer Graphics*, pages 38–42, May 1997.

[52] P. Leach, P. Levine, B. Douros, J. Hamilton, D. Nelson, and B. Stumpf. The architecture of an integrated local network. *IEEE Journal on Selected Areas in Communications*, SAC-1(5), 1983.

[53] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 148–159, May 1990.

[54] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The stanford dash prototype: Logic overhead and performance. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, May 1992.

[55] K. Li, H. Chen, Y. Chen, D. W. Clark, P. Cook, S. Damianakis, G. Essl, A. Finkelstein, T. Funkhouser, A. Klein, Z. Liu, E. Praun, R. Samanta, B. Shedd, J. P. Singh, G. Tzanetakis, and J. Zheng. Early experiences and challenges in building and using a scalable display wall system. *Computer Graphics and Applications*, 220:671–680, 2000.

[56] C. Liao, M. Martonosi, and D. W. Clark. Performance monitoring in a Myrinet-connected SHRIMP cluster. In *Proc. of 2nd SIGMETRICS Symposium on Parallel and Distributed Tools*, Aug. 1998.

[57] R. Lipton and J. Sandberg. Pram: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, Sept. 1988.

[58] A. Majumder, H. Towles, and G. Welch. Color matching of projectors for multi-projector displays. In *submitted for publication*, 2000.

[59] T. Mayer. New options and considerations for creating enhanced viewing experiences. In *Computer Graphics*, pages 32–34, May 1997.

[60] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equation of state calculation by fast computing machines. *Journal of Chemical Physics*, 21:1087–1092, 1953.

[61] L. R. Monnerat and R. Bianchini. Efficiently adapting to sharing patterns in software DSMs. In *Proceedings of 4th International Symposium on High-Performance Computer Architecture*, Feb. 1998.

[62] B. J. Nelson. *Remote Procedure Call*. PhD thesis, Carnegie-Mellon University, May 1981.

[63] S. Nugent. The iPSC/2 direct-connect communication technology. In *Proceedings of 3rd Conference on Hypercube Concurrent Comput ers and Applications*, pages 51–60, Jan. 1988.

[64] N. I. of Health. Electronic imaging: Report of the board of regents, u.s. department of health and human services, public health, national institutes of health. NIH Publication 90-2197, 1990.

[65] S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois fast messages (fm) for myrinet. In *Proceedings of Supercomputing '95*, 1995.

[66] J. Palmer. The NCUBE family of high-performance parallel computer systems. In *Proceedings of 3rd Conference on Hypercube Concurrent Computers and Applications*, pages 845–851, Jan. 1988.

[67] J. Peddie. *High-Resolution Graphics Display Systems*. Windcrest/McGraw-Hill, New York, 1994.

[68] P. Pierce. The Paragon implementation of the NX message passing interface. In *Proceedings of Scalable High-Performance Computing Conference (SHPCC) 94*, 1994.

[69] C. A. Poynton. *A Technical Introduction To Digital Video*, pages 6–7. John Wiley and Sons, Inc., New York, 1996.

[70] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Simulated Annealing Methods*, chapter 10, pages 444–455. Cambridge University Press, 2 edition, 1992.

[71] S. A. Przybylski. *Cache and memory Hierarchy Design: A Performance-Directed Approach*. Morgan Kaufmann Publishers, 1990.

[72] R. Raskar. Immersive planar display using roughly aligned projectors. In *IEEE Virtual Reality 2000*, Mar. 2000.

[73] R. Raskar, M. Cutts, G. Welch, and W. Stuerzlinger. Efficient image generation for multiprojector or multisurface displays. In *9th EuroGraphics Rendering Workshop*, 1998.

[74] R. Raskar, G. Welch, M. Cutts, and A. Lake. The office of the future: A unified approach to image-based modeling. In *SIGGRAPH 98*, pages 179–188, July 1998.

[75] S. Reinhardt, J. Larus, and D. Wood. Tempest and typhoon: User-level shared memory. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 325–336, Apr. 1994.

[76] R. Samanta, A. Bilas, L. Iftode, and J. P. Singh. Home-based svm protocols for smp clusters: Design and performance. In *Proceedings of 4th International Symposium on High-Performance Computer Architecture*, 1998.

[77] R. Samanta, J. Zheng, T. Funkhouser, J. Singh, and K. Li. Load balancing for multi-projector display systems. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, 1999.

[78] R. W. Scheifler and J. Gettys. The x window system. *ACM Transactions on Graphics*, 5(2):79–109, Apr. 1986.

[79] D. R. Schikore, R. A. Fischer, R. Frank, R. Gaunt, J. Hobson, and B. Whitlock. High-resolution multiprojector display walls. *Computer Graphics and Applications*, 20:38–44, 2000.

[80] I. Schoinas and M. D. Hill. Address translation mechanisms in network interfaces. In *Proceedings of 4th International Symposium on High-Performance Computer Architecture*, 1998.

[81] M. D. Schroeder, A. D. Birrell, M. Burrows, H. Murray, R. M. Needham, T. L. Rodeheffer, E. H. Satterthwaite, and C. P. Thacker. Autonet: a high-speed, self-configuring local area network using point-to-point links. *IEEE Journal On Selected Areas in Communication*, 9(8):1318–1335, Oct. 1991.

[82] S. L. Scott. Synchronization and communication in the t3e multiprocessor. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–36, Oct. 1996.

[83] J. Smith and C. Traw. Giving applications access to gb/s networking. *IEEE Network*, 7(4):44–52, July 1993.

[84] A. Z. Spector. Performing remote operations efficiently on a local computer network. *Commun. ACM*, 25(4):260–273, Apr. 1982.

[85] G. Stoll, B. Wei, D. Clark, E. Felten, K. Li, and P. Hanrahan. Evaluating multi-port frame buffer designs for a mesh-connected multicomputer. In *Proceedings of 22nd International Symposium on Computer Architecture (ISCA)*, pages 96–105, Santa Margherita, Italy, June 1995.

[86] R. Surati. *A Scalable Self-Calibrating Technology for Large Scale Displays*. PhD thesis, MIT, 1999.

[87] C. V. Systems. Comview viewscreen and viewmaestro product sheets. *http://www.comview-vs.com*.

[88] R. Szeliski and H. Shum. Creating full view panoramic image mosaics and environment maps. In *Computer Graphics (SIGGRAPH '97 Proceedings)*, pages 251–258, 1997.

[89] E. R. Tufte. *Visual Explanations: Images and Quantities, Evidence and Narrative*, pages 444–455. Graphics Press, Cheshire, Connecticut, 1 edition, 1997.

[90] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th Annual Symposium on Operating System Principles*, pages 40–53, Dec. 1995.

[91] S. Webber and J. Beirne. The stratus architecture. *21st Int. Symp. on Fault-Tolerant Computing (FTCS-21)*, pages 79–85, 1991.

[92] B. Wei, D. Clark, E. Felten, K. Li, and G. Stoll. Performance issues of a distributed frame buffer on a multicomputer. In *Proceedings of 1998 Eurographics/SIGGRAPH Workshop on Graphics Hardware*, Lisbon, Spain, Aug. 1998.

[93] B. Wei, C. Silva, E. Koutsofios, S. Krishnan, and S. North. Visualization research with large displays. *Computer Graphics and Applications*, 20:50–54, 2000.

[94] G. Welch, H. Fucks, R. Raskar, H. Towles, and M. S. Brown. Projected imagery in your office of the future. *Computer Graphics and Applications*, 20:62–67, 2000.

[95] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. Methodological considerations and characterization of the splash-2 parallel application suite. In *Proceedings of the 22nd Annual Symposium on Computer Architecture*, May 1995.

[96] Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In *Proceedings of the Operating Systems Design and Implementat ion Symposium*, Oct. 1996.