Scalable and Ultra-High Resolution MPEG Video Delivery on Tiled Displays

Han 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

November 2003

Abstract

Video is a powerful means for people to communicate with each other across space and time. Its effectiveness is largely dependent on resolution and scale. Unlike most other information technologies, video resolution was increasing slowly during the past two decades. One of the major limiting factors was display resolution, which has been improving only at a rate of about 5% per year, while processor performance, memory density, network bandwidth and disk storage capacity are doubling every 18 to 24 months as predicted by Moore's Law. To overcome the resolution limit, researchers have recently proposed ways to construct multi-projector tiled display systems driven by commodity PC clusters. However, ultra-high resolution videos have not been available on such displays for several reasons. First, video resolution has been limited to the HDTV standard. Second, it has been difficult to parallelize MPEG video decoders to run on a cluster due to limited communication performance in the typical architecture. Third, to appear seamless, ultra-high resolution videos require accurate projector calibration, which is difficult to achieve on large-scale tiled displays.

This dissertation presents a framework for decoding and displaying ultra-high resolution videos on a scalable multi-projector tiled display system driven by a PC cluster. It proposes a classification of three kinds of scalable resolution videos for tiled displays. First, a single ultra-high resolution video can be used for applications such as digital cinema or planetarium. Second, in remote scientific visualization, multiple low-resolution videos can be tiled to create scalable video resolution. Third, for immersive tele-presence applications, a high-resolution video can be formed by overlaying multiple low-resolution videos with different fields of view.

Towards realizing this framework, this dissertation presents the design, implementation and evaluation of several key components necessary for building a scalable

MPEG decoding system for tiled displays. The first component, a high-performance software MPEG video decoder, is used as the underlying building block for the scalable decoding system. Secondly, we propose a hierarchical parallel algorithm to decode and display ultra-high resolution MPEG streams on a PC cluster. Finally, sub-pixel accurate projector alignment is achieved with the camera homography tree algorithm; improved color consistency is attained using a new full-gamut color matching system.

With these implementations, the system is able to accomplish a demanding task—it plays an IMAX quality, 3840 by 2800 pixel video at about 39 frames per second on a tiled display driven by a cluster of 21 PCs.

Acknowledgments

This dissertation is but a small first step in my pursuit of knowledge and truth as a computer science researcher. It nonetheless represents a milestone in my life. I have many people to thank for helping me achieve this.

First and foremost, my deepest gratefulness goes to my advisor Kai Li. For six years, Kai has helped me stay focused and offered me encouragement and guidance through the odyssey. I would not have gone this far if not for Kai. Being a leader of many projects, Kai has a keen insight and vision in both acamedic and industrial researches. It would be my ultimate success if I could learn and benefit from them.

I would also like to thank all my other thesis committee members: Doug Clark, Tom Funkhouser, JP Singh, Szymon Rusinkiewicz, and Adam Finkelstein. I was very fortunate to have Doug and Tom as my readers. They have given me invaluable feedback and made my dissertation logically and structurally sound.

I have collaborated with many other researchers on my dissertation topics. Being my internship mentor at AT&T lab, Bin Wei helped me in many aspects of the development of both the uni-processor and parallel MPEG video decoders. My discussions with Minerva Yeung and Yen-Kuang Chen of Intel Microprocessor Research Lab proved helpful too. It was my privilege to work with Rahul Sukthankar and Tat-Jen Cham at Compaq CRL on the scalable alignment system. For the last three years, Grant Wallace has helped me in developing, testing, and publishing of many projects; I really can not thank him enough. Finally, kudos to Ben Shedd, whose courses have turned the otherwise dull lab into a refreshing art gallery periodically.

I am grateful to our graduate coördinator Melissa Lawson for assisting me in navigating through the entire PhD process. She has made life much easier for me, actually for all of us graduate students. The display wall project would not have been

successful without support from our excellent technical staff members: Chris Tengi, Jim Robert, Joe Crouthamel, Chris Miller, Steve Elgersma, and Brian Jones.

My parents, Ruitao Chen and Xiaoli Xie, deserve my utmost gratitude. It is they who gave me my life. It is they who fostered and cherished me as a child. It is they who sparked my curiosity in science and nature. And it is they who supported me through all twenty plus years of education. Although our times together were short, I am lucky enough to have a brother, Yi Chen, with whom I shared many common interests. I am also indebted to my girlfriend and fiancée, Ying Zhang, who has made my life rich, complete, and enjoyable.

Being one of the few students who have stayed in the same office for six years, I thank my officemates: Scott Karlin, Angelos Bilas, Lisa Worthington, Tassos Viglas, Ting Liu, Nitin Garg, Limin Wang, Elena Nabieva, and Subhash Khot. Thanks to fellow graduate students and friends: Yuqun Chen, Stef Damianakis, Georg Essl, Allison Klein, Zhiyan Liu, Emil Praun, Rudro Samanta, George Tzanetakis, Jiannan Zheng, and Jie Chen, Mao Chen, Aki Nakao, Xiang Yu. Thanks also to my roommates Lintao Zhang and Zhijie Shi. Finally, I enjoyed all the tennis with Xiaohu Qie.

A large project like the display wall is not possible without the generosity of our funding agencies. The project is supported by Department of Energy under grant ANI-9906704 and grant DE-FC02-99ER25387, by National Science Foundation under grant CDA-9624099, grant EIA-9975011, and grant EIA-0101247, by NCSA under grant ACI-9619019 (through NSF), by Intel Research Council and Intel Technology 2000 equipment grant. An early stage of the decoder reseach was supported in part by AT&T Labs-Research. I am also honored to be a Gordon Wu fellow for four years.

To my parents

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Among all the media, video is one of the most powerful ways to convey information across both space and time. The effectiveness of a video depends largely on its resolution[1] and scale, as most people who have been to an IMAX film [38] or an immersive virtual reality system, such as a CAVE [22, 21], can attest. The last four decades have seen the exponential growth of almost every aspect of computing technologies. Microprocessor performance, memory density, network bandwidth, disk storage capacity have been improving at a rate predicted by the Moore's Law [66], that is, doubling every 18 to 24 months. However, display resolution is among the few exceptions that did not enjoy this growth rate; it has been improving about 5% a year. This is due to the physical difficulty in manufacturing larger size CRT tubes, LCD/Plasma panels, or DLP chips [3, 65, 64, 91].

Researchers have recently proposed ways to construct tiled displays driven by commodity PC clusters [59, 31, 80, 93, 78, 33]. A typical projector-based tiled dis-

[1]Resolution has two distinct meanings. It can refer to the number of lines in the vertical direction and number of dots in the horizontal direction of a digital image or video frame. It can also refer to the number of lines/dots per unit length, as measured by *Dot Per Inch* (dpi). Throughout this dissertation, unless otherwise stated, we refer to the first meaning of resolution.

play consists of a large back projection screen[2], a number of projectors forming an rectangular array behind the screen, a cluster of PCs each driving one projector, and the necessary networking equipment, I/O devices, file server, etc.

Three properties of such tiled displays are immediately noticeable. First, they deliver very high resolution imageries, often in the range of several thousand pixels wide and high. This is made possible by the large number of projectors in the system. Second, projector-based tiled displays offer extremely large display surface area, usually measured in feet in both directions. Third, the cluster architecture provides excellent scalability for the display resolution. Additionally, projectors allow the physical size of the display to be scaled up or down by adjusting their throw distances.

With the display resolution limit finally being shattered by such scalable, large-scale, high-resolution display systems, tremendous opportunities exist for creating scalable ultra-high resolution[3] video delivery systems using these tiled displays.

With opportunities come challenges. First, due to historical and technical reasons digital videos produced so far have been limited to the High Definition Television (HDTV) resolution. Second, building a parallel video decoder on PC clusters has been difficult due to the limited network performance in typical setups. Third, to appear seamless, ultra-high resolution videos require accurate geometric alignment, photometric balancing, and color matching among projectors.

This dissertation addresses these issues by first presenting a framework for scalable video delivery. We then describe three major components used for a scalable ultra-high resolution video decoding system for tiled displays.

[2]Front projection is also possible, when the number of projectors are relatively small. For example, a flight simulator may use three projectors with a front projection screen.

[3]Before the advent of tiled display systems, one or two mega-pixel displays or videos qualify for high resolution. For example, an Ultra Extended Graphics Array (UXGA) display has a resolution of 1600×1200, the highest resolution HDTV is 1920×1080. In this dissertation, we are interested in videos with resolution beyond that of HDTV. Sometimes it is measured in tens of millions of pixels; without a more proper name, we call this ultra-high resolution.

1.1 Framework for Scalable Video Delivery

Low display resolution has limited the video resolution in the past. Now that large scale ultra-high resolution displays become available, we propose a general framework for scalable and flexible video delivery. As an end-to-end system, it includes the following components: source, encoder, channel, decoder, and display.

We classify video sources into four types and video encoding into three classes. The four video sources are single camera, single computer, multiple cameras, and multiple computers. The three classes of video encodings are the following. First, a single ultra-high resolution video can be used for applications in digital cinemas or planetariums. Second, in remote scientific visualization, multiple low-resolution videos can be tiled to create scalable video resolution. Third, for immersive tele-presence applications, a high-resolution video can be formed by overlaying multiple low-resolution videos with different fields of view.

The resulting combinations of video source and video encoding allow a system builder to tailor the video delivery system to the targeted application's need.

Chapter 2 presents a detailed description of the components in the framework. It discusses all possible source/encoding combinations, including their advantages, disadvantages, and applications. It also proposes a unified representation scheme for all types of videos in the framework.

1.2 High Performance Software Video Decoding

It is rather obvious that to decode ultra-high resolution videos on a tiled display requires a parallel decoder suitable for a cluster architecture. The road to this ultimate parallel decoder starts from a more humble uni-processor version. We chose to develop

a software-based uni-processor video decoder for several reasons. First, a software decoder gives access to all functions and states, allowing us to experiment with many different parallelization methods. Second, software decoders are more cost-effective and can ride on the rapid growth of the underlying hardware.

We use the reference design from the MPEG Software Simulation Group (MSSG) as the basis for our software video decoder. Our goal is to optimize it to achieve maximum performance, and in the meantime keeping its high-level structure intact to facilitate later parallelization.

There have been many previous efforts to improve video decoding performance by the means of Multimedia Instruction Set Extension to traditional general purpose processors. As we will see in Chapter 3, by aggressively using these multimedia instructions, we can readily improve the reference decoder from MSSG by a factor of two. The resulting decoder sports frame rates close to those of the state-of-the-art commercial products.

However, further improvement is hard to come by with multimedia instructions alone, because as our study indicates, the decoder has become mostly memory bound. To alleviate the memory bottleneck, we propose several data structural and procedural optimizations to the decoder [15]. Our experiments show that these techniques can further improve this multimedia instruction optimized decoder by another factor of two. This high performance video decoder forms the basis of our scalable video delivery system.

1.3 Scalable Parallel Video Decoding

With the high performance uni-processor video decoder handy, we set out to design our parallel video decoder. Like any parallel systems, we want this decoder to be

scalable. To be more specific, we want to achieve resolution scalability, that is, it should decode higher resolution videos at real time frame rate with more nodes.

The challenge of delivering ultra-high resolution videos on a cluster-based system is two-fold. First, to decode such videos requires a large amount of computation. Second, to distribute the pixels to the displays requires very high communication bandwidth. These call for a carefully designed parallel MPEG video decoder, which can distribute the computation evenly among all cluster nodes, while minimizing communication for pixel redistribution.

We studied previous functional and data driven parallelizations of MPEG video decoders. As Chapter 4 points out, functional parallelization is suitable for multi-processor computers with shared memory, but it does not map to cluster architectures well. Simple data parallelizations also fail to scale on a cluster; a fine-grained parallelization usually results in computation bottlenecks, while a coarse-grained one causes communication log-jams.

Our solution is a novel hierarchical parallel MPEG video decoder [14]. It is a concatenation of a coarse-grained, picture-level parallelizer and several fine-grained, macroblock-level parallelizers. It preserves the low communication requirement of a fine-grained parallelizer, while eliminating its computation hot-spot by using multiple of them simultaneously. Experiments show that the decoder scales to very high resolution videos on very large tiled displays—it plays an IMAX quality, 3840 by 2800 pixel video at about 38.9 frames per second on a tiled display driven by a cluster of 21 PCs.

This parallel video decoder in itself does not handle all the encoding schemes presented in the framework. It nevertheless forms a solid foundation where new functionalities can be easily added to fully realize a scalable and flexible video delivery system on tiled displays.

1.4 Seamless Video Display

Although a projector-based tiled display is a relatively easy and inexpensive way to create a large-scale high-resolution display, this does not come without a cost. As we will discuss in much more detail in Chapter 5, without proper projector calibrations, a tiled display will exhibit severe distortions; the quality of images or videos is thus greatly reduced. There are three major causes of visual artifacts in a projector-based tiled display. First, mis-aligned adjacent projectors create overlaps and discontinuities. Second, differences in the bulbs and imperfections in the optics generate both inter- and intra-projector brightness non-uniformity. Third, differences in imaging technologies, color filters, and bulbs result in color variations among projectors.

For static content, small artifacts of an uncalibrated display may very well be blended into the images themselves. This means that a casual calibration often yields satisfactory results. However, moving objects in a video makes even tiny artifacts glaring, because the human visual system is surprising good at tracking moving objects and separating them from a static distortion field. Therefore, to present a high resolution video seamlessly requires a high precision calibration of the tiled display in the following three aspects.

Geometric Alignment. Geometric alignment is needed to remove discontinuities caused by mis-aligned projectors in the overlapped regions. Several vision-based software solutions [19, 77, 96] were proposed to address this problem. Single camera-based algorithms suffer from scalability problems when the resolution of a tiled display far exceeds that of the camera. On the other hand, the multi-view algorithm proposed in [19] does not explicitly exploit the projective geometry available in planar tiled displays; it resorts to Simulated Annealing for solving a large scale non-linear optimization problem and sometimes suffers

from slow convergence when the initial manual alignment can not be sufficiently approximated by simple translations and scales. We propose a multi-view algorithm that exploits 2D homographies to achieve sub-pixel accurate geometric alignment in a scalable way [16].

Luminance Balancing. Even with sub-pixel accurate geometric alignment, luminance imbalance within and among the projectors can still cause obvious and severe visual artifacts. Majumder *et al.* proposed the use of *Luminance Attenuation Map* (LAM) to equalize the luminance output across a tiled display [63].

Color Matching. Finally, commodity projectors typically exhibit different color characteristics even for units of the same model. The problem is often worse, when a tiled display contains mixed vendor/model projectors. The latest DLP projectors complicate the problem further with the use of White Enhancement [52], which makes their color gamuts non-additive. In the past, researchers have proposed ways for color matching tiled displays [62, 85]. But these methods usually assume chromaticity constancy and an additive gamut. Thus, they only work for a homogeneous array of LCD projectors from the same manufacturer. We present a non-parametric color gamut matching algorithm for displays made of DLP projectors or mixed vendor/technology projectors [92].

As we will describe in Chapter 5, a complete calibration system generates correction information in the form of a mesh for geometric alignment, an alpha mask for luminance balancing, and a color map for color matching. With the latest programmable graphics technology, they can be combined with the maps already present in the unified representation scheme, and executed in real-time for full frame videos.

1.5 Thesis Contribution

The main contributions of this dissertation are:

- We propose a framework for scalable video delivery on tiled displays. It classifies the video sources into four types, and the video encoding methods into three classes. This gives system designers a great variety of choices to build applications ranging from immersive tele-presence to digital cinema. We also present a unified representation method of the three classes of videos.

- We design a high performance uni-processor video decoder. It is based on the open source reference decoder from the MPEG Software Simulation Group. We propose optimization techniques to achieve high performance video decoding with minimal modifications to the decoding algorithm, making it easy for later parallelization. We accomplish a two fold performance increase on top of known SIMD instruction optimizations by exploiting the concurrency among processor, memory, and graphics port.

- We propose a scalable high performance parallel MPEG video decoder for PC clusters. By using two levels of splitters, it can eliminate both computation and communication bottlenecks in a data driven parallel decoder. The parallel decoder is able to decode and play ultra-high resolution videos at real-time frame rate on a commodity tiled display.

- We design a scalable geometric alignment system and a full gamut color matching system as two major components of a complete projector calibration system for tiled displays. We propose a novel homography tree algorithm that is able to incorporate multiple camera views to achieve sub-pixel accurate geometric

alignment. Simulation shows that it scales to tiled displays containing hundreds of projectors. The color matching system tackles the non-additive gamut problem of commodity single-chip DLP projectors by using a non-parametric color matching model. Experiments show that it can match DLP projectors or mixed DLP/LCD projectors to within 1.5% color consistency. Finally, we also describe how to implement real-time imagery correction using programmable graphics hardware.

We have implemented this scalable video decoding system on a real tiled display, and have demonstrated that it is both scalable and versatile.

Chapter 2

Framework for Scalable Video

Unlike most other information technologies, which have enjoyed an exponential growth for the past several decades, display resolution has largely stagnated. Low display resolution has in turn limited the resolution of digital videos. Now that large-scale, high-resolution displays have become available, it opens up opportunities for scalable high resolution digital video delivery. Here, we propose a general framework for scalable and flexible video delivery.

It is an end-to-end system encompassing the following five major components from source to destination:

Source. The source generates the input raw pixels to form a video.

Encoder. The encoder compresses the raw video into a more manageable bitstream.

Channel. Compressed video streams are sent via a channel to the decoder.

Decoder. The decoder receives the bitstream and restores it to raw pixels.

Display. Finally, the pixels are shown on a display to be seen by the viewers.

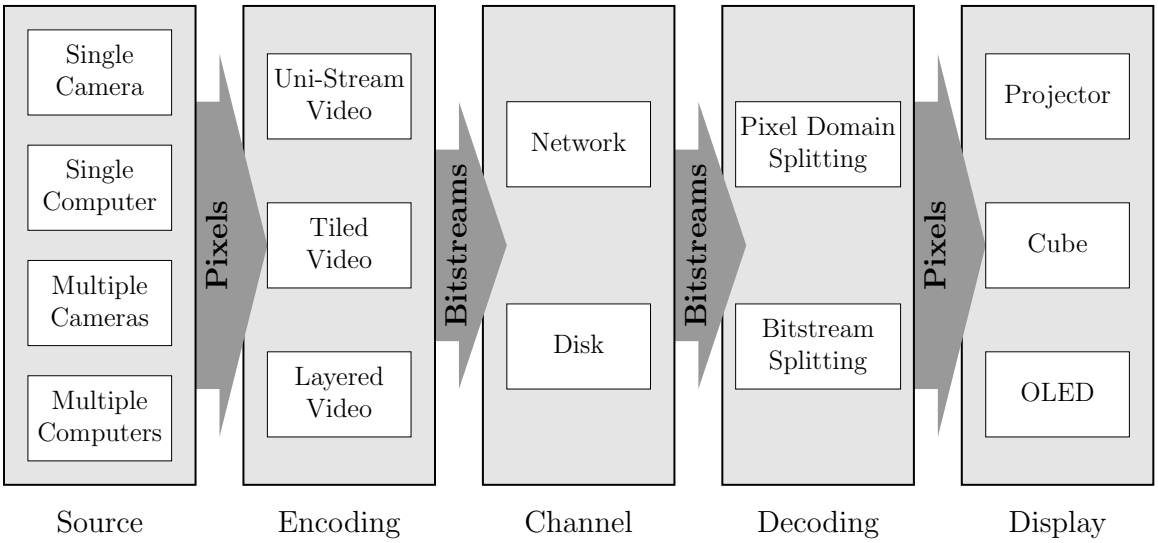


Figure 2.1: Framework of an End-to-end Video Delivery Pipeline. Each stage of the pipeline contains multiple design alternatives shown in small white boxes.

2.1 Design Choices for the Framework

This framework provides system designer a variety of options in building a scalable video delivery system. As Figure 2.1 indicates, there are multiple alternatives for each component, which will be the topics for the subsequent subsections.

2.1.1 Video Source

A video can come from various sources. In most cases, the source of video can be classified into two major categories: natural images from cameras and synthetic images rendered by computers.

Commodity video cameras typically have limited resolutions. To build a scalable resolution video delivery system, we need to consider using multiple cameras to achieve greater resolution. The same also applies to computer graphics—limited rendering capability of a single PC calls for a cluster-based parallel rendering archi-

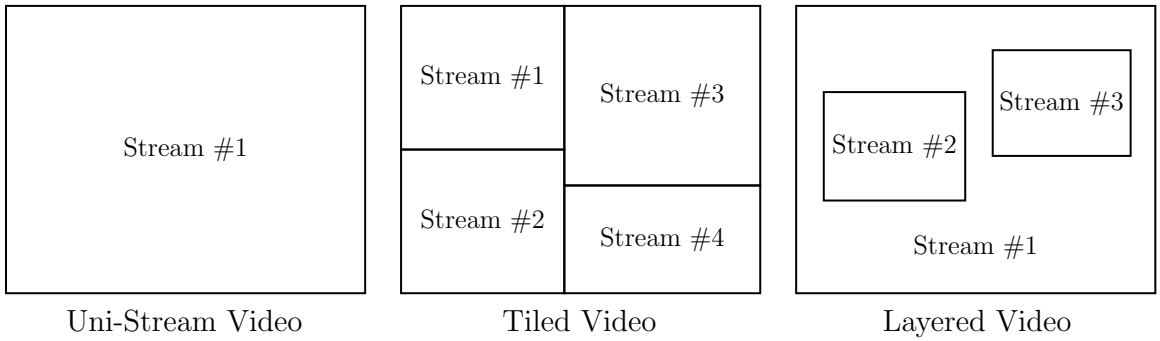


Figure 2.2: Three Classes of Video Encoding.

tecture. This gives us four possible sources of digital videos: single camera, multiple cameras, single computer, and multiple computers.

An additional source of digital video is from the scanning of existing film stocks. Films were usually shot with one camera; the resolution of the digital scan depends on the film format and scanner resolution. Therefore, we consider it to be covered by the case of "single camera", albeit its resolution is typically much higher than that of a commodity video camera.

2.1.2 Video Encoding

We propose three classes of video encoding methods for a scalable video delivery system, namely, Uni-Stream Video, Tiled Video, and Layered Multi-Resolution Video, as shown in Figure 2.2.

Uni-Stream Video. This video consists of only a single compressed stream. It is the simplest among all three classes. When there is enough network bandwidth, imaging device performance, and video encoding speed, one can encode an ultra-high resolution video in a single stream, thus achieving the highest possible visual quality. Applications that benefit from ultra-high resolution uni-stream videos include digital cinema, planetarium, high-end simulator, etc.

Tiled Video. When imaging device performance and/or video encoding speed are not sufficient for generating uni-stream videos, one can use multiple video sources, either cameras or computers. Each source is responsible for generating a portion of the video frame, and the final video is the result of tiling these smaller parts. Conceptually, this is just like a high resolution tiled display that is created from an array of low resolution individual displays. These sub-videos are encoded independently and the collection of them forms what we call a tiled video. Typical applications of tiled video include low-cost high resolution camera arrays and cluster-based remote rendering.

Layered Multi-Resolution Video. The large scale and high resolution offered by a tiled display are ideally suited for immersive tele-presence applications. However, sending full frame ultra-high resolution video across the Internet is not feasible due to its bandwidth requirement. On the other hand, it is not necessary either—although the human visual system has a wide field of view, only a small surrounding region of the *fovea centralis* has high visual acuity.

This observation leads to the notion of *Layered Multi-Resolution Video*[1], a compromise between visual quality and bandwidth requirement. Such a video consists of a collection of sub-videos stacked together, each of which has a different field of view, spatial resolution (in the dpi sense), and/or even temporal resolution [30]. For example, in tele-conferencing, two video layers may be used—a low-resolution video covers the whole conference room, while a higher-resolution one focuses on the speaker.

With four possible video sources, this classification creates 12 possible combinations. An in-depth description and analysis of them will be presented in Section 2.2.

[1]For succinctness, this will be referred to just as "Layered Video" in the following text.

2.1.3 Transmission Channel

Encoded video bitstreams are sent to the decoder across space and/or time via the transmission channel. In the spatial case, it is usually a network link between the encoder and the decoder. In the temporal case, this is typically a storage device, such as a disk drive.

For video delivery, some of the most pertinent and important characteristics of the transmission channel are its available bandwidth, latency, and jitter. Local area network or disk offers very high bandwidth, low latency and low jitter. Therefore, they can be used in conjunction with all three kinds of video encoding. On the other hand, wide area network, such as the Internet, provides only limited bandwidth, relatively high latency and jitter, making uniformly high resolution content impractical. In this case, layered video is more suitable.

However a system chooses its transmission channel, it should make sure that multiple streams arrive at the decoder at about the same time to minimize the end-to-end latency. This is especially important for real-time applications such as teleconferencing. It can be achieved by packetizing multiple streams and interleaving them in a transport stream.

Detailed treatment of the transmission channel is outside the scope of this dissertation. We limit our discussion to this subsection.

2.1.4 Video Decoding

The decoding stage restores a compressed video stream into its raw pixel format, which will later be shown on the display. In order to bring ultra-high resolution videos to the users, a video decoder must harness all available processing powers in a system. Video decoding schemes can be roughly classified into the following types:

Pixel Domain Splitting

In the simplest design, one decoder is used to decompress the incoming video stream. The decoded frames are split into tiles, with their boundaries matching those of the projectors. These sub-frames are then sent to the display nodes.

This approach is conceptually simple, and easy to implement. However, due to the extremely high computation and bandwidth requirement of decoding and sending the videos, it is only practical for relative low resolution videos. For example, sending a modest 720p HDTV video[2] at 60 frames per second requires

$$1280 \times 720 \times 12 \times 60 = 663.6 \quad \text{Mbits/s},$$

which is close to what Gigabit Ethernet links can typically provide.

Pixel Domain Splitting with Re-encoding

A small improvement can be made to reduce the bandwidth requirement of the previous approach, and it is used in early efforts to play high resolution videos on tiled displays. In this decoding scheme, an input video stream is decoded and split in the pixel domain as described before. The resulting sub-videos are then re-encoded into compressed forms. A collection of independent decoders runs on the nodes; each decodes one video stream and synchronizes at frame buffer swapping.

This approach reduces the bandwidth and/or storage requirement, and is still easy to implement. Very few modifications are needed to synchronize multiple decoders. The preprocessing stage can also be implemented using any existing video decoder and encoder. Thus, this used to be a quick way of bringing high resolution video contents to a tiled display.

[2] Assuming 4:2:0 video, one pixel requires 12 bits. Higher quality videos need more bits per pixel.

However, the disadvantage of this approach is obvious. First of all, the preprocessing stage requires decoding, splitting, and re-encoding a high resolution video. This is a very time consuming process, especially the re-encoding stage; therefore, except for very low resolution videos, it is usually done offline. This makes any online or streaming video system impossible. Secondly, the splitting of video into tiles is predetermined based on the configuration of the tiled display. When the display configuration changes, one has to re-encode the video, making this solution extremely inflexible. Finally, the visual quality of a video may be reduced after the decompression-recompression cycle.

Compressed Domain Splitting

A further improvement to the decoding system can be achieved by splitting the incoming video stream in the compressed domain. Such a decoding system consists of a coordinator and multiple decoders. The coordinator partitions an input video stream into small chunks of work units, and sends them to the decoders. The coordinator and the decoders form a truly parallel video decoder.

The most important distinction between this approach and the previous ones is where the splitting takes place. A compressed domain splitter runs much faster, because there is no need for re-encoding. This can typically be done in real time with a proper implementation. Therefore an online or streaming video system is possible. Further, the coordinator can easily adjust the assignment of work units according to the display configuration, making this method flexible. Finally, because the splitting is done in compressed domain, there is no quality loss.

The down side is the requirement for a parallel software decoder, which will be one of the main focuses of this dissertation and discussed in details in Chapter 4.

With the introduction of true parallel decoders, the pixel domain splitting scheme has become a historic relic. Its extremely limited capability makes it impractical for real uses. We only present this method here for completeness' sake. For the rest of this dissertation, we will consider it an obsolete method, and focus only on true parallel decoding with compressed domain splitting.

2.1.5 Display Technology

Due to manufacturing difficulties it is very hard to provide ultra-high resolution on a single display device. When this is possible at all, such as the QUXGA-W (3840×2400) resolution IBM T221 LCD monitor, the limitation in cable signal bandwidth makes it necessary to drive this display with four DVI inputs, essentially turning it into a miniature tiled-display-in-disguise [51]. Even if the signal bandwidth problem is solved in the future, it will still be very difficult to produce wall-size display devices in a single piece. Therefore, we believe that that tiled display is, and will be for the near future, the only feasible and economical way of building a large-scale, high-resolution display.

In a typical embodiment of a tiled display, an array of projectors is arranged behind a rear-projection screen to form a large display area. Alternatively, rear-projection cubes can be stacked to form an array. In the future, one can imagine tiling multiple flat *Organic Light Emitting Diode* (OLED) [88] devices to form such a display.

One unique aspect of projector-based tiled display is the need for projector calibration. This includes geometric alignment, photometric balancing (including edge blending), and color matching. Without proper calibration, a projector array can exhibit artifacts that severely reduce the quality of a video. Cube-based displays eliminate the alignment and blending requirement, but it is hard, if not impossible,

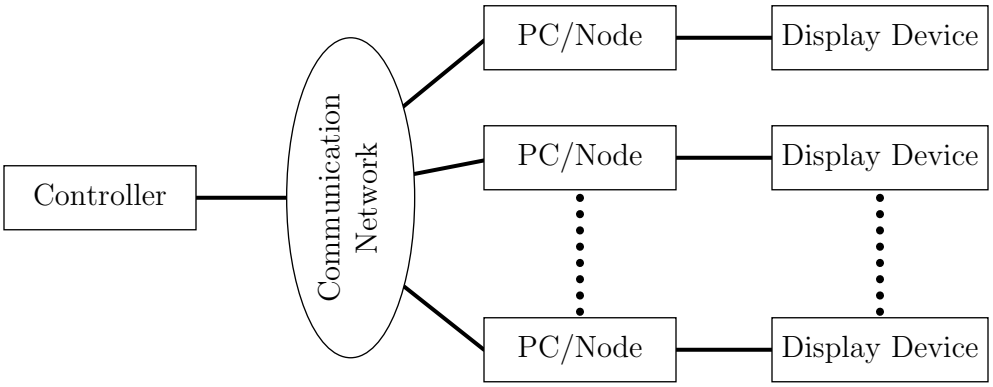


Figure 2.3: A Generalized Tiled Display

to form a truly seamless display using cubes due to the width of the framing material. Furthermore, a cube array may still need careful color and brightness balancing. OLED offers a potentially calibration-free solution, but it is unclear whether it will suffer from the same "seam" problem of cubes. Due to its ubiquity and seamlessness, we will only consider projector-based tiled displays in this dissertation. From this point on, they are simply referred to as tiled displays, unless otherwise noted.

Regardless of the underlying display technology employed for each tile, it has to be driven by some computing device with a graphics or video output. This device may take many different forms, such as, a full fledged PC, a thin-client, a set-top box, etc. For the purpose of this dissertation and without loss of generality, it can be considered to contain some amount of processing power, memory space, graphics or video capability, and network connectivity. For the sake of clarity, we simply call it a PC or a node. Figure 2.3 depicts such a generalized tiled display system.

The particular tiled display system that we use to perform experiments is the Princeton Scalable Display Wall from the Computer Science Department of Princeton University [59, 17]. In its second generation, this display system consists of 24 Compaq MP1800 portable DLP projectors. The projectors are arranged in a 6-wide by 4-high

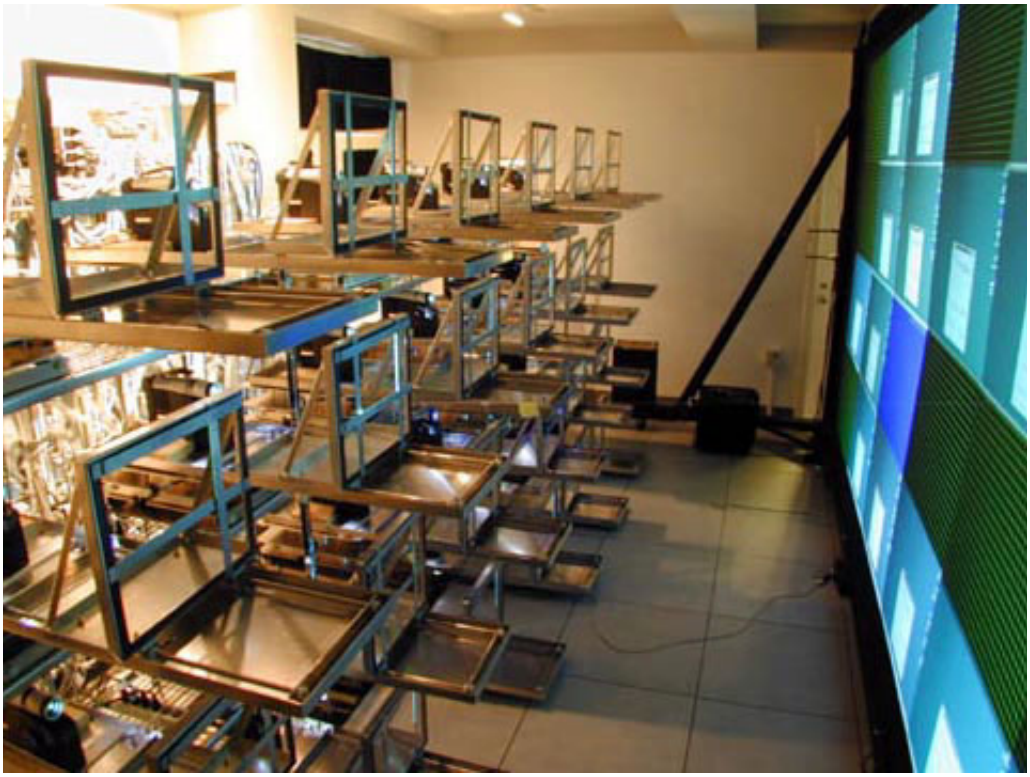


Figure 2.4: A rear view of the Princeton Scalable Display Wall showing 24 portable DLP projectors projecting onto a Jenmar Blackscreen.

array behind a Jenmar Blackscreen, as shown in Figure 2.4. Each projector has a native XGA (1024×768) resolution. There is roughly a 40-pixel overlap between adjacent projectors for edge blending. Thus, the overall effective resolution of the display is 6000×3000. Each projector is driven by a PC with a 733 MHz Pentium III processor, 256 MB of RDRAM, and an NVIDIA GeForce2 GTS graphics card. A 25th PC is used as the "console" to control the tiled display. It has two 550 MHz Pentium III processors with 1 GB SDRAM. There are also various other PC's in the cluster, such as a file server, a sound server, an input server, etc. They will be ignored for our discussion in this dissertation. All PC's are connected together via two networks—a 100 Mbits/s FastEthernet and a 1 Gbit/s Myrinet [11].

Similar projector-based tiled display systems are also built in various other universities and laboratories, such as Lawrence Livermore National Laboratory [80], AT&T Labs Research [93], University of North Carolina at Chapel Hill [78], Argonne National Laboratory [33], to name just a few.

2.2 Discussion of the Framework

The combination of four types of video sources and three classes of video encoding methods creates 12 possible choices for designing a video delivery system. Each of these methods represents a design trade-off among processing speed, visual quality, bandwidth requirement, etc. Each option has its own advantages and disadvantages, as well as its unique challenges for a system builder. None of them can fit the needs of all video delivery systems, and some of them are not particularly interesting or even useful at all. However, as a whole, they cover most common applications in communication, scientific visualization, entertainment, etc. Table 2.1 summarizes these combinations, followed by detailed discussions.

2.2.1 Uni-Stream Video

Single Source

Creating a uni-stream video from a single source is relatively straightforward. First frames are sampled from an imaging device, rendered by a computer, or scanned from existing films. These frames are then fed through a video encoder to generate one compressed bitstream.

The first advantage of this combination is its simplicity—one source, one encoder, and one output stream. Secondly, compressing the entire frame in one video stream

Table 2.1: Combinations of Video Sources and Encoding Methods.

	Uni-Stream Video	Tiled Video	Layered Video
Single Camera or Single Computer	**Advantages:** Simplicity, good compression	**Advantages:** Quick way to bring video to tiled displays	**Advantages:** Simple optical design, no need for registration.
	Disadvantages: For high resolution videos, it requires high resolution imager, high computation power, and high network bandwidth.	**Disadvantages:** Time consuming preprocessing, additional storage, and inflexibility. Primarily used with synchronized decoding.	**Disadvantages:** Potentially limited spatial resolution.
	Applications: Video conferencing, television, cinema	**Applications:** Early attempts in bringing videos to tiled displays	**Applications:** Immersive tele-presence, virtual reality.
Multiple Cameras	**Advantages:** No need for high resolution cameras	**Advantages:** No need for a parallel encoder.	**Advantages:** Fits human visual system naturally, low bandwidth requirement.
	Disadvantages: Additional computation for registration	**Disadvantages:** Extra processing at decoder, requires proximity of optical centers and synchronized focusing	**Disadvantages:** Requires proximity of optical centers, synchronized focusing.
	Applications: Low cost camera array.	**Applications:** Low cost camera array.	**Applications:** Tele-presence, attentive displays.
Multiple Computers	**Advantages:** Utilizes the processing power in multiple PC's	**Advantages:** Simplified encoder design.	**Advantages:** Fits human visual system naturally, low bandwidth requirement.
	Disadvantages: Requires a parallel encoder.	**Disadvantages:** May require a coarser grained load balancer	**Disadvantages:** Typically only works for a single viewer.
	Applications: Remote rendering or visualization.	**Applications:** Remote rendering or visualization	**Applications:** Virtual reality, attentive display.

allows the encoder to achieve a better compression ratio—this means higher visual quality at the same bit-rate, or lower bit-rate with the same quality.

There are three major disadvantages of this approach. First, for high resolution videos, a high resolution camera is needed. The cost of HDTV resolution video cameras is very high, and higher resolution ones are simply non-existent. The second disadvantage is the high computation requirement for compressing high resolution videos. It is hard to achieve real-time MPEG compression on commodity PC hardware for even HDTV resolution videos. For higher resolution videos, this has to be done offline. Finally, the transmission of high quality, ultra-high resolution videos demands high network throughput. Thus it is most suitable for local delivery instead of streaming across the Internet.

The disadvantages translate to challenges: high throughput imaging devices, high performance video encoding, and high bandwidth networking technology. Furthermore, high performance parallel decoding is also required in order to play ultra-high resolution uni-stream videos at real-time frame rate.

Low resolution applications of this combination include desktop video conferencing, desktop remote rendering/visualization, etc. High resolution applications include digital cinema, planetarium, and so on.

Multiple Camera—Camera Array

Alternatively, one can create a uni-stream video from multiple cameras. High resolution imaging devices are usually too slow to capture full motion videos. An inexpensive way of emulating a high resolution video camera is to use an array of low resolution cameras. One example of this is the Light Field Video Camera [94]. When all cameras are co-incidental, we can treat the system as one virtual camera. The frame of any physical camera and that of the virtual camera is then related by a

2D homography. By warping the frames from physical cameras, and pasting them onto the virtual frame, we can obtain the equivalent of a high resolution camera. We call this approach a *Camera Array*. Videos from the camera array are registered and stitched together before being encoded in one stream.

Like a uni-stream video from a single camera, this approach has the advantage of simplicity and high compression ratio. It also eliminates the need for high resolution imaging devices.

Besides its computation and bandwidth requirement, the main disadvantage, and challenge, of this solution is the need for choreographing multiple cameras. First of all, the cameras have to be positioned and oriented properly to avoid gaps in the final virtual frame and, at the same time, to maintain the proximity of all optical centers. Next, when the effective focal length of the virtual camera needs to be changed to achieve the "zoom" effect, adjustment needs to be made to the orientations of all cameras, in order to avoid gaps in the case of increasing focal length (zooming in), or unnecessarily large overlapping areas conversely (zooming out). Furthermore, the foci of all cameras should be synchronized so that there is a uniform depth of field across the virtual frame. Finally, multiple video streams need to be registered and stitched, which requires additional computation.

Applications of this combination include low-cost high resolution cameras for telepresence or cinematography.

Multiple Computers—Remote Parallel Rendering

In the scientific computing community, researchers often need to render or visualize complex data sets residing on remote computers. There are basically three levels at which the communication can happen: raw data, primitive, or pixel. At the raw data level, the remote computer acts simply as a file server, while the client computer does

the computation, rendering and display. At the primitive level, the remote computer performs the actual computation and geometric transform, the client computer is only responsible for rendering the 2D or 3D fragments (primitives). This more or less corresponds to the UNIX X Window system or the WireGL/Chromium model [35, 36]. At the pixel level, the remote computer rasterizes the scene and transmits the final pixels to the client to be displayed.

As the display size grows larger, the number of pixels increases. In order to maintain the same triangle size (in pixel) on screen, the complexity of data sets or the size of 3D models has to increase as well. One immediate consequence is that, beyond some point, it becomes more economical to transmit the rendered results in pixels instead of primitives[3]. This is where videos come in handy.

The advantage of this arrangement is its simplicity, high visual quality, and high performance parallel rendering. The main disadvantage is the computation and bandwidth requirement. Furthermore, to form a uni-stream video, a parallel video encoder is required to generate the bitstream from a cluster of rendering nodes.

2.2.2 Tiled Video

Single Source

As we have described in Section 2.1.4, this approach was used in conjunction with the pixel domain splitting method, in early efforts to bring high resolution videos to tiled displays. A high resolution video is pre-processed to generate multiple smaller video streams each matching a tile in the display. The advantage of this approach is

[3]A triangle is represented by its vertices, surface normal, diffuse color, specular color, texture coordinates, etc. These can easily add up to 100 bytes per triangle. With geometry compression [26, 89], this number can be reduced to about 10 bytes per triangle. A modest 200K triangle model thus needs 2 MB, about the same for an uncompressed 24-bit 1024×768 frame. For 3D models with millions of triangles [58], transmitting videos can result in significant bandwidth saving.

that it is relatively easy to implement, and does not require a parallel video decoder. However, it has some serious drawbacks. First, the pre-processing is a time consuming decompression-recompression cycle. Therefore, it has to be done offline, and requires additional storage space for the tiled video. Second, video quality typically degrades due to recompression. Third, it is not flexible; when the screen configuration changes, a new tiled video has to be created.

Multiple Sources—Camera Array and Remote Parallel Rendering

As described in the previous subsection, videos can be generated from multiple sources, as in the camera array and remote parallel rendering cases. Instead of merging the fragments of video at the encoding end and transmitting only one stream, one can encode individual fragments into multiple streams. This forms a tiled video.

Compared with a uni-stream video, the added advantage is that it reduces the need for a high performance (parallel) video encoder, because tiles can be encoded simultaneously and independently on a cluster of nodes. There are two main disadvantages. First, the overall compression ratio will suffer slightly due to reduced search ranges for the video encoder. Second, the complexity of the video decoder is increased—it has to be able to decode multiple streams in parallel, and warp and blend the tiles before they can be displayed.

An additional challenge exists for remote parallel rendering. When the images are being rendered on a remote cluster, the scene is usually partitioned in the screen space in a sort-first fashion to balance the load. This typically results in disjoint rectangular screen regions for the servers, when a load balancing algorithm such as KD-Tree is used [79]. As the scene complexity distribution varies across frames, the servers may adjust the size of each video stream to achieve optimal load balancing. Therefore, care needs to be taken to accommodate the changing resolution of a sub-

video. This can be handled by leaving a certain percentage of headroom in the declaration of the sub-video resolution. When the actual video size is less than the declared resolution, black (or empty) blocks are used to fill in the voids. These empty blocks can be encoded in a few bits with compressions such as MPEG; therefore, the overhead is negligible. Proper alpha masking is then used to filter out these blocks at the receiving end; see Section 2.2.4. Another solution is to choose a relatively coarse temporal granularity for the balancing algorithm, so that a sufficient number of frames of the same resolution can be generated successively to form a relatively large syntactical entity in the video stream, such as an MPEG sequence.

2.2.3 Layered Video

Single Source

Layered video can be generated from a single source. For example, we consider the use of one single camera. Advancement in micro-electronics has enabled engineers to create single CCD or CMOS photosensors with tens of millions of pixels. However, their use is confined to still cameras, due to the limited read-out bandwidth of these sensors and the tremendous processing power and transmission bandwidth required to create full frame videos. Conceivably, one can modify the design of such sensors to allow selective down-sampling. For instance, in a 4000×3000 CCD device, a selectable 640×480 region outputs pixels at native resolution, while everywhere else a $5\times$ down-sampling factor is used, generating an 800×600 low-resolution output. This, in effect, creates two virtual cameras with coincidental optical centers, without the bulk of multiple physical cameras and potentially troublesome optical design.

The intrinsic advantages of layered video are all preserved—natural match of the human visual system, flexibility, and low bandwidth requirement. Additional benefits

come from the fact that a single source is used. Because the layers are obtained from the same physical source, they share the same optical center, focal length, color space, brightness, etc. Also, generally speaking, no image registration is needed to align the layers of videos, which saves a lot of computation time at the encoding end.

The disadvantage is that the highest spatial resolution is limited to the native resolution of the underlying CCD device.

We believe that this device is a very useful tool for tele-presence and tele-conference applications. The major challenge is to actually build such a multi-resolution camera.

Multiple Cameras

Naturally, layered video can be created from multiple cameras with different fields of view. The images from two cameras are related by a 2D homography, if the their optical centers coincide. Thus the rendering of layered video can be accelerated with graphics texture mapping.

The advantage of this approach is that commodity video cameras are readily available so this camera system can be built easily. It also has all the other advantages of layered videos mentioned before. Being a multiple camera system, it has the disadvantages discussed in Section 2.2.1. Some physical tricks can be played to get around some of the problems. For example, when only two cameras are concerned, a half mirror can be used to align the optical centers, as illustrated in Figure 2.5.

Multiple Computers

Multiple computers can also be the source of a layered video, as in the case of remote virtual reality applications. Instead of rendering the full scene in high resolution, one can use one computer to render a lower resolution background, and another computer to render a much higher resolution inset for the foreground. This is analogous to using

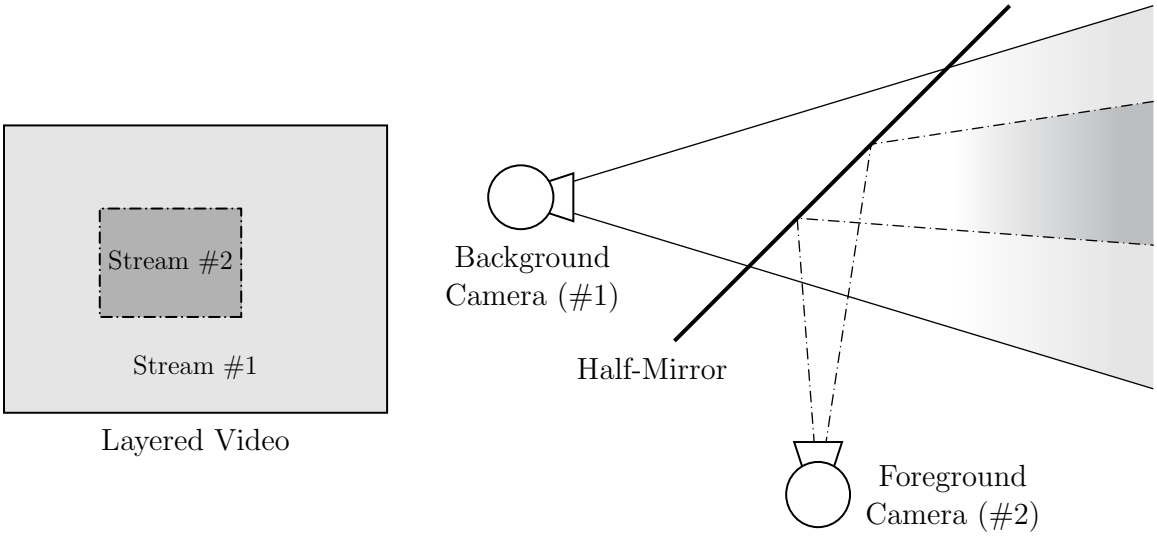


Figure 2.5: Creating Layered Video With Two Cameras

multiple cameras only without the trouble of dealing with multiple physical cameras.

Control Models

Regardless of the video source, the placement of the high resolution inset within the low resolution background should be properly determined. This control process can be either *feedforward* or *feedback*. In a feedforward model, one may choose to focus the high resolution inset on where the most motion is occurring, such as the speaker's face, hand, etc. This can be done either manually by an operator, or automatically through the use of motion tracking devices or by motion analysis of the background video. In a feedback model, the camera control signals are generated at the receiving end and sent via a back channel to the camera system. For instance, the gaze of the viewer at the receiving end can be tracked, and the high resolution inset is positioned automatically in the corresponding direction, thus creating an *Attentive Display* [61, 7].

2.2.4 Unified Representation Scheme

As we have seen in the previous discussions, the framework for scalable video delivery allows many different types of systems to be built. It is highly desirable to design a unified representation scheme to encapsulate all three video encoding methods with every possible video source in a common environment. There are several challenges:

- First, as mentioned above, when two cameras are coincidental, the geometric relationship between their frames can be modeled as a simple 2D homography with 8 parameters. However, it is difficult to truly align the optical centers of multiple cameras. Therefore, motion parallax can occur.
- To complicate the matter further, optics in the cameras are not perfect, resulting in radial distortions, luminance non-uniformity and color mismatch.
- Finally, layers and tiles need to be edge blended to create a seamless video.

Our unified representation scheme addresses these issues. It consists of the following elements:

Virtual Frame. First, a virtual Cartesian coordinate frame is defined. This virtual frame defines the geometry of the final video.

Sub-Streams. Video stream are encoded independently. We call them sub-streams.

z-Ordering. Each sub-stream is assigned a z-order logically—a smaller z-order sub-stream "sits" atop a larger z-order sub-stream and typically has a higher resolution, thus enhancing the latter's resolution.

2D Warp. The desired location of each sub-stream pixel in the virtual frame is encoded in an array—the 2D Warp. The warp can be generated using registration algorithms either online or offline, depending on the system's intended use.

Alpha Map. For each pixel of a sub-stream, its desired transparency is encoded in another array—the Alpha Map. The purpose of the Alpha Maps is to properly blend the sub-streams to form a seamless video.

At the receiving end, a decoder performs the following to present a video to viewers. It first de-multiplexes all individual sub-streams along with their warp and alpha map from the transport stream, if necessary. It then decodes each sub-stream. Finally, the frames and maps are divided into multiple tiles and sent along with the z-order to corresponding display modules. The function of a display module is to warp the frames according to the 2D warp, attach the alpha (transparency) channel, and finally render them in the low-to-high z-order.

This procedure applies to all three classes of encoding methods. Layered Video fits this representation naturally. Like layered video, Tiled Video can be represented using the exactly same syntax, except that the z-orders are irrelevant. We can also view a tiled video as a layered video with an imaginary base layer, which contains nothing and is not encoded or transmitted. Uni-Stream Video is treated as a special case of Layered Video. It contains only one sub-stream—the base layer. The warp, alpha map and z-order are all irrelevant in most cases. However, they can be used to achieve some desired special effects, such as warping of the video, fade-in and fade-out, etc.

Note that the Warp and the Alpha Map have to be updated for every new camera position. In some scenarios, this could mean every frame, such as a layered video system with dynamical tracking, a tiled video from a zooming camera array, or a dynamically load balanced remote rendering system. To reduce the computation and bandwidth requirements in generating and transmitting these maps, spatial sub-sampling can be used. A decoder can regenerate the full maps through interpolation.

As we have mentioned in Section 2.2.2, some special care also needs to be taken to accommodate the time-varying resolutions of sub-streams in a tiled video generated for remote parallel rendering.

Finally, we remark that this representation is logical by nature. The actual embodiment is immaterial to our discussion. It can be either encapsulated in an MPEG-4 environment or encoded as multiple MPEG-2 videos with maps (either embedded as user data or sent separately). However a system chooses to implement this, it should make sure that multiple streams arrive at the receiver at about the same time to minimize the end-to-end latency. This can be achieved by packetizing multiple streams and interleaving them in a transport stream. Again, the design choice is not the main concern of this dissertation.

Chapter 3

High Performance Video Decoding

At the core of the scalable video decoding system that we built is a scalable parallel MPEG video decoder. To develop such a parallel decoder and to experiment with different parallelization techniques require access to the underlying decoder. This is generally not possible with hardware-based decoder boards or closed-source commercial software codecs. This leads us to develop our own software-based decoding solution. A software decoder also has the additional advantage of being cost-effective and tracking technology well. With microprocessor frequency measured in GHz and various multimedia instruction extensions, it is now routine to decode and play DVD or even HDTV resolution contents in software, a task that had required dedicated hardware decoder boards just a few years ago.

The design goals of this software video decoder are two-fold. First, the decoder has to be easy to understand and easy to parallelize. Second, it has to be a high performance decoder. As we have described in Chapter 1, in a scalable video delivery system for tiled displays, a decoder sometimes needs to decode multiple high resolution video streams in real-time. This motivates us to study methods to extract the maximum performance from a software video decoder.

We base our study on the open source reference decoder developed by the MPEG Software Simulation Group. As a reference decoder, it is highly structured and easy to understand, although at the expense of very low performance. In order to achieve the two design goals, our final decoder has to require only minimum algorithmic modifications to the original algorithm, while achieving high performance.

In the past, the key to improving the performance of software decoding has been to develop ways to satisfy its computational requirements. Much of the previous work on improving software MPEG decoding [71] has focused on multimedia instruction extensions and effective ways of using such instructions to optimize certain core functions [37, 55, 90, 99]. As memory performance has been improving at a much slower rate than microprocessors during the past decades, the performance bottleneck of a software decoder has now been shifted to memory operations.

To understand the extent of the problem, we analyzed the distribution of *Cycles-Per-Instruction* (CPI) [72] of a software MPEG-2 decoder optimized by extensive uses of MultiMedia eXtension (MMX) and Streaming SIMD Extensions (SSE) instructions [73]. We found that stallings of memory operations increase the CPI significantly in memory-intensive functions. Our profiling results in Section 3.3.2 show that, on a PC with a 933 MHz Pentium III CPU, the average CPI of motion compensation is 1.81 and that of display is 10.57. These are several times more than the average CPI of 0.57 for the computation-intensive IDCT functions.

Our approach to solving the memory performance bottleneck problem is to exploit the concurrency between the CPU and the memory sub-system in a modern computer. We first introduce a new frame buffer layout, called *Interleaved Block-Order* (IBO), to improve the CPU's cache performance. We then describe an algorithm to prefetch macroblocks explicitly for motion compensation. Finally, we present an algorithm to schedule interleaved decoding and displaying at the macroblock level.

We implemented our proposed methods on a PC with a 933 MHz Pentium III processor. In Section 3.4.4, our tests with several DVD and HDTV streams will show that the optimizations improve the performance of a software decoder, already extensively optimized with multimedia instructions, by another factor of two. Our optimizations successfully reduce the CPIs of memory-intensive functions. The CPI of motion compensation functions is reduced to 0.7, and the CPI of display function is reduced to 1.07. As a result, the improved software decoder decodes and displays 720p (1280×720) format HDTV streams at over 62 frames per second.

The rest of this chapter is organized as follows. Section 3.1 gives a brief introduction to the MPEG video compression standards and discusses previous related work in software decoding. Section 3.2 first describes our methodology and testing environments. Section 3.3 shows various components of a software decoder and analyzes the bottleneck in it. Section 3.4 presents and evaluates our optimization techniques. Finally, Section 3.5 summarizes our study.

3.1 Background and Related Work

Due to the overwhelming amount of data present in digital videos, it is impractical to store and transmit them in their raw format, except in places where absolute quality is required, such as mastering studio. Video compression technologies are used to reduce a digital video to up to 1/100 of its original size without noticeably degrading its visual quality. There are many video compression methods available, such as Motion JPEG, MPEG [57, 43], H.261 [60], H.263 [45], H.264 [46], etc. Because of its wide acceptance, good compression rate, and high visual quality, we use MPEG as an representative in our discussion. Many of the properties of MPEG video streams also apply to other compression methods.

3.1.1 MPEG Video Compression

MPEG, or Moving Picture Experts Group, is a collective name that actually refers to several audiovisual compression standards developed over the years. Currently, there are four major MPEG standards, namely, MPEG-1 [57], MPEG-2 [43], MPEG-4 [44], and MPEG-7 [69].

MPEG-1 was designed to provide adequate quality for CIF videos (352×240 for NTSC) at Compact Disc (CD) data rate, i.e. 1.5 Mbps. It is commonly used in VideoCD's. MPEG-2 is an extension to MPEG-1. It allows much higher resolution videos and introduces new techniques for better visual quality and compression ratio, especially for interlaced video sources. MPEG-2 is the basis for some of the most widely used digital video technologies today, such as Digital Video Disc (DVD), Direct Satellite System (DSS), Digital Video Broadcasting (DVB), High Definition Television (HDTV), etc.

MPEG-1 and -2 are intended for relatively high quality film or TV contents. They do not perform particularly well at low bit rates. MPEG-4 was designed to address this issue. In addition to new motion estimation/compensation methods for higher compression ratio, it also provides an object-based scene management facility. Additionally, parametric face animation models can be used for extremely low bit rate video communications.

Finally, there is the aptly named MPEG-7 standard that encompasses all the previous three standards, $(7 = 1 + 2 + 4)$. MPEG-7 itself does not provide any new compression techniques beyond those introduced in MPEG-1, -2, and -4. Instead, its main goal is to describe the content semantically. Therefore, MPEG-7 is irrelevant to the discussion in this dissertation.

Due to its popularity and ubiquity, we use MPEG-2 as the representative MPEG

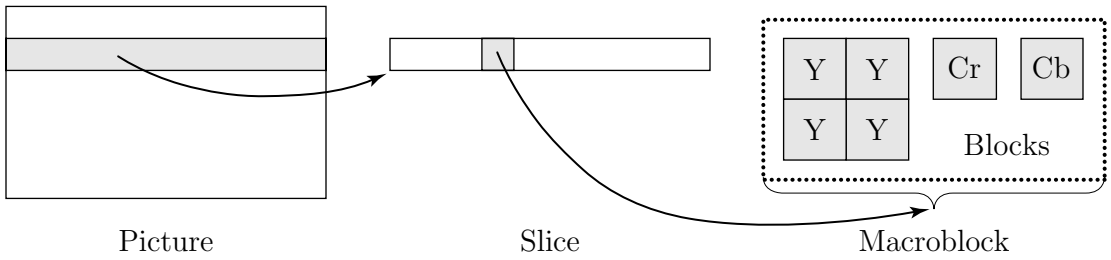


Figure 3.1: Elements in an MPEG-2 Video Stream

standard throughout this dissertation, unless otherwise mentioned. MPEG-2 is a set of ISO standards for compressing digital video and audio. It consists of a video compression standard, an audio compression standard, and a system layer standard for multiplexing them. Our focus is on the MPEG-2 video streams.

To achieve maximum compression ratio, MPEG-2 video compression removes both temporal and spatial redundancies from the video data. In encoding a video stream, an encoder first converts pixels in a picture into YCrCb color space with optional sub-sampling of chroma signals. Depending on the input source of video, a picture can be either a frame in a progressive sequence or a field in a non-progressive sequence. A picture is then divided into 8×8-size blocks. Four luma blocks along with 2, 4, or 8 chroma blocks are grouped together to form a macroblock, for 4:2:0, 4:2:2, or 4:4:4 digital component videos[1], respectively. Figure 3.1 illustrates this hierarchy of syntactic elements.

There are three types of pictures in an MPEG-2 video stream: *Intra* (I), *Predicted* (P) and *Bi-directional predicted* (B). MPEG-2 compresses an I-picture in the same way as JPEG does. It performs *Discrete Cosine Transform* (DCT) on a block basis, and uses *Quantization* and *Run Length Encoding* (RLE) to remove spatial re-

[1]The chroma signals can be optionally sub-sampled in both horizontal and vertical directions to save transmission bandwidth. In a 4:2:0 component video, chroma signals are sub-sampled by a factor of two both horizontally and vertically. In a 4:2:2 video, chroma signals are only sub-sampled horizontally. Chroma signals are not sub-sampled in a 4:4:4 video.

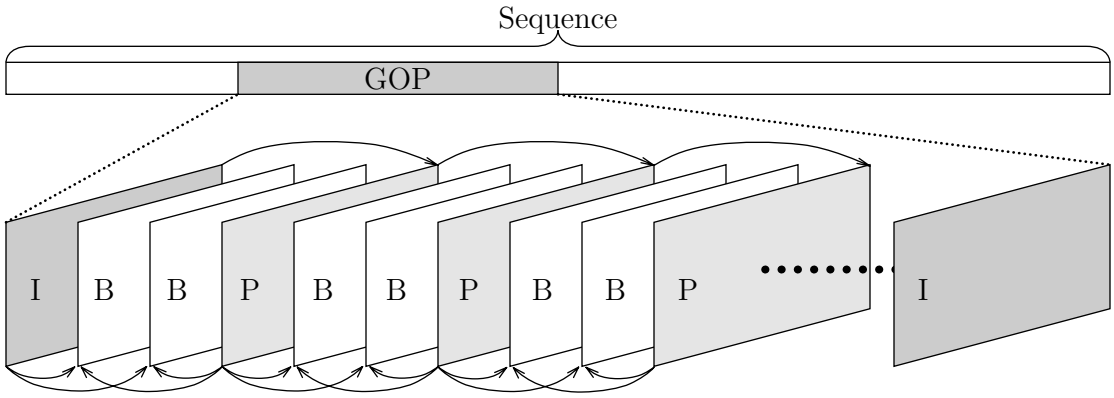


Figure 3.2: A Series of Pictures

dundancy. *Motion Estimation* is used to further remove temporal redundancies. For each macroblock in a P- and B-picture, one to four *Motion Vectors* are used to predict it from up to two reference pictures; the residual (or difference) is then DCT coded.

A series of I-, P-, and B-pictures are grouped together to form a *Group of Pictures* (GOP), which, in turn, forms a sequence, as illustrated in Figure 3.2.

In encoded video bitstreams, a 32-bit byte-aligned start code is provided for each sequence, GOP, picture, and slice. However, a macroblock does not have a start code, nor is its start or end necessarily byte-aligned. It will become clear in Chapter 4 that this lack of start code for macroblock creates challenges for designing a scalable parallel MPEG decoder.

3.1.2 Related Work

Patel *et al.* first investigated the performance of a software MPEG-1 video decoder [71], which was later commonly referred to as the "Berkeley Code." Because of the lack of hardware color space conversion and true-color display, much effort was directed to optimizing dithering performance.

During the past few years, much work has focused on introducing and using mul-

timedia instructions. In 1995, Lee published her MPEG-1 video decoder on a HP PA-RISC processor with multimedia instruction extensions [55]. Her decoder was able to achieve real time MPEG-1 decoding on an HP712 workstation. Zhou *et al.* discussed MPEG-1 decoding on a Sun UltraSPARC with VIS extensions [99]. With the growing popularity of DVD, several companies such as CineMaster, Cyberlink, InterVideo, and Xing developed software DVD players for desktop PCs. Recently, Tung *et al.* studied MMX optimizations for software MPEG-2 decoding and did a performance evaluation based on Cyberlink's old non-MMX decoder [90]. Ranganathan *et al.* evaluated the benefits of multimedia extensions on different processor architectures [75]. Their benchmarks showed the performance speedups is close to two.

A large body of literature exists on the topic of improving caching locality. Early researches have proposed ways of rearranging data structures and altering algorithms to reduce page faulting in virtual memory [1, 29]. Tiling has become a well known software technique for using the memory hierarchy effectively [20, 32, 54]. It can be applied to any levels of memory hierarchy, including virtual memory, caches, and registers. Philbin *et al.* [74] also proposed fine granularity thread scheduling for improving data cache locality. These efforts were aimed primarily at scientific programs.

Software and hardware prefetching techniques have been well studied in the past to address the issue of the widening gap between the performance of processor and memory. Early hardware prefetching techniques [48, 83] only work for programs with sequential accesses. Reference prediction table based preloading mechanism was proposed by Baer *et al.* [5, 18]. Klaiber *et al.* [50] and Callahan *et al.* [12] studied software controlled prefetching, and Mowry *et al.* [68, 67] proposed compiler-based algorithm for automatic insertion of prefetching instructions. Ranganathan *et al.* studied the interactions of software prefetching with ILP processors [76].

Most such studies are targeted for general purpose applications. Soderquist and

Leeser [81] studied the data cache performance of software MPEG-2 video decoders with a hardware simulator. They proposed a few architectural methods to improve the performance of a software MPEG-2 decoder. However, they did not provide implementation or simulation results of these methods. Zucker *et al.* studied several prefetching techniques for MPEG-2 video decoding [100, 101, 102, 103]. Their studies focused on hardware prefetching or compiler-based software prefetching techniques. Cucchiara *et al.* [23, 24] recently also proposed several other architectural ideas to improve multimedia applications.

These studies did not explore the possibility of using data structural and algorithmic changes to exploit concurrency in an MPEG decoder.

3.2 Methodology and Environment

Our study uses Cycles-Per-Instruction as a measure to see how well the instructions in the core functions of a software decoder perform. This is a well known method in computer architecture research to understand the degree of instruction level parallelism [72]. Although it is a very powerful tool, CPI can not be used directly as a performance metric in this study, because the program itself is changed between optimization steps. We use the final decoding frame rate as the performance metric and CPI only as an indicator for identifying performance bottlenecks and thus optimization opportunities in the decoder. This section describes the hardware platform, software tools, and video streams used in our tests.

3.2.1 Software Tools and Measurements

We use Microsoft Visual C++ 6.0 with Service Pack 5 to develop the software decoder. "Release build" with "maximum optimization for speed" options is used to compile

the programs. We use Intel VTune 4.0 Performance Analyzer [42] to estimate the CPI for functions with sequential or iterative structures. It is calculated as follows:

$$\text{CPI} = \frac{Ft}{nI},$$

where F is the CPU clock frequency, t is the time spent in the measured function[2], n is the call count of the function, and I is the number of instructions in the function. We obtained n by using the profiling tool in VTune, I by hand counting (since VTune does not have such a feature), and t by VTune's time measure.

We also use CyberLink PowerDVD 2.55, a popular commercial software DVD player, for comparison purpose. Since the rate control in PowerDVD cannot be turned off, we use VTune to measure the total time, t_D, spent in its decoder module. Then we calculate the effective frame rate:

$$\text{fps} = f/t_D.$$

3.2.2 Test Platform

Our test PC has a 933 MHz Pentium III processor, 256 MB of PC133 SDRAM, and an NVIDIA GeForce256 AGP 4X graphics card. The PC runs Windows 2000 Professional with Service Pack 1. DirectX 7.0 is used for direct frame buffer access.

Our tests do not use any built-in hardware support for MPEG-2 video decoding in the graphics card; we only used the hardware color space conversion feature. To do this, we use a DirectDraw overlay surface with YUYV pixel format to display

[2]VTune is an event-based sampling tool. The performance analyzer collects samples of the program counter at regular intervals, typically 1 ms. The execution time of a function is then deduced from the number of samples that fall into the address range of this function. Unlike compiler-based instrumentation, this method generates no measuring overhead and excludes interrupts properly.

Table 3.1: Test MPEG-2 Video Streams.

Stream	Resolution	I-Pictures	P-Pictures	B-Pictures	Total Pictures	Size(MB)
`spr`	720×480	213	631	1,670	2,514	58.7
`matrix`	720×480	194	580	1,546	2,320	57.5
`fish`	$1,280 \times 720$	27	240	505	772	27.3
`fox5`	$1,280 \times 720$	24	216	480	720	22.5

the video frames. YCrCb to RGB conversion is performed by the overlay hardware in real-time. We remark that, when memory consumption is concerned, YUYV[3] is a sub-optimal format for 4:2:0 video. It uses 16 bits for every pixel, while only 12 bits are needed. In effect, we upsample the 4:2:0 video to 4:2:2. Although the video quality is not improved in this process, we choose this format because of its ubiquity.

3.2.3 Test Sequences

To test the performance of a decoder at different resolutions, we use several MPEG-2 video streams. We choose a total number of four 720×480 and 1280×720 resolution video streams, as shown in Table 3.1, to represent mainstream DVD and high-end HDTV applications. Also, a 720p HDTV video frame has approximately the same amount of pixels as in a commodity XGA projector. Therefore, this represents the natural workload of a decoder when it is parallelized to work on a tiled display.

`Spr` (Saving Private Ryan) and `matrix` (The Matrix) are two clips ripped from the respective movie DVD's. The `fish` clip is a shot of a fish tank taken with an HDTV video camera, courtesy of Intel Microprocessor Research Lab. `Fox5` is a clip recorded from the HDTV broadcast of the FOX5 TV station of New York City.

[3]YCrCb is the technically correct term to use in the context of digital videos [47]. We use YUV to describe the pixel formats of a graphics card due to its already widespread use.

3.3 Performance Bottleneck

To identify the performance bottlenecks in a decoder, we measure a software MPEG-2 video decoder that has been optimized by extensive use of the MMX/SSE instructions. To calibrate its performance, we compare the decoder with the PowerDVD software player. Our results show that the performance bottleneck is at memory operations in memory intensive functions including motion compensation and display.

3.3.1 The Baseline MPEG-2 Video Decoder

The baseline software decoder used in our experiments is an MPEG-2 video codec developed by the *MPEG Software Simulation Group* (MSSG) [28]. To better understand the components of the software MPEG-2 decoder, we first describe its algorithm and then discuss in detail the functions of its main modules.

Figure 3.3 shows the block diagram of a typical software MPEG video decoder, such as [28]. Figure 3.4 lists the corresponding high-level algorithm. The decoder iterates on decoding a picture and sending the decoded pixels to the frame buffer of graphics card. Within a picture, the decoder processes each macroblock through three steps: Variable Length Decoding and Inverse Quantization (referred to as VLD for succinctness), Inverse DCT (IDCT), and Motion Compensation (MC).

Each of these processing steps along with the display has its own characteristics in terms of computation and memory bandwidth requirement.

VLD. The VLD module parses an input stream, decodes macroblock headers, motion vectors, and DCT block coefficients. In this step, a small amount of compressed data is read from disk or network, which can easily fit into the L1 cache. VLD is the process of inverse Huffman coding, it involves table lookup, bit shifting

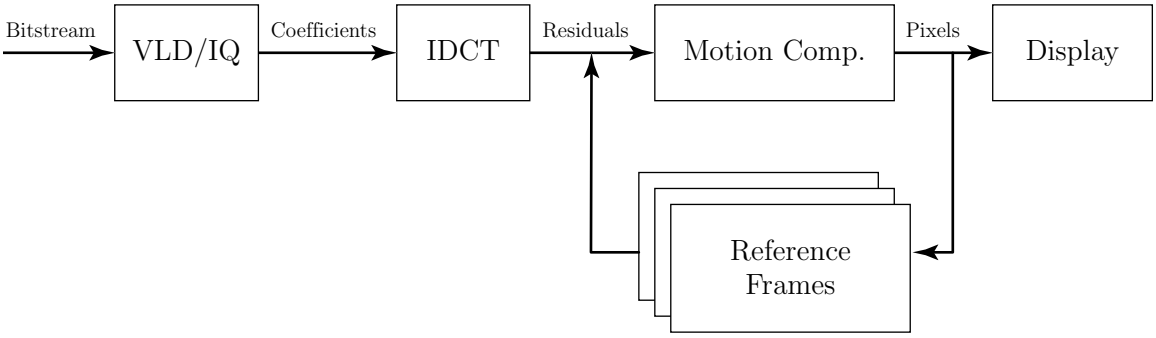


Figure 3.3: Block Diagram of a Typical Software MPEG Video Decoder.

operations, and branches. Because general purpose processors are usually not optimized for these operations, VLD is mostly computation intensive.

IDCT. The IDCT module restores DCT coefficients into a block of pixels or prediction residuals. This step needs only one block (64 short integers) of pixel data along with some tables of constants [4]. The data can fit into the L1 cache of a processor easily. However, it does take a lot of cycles to compute the result, making IDCT an computation intensive procedure.

MC. The MC module uses motion vectors to form a prediction of the current macroblock from previously decoded pictures. It then combines the prediction with the residuals from the IDCT module to produce the final picture. It is computation intensive for three reasons. First, it needs to calculate the average of two or four pixels when half-pixel accuracy motion vectors are used. Second, it has to average two macroblocks when bi-directional predictions or dual prime predictions are used. Third, it needs to saturate the sum of prediction and residual when it exceeds the representation range of a byte. Motion compensation is also memory intensive because it is essentially a series of memory copies. For even a moderate resolution video, the working set size of three internal frame buffers cannot fit into the L1 or L2 cache of a general purpose processors.

```
Video Decoder:
for each picture
    for each macroblock
        Decode macroblock header, motion vectors
        for each block
            if a block is coded
                Decode and inverse quantize a block
                IDCT the block
        Motion compensation
    if the current picture is B-type
        Display the current frame
    else
        Display the forward reference frame
```

Figure 3.4: Algorithm of a Typical MPEG Video Decoder.

Display. The function of display module is to display a frame in YCrCb format on the monitor. This function used to be computation intensive, when color space conversion was performed by the CPU [71]. Nowadays virtually every graphics card has built-in capability of YCrCb-to-RGB conversion. Thus, display has become a pure memory copy. Its speed is limited by the bandwidth of the memory bus and/or the graphics bus.

3.3.2 Identifying Performance Bottlenecks

We first apply known optimization techniques to the MSSG reference decoder. The reasons are two-fold. First, it helps us understand the benefit of multimedia instruction set extensions. Second, it allows us to identify performance bottlenecks in the resulting optimized video decoder.

The optimizations that we incorporate include a fast IDCT routine using MMX

Table 3.2: Frame Rates of V0, V1, and PowerDVD.

Stream	V0	V1	PowerDVD
`spr`	33.5	74.9	80.3
`matrix`	33.0	77.6	83.1
`fish`	14.0	32.0	N/A
`fox5`	14.4	33.6	N/A

instructions, fast motion compensation routines using MMX/SSE instructions, fast bitstream processing functions using MMX instructions, and 1×1 and 4×4 IDCT fast paths for blocks that contains only low frequency coefficients. We call the reference decoder V0 (Version 0), and the optimized decoder V1 (Version 1).

We play all four test streams on V0, V1 and the PowerDVD player[4]. Table 3.2 shows the frame rates. Notice that V1 is about 2.2 times faster than V0, and only about 7% slower than PowerDVD. This indicates that extensive use of MMX/SSE instructions is able to provide state-of-the-art software decoding performance. The 2× performance improvement also confirms the results in [75].

Using the profiling tools in VTune, we analyze the running time of each major component in V1. Table 3.3 shows the time spent in the VLD, IDCT, MC and display modules for all four video clips. The table also includes the calculated CPIs of core routines in IDCT, MC, and display. We did not calculate the CPI of VLD, due to the difficulty of hand counting its number of instructions.

The results show that the MC and display modules dominate the running time. The average CPI of IDCT is 0.57, indicating that the processor executes almost two instructions per CPU cycle. In other words, the two MMX pipelines are working nearly at full throughput. This strongly suggests that IDCT is not memory limited.

[4]PowerDVD does not play HDTV. Therefore, no frame rates are available for `fish` and `fox5`.

Table 3.3: Running Time Breakdown and CPIs of V1. Running time is measured in ms, and the corresponding CPI is noted in parentheses.

	VLD	IDCT	MC	Display	Other
`spr`	7,715 (n/a)	2,410 (0.56)	11,902 (1.87)	10,384 (10.44)	1,145 (n/a)
`matrix`	7,168 (n/a)	2,410 (0.57)	9,976 (1.81)	9,583 (10.43)	1,041 (n/a)
`fish`	4,276 (n/a)	1,437 (0.57)	9,045 (1.74)	8,716 (10.71)	628 (n/a)
`fox5`	3,672 (n/a)	1,087 (0.59)	7,970 (1.82)	8,122 (10.71)	573 (n/a)

The average CPI of MC is 1.81, which is significantly greater than that of IDCT. This shows that there are stalls in the MC module. By analyzing the code, we found that the core MC functions have simple sequential structures, with branches accounting for less than 2% of the total instructions. Thus branch mis-predictions are unlikely to be the cause. As we have described before, MC is essentially a series of memory copies. Therefore, it is the memory accesses in MC that are stalling the CPU. It shows in two forms—cache read misses or the piling up of writes.

The average CPI of the display module is 10.57, showing that the CPU stalls severely. Because the display function is a sequential copy from main memory to graphics memory, we can preclude branch mis-prediction as the cause of the high CPI number. We believe the main reason is that it takes very few instructions but many CPU cycles to transfer data in the write buffer to an AGP device (graphics card). When the write buffer in the CPU is full, further write instructions have to stall until an entry in the write buffer is retired.

We notice that the distributions of CPIs exhibit the same pattern for all four video clips. In the following sections, we choose to use `fish` as a representative clip to evaluate the decoder's performance after each optimization technique is incorporated.

Figure 3.5 illustrates the time sequences of different tasks in the CPU, memory

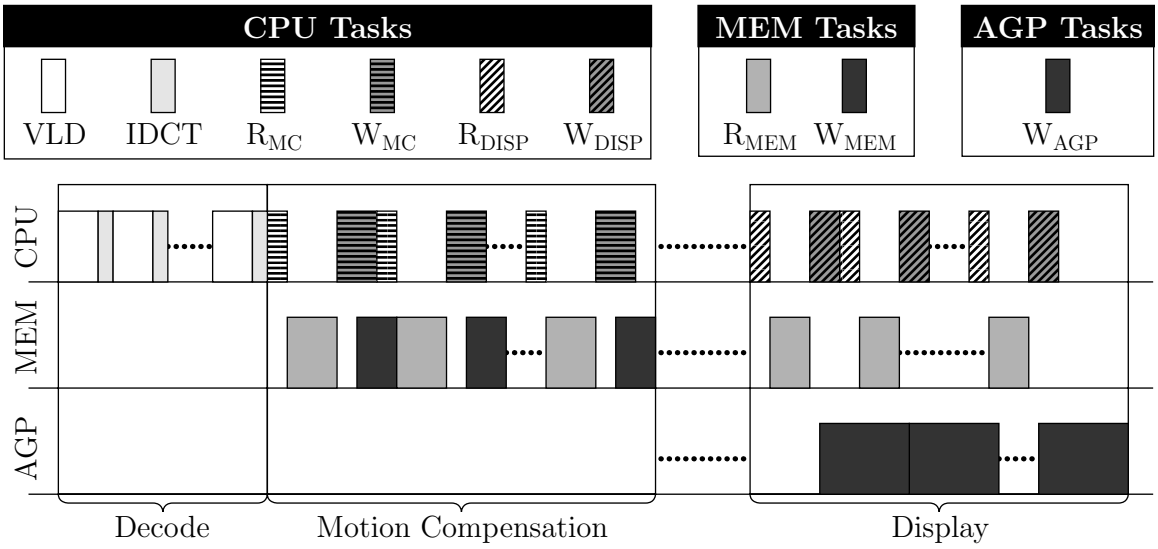


Figure 3.5: System Resource Utilization in an MPEG-2 Decoder. This is an abstract view of the algorithm; items are not drawn to proportion.

bus, and AGP bus with the software decoder. The time line goes from left to right horizontally. The bars along the CPU line indicate the tasks of the core functions in a software decoder and the width of the bars indicate the amount of time taken. The tasks include VLD, IDCT, R_{MC} (reading in MC), W_{MC} (computation and writing in MC), R_{DISP} (reading in Display), and W_{DISP} (computation and writing in Display). The bars along the MEM line and AGP line indicate the amount of time taken to read (R_{MEM}) and write data (W_{MEM} and W_{AGP}) in the memory sub-system and the frame buffer across the AGP port, initiated by the tasks along the CPU line.

With this video decoding algorithm, the CPU has to stall frequently while waiting for data to be read in MC, and to be written in display. This illustration explains why the CPIs of MC and Display are so high.

3.4 Macroblock Level Concurrency

To remove or alleviate performance bottlenecks in the software MPEG decoder, we propose three techniques to exploit concurrency among the CPU, the memory subsystem, and the frame buffer. In each of the following three subsections, we first describe a method and then evaluate its performance improvement. We will then provide an overall evaluation of a software decoder with all three optimizations.

3.4.1 Interleaved Block-Order of Frame Buffer

Description

Caches in a memory hierarchy help to exploit 1D spatial localities that exist in many applications. In MPEG-2 video decoding, and many other image and video applications, the data reference pattern exhibits a 2D spatial locality, that is, when one pixel is accessed, the pixels in its neighboring columns and rows are also likely to be accessed. For example, in MPEG-2 video decoding, when motion compensation is being performed, 16×16-size pixel blocks are read and written at once.

In a typical software video decoder, such as the MSSG decoder, scanline ordered internal frame buffers is used. Although this layout is easy to implement, it proves to be not the most efficient. As we can see in Figure 3.6, pixels within an 8×8 block are scattered across 8 cache lines. This means decreased cache locality for the decoder.

To improve the memory write performance for multimedia applications, some architectures, such as Pentium III, provides a mechanism called write-combine without write-allocation [39, 40]. Successive partial writes to a cache line can be buffered and collapsed to form a single cache line write to the memory. This saves an unnecessary read when a cache line is first allocated. Most processors have only a handful of these write-combine buffers. For instance, Pentium III has four. It is therefore im-

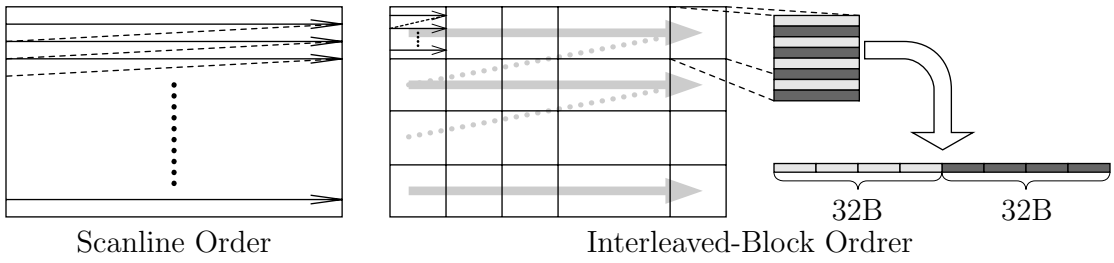


Figure 3.6: Interleaved Block-Order Layout.

portant that these partial writes happen close to each other. When scanline ordered internal buffers are used, writing a macroblock causes at least 32 partial writes (16 for the luma component, and 8 for each of the chroma components). This makes write-combine impossible.

To improve the cache locality and take advantage of the write-combine feature, we propose a new layout for internal frame buffers, called Interleaved Block-Order (IBO)[5]. In this layout, a frame buffer is first row-major ordered on an 8×8-block basis. Within each block even scanlines are first stored together, then followed by odd scanlines, as shown in Figure 3.6. With this arrangement, all 64 bytes for a block are stored together, so that they can fit into one or two cache lines. This not only increases cache locality but also makes write combine possible. The interleaving structure also benefits field pictures and field predictions.

Evaluation

We implement the interleaved block-order frame buffers by modifying the motion compensation routines and the display routine in V1. We call this version V2.

We profile V2 playing the `fish` clip, and calculate the CPI of major components. Table 3.4 shows the comparison between V1 and V2.

[5]This is inspired by the early tiling works [20, 32, 54]. Similar techniques were also employed for texture buffers in graphics accelerators such as the Evans & Sutherland RealImage chipset.

Table 3.4: Comparison Between V1 and V2 (`fish`).

	V1		V2	
	Time(ms)	CPI	Time(ms)	CPI
VLD	4,276	—	4,117	—
IDCT	1,437	0.57	1,390	0.55
MC	9,045	1.74	6,984	1.52
Display	8,716	10.71	8,336	10.41
Other	628	—	1,161	—
Total	24,102	—	21,988	—
FPS	32.02		35.13	

We notice that the time spent in the MC module is reduced from 9 seconds to 7 seconds. Because of the sequential nature of the display function, branch mispredictions should not play a major role. This leads us to attribute the performance enhancement to the improved data cache locality provided by the interleaved block-order frame buffer layout. We remark that this is a 23% improvement for an optimized motion compensation module by simply reorganizing the memory layout. The CPI of MC is reduced from 1.74 to 1.52. This means that the CPU stalling is reduced due to better cache locality and less memory traffic by enabling write-combine.

The display time decreases from 8.7 to 8.3 seconds. This is likely due to better cache locality. With a CPI of 10.41, display is still severely memory bound.

3.4.2 Explicit Prefetching of Macroblocks

Description

In a video decoder, the working set size is at least 3 frame buffer size—two for reference frames and one for the current frame[6] . For a modest 720p format (1280×720) stream,

[6]The original definition of working set is due to Denning [27]; here we use its informal meaning.

it equals about 4.1 MB of memory. The working set can hardly fit into the largest L2 or L3 cache of today's commodity processors[7], let alone the L1 cache. The sequential decoding of macroblocks translates to compulsory and/or capacity cache miss [34] for almost every macroblock.

Software controlled cache prefetching method [102, 103] was proposed to alleviate this problem. A compiler first instruments the decoder and then inserts prefetch instructions according to the profiling results. Because the source address of a reference macroblock is data dependent, the automatically inserted prefetch instructions are speculative at best. When too few prefetches are used, cache misses can still occur; but when too many prefetches are used, memory traffic can be unduly increased, thus exacerbating the problem.

An ideal software prefetching mechanism should avoid these problems by reducing or eliminating cache misses without creating extra memory traffics. Fortunately, this is possible for MPEG-2 video decoding. We notice that motion vectors are coded in the macroblock header. Therefore, a decoder knows the source addresses of reference blocks before decoding the blocks and IDCT. As we have shown before, VLD and IDCT are not memory intensive; this provides a perfect opportunity to prefetch reference macroblocks during these steps. By doing so, the decoder can distribute memory accesses among all three processing steps, hiding the memory read latency.

Figure 3.7 shows the modified decoding algorithm with prefetching. In this algorithm, we set up the source addresses of reference blocks right after motion vectors are decoded. Prefetch instructions are then inserted between the decoding and IDCT of each block. These prefetches bring the reference blocks to cache.

Figure 3.8 illustrates how the algorithm works. Comparing with Figure 3.4, we

[7]Intel Pentium 4: 512KB L2. Intel Xeon: 512KB L2. AMD Athlon XP: 256/512KB L2. AMD Opteron: 1MB L2. Apple PowerPC G4: 256KB L2, 1/2MB L3. Apple PowerPC G5: 512KB L2

Video Decoder:
for each picture
    for each macroblock
        Decode macroblock header, motion vectors
        Calculate source addresses of reference blocks
        for each block
            Prefetch a reference block
            if a block is coded
                Decode and inverse quantize the block
                IDCT the block
        Motion compensation
    if the current picture is B-type
        Display the current frame
    else
        Display the forward reference frame

Figure 3.7: Decoding Algorithm with Prefetching. (Changes are shown in *italics*.)

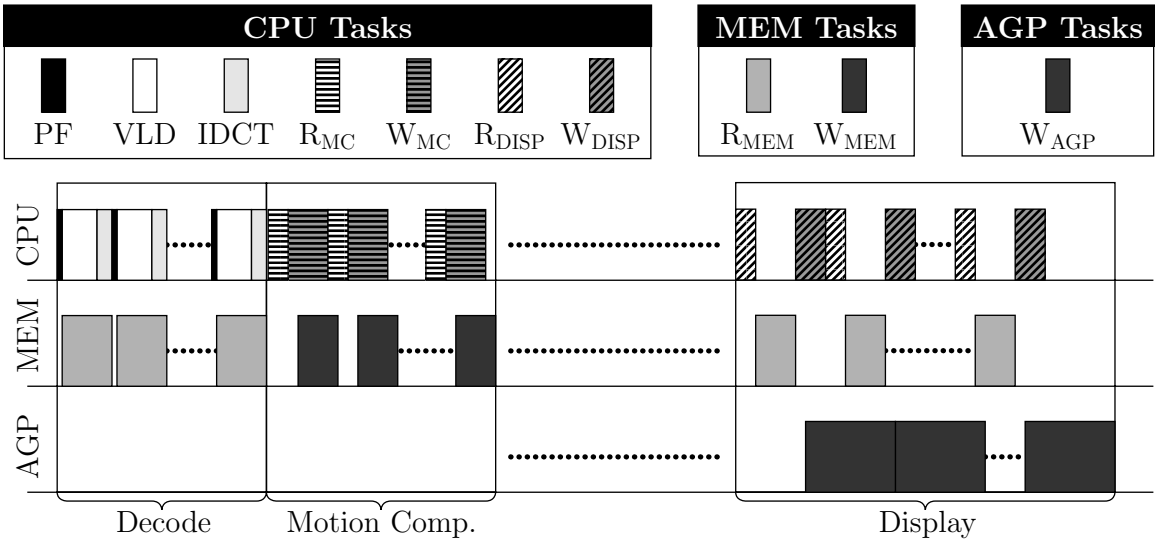


Figure 3.8: Improved Resource Utilization with Explicit Prefetching. This is an abstract view of the algorithm; items are not drawn to proportion.

Table 3.5: Comparison Between V2 and V3 (`fish`).

	V2		V3	
	Time(ms)	CPI	Time(ms)	CPI
VLD	4,117	—	4,705	—
IDCT	1,390	0.55	1,509	0.59
MC	6,984	1.52	3,922	0.65
Display	8,336	10.41	8,350	10.42
Other	1,161	—	835	—
Total	21,988	—	19,321	—
FPS	35.13		39.96	

notice that the memory reads are moved from the motion compensation phase to decoding. This has the effect of reducing or even eliminating stalls in motion compensation, while utilizing the otherwise idling memory resource in the decoding stage.

We remark that an additional benefit of the interleaved block-order is that it reduces the number of prefetch instructions, because the pixels of a block are stored together. In most processors, such as the Pentium III, a cache line consists of 32 bytes. Thus a block can be completely prefetched with two instructions.

Evaluation

To implement the explicit prefetching, we move the reference block address calculation from the motion compensation module to the VLD module. They are placed after the macroblock header decoding code. We then manually insert prefetch instructions, and intersperse them with block decoding functions. We call this decoder V3. The running time breakdown and CPI's of V3, as compared to V2, are shown in Table 3.5.

The total time in MC is reduced from 6.984 seconds to 3.922 seconds. The accurate, explicit macroblock prefetching has effectively removed the memory bottleneck

in motion compensation. As a result, the CPI of MC is now reduced from 1.52 to 0.65. This indicates a much improved utilization of the MMX/SSE units.

However, prefetching does introduce some overheads. The time spent in the VLD module is increased from 4.117 seconds to 4.705 seconds and that in the IDCT module has increased from 1.39 seconds to 1.509 seconds. But these overheads are much less than the saving of 3.061 seconds in the MC module. As a result, the overall frame rate of playing the fish video clip is increased from 35 to 40.

3.4.3 Interleaved Output and Decode

Description

Although the previous two optimizations can remove the memory performance bottleneck in the motion compensation module, they are not able to do much for the display phase in the decoder. From the running time measurement of V3, we find that the average observed bandwidth for copying pixels is only about: $772 \times 1280 \times 720 \times 2/8.350 = 170.4$ MB/s. This is far short of the available bandwidth on an AGP 4X port. It takes about $8350/772 = 10.8$ ms to copy a 1280×720 frame in packed YUYV format. In order to achieve real time decoding of 720p at 60 fps, a decoder has only about 16 ms to decode and display a frame. Clearly the display is still a bottleneck.

From Figure 3.8 we notice that the graphics bus idles during the decoding phase. It is only used during the display phase, where an entire picture is written to the frame buffer. The 10.42 CPI indicates that the CPU stalls frequently to wait for data to be sent across the graphics bus.

To exploit the concurrency between the CPU and the frame buffer across the AGP port, we propose an algorithm to interleave decoding and displaying at macroblock level. Instead of copying the entire picture after it is decoded, we break the copying

process into small units. To find out the appropriate granularity, we perform the following simple experiment.

We write a program that allocates a DirectDraw surface on the graphics card and writes bytes to the buffer, just as an MPEG-2 video decoder does. The program iterates n times on a loop. Within the loop, it first performs some simulated computation that does not reference any memory locations. It then optionally writes B bytes to consecutive addresses on the display surface.

The program is run in two modes. In the first mode, the optional writes to the frame buffer are disabled, and we measure the total running time as t_c, that is, time for computation alone. In the second mode, the optional writes are enabled, and the total running time T is again measured. The difference in the two running times is caused by the writes. We can then calculate the effective write bandwidth:

$$\text{EBW} = nB/(T - t_c)$$

We vary B from 128 bytes to 256K bytes in powers of 2. The observed effective write bandwidth[8] across the AGP port versus the write granularity is plotted in Figure 3.9. From the plot, we observe that when B is large, the curve is flat. Here we essentially see the sustained throughput of the write operation, which is mostly limited by the on-board graphics memory speed. However, as B decreases, the effective bandwidth increases drastically. In the finest granularity, it even exceeds the theoretical limit of 1.066 GB/s in the AGP 4X specification. The reason is that the algorithm has successfully exploited the concurrency between CPU and the AGP port. It hides the bus transactions in the computation. In effect, it sees the CPU

[8]Due to equipment reason, this experiment was performed on a PC with slightly different configurations as the one used to evaluate the decoder's performance. The absolute numbers in the plot should not be taken at their face value. Instead the trend of this curve is what interests us most.

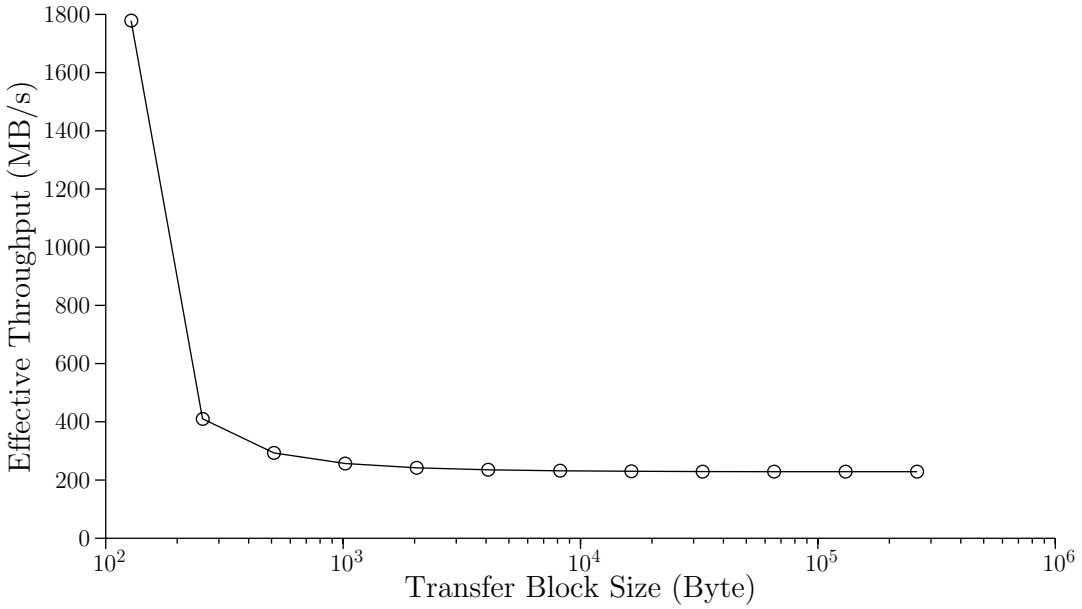


Figure 3.9: Effective AGP Write Bandwidth as a Function of Write Granularity.

write buffer speed instead of the AGP speed.

In the test, we conclude that the smaller the write granularity, the higher the write bandwidth, and thus the higher the overall frame rate. In a real MPEG-2 video decoder, using too small a granularity will inevitably introduce overheads which eventually negate the performance gain. We decide to use macroblock as the granularity. It is small enough to gain from this effect. On the other hand, it is also large enough so that it requires very little algorithmic change, because a picture is naturally decoded one macroblock at a time.

We use a pointer `output-frame` to indicate which frame[9] to be output during the decoding. It is the current frame for a B-picture, and the previous reference frame for an I- or P-picture. After the motion compensation of a macroblock in the current frame, one macroblock from the `output-frame` will be copied to the graphics card. The latency of graphics bus is hidden in the decoding of next macroblock. To further

[9] A *frame* in a progressive sequence or the combined even and odd *fields* in an interlaced sequence.

Video Decoder:
for each picture
  if *current picture is B-type*
    Set `output-frame` *to* `current-frame`
  else
    Set `output-frame` *to* `forward-reference-frame`
  for each macroblock
    Decode macroblock header, motion vectors
    Calculate source addresses of reference blocks
    for each block
      Prefetch a reference block
      if a block is coded
        Decode and inverse quantize a block
        IDCT a block
    Prefetch a macroblock from `output-frame`
    Motion compensation
    Display the output macroblock

Figure 3.10: Decoding Algorithm with Interleaved Output and Decode.

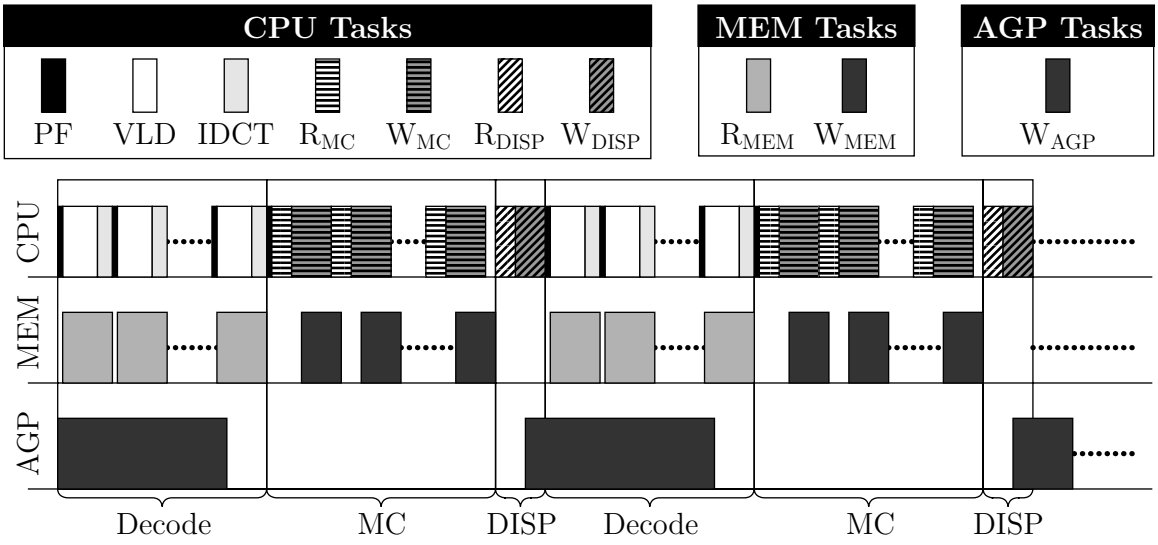


Figure 3.11: Optimized Resource Utilization with Prefetching and Interleaved Output. This is an abstract view of the algorithm; items are not drawn to proportion.

Table 3.6: Comparison Between V3 and V4 (`fish`).

	V3		V4	
	Time(ms)	CPI	Time(ms)	CPI
VLD	4,705	—	5,059	—
IDCT	1,509	0.59	1,454	0.57
MC	3,922	0.65	4,216	0.70
Display	8,350	10.42	882	1.07
Other	835	—	830	—
Total	19,321	—	12,441	—
FPS	39.96		62.09	

reduce the time spent in reading a macroblock, we prefetch the output macroblock before the motion compensation.

Figure 3.10 shows the modified decoding algorithm with interleaved output and decode. Figure 3.11 illustrates how the algorithm alleviates the performance bottleneck. When display is the only purpose of decoding a stream, we further reduce the memory requirement by not storing B pictures in memory. This option can be easily integrated with interleaved output.

Evaluation

We implement a display routine that outputs one macroblock at a time, and interleave the output with decoding. We call this new version V4. Table 3.6 shows the running time break down and CPI's of V4, as compared to the previous version V3.

The result shows that the bottleneck at the display phase has been completely removed. The display time is reduced from 8.35 seconds to 0.882 seconds, a speedup by about a factor of 10. The CPI is reduced from 10.42 to 1.07. Again, this shows that the MMX/SSE units are working at full throttle. The effective write bandwidth

Table 3.7: Performance Comparison of V0, V1 and V4.

Stream	V0 (fps)	V1 (fps)	V4 (fps)	Speedup V4 vs. V0	Speedup V4 vs. V1
`spr`	33.48	74.92	132.72	3.96	1.77
`matrix`	33.03	77.64	133.97	4.06	1.73
`fish`	14.06	32.02	62.09	4.42	1.94
`fox5`	14.38	33.61	67.57	4.70	2.01

is $772 \times 1280 \times 720 \times 2/0.882 = 1.613$ GB/s, well exceeding the AGP 4X port limit.

As a result of these combined efforts to remove memory access bottlenecks, the new decoder now plays the fish video stream at 62 frames a second.

Because we move the writes across AGP bus to the decoding phase, and prefetch the output macroblock before motion compensation, the performances of these two components do suffer slightly. This is indicated by the increase in running times of VLD and MC.

3.4.4 Overall Comparisons

To see the overall effects of all three optimizations, we run all four test streams with V0, V1 and V4. Table 3.7 shows the frame rates of all three versions.

The results show that the combined three optimizations can speedup V1 (optimized with extensive use of MMX/SSE instructions in the core functions) by a factor of 1.7 to 2.0. The overall speedup over the original MSSG decoder is about 4.0 to 4.7. The resulting software MPEG-2 decoder can now play HDTV video streams on a PC with 933 MHz Pentium III CPU at real-time frame rates.

We also notice that our optimizations improve higher resolution HDTV videos better. This is because the larger memory footprint of decoding them more adversely

impacts the original decoding algorithm. Thus there is more to gain when the memory bottleneck is lifted.

3.5 Summary

In this chapter, we presented a high performance software MPEG video decoder. We have successfully achieved the two design goals. First, the data structural and algorithmic changes to the original reference decoder are small. The optimized version still retains much of the original structure; this makes later parallelization a much easier task. Second, on the performance side of the equation, we notice that extensive use of multimedia instructions can speed up the original MSSG software decoder by a factor of about two, whereas the memory performance optimizations presented here can further improve the decoder by another factor of about two.

We analyze the distributions of cycles-per-instruction (CPI) in the core functions of an optimized software MPEG-2 decoder and find that its performance bottleneck on today's computers is at memory operations. The CPI of the motion compensation module is 1.81 while the CPI of the display is 10.57. Based on the principle of concurrency, we proposed and evaluated three optimization techniques to remove the performance bottleneck, including an interleaved block-order data layout to improve the data cache locality, an algorithm to explicitly prefetch macroblocks, and an algorithm to schedule interleaved macroblock decoding and displaying. Our evaluations show that each optimization improves certain aspect of the memory performance and that combining all three optimizations can remove the memory performance bottlenecks in a software MPEG-2 decoder almost completely. The resulting software decoder can decode and display 1280×720-resolution HDTV streams at over 62 frames per second on a 933 MHz Pentium III PC without special hardware support.

We have not tested the decoder with 1080i format HDTV streams, because the graphics card does not support overlay surfaces as large as 1920×1080. However, the decoder itself is not limited by the resolution. We plan to find means to evaluate our algorithm for higher resolution streams.

Although we have only implemented and evaluated the proposed methods on a Pentium III platform, our methods can be applied to other similar architectures that may be used to drive a tiled display, except for the specific prefetching instructions used.

Finally, we remark that, as with many other cache and memory related research, the design choices made here are heavily dependent on the characteristics of the underlying architecture, for example, the size of the L1 and L2 cache, the relative clock speed of the processor and memory, the implementation of the write buffer, to name just a few. Dramatic architectural changes in the future would invalidate some of the techniques proposed, and thus provide new research opportunities.

Chapter 4

Parallel MPEG Video Decoding

As we recall, the framework of scalable video delivery on tiled displays allows three classes of video encodings, and four types of video sources. This creates a full spectrum of videos in terms of resolution and quality. At one end is layered videos constructed from multiple relatively low resolution video streams. At the other end, the framework allows ultra-high resolution video to be encoded in one single stream. To handle all these videos with confidence requires a powerful parallel video decoder.

Just like the architecture of a cluster-based tiled display, a parallel MPEG video decoder for such displays consists of the following major components, as depicted in Figure 4.1. A splitter first divides the input stream into small work units and sends them to the decoders. The decoders decompress the work units and restore them to pixels. In the process, they might need to communicate with each other. Finally, the decoded pixels may be redistributed before being displayed.

There are two challenges in successfully building such a parallel decoder: high performance and scalability. High performance means that the decoder should use the processors of all nodes in a tiled display in the most efficient way. We achieve this goal by using the MPEG video decoder described in Chapter 3 as the basis for our parallel

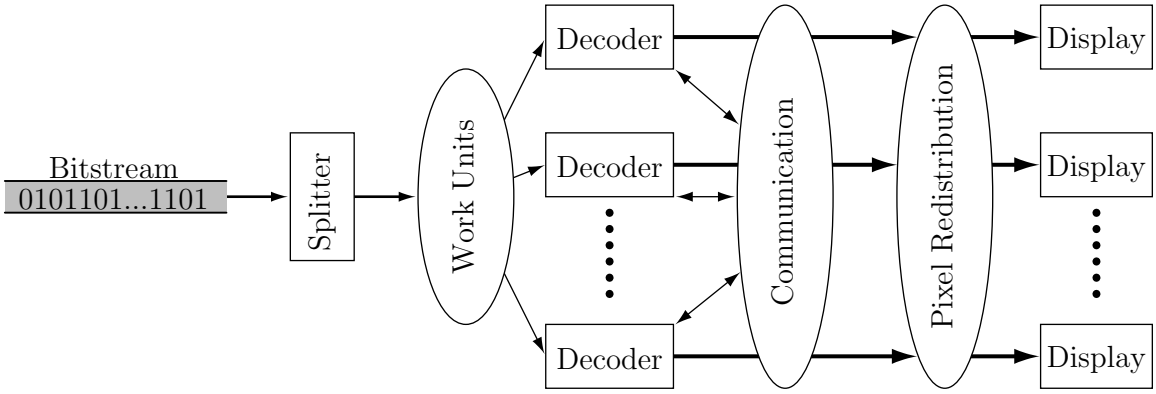


Figure 4.1: A Generalized Parallel MPEG Decoder for PC Cluster.

video decoder, and we balance the work load among the parallel decoders so that the maximum processing time of any component in the system is minimized. In a parallel or cluster system, good scalability allows it to tackle larger scale problems with added resources. In our case, this translates to higher resolution videos with more nodes. To achieve scalability requires us to remove both computation and communication bottlenecks. As in most commodity clusters, communication bottlenecks generally result from the limited bandwidth provided by off-the-shelf networking technologies.

In this chapter, we present a hierarchical parallel MPEG video decoder that is both high performance and scalable. It extends the general parallel decoder architecture by introducing two levels of splitters. A root splitter first splits a video stream at the picture level. These pictures are then passed to a number of second-level splitters, each of which splits the pictures at macroblock level. Finally, the macroblocks are sorted according to their screen locations and sent to the decoders, each of which is connected to a projector. We call such a parallel video decoder a 1-k-(m, n) system, where k refers to the number of second-level splitters, and $m \times n$ decoders drive a projector array of m-wide and n-high.

Our experiments in Section 4.3 show that such a system is highly scalable and

has a low and balanced communication requirement among the cluster nodes. In a 1-4-$(4, 4)$ setup, it decodes and plays a 3840×2800 resolution MPEG video stream at 38.9 frames a second, or an equivalent bit rate of 130 Mbps.

The rest of this chapter is organized as follows. Section 4.1 discusses the issues in parallel decoding for a tiled display system and related previous work. Section 4.2 presents our parallel decoder and discusses design issues. Section 4.3 reports our experimental results. Finally, Section 4.4 summarizes what we have learned in our study and points out future work.

4.1 Background and Related Work

Patel *et al.* first described their software MPEG video decoder in [71]. Subsequent work has followed two directions to improve decoding performance. One of them is the use of multimedia instruction set extension, which has been the subject of discussion for the last chapter. Another way is to parallelize the decoder so that it can run on a number of processor/node. This can be done in different ways and at different levels.

4.1.1 Parallelization of MPEG Video Decoding

There are two main approaches to the parallelization of MPEG video decoding: *Functional Parallelization* and *Data Parallelization.*

In functional parallelization, a decoder is first partitioned into several functional units, for example, VLD, IDCT, MC, and display, as mentioned in Chapter 3. Each unit is then assigned to a processor in an SMP machine or a node in a cluster. The input stream is transformed at each processing stage and finally becomes a frame to be displayed. To draw an analogy from microprocessor architecture, this is like a pipeline design. The advantage of functional parallelization is its simplicity. It can be

achieved by simply running each functional unit as a thread and passing data between units through shared memory or remote procedure calls (RPC). However, there are many disadvantages to this approach. First, the number of functional units in a decoder is limited. As the number of processors/nodes grows, it becomes increasingly hard to partition a decoder in successively smaller logical units. Second, it is not easy to balance the loads on processors, because the times a decoder spends on all units are not necessarily the same. Third, the amount of communication needed between units is very large. For example, after the VLD stage, all communication happens in pixel domain, even a modest 1080i HDTV video (1920×1080 interlaced at 30 fps) requires close to 1 Gb/s bandwidth between units. Therefore, functional parallelizations only work well on SMP's with a small number of processors. It does not map well to a cluster architecture, let alone being scalable.

In data parallelization, the decoder itself is not the target of partitioning, instead the video stream (data) is split into multiple work units. Several decoders work on these work units in parallel to decode the video. To continue the analogy of microprocessor architecture, this is like a superscalar design. Because MPEG video streams are defined to contain a hierarchy of elements, each of which can be the level where parallelization happens, we have to consider each level of parallelization and determine its advantages and disadvantages.

For simplicity, we assume that the tiled display consists of $m \times n$ nodes each with one processor and one attached projector arranged in a rectangular array of m-wide and n-high. Assuming readers' familiarity with the MPEG terminologies described in Section 3.1, we discuss four parallelization levels with regard to potential bottlenecks, that is, the cost of splitting, the inter-decoder communication cost, and the pixel redistribution cost.

Group of Picture or Sequence: This represents some of the easiest ways to parallelize an MPEG video decoder. Because there are byte-aligned start codes for GOP's and sequences in an video stream, the cost of splitting is next to nothing. A splitter only needs to scan the bitstream once, therefore it can process the input video stream as fast as it can read the data from either the network or the disk. Furthermore, sequences and most GOP's (closed GOP's) are self-contained, that is, no motion estimation references pictures outside this work unit. Thus, the decoders do not need to communicate with each other. However, these advantages do not come without some extremely high cost of pixel redistribution. For each picture in a GOP or sequence, a decoder needs to send $(mn-1)/(mn)$ of it to other nodes for display. Besides this, a GOP/Sequence level parallel decoder has to buffer a large number of pictures before they can be displays, this could be problematic for nodes with limited memory resource.

Picture: Like its higher level counterparts, a picture is marked with a byte-aligned start code too. Therefore, the splitting cost is minimum at this level. Unlike a closed GOP or a sequence, a picture is usually not self-contained. Only I-pictures can be fully decoded alone. For P- or B-pictures, the decoders need to fetch reference blocks from previously decoded pictures. Most of the time, they reside on remote nodes. The exact amount the data that need to be transferred depends how the video stream is encoded; up to one entire picture is needed for a P-picture, and two for a B-picture. The pixel redistribution cost is the same as in GOP level parallelism, although the buffering requirement is much lower than at GOP level—only one picture needs to be buffered at any node.

Slice: Slice is the lowest level at which a start code is provided in the video stream. This means that the splitting cost remains minimum. Inter-decoder commu-

nication is somewhat reduced, when decoders are assigned slices from fixed locations of the video frame. This way, a decoder only needs to fetch remote data when a macroblock references data outside the screen region covered by the work unit. In typical MPEG video streams, a slice covers an entire row of macroblocks. A decoder needs to ship out $(m-1)/m$ of it to other nodes for display. Therefore the pixel redistribution cost is lower then that of a higher level parallelization mode. A decoder needs to collect and buffer $1/mn$ of a picture before it is displayed.

Macroblock or Block: At this level, the dynamics of the decoder start to change. First of all, the splitting cost becomes high, due to the lack of start codes for macroblocks and lower level elements. A splitter has to parse an entire picture in order to access a macroblock or block. At this fine granularity, the splitter can sort macroblocks into $m \times n$ bins, each of which represents a rectangular region as defined by the projector's screen location. Thus, inter-decoder communication is further reduced. A decoder needs to fetch remote blocks only when a motion vector crosses the boundary of the rectangle. The greatest advantage of this parallelization model is that the pixel redistribution is completely eliminated, because macroblocks are only sent to where they will be displayed. The buffering cost is also $1/mn$ of a picture, as in a slice level parallel decoder.

We summarize these discussions in Table 4.1. It is very clear that none of these parallelization methods suffices in itself. In a coarse-grained parallelization, i.e., at slice or higher level, the splitting cost is very low, but the communication cost is high. In a fine-grained parallelism, i.e., at macroblock or lower level, the communication cost is low and distributed, but the splitting cost is high, which becomes a bottleneck when the number of decoders increases.

Table 4.1: Comparison of Different Types of Parallelization.

Level	Splitting Cost	Communication Requirement	Pixel Redistribution
Sequence	very low	none	very high
GOP	very low	none or low	very high
Picture	very low	very high	very high
Slice	very low	moderate to high	moderate to high
Macroblock or Block	high	moderate	none

4.1.2 Previous and Related Work

Bilas *et al.* proposed a parallel decoder on a shared memory SMP [10] and investigated parallelism at both GOP and slice levels. Running on an SGI Challenge with 16 150 MHz MIPS R4400 processors, it achieves real time frame rate for DVD resolution videos and 7 fps for HDTV resolution videos. Kwong *et al.* designed a GOP level parallel decoder on an IBM SP parallel computer using a HIPPI network [53]. It decodes 352×240 videos at 30 fps with 16 nodes of IBM RS/6000 model 370.

Bala *et al.* proposed a functional parallelization on a 2-CPU SMP [6]. One thread is used for run length decoding, IDCT, and motion compensation, while another does the dithering and display. As we have argued in Section 4.1.1, this approach has scalability problems, and does not map to a cluster architecture well.

In the hardware front, Lee *et al.* described an MPEG codec on a single-chip multiprocessor [56] exploiting both instruction level and functional parallelism. Yang *et al.* proposed a variable issue architecture for multi-threading MPEG processing [97], but no parallelism model and performance data were given. In the related MPEG encoding field, Yu *et al.* investigated macroblock and slice level parallelism in encoding [98]. Akramullah also proposed a data parallel encoder [2].

Most of these methods use a shared memory SMP or equivalent platform. They focus on improving performance with more processors, but have not considered the issues in scaling video resolutions. Because of the high memory bandwidth requirement of ultra-high resolution videos, it is impossible for an SMP to display such videos even if it can decode them with enough number of processors.

It is clear that none of the previous work applies to a cluster architecture, especially for high resolution videos. Although a data driven parallel decoder is the obvious choice, none of the simple parallelization methods is sufficient for building a scalable video decoder for tiled display, due to bottlenecks in either computation or communication. This motivates us to study and design a new parallelization model that is bottleneck-free.

4.2 Hierarchical Decoding

Here we present a new parallelization method of MPEG video decoding—Hybrid Granularity Hierarchical Decoder. This is a data parallel decoder that overcomes the limitation of the four simple parallelizations described before.

We have two main design goals: high performance and good scalability. To put it another way, it should decode video streams at high frame rate, and be able to handle higher resolution videos with more nodes. To achieve high performance video decoding, we leverage on the uni-processor decoder described in Chapter 3. We also need to balance the computation work loads of multiple decoders[1]. Scalability requires the system to be bottleneck-free in terms of both computation and communication.

[1]Strictly speaking, we need to minimize the maximum of the execution times of all decoders in order to achieve maximal performance. Generally, this is not the same as achieving the most balanced work load; because migrating work load among nodes will incur additional communication overheads. Nevertheless, a balanced work load is a good indicator of high performance, especially when all the nodes are homogeneous, as in our case.

4.2.1 Hybrid Granularity Hierarchical Decoder

As the discussions in the previous section point out, a macroblock level parallel decoder has low and balanced communication requirement, but the high splitting cost causes the splitter to become a computation bottleneck for videos with resolution higher than HDTV.

Naturally, we want to use more than one macroblock-level splitters so that all the decoders can be fed at full speed. This leads to a hierarchical parallel decoder. The question that remains is at what granularity the multiple macroblock-level splitters should be fed. Clearly, it has to be picture or higher, so that the macroblock-level splitters can still sort the macroblocks according to their screen locations. One might be tempted to choose GOP or sequence level, for they are self-contained. However, this requires long buffering in the second-level splitters and causes long latency in the entire decoder. Fortunately, this is not necessary.

As noted before, P- and B-pictures have dependencies on previous pictures when they are being decoded. However, this dependency does not exist when they are only being split—a splitter only parses and decodes a picture, but does not perform the actual motion compensation. Thus multiple pictures can be assigned to independent splitters simultaneously, thereby eliminating the bottleneck.

Here is how a hybrid granularity hierarchical decoder works. The system consists of one or two levels of splitters and a set of decoders. For relatively low-resolution videos, such as DVD or HDTV, a single macroblock-level splitter is adequate. For higher resolution videos, the system uses a root splitter to first split the input video at picture level. We call it a P-Splitter. It passes pictures to multiple second-level splitters, which split them at macroblock level to feed the decoders. We call a second-level splitter an M-Splitter. All together, we call the entire decoder a 1-k-(m, n)

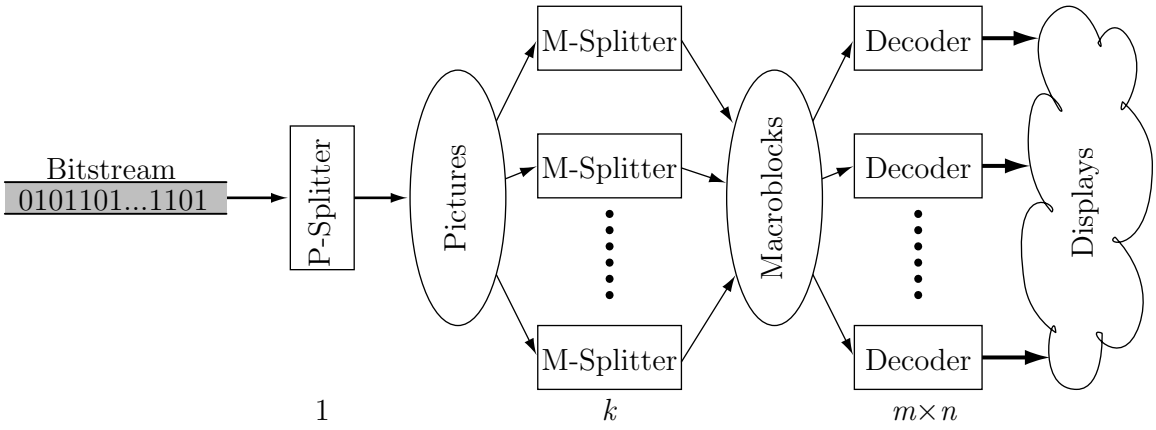


Figure 4.2: Hierarchical Parallel Video Decoder: a 1-k-(m, n) System.

system for a hierarchy of one root P-Splitter, k second-level M-Splitters, and $m \times n$ decoders in an $m \times n$ tiled display system, as illustrator in Figure 4.2. The P-Splitter runs on a console node while the k second-level M-Splitters run on k additional nodes. A single-level macroblock parallel decoder is a special case where $k = 1$; we call it a 1-(m, n) system.

Figure 4.3 lists the high level algorithms for the P-Splitter, the M-Splitters, and the decoders.

P-Splitter It scans a bitstream to find out where a picture starts and ends using the start code. It then copies the picture data to an output buffer and sends it to one of the k second-level splitters. For Constant Bit Rate (CBR) MPEG video streams, each picture contains approximately the same bits. Therefore, we choose M-Splitters in a round-robin fashion as a simple yet effective way to balance the workload[2].

M-Splitter A second-level M-Splitter parses a picture into macroblocks, and sorts

[2]For Variable Bit Rate (VBR) video streams, the current lightest-loaded M-Splitter can be chosen for the next picture to balance workload. The picture ordering algorithm proposed in Section 4.2.5 can be readily modified to accommodate this, although we have not implemented this.

```
P-Splitter:
while there are more pictures in the stream
    Copy the current picture P into an output buffer
    Select an M-Splitter
    Send the picture to the M-Splitter
```

```
M-Splitter:
while splitting has not finished
    Receive a picture P from the P-Splitter
    for each macroblock M
        for each decoder D_i
            if M lies in the rectangle of D_i
                Append M to buffer SP_i
    Send SP's to D's
```

```
Decoder:
while decoding has not finished
    Receive SP from M-Splitter
    Decode SP; fetch blocks remotely if necessary
    Display the SP
```

Figure 4.3: High Level Algorithms of a Hierarchical Decoder.

them into $m \times n$ output buffers. We call them sub-pictures (SP). They do not necessarily conform to MPEG syntax. The splitter then sends the sub-pictures to the corresponding decoders. Because there is no inter-picture dependency, the M-Splitters operate independently.

Decoder A decoder (D) receives a sub-picture and decodes it one macroblock at a time. When a macroblock's motion vector crosses the boundary of the decoder's screen rectangle, the decoder needs to fetch blocks remotely. Finally, the sub-picture is displayed after all macroblocks have been decoded.

The description of the algorithm above is straightforward. But careful designs are

required to make it efficient. In the following subsections we discuss several design issues we face, and the choices we make.

4.2.2 Cooperative Prefetching of Remote Macroblocks

Although conceptually simple, fetching remote blocks on demand can be inefficient. First, when a decoder needs remote macroblocks, it has to issue an network request and then block and wait. Second, a dedicated server thread is required in each decoder to handle these requests from others; this introduces many context switches.

We can solve this problem by using the idea of prefetching. Although a decoder can not foresee which macroblocks it will need when decoding a sub-picture, an M-Splitter is in the perfect position to accurately predict this. Because it parses an entire picture during splitting, it can decode the motion vectors associated with each macroblock and determine whether any of them crosses the rectangular boundary of the target decoder, in which case, the decoder needs to fetch a reference macroblock remotely. Put it another way, after a picture is split, the M-splitter has all the knowledge of macroblock exchanges that will happen later.

To take advantage of this, we add an additional data structure called *Macroblock Exchange Instructions* (MEI) to each decoder. When a motion vector of macroblock M in the rectangular decoding region of D_i crosses the boundary and references blocks from another decoder D_j, the M-Splitter appends an instruction SEND(x, y, i) to MEI$_j$ and RECV(x, y, j) to MEI$_i$. Here (x, y) is the coordinate of the reference block. After the entire picture is split, the splitter sends MEI$_i$ and SP$_i$ to D_i.

When a decoder receives the MEI and SP, it executes the SEND instructions before decoding the sub-picture. This is possible because in MPEG video streams the reference blocks are always from previously decoded pictures. These blocks are then

stored in the buffers provided to the network transport. When user-level communication protocols are used, these buffers are in the user space, therefore no extra copying is required. During decoding, when a decoder needs a remote block, it can read the block from the buffers *locally* without blocking. The RECV instructions provided in the MEI serve as a verification, they essentially tell the decoder which decoder and where on screen this remote block is from.

We call this method *Cooperative Prefetching of Remote Macroblocks*. It eliminates the need for multi-threading in decoders and reduces the overhead of blocking receive to a minimum. The message exchanges among the decoders also implicitly serve as a way of synchronization, so that no two are off by more than one frame.

4.2.3 Decoder State Propagation

MPEG video compression tries to remove redundancy in almost every aspect of a video stream. Within a slice, the Direct Current (DC) coefficient and motion vectors of a macroblock are predicted from those of the last macroblock [43]. Only the very first macroblock in a slice uses known fixed values for these predictors. After an M-Splitter parses a picture into sub-pictures, each decoder might only get a portion of an original slice. To decode these partial slices properly, a decoder needs assistance to obtain the initial predictors.

Fortunately, as with the Cooperative Prefetching of Remote Macroblocks, an M-Splitter has enough knowledge to provide these predictors to all decoders. To propagate the information efficiently, we create a *State Propagation Header* (SPH) for sub-picture bit streams. When two adjacent macroblocks in a sub-picture are not from the same slice, we insert an SPH between them. The header contains all crucial information to start a partial slice, including the macroblock mode, current DC

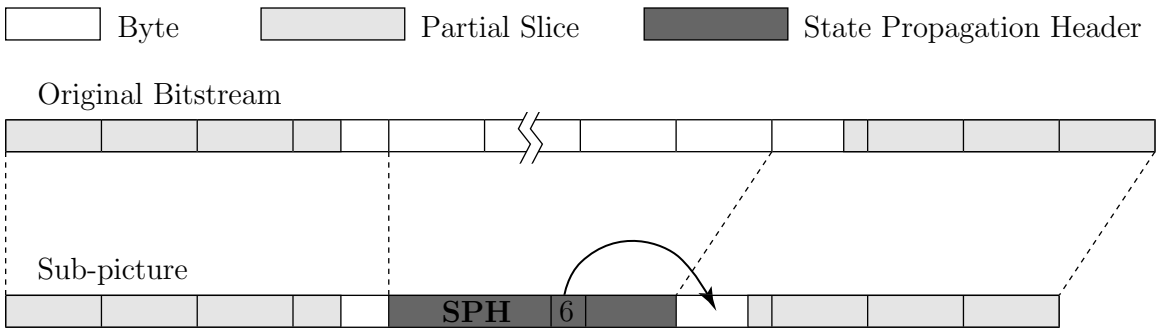


Figure 4.4: Decoder State Propagation for Partial Slices.

coefficients, motion vector predictors, and macroblock address increment.

As mentioned in the MPEG overview of Section 3.1, macroblocks do not necessarily start or end at byte boundaries. To avoid costly bit shifting operations for re-aligning partial slices, we copy all the bytes that contain a partial slice from the original stream, and specify in SPH how many bits (0 to 7) should be skipped at the beginning; see Figure 4.4. Although SPH and the unused bits increase the size of bitstream, every row of macroblocks in a sub-picture needs only one header. We show in the evaluation section that the overhead introduced is small.

4.2.4 Zero-Copy Data Transfer

We use the GM library [70] over Myrinet [11] for fast user-level communication. To avoid memory copies in sending and receiving network messages, we design the protocol such that the receiver always posts receive buffers before the sender sends any data. A receiver can be either a second-level M-splitter or a decoder, and a sender is either the root P-splitter or an M-splitter, respectively.

We use two receive buffers to implement a simple flow control; see Figure 4.5. Initially, a receiver posts two receive buffers. After it receives and consumes a message,

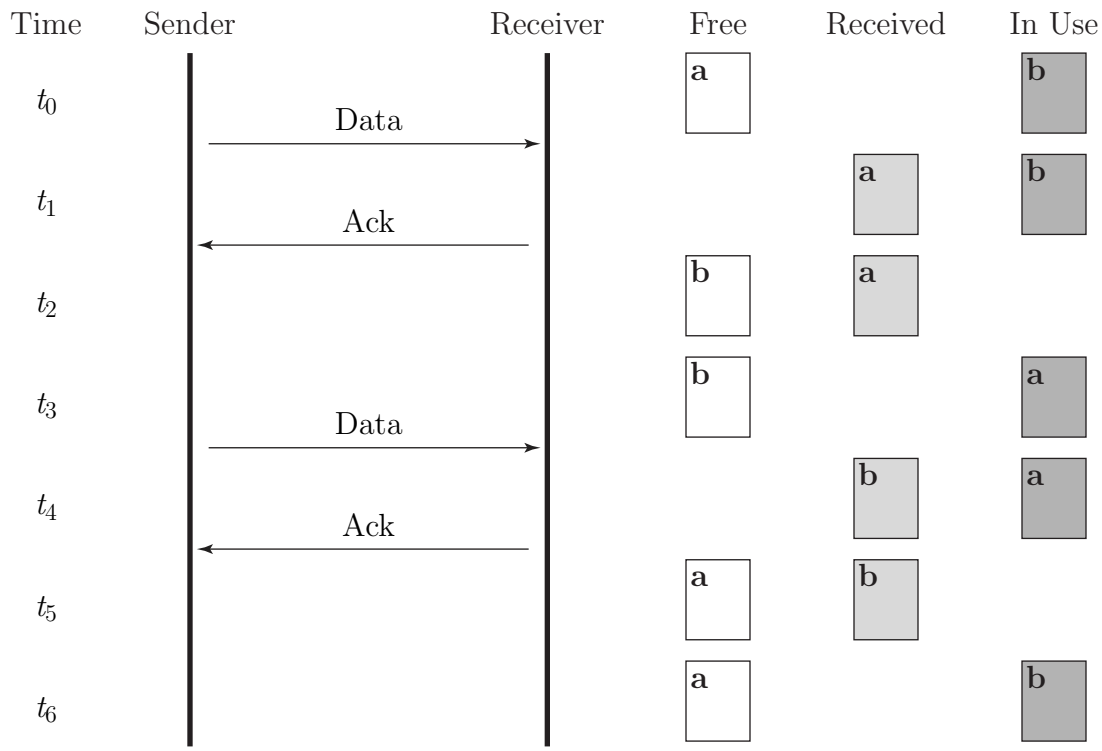


Figure 4.5: Flow Control Protocol with Double Buffering

the receiver recycles the previous receive buffer and sends an ack/go-ahead message to the sender to indicate that a new receive buffer is available. A sender first performs its work. Then, except for the first time, it waits for an ack from the receiver before sending the data. Because these buffers reside in user address space, it eliminates memory copy and minimizes the time spent on blocking receives.

4.2.5 Ordering of Pictures

Because GM does not maintain the order of messages from different senders, a protocol is required to keep the proper arriving order of sub-pictures at the decoders. A naive solution is to attach a frame number to each sub-picture. However, this requires the decoders to keep a queue of incoming pictures and reorder them. Instead, we make use of the ack/go-ahead messages in our protocol to eliminate queuing.

When multiple M-splitters exist, we re-direct a decoder's ack message to the next M-splitter instead of the sender of the sub-picture. Consider k M-splitters. When the splitter a sends sub-pictures to the decoders, it also sends the node id of the M-splitter that is responsible for the next picture, i.e., $b = (a + 1) \mod k$. We call it `ack-node-id` (ANID). When a decoder receives a message from any M-splitter, it first extracts the ANID; then instead of sending the ack message to the sender, it sends the ack to node ANID. This allows splitter b to send the next picture. In essence, this creates a time sharing scheme of the original flow control.

To make the number of second-level M-splitters flexible, we hide the information about M-splitters from each other. Instead, the root P-splitter will send the `next-splitter-id` (NSID) along with the picture data to an M-splitter. This algorithm can readily accommodate dynamic load balancing for M-splitters; the P-splitter picks the next splitter according to the current loads of M-splitters, instead of using a static round-robin algorithm. No changes to M-splitters or decoders are needed.

Figure 4.6 lists the refined algorithms for the P-splitter, the second-level M-splitters, and the decoders. Figure 4.7 shows the flow of work units and messages in a system where $k = 2$. Since message exchanges among decoders are irrelevant for this discussion, we treat them as one entity and use one column for all the decoders.

4.2.6 Configuration Determination

Currently, we determine the configuration of the 1-k-(m, n) system empirically, based on the video stream resolution.

We determine m and n by matching the video resolution with the resolution of a tiled display wall. For example, a 3840×2800 video stream would require $m = 4$ and $n = 4$ if the resolution of each tile is 1024×768.

P-splitter:

$a = 0$

while there are more pictures

Copy a picture P to send buffer

Wait for ACK from any splitter, except for the first picture

Send P to splitter a, with NSID $= (a + 1) \mod k$

$a = (a + 1) \mod k$

M-Splitter:

Post two receive buffers for incoming messages

while there are more pictures

Recycle the previous receive buffer

Receive picture P from root, with NSID $= b$

Send ACK to root

for each macroblock M

for each decoder D_i

if M lies in the rectangle of D_i

Append M to buffer SP_i

if M references data at (x,y) of D_j

Append SEND(x,y,i) to MEI_j

Append RECV(x,y,j) to MEI_i

Wait for ACK from all decoders, except for the very first picture in a stream

Send MEI's and SP's to D's, with ANID $= b$

Decoder:

Post two receive buffers for incoming messages

while there are more pictures

Recycle the previous receive buffer

Receive MEI and SP from splitter, with ANID $= b$

Send ACK to splitter b

Execute SEND instruction in MEI

Decode SP; get remote macroblocks according to RECV's

Display the picture

Figure 4.6: Refined Algorithms of a Hierarchical Decoder.

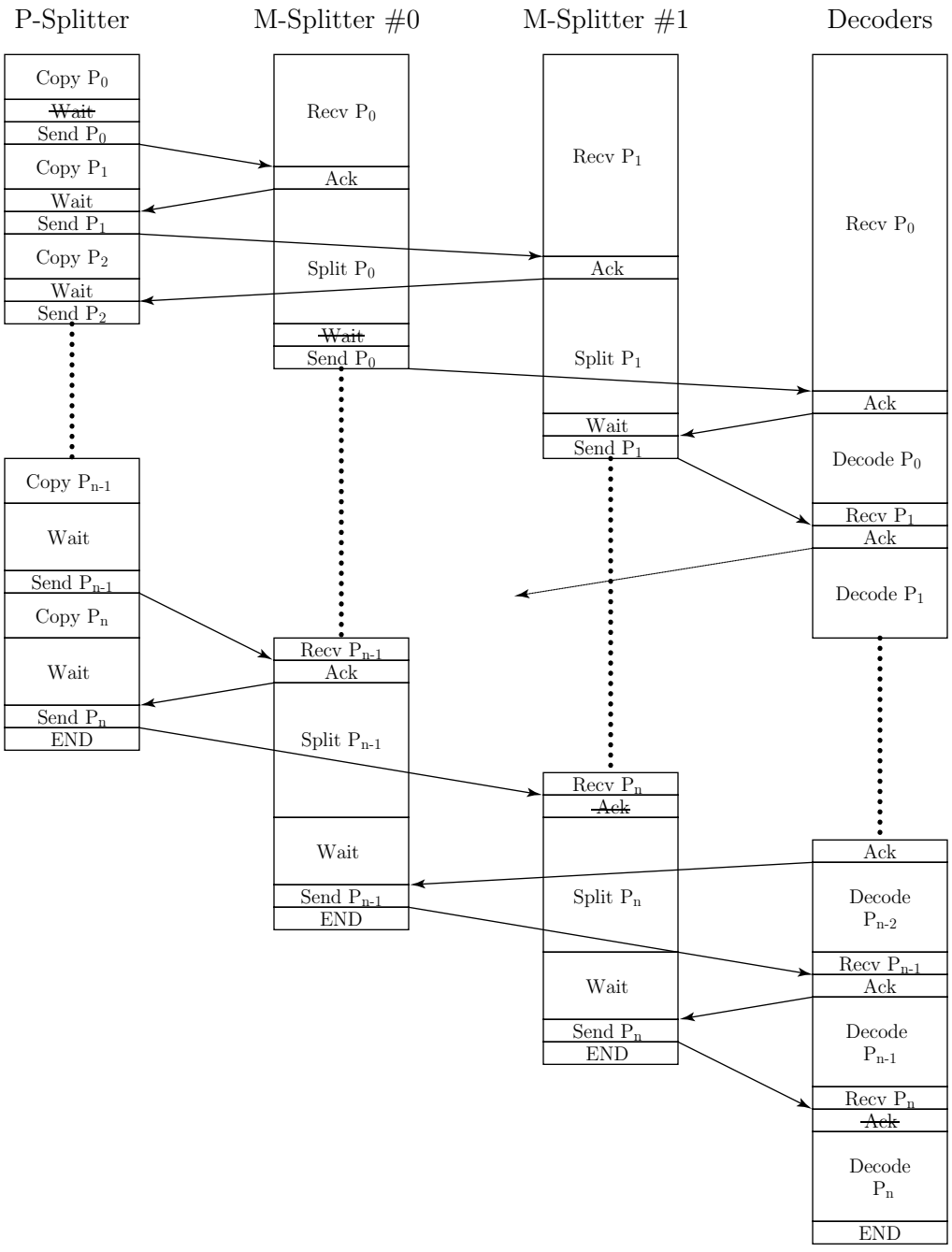


Figure 4.7: Flow of Work Units and Messages in a Hierarchical Decoder.

We determine k by matching the speed of M-splitters and decoders. Suppose it takes t_s to split a picture at macroblock level; the aggregated throughput of all M-splitters, calculated as frames per second, is $F_s = k/t_s$. If it takes the decoders t_d to decode and display the sub-pictures, the maximum frame rate achievable is then $F_d = 1/t_d$. The overall performance of the system is determined by the slowest link, that is, the frame rate is:

$$F = \min(k/t_s, 1/t_d)$$

When $t_s > k \cdot t_d$ the M-splitters are the bottlenecks in the system, and the decoders are not running at full speed. When $t_s \leq k \cdot t_d$ the decoders are running at full speed. Therefore the optimum value of k is $\lceil t_s/t_d \rceil$. If this value equals 1, we can save the second-level splitter by using a 1-(m, n) system.

4.3 Experiments and Results

We implement the hierarchical parallel MPEG decoder based on the high performance video decoder presented in Chapter 3. We evaluate its performance on the Princeton Scalable Display Wall system. The goal of the evaluation is to verify 1) whether the proposed parallel decoder is scalable and ultimately 2) whether it maps well to the framework of scalable video decoding on tiled display.

We investigate its scalability by asking the following questions:

- When do we need two-level splitting?
- For a given video, does the performance scale with increasing number of nodes?
- Does the decoder scale with increasing video resolution?

We further prove that the decoder satisfies the need for the framework by demon-

strating that it decodes an ultra-high resolution video at interactive frame rate while using only low network bandwidth.

4.3.1 Methodology

We conduct our experiments on a tiled display with commodity parts. Different portions of the display are used to evaluate the frame rate scalability of the decoder. Further, we select a number of video streams with increasing resolutions to test the resolution scalability of the decoder.

Test Platform

We have briefly described the tiled display system used for experiments in Section 2.1.5. To recap, it consists of 24 DLP projectors in a 6×4 configuration. The resolution of each projector is 1024×768. After accounting for the overlaps, the effective total resolution is about 6000×3000. The display wall has 25 PCs in a cluster connected by Myrinet. The console PC has a 550 MHz Pentium III processor and 1 GB of SDRAM. Each projector is driven by a PC with a 733 MHz Pentium III processor, 256 MB of RDRAM, and an NVIDIA GeForce2 GTS graphics card.

Our experiments use a fraction of the entire system. We use the console PC as the P-splitter, up to 5 PCs in the display cluster as second-level M-splitters, and up to 16 PCs in the display cluster as decoders. The screen configurations in our experiments include 1×1, 2×1, 2×2, 3×2, 3×3, 4×3, and 4×4.

Because of the overlapping regions between adjacent projectors, some macroblocks are sent to multiple decoders. This causes some overhead, especially for low resolution videos, as we will notice in the evaluations.

Table 4.2: Characteristics of Test Video Streams.

Stream	Name	Resolution	Average Frame Size (Byte)	Bit Per Pixel
1	spr	720×480	28873.75	0.668
2	matrix	720×480	25032.65	0.579
3	t2	720×480	32810.20	0.759
4	anim7.5	1000×750	31734.24	0.338
5	fish2	1280×720	35223.68	0.306
6	fish3	1280×720	35185.45	0.305
7	fish6	1280×720	35168.18	0.305
8	fish8	1280×720	35239.03	0.306
9	fox5	1280×720	30925.03	0.268
10	nbc4	1920×1080	74249.08	0.286
11	cbs3	1920×1080	75261.80	0.290
12	anim30	2000×1500	125693.42	0.335
13	orion40	2880×1440	166852.45	0.322
14	orion60	2880×2160	250388.21	0.322
15	orion80	3840×2160	333758.54	0.322
16	orion100	3840×2800	416968.77	0.310

Test Video Streams

We use a total of 16 MPEG video streams to test the performance and scalability of the system; see Table 4.2. Their resolutions range from DVD to near IMAX.

Stream 1 to 3 are clips from the movie Saving Private Ryan, The Matrix, and Terminator 2, respectively. Stream 4 is a scene from a short animation made by Adam Finkelstein. Streams 5 through 8 are shots of a fish tank taken with an HDTV video camera, courtesy of Intel MRL. Stream 9 is a video clip recorded from the HDTV broadcast of FOX5 in 720p format. Streams 10 and 11 are clips recorded from NBC4 and CBS3 in 1080i format respectively. Stream 12 has the same content

as stream 4 but is rendered at quadrupled resolution. Streams 13 through 16 are compressed from results of a fly-by visualization of the Orion Nebula, courtesy of UCSD supercomputing center.

All streams except the first three have about the same bit rate per pixel, 0.3 bpp. This translates to a bitstream rate of about 20 Mbps for 720p HDTV streams 60 fps, 1080i streams at 30 fps, and about 100 Mbps for the highest resolution Orion fly-by sequence at 30 fps. The first three streams are compressed for movie DVD's, and have a higher bit rate. Each sequence contains 240 frames.

4.3.2 Performance of One-Level Splitting

The goal of this experiment is to evaluate the performance of a one-level system and to investigate when the M-splitter in a 1-(m, n) system becomes a bottleneck. We played the streams `spr` (DVD) and `fish8` (720p HDTV) on a one-level system with screen configurations ranging from 1×1 to 4×4. The frame rates are listed in the left half of Table 4.3. A plot of frame rate versus total number of nodes $(1 + mn)$ are shown as the dashed lines in Figure 4.8.

We can see that when the number of decoders exceeds 4, the M-splitter can no longer keep up with the decoders and becomes a computation bottleneck. When the number of decoders further increases, the frame rates actually drops slightly. We will explain this in the next subsection.

4.3.3 Two-Level Splitting Frame Rate Scalability

In this experiment, we try to remove the computation bottleneck with a hierarchical decoder. A two-level 1-k-(m, n) system is used, and we determine k by increasing it until the overall frame rate stops increasing. The frame rates are shown in the right

Table 4.3: Frame Rate of One-Level and Two-Level Systems.

One-Level System				Two-Level System			
`spr`		`fish8`		`spr`		`fish8`	
Config	fps	Config	fps	Config	fps	Config	fps
1-(1, 1)	97.8	1-(1, 1)	49.9	1-(1, 1)	97.8	1-(1, 1)	49.9
1-(2, 1)	160.3	1-(2, 1)	90.7	1-(2, 1)	160.3	1-(2, 1)	90.7
1-(2, 2)	242.2	1-(2, 2)	153.2	1-2-(2, 2)	245.1	1-2-(2, 2)	155.4
1-(3, 2)	260.2	1-(3, 2)	142.4	1-2-(3, 2)	292.0	1-2-(3, 2)	191.1
1-(3, 3)	243.8	1-(3, 3)	133.2	1-2-(3, 3)	330.0	1-3-(3, 3)	242.3
1-(4, 3)	231.4	1-(4, 3)	125.0	1-3-(4, 3)	360.3	1-4-(4, 3)	274.8
1-(4, 4)	219.8	1-(4, 4)	116.6	1-3-(4, 4)	398.4	1-5-(4, 4)	315.9

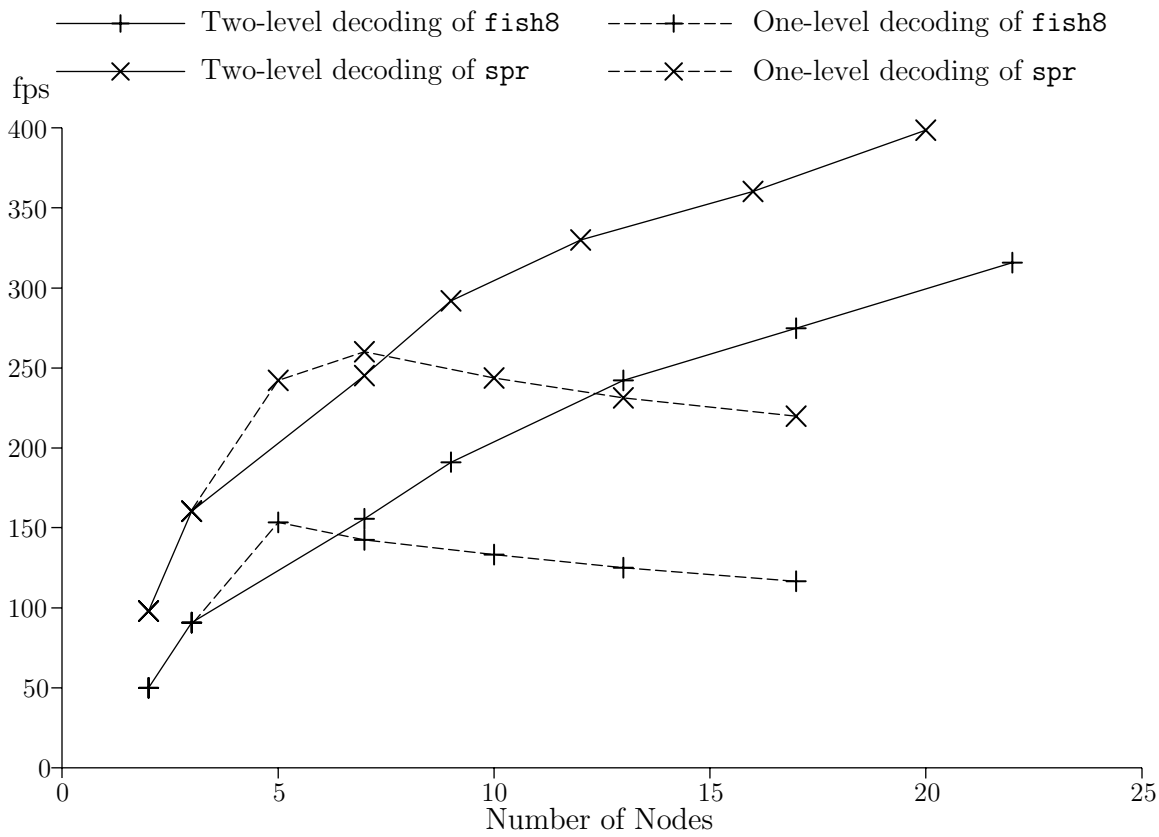


Figure 4.8: Frame Rate of One-Level and Two-Level systems.

half of Table 4.3. Two plots of frame rate versus total number of nodes $(1 + k + mn)$ are shown as the solid lines in Figure 4.8. It is clear that the computation bottleneck is removed. The system is able to decode both videos at higher frame rates as the number of decoders increases.

However, the acceleration is less than linear. This occurs for the same reason that the frame rate of a one-level system drops after saturation. Given a fixed resolution video stream, each decoder is responsible for fewer macroblocks when the number of decoders increases. Thus, the percentage of macroblocks that reference remote blocks increases. As a result, the decoders spend more time in performing Cooperative Prefetching. To illustrate this, we profile the decoders in both 2×2 and 4×4 settings for `fish8` and break down the running time into five parts:

Work: the time to decode and display a picture

Serve: the time to prepare data for remote decoders

Receive: the time waiting for sub-picture from M-splitters

Wait: the time waiting for remote blocks

Ack: the time to send acks to M-splitters

Figure 4.9 shows the running time breakdown of each decoder and their average for both the 2×2 and 4×4 setups. We can see that the majority (about 80%) of the running time is spent in decoding in a 1-2-$(2, 2)$ system. However, only about 40% of the total running time is spent in decoding in a 1-5-$(4, 4)$ system. The percentage of serving remote decoders increases significantly because more macroblocks reference remote blocks. Also, increased contention in the network causes the receiving time to increase slightly. All these factors contribute to the slow down of the overall system.

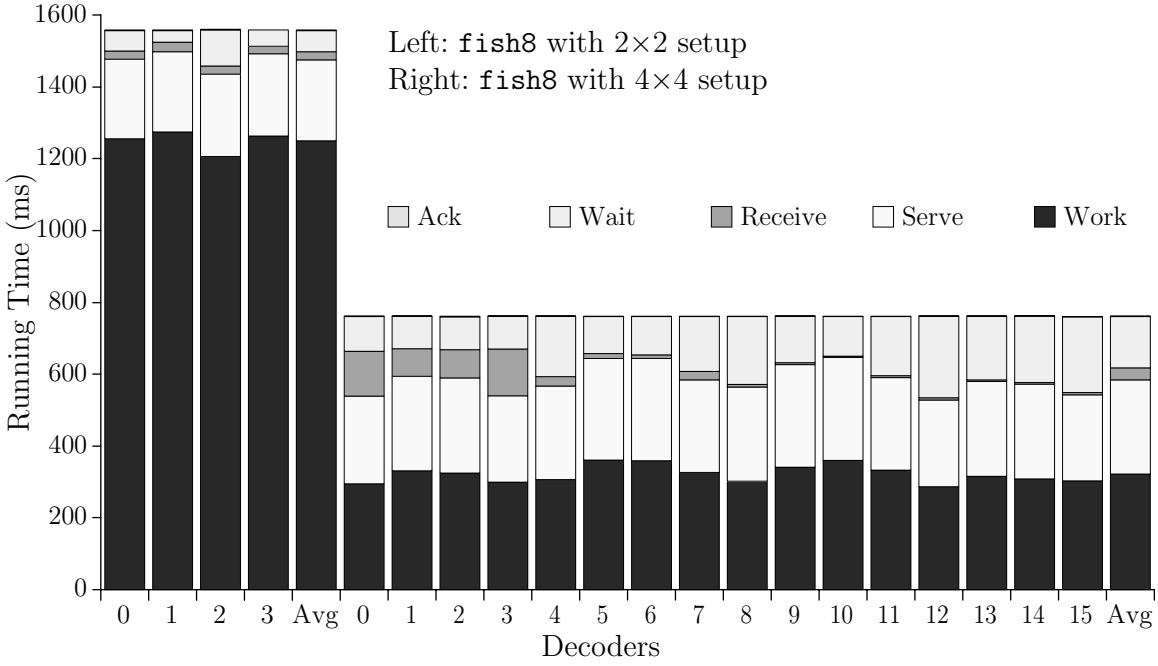


Figure 4.9: Running Time Breakdown of Decoders.

As stated before, frame rate scalability is not critical once we achieve the real time frame rate. The more important question is whether the system can decode higher resolution streams with more decoders.

4.3.4 Two-Level Splitting Resolution Scalability

To test the resolution scalability of our two-level system, we run each of the 16 streams with an appropriate number of decoders such that the video resolution matches the screen resolution. As in Section 4.3.3, the number of M-splitters k is chosen to keep the decoders running at full speed. Table 4.4 shows the screen configuration, frame rate, and the rate of total decoded pixel for each stream. Because the number of pixels per decoder is not constant for these 16 streams, frame rate is not a consistent metric of system performance. Total decoded pixel is a much more accurate measure of how

Table 4.4: Frame Rate of All Test Streams in Two-Level Systems.

Stream	Config	Num of Nodes	Frame Rate (fps)	Pixel Rate (Mpps)
1	1-(1, 1)	2	97.8	33.8
2	1-(1, 1)	2	105.0	36.3
3	1-(1, 1)	2	99.8	34.5
4	1-(2, 1)	3	102.1	76.6
5	1-(2, 1)	3	94.6	87.2
6	1-(2, 1)	3	87.4	80.5
7	1-(2, 1)	3	89.2	82.2
8	1-(2, 1)	3	90.7	83.6
9	1-(2, 1)	3	105.0	96.8
10	1-2-(2, 2)	7	88.8	184
11	1-2-(2, 2)	7	74.0	154
12	1-2-(3, 2)	9	60.2	181
13	1-2-(3, 2)	9	45.3	188
14	1-3-(3, 3)	13	43.0	268
15	1-3-(4, 3)	16	38.9	323
16	1-4-(4, 4)	21	38.9	418

fast the system decodes the streams. Figure 4.10 shows a plot of total pixel decoding rate versus number of nodes. When multiple streams exist for one configuration, we use the average pixel rate for the data point. It is obvious from the plot that the two-level system achieves a near linear acceleration, and scales well.

There is a slight drop of performance for the four highest resolution videos. We notice that in these videos, the majority of motion and visual details are located in a portion of the entire screen. Because an MPEG video encoder can allocate bits according to the scene complexity within a picture, in a sub-picture containing complex motion and detail, the bit-rate of the corresponding decoder is much higher than that

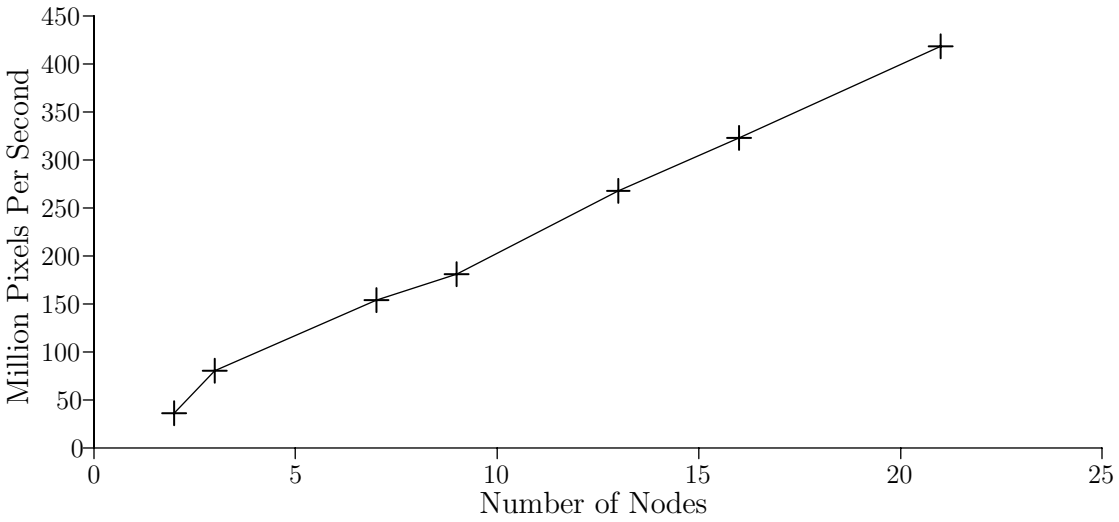


Figure 4.10: Pixel Decoding Rate of Two-Level System.

of a decoder with less motion and detail. When the decoders are synchronized, the overall frame rate is determined by the slowest decoder.

4.3.5 Bandwidth Requirement

In this experiment, we measure the send and receive bandwidth of each decoder and M-splitter in a 1-4-$(4, 4)$ system decoding stream 16 (`orion100`). The results are shown in Figure 4.11. We can see that even for an ultra-high resolution video with localized details, the communication requirement is still low; it is well within the range of current commodity network technologies. The SPH headers in sub-pictures cause the send bandwidth of an M-splitter to be larger than its receive bandwidth. However, the overhead is only about 20%.

The decoder plays a 3840×2800 resolution video at 38.9 frames per second using only a fraction of the available bandwidth of a commodity network. It shows that our parallel decoder can be used for a scalable video delivery system on tiled displays.

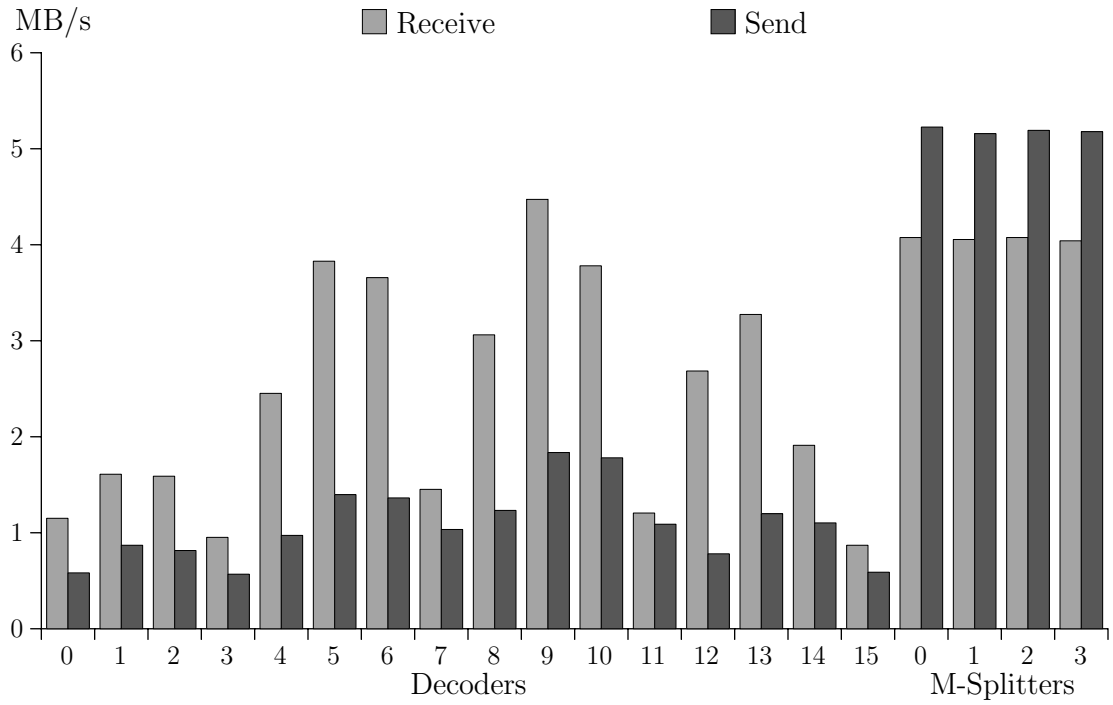


Figure 4.11: Bandwidth Requirement of a 1-4-$(4, 4)$ System Decoding `orion100`.

4.4 Summary

In this chapter, we presented the cornerstone of a scalable video delivery system for tiled displays—a hierarchical parallel MPEG video decoder. Our approach combines picture level splitting with macroblock level splitting to avoid the bottlenecks in a typical parallel decoder. Our experiments on a tiled display system show that our decoder has high performance and is highly scalable. It supports very high frame rate decoding of normal resolution video. At the same time, it can also decode ultra-high resolution video streams (stream 13–16) at real time frame rates.

Because of the existence of multiple M-splitters, there are no longer computation bottlenecks in the system. The system can scale as much as the P-splitter can parse the input stream. As we have argued before, splitting a video stream at pictures

requires a simple one-pass scanning of the input byte streams. The P-splitter can run as fast as it can read bytes from either the network or the disk. Therefore, we expect our system to perform well beyond the scales and resolutions reported here. This can be useful for scientific visualization, and immersive and collaborative environments.

Several aspects of the parallel decoding can be further studied and improved. First, each of our graphics cards drives a single projector. As we have argued before, the improvement in processor speed, memory density, network bandwidth, etc. have far out-paced that of display resolution; it would be useful to experiment with graphics cards that can drive multiple displays to further evaluate the system. Second, the decoding system currently balances workloads statically. We expect that a dynamic load-balancing method can help the splitter distribute work more evenly to fully utilize the decoders. Finally, the configuration of the system is currently chosen empirically for each sequence to maximize the performance. This is an trial-and-error process. A relatively simple technique can solve this problem—the root P-splitter automatically increases the number of M-splitters until the target frame rate is reached. Performance prediction based on video bit rate [8] can be used to estimate the required number of M-splitters so that the system can converge very quickly. Together with dynamic loading balancing, this can make the system more flexible.

Chapter 5

Seamless Video on Tiled Displays

Although a projector array is not the only way of building tiled displays, it will no doubt remain the most economical and technically feasible solution for the time being. A projector-based tiled display requires calibration to appear seamless. There are three main sources of visual artifacts: the discontinuity in overlapping regions caused by mis-aligned projectors, the brightness (or luminance) variation both within and among projectors, and the different reproductions of colors across the display.

The combined effects of these artifacts can be viewed as a modulation function applied to the image signal. For a single frame of static image, it is sometimes very hard for the human visual system to separate the modulation and the signal, especially for images with lots of high frequency details. A casual calibration performed manually is usually satisfactory in this case. However, for moving pictures, especially panning shots, the problem is much worse. With a moving content (signal), the distortion field can be easily separated and picked up by the human eyes. Therefore, high quality video delivery on tiled displays calls for a very precise calibration in all three aspects, that is, geometric alignment, luminance balancing, and color matching. A fourth implicit component of a full system is the software infrastructure for applying these

corrections to images and videos in real-time.

In this chapter, we present two components of a calibration system for tiled displays. We also describe techniques for real-time imagery correction using programmable graphics hardware.

The first component is a scalable tiled display alignment system. Existing vision-based algorithms typically rely on a single camera view and degrade in accuracy[1] as the display resolution exceeds the camera resolution by several orders of magnitude. To avoid this limitation, our system effectively integrates multiple zoomed camera views, creating a high resolution virtual camera. Our algorithm builds and refines a camera homography[2] tree to automatically register any number of uncalibrated camera images. The resulting system is fast and accurate. It also scales to displays with hundreds of projectors, as our simulation validates.

The second system is a full-gamut color matching system for tiled displays. Previous methods work on displays made of homogeneous LCD projectors, but do not handle DLP projectors well, due to the White Enhancement [52, 84] in these units. In contrast, our system is able to color match tiled displays composed of DLP projectors or mixed types of projectors using an inexpensive colorimeter. Our results show that it can achieve 1.5% color variance among projectors.

5.1 Background and Related Work

Geometric alignment is the very first problem people try to solve while building a tiled display. Although it is possible to achieve precise geometric alignment manually using a well designed and engineered projector mounting system with at least six

[1] Throughout this chapter "accuracy" is intended to mean "local registration accuracy" unless otherwise stated.

[2] We use homography synonymously with collineation: a planar transformation which maintains collinear points.

Figure 5.1: Manually Adjustable Projector Mounts with Six Degrees-of-Freedom. Left: mount designed by Princeton University and manufactured by Intel. Right: an improved design by Argonne National Laboratories.

degrees of freedom, such as those shown in Figure 5.1, this process is unfortunately very labor-intensive and time-consuming. Furthermore, the display requires frequent re-alignment because the projectors will inevitably shift due to vibration and thermal flexing in the mounts.

Recent advances in commodity graphics hardware have made it possible to pre-warp imagery in real-time to correct misalignment in a tiled display. This enables software-based alignment solutions without compromising image quality. Several ideas for camera-based automation of projector array alignment have recently been proposed. Surati [87] builds lookup tables that map pixels from each projector to points on the display surface; this is done by physically attaching a calibration grid (printed by a high-precision plotter) onto the surface. While this approach is adequate for a 2×2 array of projectors, it scales poorly for larger displays since creating and accurately mounting an absolute measurement grid onto the display surface is infeasible. The PixelFlex system [96] uses a single, wide field-of-view camera in conjunction with structured light patterns from each projector to determine camera-projector homographies, enabling automatic alignment of a reconfigurable tiled display. These

methods assume that the entire display surface is small enough to be completely visible in one camera's field of view. As tiled displays become larger, capturing a single camera image of the entire display surface becomes increasingly impractical: a single pixel in the camera maps to unacceptably large areas on the display surface once the display grows beyond a certain size.

This motivates approaches that can integrate information about projector geometries from a set of camera images, each of which observe only a small portion of the display surface. Raskar *et al.* [77] employ two calibrated cameras in conjunction with projected patterns to recover the 3-D model of a possibly non-planar projection surface. The need to calibrate camera views to the system makes this approach difficult to scale. Chen *et al.* [19] use a pan-tilt-zoom camera to observe individual overlap regions in a projector array. Information about local discontinuities, e.g. point-matches and line-matches across the seam, is acquired using an iterative process, and a large global optimization problem is constructed from this data. Simulated annealing is used to find the pre-warps that minimize discontinuity errors. The primary advantage of their algorithm (referred to as SimAnneal in the remainder of this paper) is that, in principle, it scales well to large tiled displays since the uncalibrated camera can easily scan the overlap regions. However, simulated annealing converges slowly, particularly in high dimensional parameter spaces. Furthermore, unless the initial manual alignment between projectors is good, the optimization algorithm sometimes gets stuck in local minima, resulting in reduced alignment accuracy.

Luminance balancing is largely a solved problem due to Majumder *et al.*. In her system, a calibrated high dynamic range camera [25] is used to capture the radiance output characteristics of the entire tiled display. An inverse of this field called the *Luminance Attenuation Map* (LAM) can then be applied to all images and videos to equalize the luminance output across the display [63].

Even with sub-pixel accurate geometric alignment and uniform luminance across the display, color mismatches among the projectors can still create very distracting artifacts. Majumder *et al.*presented a generalized description of the color matching problem for tiled displays and proposed a method to partially address the issue through an independent intensity matching on red, green and blue channels of all projectors [62]. Stone presents an algorithm for finding the standard gamut of LCD projectors and gives a characterization of the problems DLP projectors present in color balancing [85]. Stone also proposes *Independent Channel Balancing* (ICB).

As noted in [85], channel balancing assumes chromaticity constancy and an additive gamut. Thus, they only work for a homogeneous array of LCD projectors from the same manufacturer. However, for a tiled display containing DLP projectors or LCD projectors from different vendors, color matching becomes critical. First of all, the chromaticity values of the RGB primaries of projectors from different vendors are typically different. Second, commodity single-chip DLP projectors use a clear channel on the color wheel to boost the light output for bright colors [52, 84]. This results in a non-additive gamut, as shown in Figure 5.5, which is no longer a parallelepiped in the CIE XYZ space, and may even be a concave polyhedron. As such a matrix transformation will not work and a generalized mapping is needed.

Another issue in color matching is the gamma correction of projectors. Previous luminance balancing and color matching methods rely on the linearity of projectors. This means "gamma correcting" each projector to have a 1.0 gamma. Although this is mathematically simpler to deal with, it is not necessarily visually appealing, as the human visual system is more adapted to a gamma of 1.8 to 2.2. It also causes lost precision for brighter colors because of limited bits for the *Look-Up Table* (LUT).

In this chapter, we first present a scalable alignment system for tiled displays that is both more accurate and faster than existing approaches [16]. It is motivated by the

single projector keystone correction system described in [86], adapted to employ images taken from multiple, uncalibrated camera views. Our approach efficiently scales to projector arrays of arbitrary size without sacrificing alignment accuracy. We also present a practical solution for color matching tiled displays made of DLP projectors or mixed vendor/technology projectors. We use an inexpensive colorimeter to measure the color gamut of each individual projector. We then use a non-parametric model to find a color mapping for each projector to achieve a common gamut. We finally describe implementation methods for applying the color map along with geometric alignment and luminance balancing in real-time with graphics hardware.

The rest of this chapter is organized as follows. Section 5.2 details the camera homography tree algorithm for scalable alignment of tiled displays. Section 5.3 studies the color characteristics of DLP projectors, discusses the challenges in color matching them, and presents our non-parametric color matching system. Section 5.4 describes our evaluation methodology for both the alignment system and the color matching system. It also presents a solution for automatic measurement of alignment errors and an alignment simulator that enables scalability experiments on very large-scale tiled displays. Section 5.5 presents experimental results investigating the accuracy, scalability and running time of our algorithm and the color consistency results of our color matching system. Section 5.6 summarizes this chapter.

5.2 Scalable Alignment of Tiled Displays

We use the Princeton Scalable Display Wall described in Section 2.1.5 to study our scalable alignment system. In addition to the 24 projectors, there is a pan-tilt-zoom NTSC-resolution camera mounted on the ceiling in front of the projection screen. The video camera is used for capturing the images shown on the display.

To make our discussion general, we define the following notations. The term *Wall-*(H, V) denotes a tiled display with $H \times V$ projectors, arranged in a rectangular grid with H projectors horizontally and V projectors vertically. Each projector is assumed to have a resolution of 1024×768; thus a *Wall-*$(6, 4)$ has an approximate resolution of 6000×3000 . The term, *Cam-*$N \times N$, denotes a set of camera poses and zoom lens settings, where an $N \times N$ subset of the projector array is completely visible in each camera image. There are $n_v = \max(H - N + 1, 1) \times \max(V - N + 1, 1)$ distinct camera views available as input. For instance, a *Cam-*3×3 viewing a *Wall-*$(6, 4)$ observes a 3×3 set of projectors in each image; one could pan the camera to four horizontal and two vertical positions to obtain 8 different views of the display surface, each of which consists of a unique subset of projectors. Note that these views can be generated either from a single pan-tilt-zoom camera or from n_v fixed cameras. Our algorithm works with either scenario. We further define the term *Cam-all* to represent a single, wide-angle camera view that can see the entire display area at once, i.e. *Cam-all* = *Cam-*$M \times M$, where $M = \max(H, V)$. For example, the *Cam-all* for a *Wall-*$(6, 4)$ is *Cam-*6×6.

5.2.1 Perspective Correction with 2-D Homographies

We assume that: the positions, orientations and optical parameters of the cameras and projectors are unknown; camera and projector optics can be modeled by perspective transforms; the projection surface is flat. Thus, the various transforms between screen, cameras, and projectors can all be modeled as 2-D planar homographies:

$$\begin{pmatrix} xw \\ yw \\ w \end{pmatrix} = \begin{pmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & 1 \end{pmatrix} \begin{pmatrix} X \\ Y \\ 1 \end{pmatrix}$$

where (x, y) and (X, Y) are corresponding points in two frames of reference, and $\mathbf{h} = (h_1, \ldots, h_8)^T$ are the parameters specifying the homography. These parameters can be determined from as few as four point correspondences using standard methods. We employ the closed-form solution described in [86]. It is summarized below.

Given n feature point matches, $\{(x_i, y_i), (X_i, Y_i)\}$, $i = 1, \ldots, n$. Let

$$\mathbf{A} = \begin{pmatrix} X_1 & Y_1 & 1 & 0 & 0 & 0 & -X_1x_1 & -Y_1x_1 \\ 0 & 0 & 0 & X_1 & Y_1 & 1 & -X_1y_1 & -Y_1y_1 \\ X_2 & Y_2 & 1 & 0 & 0 & 0 & -X_2x_2 & -Y_2x_2 \\ 0 & 0 & 0 & X_2 & Y_2 & 1 & -X_2y_2 & -Y_2y_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ X_n & Y_n & 1 & 0 & 0 & 0 & -X_nx_n & -Y_nx_n \\ 0 & 0 & 0 & X_n & Y_n & 1 & -X_ny_n & -Y_ny_n \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \\ \vdots \\ x_n \\ y_n \end{pmatrix}.$$

The homography $\mathbf{h}$ is then the solution of the simultaneous equations $\mathbf{Ah} = \mathbf{b}$. When $n > 4$, the system is over-determined, we solve for the optimal $\mathbf{h}$ in the least square sense. Our system employs this technique to compute two types of homographies: camera-to-camera and projector-to-camera. Each is described in greater detail below, and illustrated in Figure 5.2.

First, camera-to-camera homographies capture the relationship between different camera views of the display surface. Although each view typically observes only a few projectors, the system combines these views to generate a reference frame for the entire display surface (see Section 5.2.3). Conceptually, this is equivalent to automatically building a panoramic mosaic from a set of photographs. One cannot directly compute a homography between two camera views that do not overlap since they share no point correspondences. Therefore, our system builds a tree of homogra-

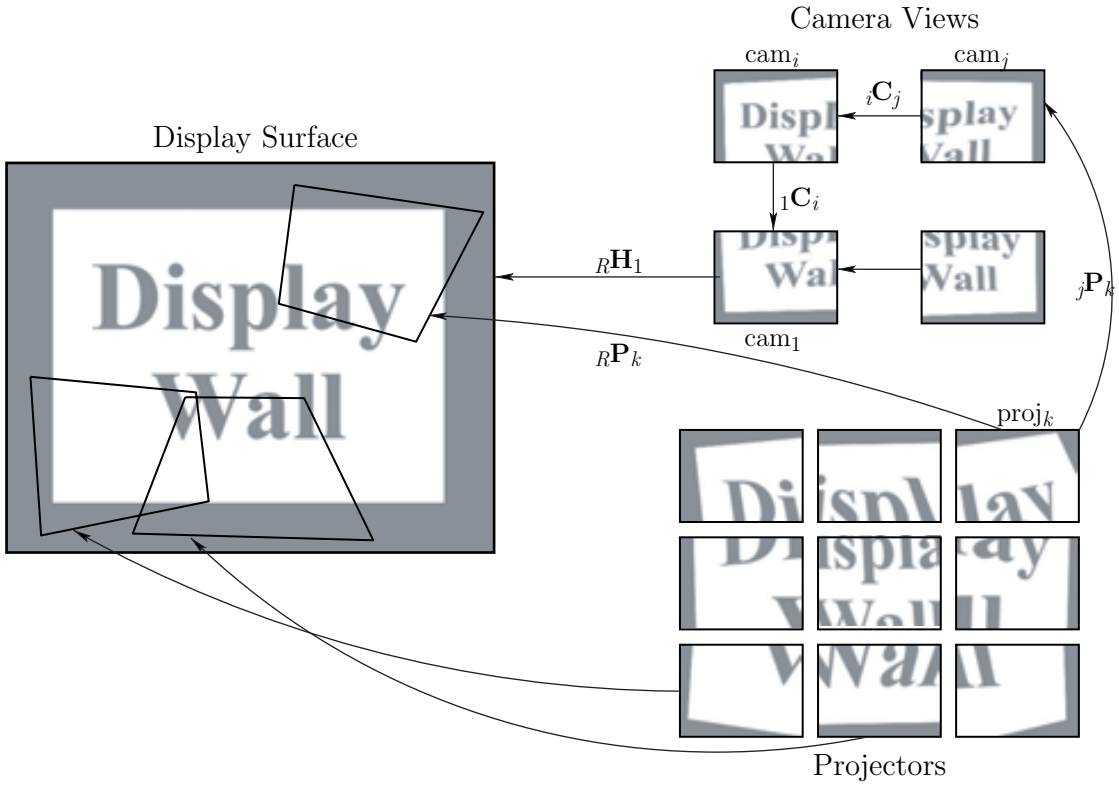


Figure 5.2: Homographies Linking the Screen, the Projectors, and the Camera Views.

phy relationships between adjacent views that spans the complete set of views; the mapping from any given view to the panoramic reference frame is determined by compounding the homographies along the path to the reference view at the root of the tree:

$$_R\mathbf{H}_j = {}_R\mathbf{H}_1 \times {}_1\mathbf{C}_i \times \cdots \times {}_i\mathbf{C}_j$$

where $_R\mathbf{H}_j$ is the homography mapping points from view j to the global reference frame, $_s\mathbf{C}_t$ are homographies connecting adjacent camera views and $_R\mathbf{H}_1$ maps the root camera view to the global reference frame.[3]

[3]The transform $_R\mathbf{H}_1$ ensures that the global frame axes are aligned with the display surface rather than the root camera view; $_R\mathbf{H}_1$ is computed by observing four known features on the display surface from any view.

Second, the projector-to-camera homographies transform each projector's area of projection into some camera's coordinate system. These homographies are determined as follows. Each projector k displays calibration slides with highly visible features, whose locations are known in projector coordinates. By observing the locations of these features in the camera image j, we can determine the relevant projector-to-camera homography ${}_j\mathbf{P}_k$. Since we know the mapping between any camera j and the reference frame, this enables us to compute ${}_R\mathbf{P}_k$, the transform from projector k to the reference frame:

$$ {}_R\mathbf{P}_k = {}_R\mathbf{H}_1 \times {}_1\mathbf{C}_i \times \cdots \times {}_i\mathbf{C}_j \times {}_j\mathbf{P}_k $$

Note that ${}_R\mathbf{P}_k$ expresses the geometric distortion induced by the projector's oblique placement. To remove this distortion, the output of each projector k's output is pre-warped by ${}_R\mathbf{P}_k^{-1}$. This process constitutes the mathematical basis for a vision-based software alignment system.

5.2.2 Sub-pixel Accurate Feature Detection

As described above, at least four point correspondences are needed to calculate a 2-D homography. We use the projectors in the tiled display to generate visible patterns. Image processing methods are used to extract feature points from these patterns.

Simple image processing techniques can typically locate features to the nearest pixel in an input image. However, since a single pixel in our camera images covers several projected pixels on the display surface, our application demands more sophisticated methods. Also, commodity video camera lenses usually exhibit noticeable distortions, making simple perspective camera models insufficient. We use the

following five-parameter distortion model from [41]:

$$x' = x + x[k_1 r^2 + k_2 r^4 + k_3 r^6] + [2p_1 xy + p_2(r^2 + 2x^2)]$$
$$y' = y + y[k_1 r^2 + k_2 r^4 + k_3 r^6] + [2p_2 xy + p_1(r^2 + 2y^2)]$$

where $r^2 = x^2 + y^2$, and (k_1, k_2, k_3) are the radial distortion coefficients, and (p_1, p_2) the tangential distortion coefficients. These distortion parameters can be obtained via standard offline calibration procedures. We have also developed a method to automatically correct the distortion along with feature extraction.

The feature detection component of our system displays a sequence of calibration slides on the projectors that are visible in each camera view. The goal is to reliably identify point features in adjacent camera views that correspond to the same location on the physical screen. The standard approach would be to project a single known pattern, such as a checkerboard, from each projector and use the checkerboard's corners as features. We improve upon this by projecting pairs of patterns: a set of horizontal lines followed by a set of vertical lines. The intersections between these line sets can be determined with greater accuracy than standard corner detection. The details are described below.

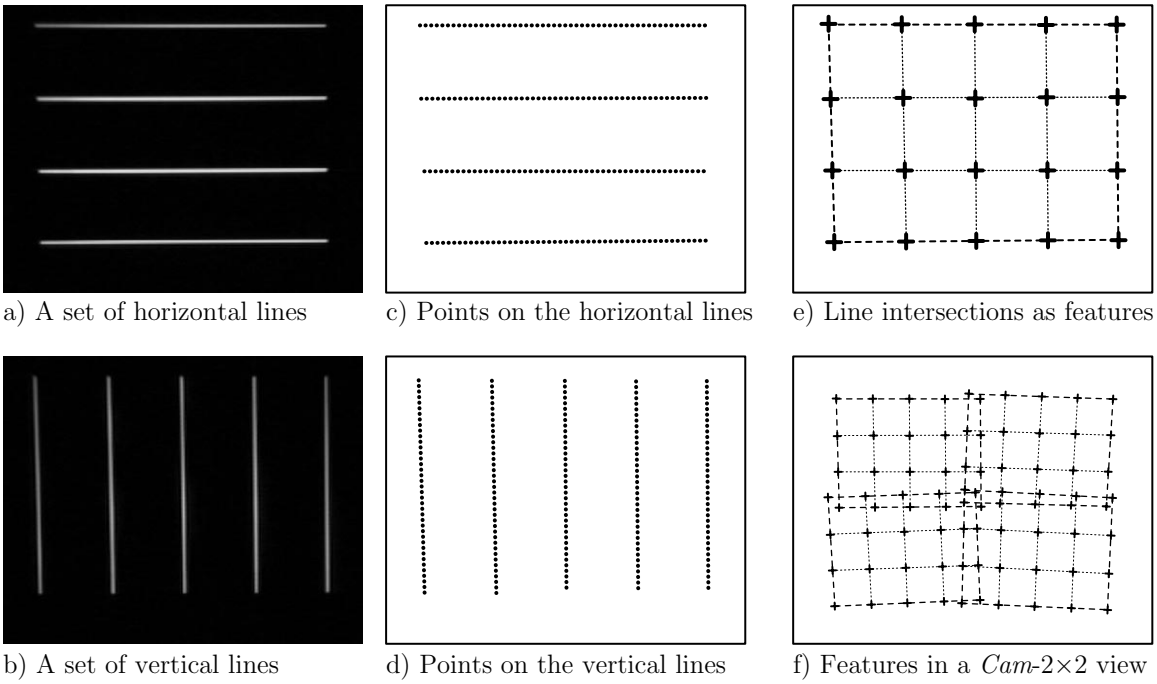
For each camera view, horizontal lines are first displayed on a projector. Figure 5.3a shows an example of the captured image. Next, vertical lines are displayed on the same projector, as shown in Figure 5.3b. Note that these lines are "horizontal" and "vertical" only in the projector's own coordinate system. They may not be horizontal or vertical on the display surface nor in the camera image since the projector and camera orientations may be oblique. Each image is then processed as follows. We fit a quadratic function to the intensity values inside every 9×1 and 1×9 window in the image. A strong peak of the function under a window indicates that a line

crosses through the window, and this provides a sub-pixel accurate estimate of the line's local position, shown as small black dots in Figure 5.3c and 5.3d. The output of this procedure is a set of position estimates with floating point precision along each visible line.

In the second step, the system determines the line equations that best fit the observed data in each camera image. Unfortunately, the observed lines are not precisely straight due to camera lens distortion. This is not easily corrected with offline calibration methods, because the camera distortion parameters change with zoom settings. This motivates the development of a novel online calibration method that combines distortion correction with line fitting. We use the sum of deviations of all points from the fitted line as the energy function. A non-linear optimizer is used to optimize the five-parameter distortion model to minimize the energy.

In the third step, the horizontal and vertical lines are intersected. This creates a set of accurate, stable point features for each camera view. A typical implementation employs calibration slides with five vertical and four horizontal lines, resulting in 20 point features per projector, as shown in Figure 5.3e. When the *Cam*-2×2 configuration is used, features from four projectors are visible in each camera view, as shown in Figure 5.3f.

These features are now used to compute the projector-to-camera and camera-to-camera homographies shown in Figure 5.2. Computing projector-to-camera homographies is straightforward since the locations of the 20 features are known *a priori* in the projector's coordinate frame. The camera-to-camera homographies are determined using all of the features that are visible in overlapping camera views. For instance, when the *Cam*-2×2 configuration is used, two common projectors are visible in adjacent camera views; this means that 40 feature points are available for the camera-to-camera homography calculation. When two camera views share no com-



a) A set of horizontal lines
c) Points on the horizontal lines
e) Line intersections as features
b) A set of vertical lines
d) Points on the vertical lines
f) Features in a *Cam*-2×2 view

Figure 5.3: Image Processing and Feature Extraction for *Cam*-2×2.

mon projector, the homography relating them must be obtained indirectly, using a chain of known homographies, as described in the following section.

5.2.3 Camera Homography Trees

Tightly-zoomed camera views give us better feature extraction precision. However, because each view only covers a portion of the entire tiled display, we need to merge these multiple views into one cohesive virtual view that covers all the projectors. To do so, we introduce the concept of a *Camera Homography Tree*, and an algorithm for optimizing it.

We call $G(V, E)$ a *Camera Homography Graph* (CHG), where each vertex in V represents a camera view and an edge in E corresponds to a directly-computable homography between two views, i.e. an edge connects two vertices only if they share

at least one common projector. For a rectangular tiled display, the CHG usually looks like a lattice. Figure 5.4 shows a *Wall*-$(6, 4)$ with *Cam*-2×2. In it, 15 views are available, forming a 5×3 lattice. A horizontal or vertical edge represents two strongly overlapping views, resulting in a better estimate of its homography; while a diagonal edge represents two weakly overlapping views, and thus a less reliable estimate.

As discussed in Section 5.2.1, as long as a CHG is connected, the homography between any two vertices can be computed by compounding a chain of homographies along a path connecting these two vertices. Ideally, the compound of homographies along any closed loop in an CHG should be identity; we call this graph a *Consistent CHG*. However, this is usually not true, due to estimation errors in the homographies resulting from imperfection in the optics and limited resolution of the imaging device. The result is an *Inconsistent CHG*. When multiple paths exist between two vertices in an inconsistent graph, the calculated homography between them may depend on the choice of path. Clearly this needs to be remedied if we want to accurately register all camera views. Similar problems exist for 2-D image mosaicing. Shum *et al.* [82] and Kang *et al.* [49] have proposed methods for global registration. Their algorithms work on continuous tone images, and have to extract features automatically. As mentioned before, we can detect features reliably with sub-pixel accuracy; this enables us to develop a novel algorithm that registers camera views precisely.

By definition, a tree is a connected graph without loops. Therefore, if a CHG is a tree, it is always consistent. Given a CHG $G(V, E)$, a *Camera Homography Tree* (CHT) is simply a spanning tree $T(G, E_T)$ of G, where $E_T \subseteq E$. In T, every pair of camera views is connected by a unique path. Although a CHT is consistent, it tends to be inaccurate when used directly—error in a single edge affects all homography paths containing that edge; also, being a subset of the original graph, only a portion of the feature correspondence information is utilized. We describe our method of

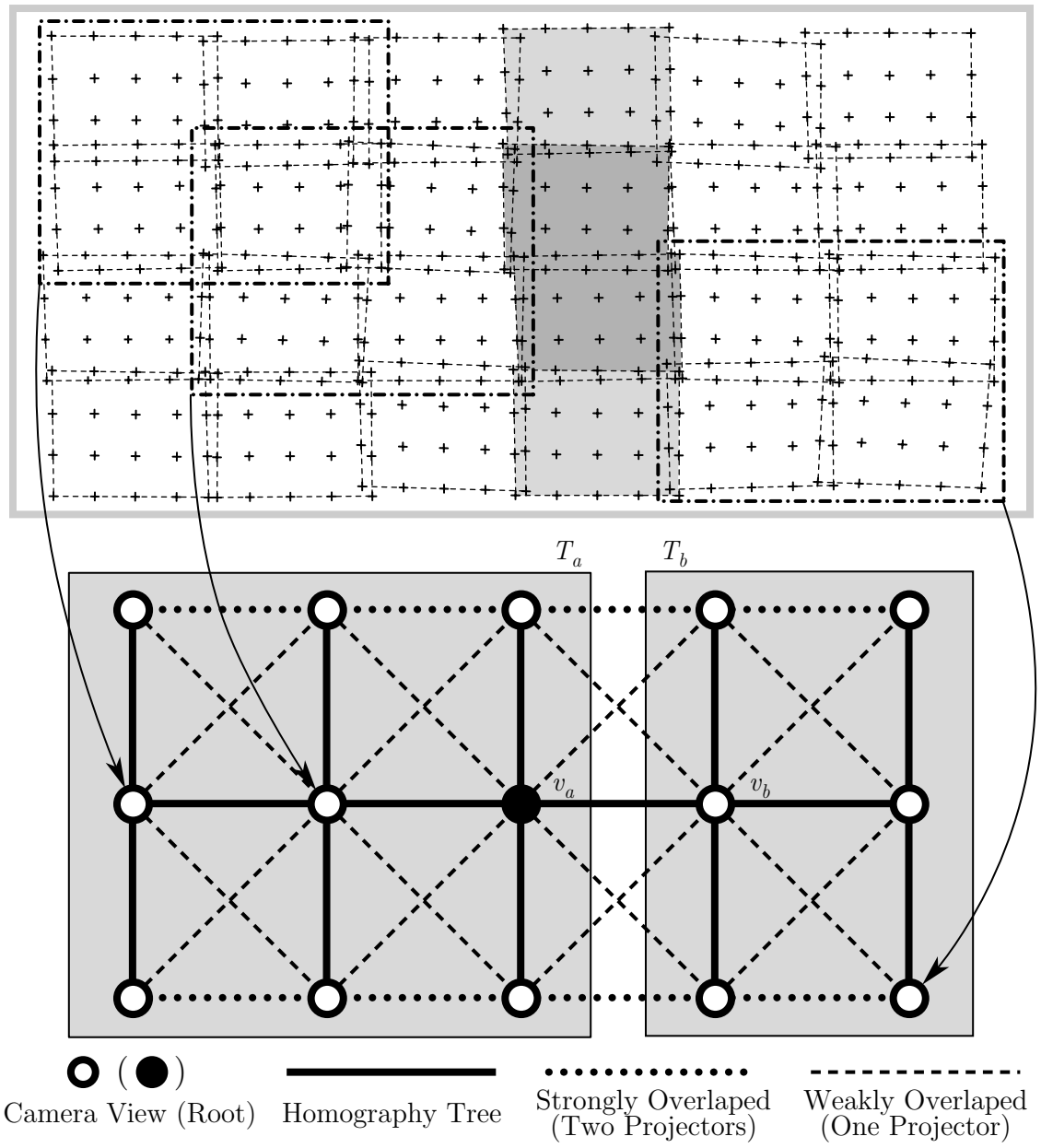


Figure 5.4: The Camera Homography Tree for *Wall*-$(6, 4)$ with *Cam*-2×2. We use a separate camera view to observe each 2×2 sub-array of the entire 6×4 tiled display, resulting in 15 views, which are shown as vertices in the bottom half of the figure. An edge in the graph represents a homography directly calculated from adjacent overlapping views. A spanning tree is constructed using the criteria described in Section 5.2.3. Each edge (homography) is then optimized iteratively. For example, the initial homography along the edge (v_a, v_b) is calculated from the features in the dark shade. It is refined with features in both dark and light shades.

constructing a initial CHT and optimizing the homographies along its edges to best represent the original CHG.

In order to reduce the error of compound homographies in the initial CHT, we minimize the path length from any vertex to the root, and the path length between any adjacent camera views. To satisfy these criteria, we pick a vertex near the center of a CHG as the root, or reference view. A fishbone-shape tree is then constructed, with its "spine" aligned with the long side of the lattice, as shown in Figure 5.4. Each edge is initialized with the homography directly computed from common features visible in both camera views, as described in Section 5.2.1.

In the optimization stage, we iteratively refine the edges of a CHT to better represent the original CHG. In each iteration, the edges are updated in a bottom-up order. Each edge $e = (v_a, v_b) \in E_T$ forms a cut set of T—when removed, T becomes two trees: $T_a(G_a, E_a)$ and $T_b(G_b, E_b)$, where $v_a \in G_a$, $v_b \in G_b$, $G_a = G - G_b$, $E_a = E_T \cap (G_a \times G_a)$, and $E_b = E_T \cap (G_b \times G_b)$. Figure 5.4 shows an example. The initial homography along the edge (v_a, v_b) is computed with features from the fourth projectors in the second and third rows, as shown in a darker shade. To refine this homography, we treat T_a and T_b as two CHT's and map features in each tree to the views of v_a and v_b. This gives us more common features between v_a and v_b, so we can compute a better homography for e. In this example, features from the entire fourth column of projectors, i.e. projectors with both dark and light shades in Figure 5.4, contribute to the refined homography. This process is continued until the variance of multiple samples of each point feature in the root view is below a threshold. We found that stable homography estimates are obtained after a small number of iterations.

This algorithm enables us to create an effective virtual camera with very high resolution from multiple uncalibrated low resolution views. It allow our system to achieve scalable sub-pixel alignment on very large tiled displays, as described in Section 5.5

5.3 Full Gamut Color Matching

In this section, we first define the general color matching process. We then discuss the challenges involved in dealing with commodity DLP projectors. We finally describe our full-gamut color matching system.

5.3.1 Generalized Color Matching Process

The color reproduction process of a display system can be described as follows. First, an RGB triple (r, g, b) in the graphics frame buffer is converted to either an analog voltage or digital bits and sent to the projector. The projector then combines lights of three primary colors proportionally to form the desired color. Given the spectrum of the output light, we can calculate the CIE[4] tristimulus values (X, Y, Z), which reflect the response of a typical human visual system [95]. The entire process can be characterized as a *Color Transfer Function* $F\colon R^3 \rightarrow R^3$, $(X, Y, Z) = F(r, g, b)$. F maps a value (color) from RGB space to the CIE XYZ space.

The gamut of a display device is the set of all reproducible colors. Commodity graphics hardware typically has an 8-bit depth in each of the RGB channels, $r, g, b \in [0, 1, \ldots, 255]$. Hence, the gamut of color transfer function F can be defined as

$$G(F) = \{F(r, g, b) | r, g, b \in [0, 255]\}.$$

We call a gamut an *additive gamut*, if it satisfies that

$$F(r, g, b) = F(r, 0, 0) + F(0, g, 0) + F(0, 0, b),$$

that is, the three RGB channels are independent of each other.

[4]Commission Internationale de l'Eclairage, or the International Commission on Illumination

Further, if the device gamma is set to 1.0, the transfer function becomes linear. It can then be expressed as a matrix transformation.

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = [f_{ij}] \begin{bmatrix} r \\ g \\ b \end{bmatrix} + \begin{bmatrix} r_0 \\ g_0 \\ b_0 \end{bmatrix}$$

where $[r_0, g_0, b_0]^T$ is the constant *black offset*. When a device does not have an additive gamut or the gamma is not 1.0, there is no such matrix $[f_{ij}]$ that satisfies the mapping.

For a tiled display to look seamless, all projectors in the system must reproduce colors in the same way, that is, they should all share a common color transfer function. However, this is usually not true in practice. Thus, there exists the need to color match the projectors.

Without access to the graphics card and projector hardware, color matching can be achieved through the use of a *color map* $M\colon [0, 1, \ldots, 255]^3 \rightarrow [0, 1, \ldots, 255]^3$, $(r', g', b') = M(r, g, b)$. The color map is applied to pixels before they are sent to the display. The equivalent color transfer function of the system can now be expressed as $F \circ M$. Given n projectors in a tiled display, each with a color transfer function of F_i, the color matching problem can be formally stated as: find M_i for $i = 1, \ldots, n$, such that $F_i \circ M_i = F_j \circ M_j$, $\forall i, j \in \{1, \ldots, n\}$.

5.3.2 Characteristics of DLP Projectors

LCD projectors usually have an additive gamut. Therefore, the color matching can be easily achieved through a 3×3 matrix multiplication in the RGB space, provided that the gamma is "corrected" to 1.0.

DLP projectors can be more difficult to color match than LCD projectors. Com-

modity single-chip DLP projectors use a spinning color wheel with primary color filters to create color channels in a time sharing fashion. They commonly use a method called "white enhancement" to increase the contrast ratio of the projector—in addition to the Red, Green, and Blue filters, a fourth White (or Clear) filter is added to the color wheel, which passes the full spectrum of the projector bulb. This is similar to the CMYK color printing process, where Cyan, Magenta, Yellow, and Black inks are used. The DLP projector chip controls how much white to add based on a function of the input RGB pixel value. As white is added, output RGB values are reduced correspondingly. Current DLP chips use a step function, adding white in 4 discrete increments [52].

One result of using white enhancement is that DLP projectors will exhibit different white points even after indepedent channel balancing is performed. The different spectral outputs of the bulbs are the result of either manufacturing tolerances or bulb decay over its lifetime.

The color gamut of a typical DLP projector is shown in Figure 5.5. As can be seen from the figure, the gamut does not form a parallelepiped in XYZ space and so is not an additive gamut. The gamut becomes stretched towards the white point due to the white enhancement. Even though it is possible to model a DLP projector gamut with a piece-wise linear model, the underlying parameters are device dependent and proprietary to the manufacturer. Therefore, we treat the color transfer function $(X, Y, Z) = F(r, g, b)$ of the projectors as a black box.

5.3.3 Measuring Color Transfer Function

With a non-parametric model, one has to measure the XYZ value for each of the 2^{24} possible input RGB value combinations. This is infeasible to implement. Instead,

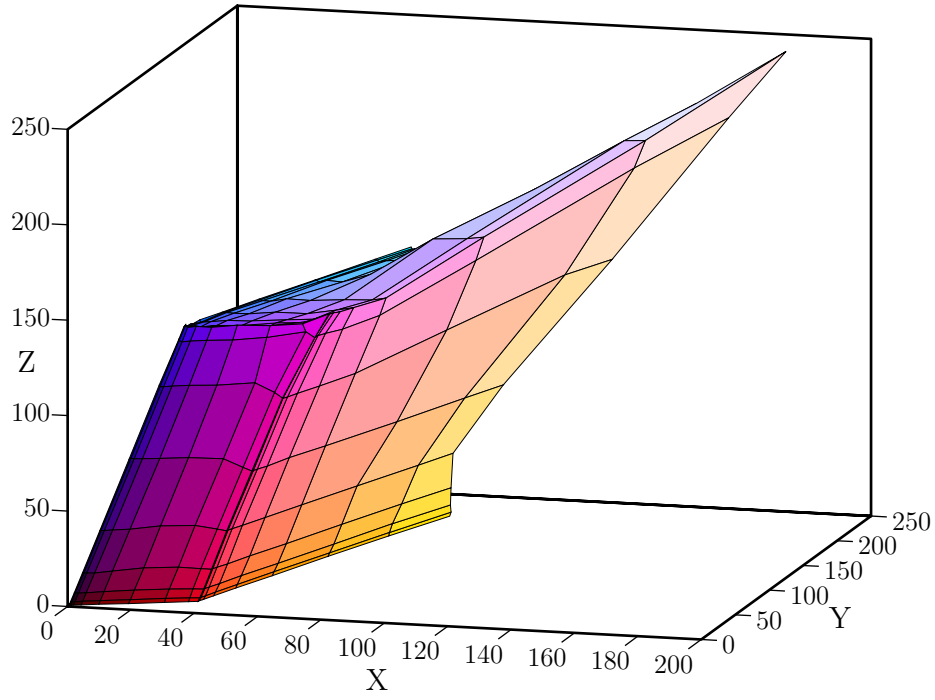


Figure 5.5: The Color Gamut of a Typical DLP Projector.

we sample F at a lower spatial frequency and use interpolation to fill in the missing values in between.

Because of the gamma curve, F changes slowly at the low end of RGB, but increases faster as RGB values grow. To accommodate this, we use a non-uniform sampling grid, with denser sampling intervals at the high end of input RGB values.

We use a colorimeter to measure the chromaticity value of an input RGB value. The colorimeter returns the x, y, z, and Y value of the input color, and we calculate the X, Y, Z value as follows

$$X = xY/y$$

$$Y = Y$$

$$Z = zY/y$$

5.3.4 Standard Color Transfer Function

Assuming monotonicity of a color transfer function F, which should be true when the projector's brightness and contrast settings are not saturated, its gamut $G(F)$ is the volume in XYZ space bounded by the following six surfaces

$$S_1 = \{F(0, g, b) | g, b \in [0, 255]\}$$
$$S_2 = \{F(255, g, b) | g, b \in [0, 255]\}$$
$$S_3 = \{F(r, 0, b) | r, b \in [0, 255]\}$$
$$S_4 = \{F(r, 255, b) | r, b \in [0, 255]\}$$
$$S_5 = \{F(r, g, 0) | r, g \in [0, 255]\}$$
$$S_6 = \{F(r, g, 255) | r, g \in [0, 255]\}$$

We use a triangle mesh generated from the sampled F data to form a polyhedral representation of G.

Let $G_i = G(F_i)$ be the gamut of the i-th projector. The common color gamut G_c that can be reproduced by all projectors is therefore the intersection of all G_i:

$$G_c = G_1 \cap G_2 \cap \cdots \cap G_n$$

By applying the polyhedron intersection algorithm [13, 9], we obtain a polyhedron representing G_c. Note that, because G_i can be concave, as in the case of DLP projectors, the intersection operation might produce a set of disjoint polyhedrons. In this case, we simply use the polyhedron with the largest volume as G_c, and discard the rest. This is dictated by the implied continuity requirement of F_c.

Once we have the common color gamut G_c, we can find a standard color transfer

function F_s. The goal is to maximize the volume of $G(F_s)$, with the constraint that $G(F_s) \subseteq G_c$.

To describe the algorithm of finding F_s, we first define a projective transform $H\colon R^3 \to R^3$.

$$x' = \frac{h_{11}x + h_{12}y + h_{13}z + h_{14}}{h_{41}x + h_{42}y + h_{43}z + 1}$$
$$y' = \frac{h_{21}x + h_{22}y + h_{23}z + h_{24}}{h_{41}x + h_{42}y + h_{43}z + 1}$$
$$z' = \frac{h_{31}x + h_{32}y + h_{33}z + h_{34}}{h_{41}x + h_{42}y + h_{43}z + 1}$$

We call two color transfer functions *projectively related*, if there exists a projective transform H, such that

$$F_1 = H \circ F_2$$

The algorithm is then described as below

1 Pick one of the color transfer functions, say F_1.

2 For each F_i, find an H_i such that the L2 distance of F_1 and $H_i \circ F_i$ is minimized.

3 Let $\bar{F} = \frac{1}{n}\sum_{i=1}^{n}(H_i \circ F_i)$

4 Maximize the volume of $G(H_s \circ \bar{F})$, with respect to H_s and with the constraint that $G(H_s \circ \bar{F}) \subseteq G_c$

5 The standard color transfer function is $F_s = H_{s,max} \circ \bar{F}$

To put the algorithm in plain English, we first obtain a starting color transfer function $\bar{F}$ by averaging F_i normalized to the shape of F_1, and then find the standard (common) color transfer function by warping $\bar{F}$, and maximizing its volume with the constraint that it has to be contained by the common color gamut G_c.

Starting from the average shape of all color gamuts allows us to preserve all the properties of the original color transfer function, such as its gamma.

5.3.5 Generating Color Maps

In order to emulate the standard color transfer function $\bar{F}$ on each of the projectors, a color map M is applied on the imageries before they are displayed, as discussed in Section 5.3.1. For a color map to be feasible, it has to satisfy the following condition

$$\forall (r,g,b) \in [0,255]^3, \quad M(r,g,b) \in [0,255]^3;$$

that is, the color map never produces out-of-gamut colors. The goal of the color map is such that $F_i \circ M_i = F_s$. Therefore,

$$M_i = F_i^{-1} \circ F_s.$$

Note that $\forall (r,g,b) \in [0,255]^3, \quad F_s(r,g,b) \in G(F_s) \subseteq G_c \subseteq G_i$. Thus, $M_i(r,g,b) \in F_i^{-1}(F_s(r,g,b)) \in [0,255]^3$. That is, M_i is indeed feasible with the definition of F_s.

Because we can only sample F_i at some discrete points, interpolation is needed when a value is not directly sampled in F_i. The final color map is a discretized version of M_i defined on $[0,1,\ldots,255]^3$ to itself.

5.3.6 Real-Time Imagery Correction

Applying the color map in CPU is a costly operation, which precludes the possibility of real-time software color matching. Access to new programmable graphics hardware, e.g., the Pixel Shader in DirectX or Texture Shader in OpenGL, has enabled us to apply the color mapping in the graphics card. To achieve this, we load the discretized

color map M as a volume texture. For each pixel, we treat its (r, g, b) color value as a volume texture coordinate (u, v, w), and sample M to find out the mapped color. This operation can be implemented with the `texreg2rgb` instruction available in the Microsoft DirectX Pixel Shader Language version 1.2 and 1.3.

As mentioned in Section 5.1, color matching is one aspect of the overall projector calibration process. To combine the geometric alignment, luminance balancing and color matching together, we propose the following rendering architecture.

1 3D scenes are rendered to a texture. In the case of 2D applications, images or video frames are loaded into a texture buffer. This is the first texture stage.

2 The discretized color map is loaded into a volume texture, and used as the second texture stage.

3 The luminance map is loaded into texture as the third texture stage.

4 Set up the first texture combiner to copy the first texture in decal mode.

5 Set up the second texture combiner to sample the volume texture using the output of the first stage as texture coordinates.

6 Set up the third texture combiner to multiply the output of the second stage with the third texture.

7 Set up the view and projection matrices to represent the geometric pre-warping.

8 Draw a rectangle.

The corresponding pixel shader is shown in Figure 5.6.

Our tests, presented in Section 5.5.5, indicate that the latest graphics cards, such as the NVIDIA GeForce4 Ti4600, can support such operations for full frame images in

```
ps.1.2
// t0 is the rendered texture
tex t0
// t1 is the color map
texreg2rgb t1, t0
// t2 is the luminance map
tex t2
mov r1, t1
mul r0, t2, r1
```

Figure 5.6: PixelShader Code for Real-Time Imagery Correction.

real-time. Future versions of the Pixel Shader language will allow more instructions, flow control and floating point precision color. With these additions, we expect that higher quality calibration can be achieved.

5.4 Evaluation Methodology

In this section, we describe our methodology for evaluating the geometric alignment system and the color matching system. We first propose metrics for evaluating the performance of a geometric alignment system. We then introduce an automated vision-based system for measuring them. To overcome the physical size limitation of our experimental tiled display, we also design a simulator for evaluating the scalability of the alignment system on arbitrarily large tiled displays. Finally, we describe the metrics for evaluating the color matching system.

5.4.1 Metrics for Tiled Display Alignment Systems

We use three metrics for evaluating the performance of a geometric alignment system for tiled displays: local alignment error, global alignment error, and running time.

Local alignment error quantifies the registration between adjacent projectors.

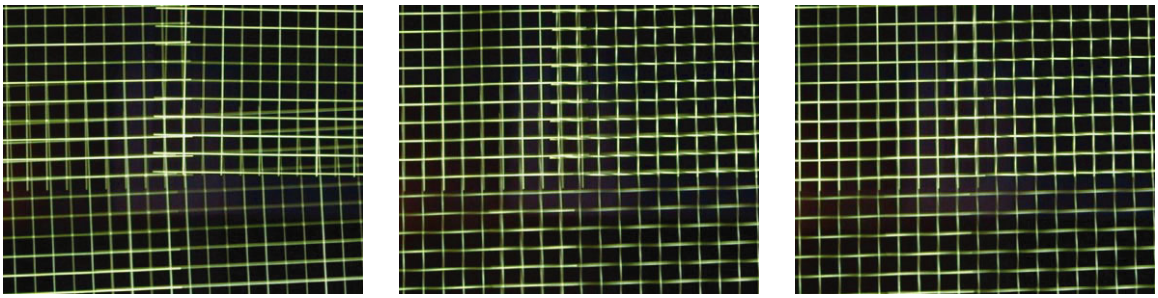
Qualitatively, misalignment creates artifacts such as discontinuities and double images in the overlap regions (see Figure 5.7a). Quantitatively, the error can be characterized by the displacement between a point shown on one projector and the same point displayed by an adjacent projector. An appropriate measurement unit for this error is the average size of a pixel projected on the display. This unit is invariant to the physical dimensions of the tiled display.

Let $\mathbf{H}_k = {}_R\mathbf{P}_k^{-1}$ be the homography that maps point $\mathbf{p} = (x, y, 1)^T$ from the display surface into projector k's reference frame. Let $\hat{\mathbf{H}}_k^{-1}$ be the alignment system's estimate for the inverse mapping. Due to alignment errors, $\hat{\mathbf{H}}_k^{-1}\mathbf{H}_k \neq \mathbf{I}$. In other words, when projector k attempts to illuminate point $\mathbf{p}$, it actually illuminates the point $\mathbf{p}_k = \hat{\mathbf{H}}_k^{-1}\mathbf{H}_k\mathbf{p}$. Let Ω be the set of all features, and Φ be the set of all projectors. We define local error to be:

$$E_l = \sum_{\forall \mathbf{p} \in \Omega} \sum_{\forall (i,j) \in \Phi \times \Phi} I(i, \mathbf{p}) \cdot I(j, \mathbf{p}) \cdot ||\mathbf{p}_i - \mathbf{p}_j||^2$$

where $I(i, \mathbf{p}) = 1$ if $\mathbf{p}$ falls within the display frustum of projector i, and $I(i, \mathbf{p}) = 0$ otherwise. This formulation of the local error does not require knowledge of absolute points on the display surface, $\mathbf{p}$. It is sufficient to examine pairs of $\mathbf{p}_i$ and and $\mathbf{p}_j$ measure the relative distance between them. In the experiments described below, we obtain local error by displaying a grid pattern and measuring the projected discrepancy between grid points which are displayed by projectors in overlap regions.

Some alignment algorithms, such as SimAnneal explicitly observe point- and line-mismatches in the overlap regions and attempt to optimize pre-warp parameters to minimize this error. Other algorithms, including ours, simply aim to register each projector to the global reference frame as accurately as possible, trusting that an accurate global registration will lead to small local errors.



a) The uncalibrated setup

b) After a single-view calibration (average local error: 1.95 pixels)

c) After *Cam*-2×2 calibration (average local error: 0.55 pixels)

Figure 5.7: Zoomed views of alignment errors on a *Wall*-$(6, 4)$. This figure shows zoomed-in views of the tiled display. The grid lines are 20 pixels apart; thus, each picture represents a screen area of about 400×300 pixels.

Global alignment error is a metric for measuring the overall registration of a tiled display. A projector array with excellent local alignment may still exhibit alignment errors for two reasons: (1) the projected image may be globally warped so that its edges are not parallel to the sides of the display surface; (2) small errors in local alignment can accumulate as homographies are chained, resulting in nonlinear distortions in the projected image. We define global alignment error to be the displacement between pixels in the projected image and their desired locations, as measured in the reference frame:

$$E_g = \sum_{\forall \mathbf{p} \in \Omega} \sum_{\forall k \in \Phi} I(k, \mathbf{p}) \cdot ||\mathbf{p} - \mathbf{p}_k||^2$$

This global error metric requires knowledge of the absolute locations of points on the display surface, thus making accurate measurements of global alignment quite difficult. We approximate the global error by measuring the nonlinearity of a regularly spaced grid pattern. Fortunately, the human visual system is tolerant of slight global misalignments while being very sensitive to local discontinuities. In practice, once the projected display is roughly aligned to the global coordinate frame, local errors dominate the user's perception of the display.

The third metric is running time. A faster alignment system is more practical, especially for setting up tiled displays in temporary venues. Fast alignment system also allows for rapid reconfiguration of the tiled display. Although we are not there yet, ultimately real-time alignment system will make tiled displays in industrial or combat environments possible. There are two components of running time: the time taken to acquire images, and the time needed for computation, including image processing and calculating homographies. We present timing results comparing our system to existing approaches.

5.4.2 Automatic Measurement of Alignment Errors

Manual measurement of local and global alignment errors is a tedious, time-consuming process. Sometimes it also results in subjectivity and non-repeatability in the results. Fortunately, we can employ the same camera hardware used for calibrating the tiled display in evaluating its local alignment accuracy.

To measure local alignment error, each projector displays a set of calibration patterns. Unlike the patterns used during the calibration phase, these patterns are aligned (to the best of the system's ability) to the global reference frame. The camera captures detailed images of the seam regions where projection areas overlap. It records the displacement between a point displayed by one projector and the same global point displayed by other projectors at the seam. The average displacement over all seam regions on the display surface gives an estimate of local alignment error (in pixels).

In principle, one must be cautious about using a camera to measure alignment accuracy, because the measurement uncertainty of the camera could very well overwhelm the alignment accuracy that we are trying to measure, especially when we are interested in achieving sub-pixel accurate alignment. For this reason, we simulated

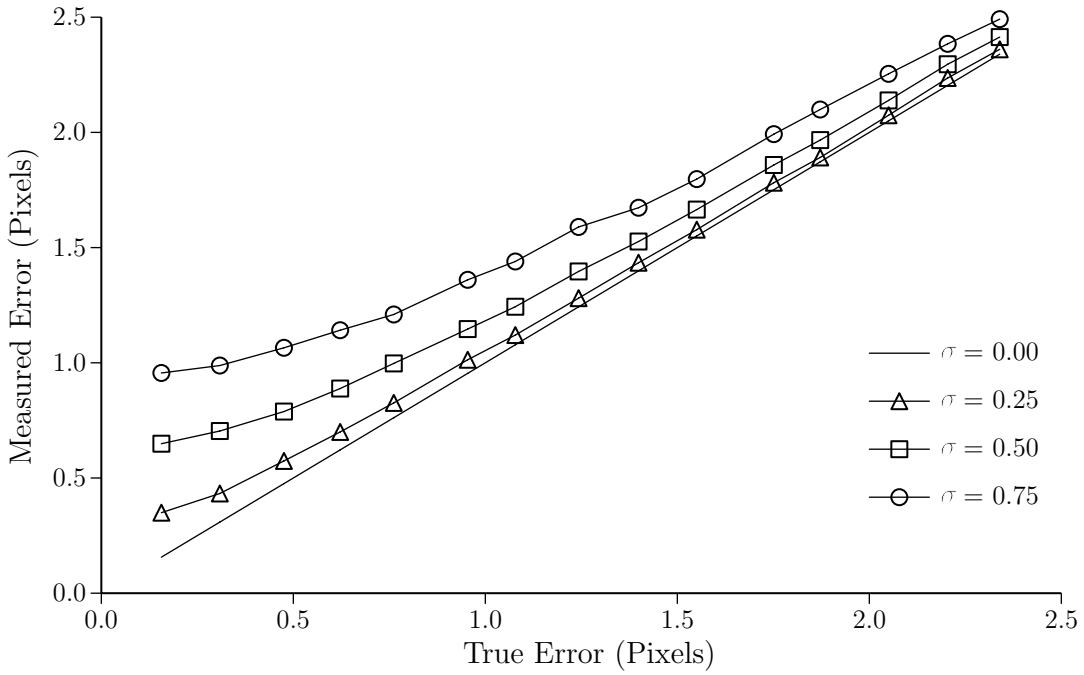


Figure 5.8: The Error Estimates of Automatic Measurement System.

the automatic measurement system in the tiled display alignment simulator (see Section 5.4.3). In this series of tests, we assumed that the noise in feature detection could be modeled using a zero-mean Gaussian distribution, $N(0, \sigma)$. Figure 5.8 plots the actual local error (ground truth available to the simulator) against the measured local error, for a range of noise models. A noise-free measurement process would obviously generate a straight line $y = x$. We note that all of the curves lie above the $y = x$ line; this means that the automated measurement system consistently overestimates errors. Our experimental data indicates that the system has $\sigma = 0.5$ pixels in both x- and y-direction. Since we use four points for each of the 38 seams on *Wall*-$(6, 4)$, the standard deviation on average local error estimates is $\tilde{\sigma} = 0.5/\sqrt{4 \times 38}$ and the 97% confidence interval is $\pm 3\tilde{\sigma} = \pm 0.12$ pixel. This indicates that we can use this automatic system to consistently evaluate a sub-pixel accurate geometry alignment result, which is indeed what our alignment system is able to produce.

5.4.3 Tiled Display Alignment Simulator

To supplement our experiments on the 24-projector tiled display we implement a tiled display alignment simulator in Matlab. The primary function of the simulator is to investigate whether our camera homography tree algorithm scales well to very large-scale tiled displays, both in terms of alignment accuracy and running time. Additionally, the simulator may enable us to determine the components of a tiled display system that are most likely to impact alignment accuracy. We broadly classify sources of alignment errors into four categories, each parameterized with one coefficient.

Projector optics: The alignment error increases when the projectors' optics cannot be accurately described using a perspective model. We simulate the projector optics with the distortion model described in Section 5.2.2. A *Projector Distortion Factor*, p, is coupled to both radial and tangential distortions:

$$(k_1, k_2, k_3, p_1, p_2) = p \times (1.0, 1.0, 0.2, 0.02, 0.005)$$

The projectors in our system exhibit little distortion; we estimate $p = 0.02$, which corresponds to an average warp of 0.32 pixels at the projected image's edge. In simulations we use p from 0.01 to 0.04.

Camera optics: Although our cameras exhibit significant distortion, especially in the widest-zoom setting, the effective distortion is greatly reduced after either offline camera calibration or our automatic distortion correction technique. However, these methods can not fully rectify the images, we model the residual distortion with a *Camera Distortion Factor*, c:

$$(k_1, k_2, k_3, p_1, p_2) = c \times (1.0, 1.0, 0.2, 0.02, 0.005)$$

We estimate $c = 0.05$ for our camera, which corresponds to an average warp of 0.23 pixels along the edges. In the simulations, we use c from 0.00 to 0.05.

Image processing: The simulator uses an abstract model for image processing. We assume that the system locates line features in the camera image, and that the position estimate for these features is corrupted with a zero-mean Gaussian noise. We define *Image Noise Factor*, n to be from 0.0 to 1.5, where $n = 1.0$ corresponds to ± 0.5 pixel error, or $N(0, 0.5)$.

Non-planar display surface: Our alignment system assumes that all transforms can be modeled using 2-D planar homographies—an assumption that relies on a planar display surface. The simulator models the shape of the display screen as a Gaussian surface parameterized on a single variable, the *Screen Curvature Factor*, s, i.e. we define the screen as a surface:

$$f(x, y) = s \times 200 \times \Psi(x, W_s) \times \Psi(y, H_s)$$
$$\Psi(x, a) = (e^{-(4x/a)^2} - e^{-4})/(1 - e^{-4})$$

where W_s and H_s are the width and height of the screen, and all units are in mm. In the experiments, we use s from 0.0 to 0.2, where $s = 0.1$ corresponds to a 20 mm central bulge, which equals to the real measurement in our $18' \times 8'$ rear-projection screen.

We present some of the simulation results in Section 5.5.3.

5.4.4 Evaluation Procedure for Color Matching System

To evaluate the performance of a color matching algorithm for tiled displays, we perform the following steps:

Data Collection: In this first step, we measure the color gamut of the projectors. We use an commodity colorimeter, the Sequel Imaging Chroma IV, to determine the chromaticity and luminance of a solid color in the CIE XYZ space. We subsample the projector RGB space with a 32 increment for values less than 128, and a 16 increment for values greater than 128. This gives us 13 points, that is, 0, 32, 64, 96, 128, 144, 160, 176, 192, 208, 224, 240, 255. This results in 13^3 (2197) samples in total. The result of such a sampling is visualized in Figure 5.5.

Map Generation: These color samples gathered in the collection phase are then fed into the color matching algorithm. We use linear interpolation on the subsampled data when performing the color matching. The resulting data structure of a color matching algorithm varies. For example, our full gamut matching algorithm produces a color map M for each projector, while the independent channel balancing algorithm generates three independent LUTs for each projector.

Consistency Measure: After the necessary color correction tools are generated in the previous step, we apply them to a number of solid colors on all projectors, and re-measure their chromaticity and luminance using a colorimeter. The color map M for the full gamut matching algorithm is applied as detailed in Section 5.3.6 and the LUT's for the independent channel balancing algorithm are loaded into the graphics cards. We subsample the entire gamut with a 32 increment for each of the RGB values, resulting in 9^3 (729) samples in total.

Following these steps, we can determine the performance of a color matching algorithm by calculating the color consistency of all projectors, both prior to and after color correction.

5.4.5 Metrics of Color Consistency

We use the sampled data described before to generate color consistency metrics. Our primary metric is the average deviation of the XYZ values of a test color from its average. Consider a color matching done over n projectors, and m test colors $c_i = (r_i, g_i, b_i)$, $i = 1, \ldots, m$. Let S_j be the set of measured XYZ values for all test colors on projector j, where $j = 1, \ldots, n$.

$$S_j = \{s_{ij} = (X_{ij}, Y_{ij}, Z_{ij}) = (F_j \circ M_j)(r_i, g_i, b_i) | i = 1, \ldots, m\}.$$

First we define the average response of a test color as

$$\bar{s_i} = \frac{1}{n} \sum_{j=1}^{n} s_{ij}.$$

The deviation from the average is

$$E_i = \frac{1}{n} \sum_{j=1}^{n} |s_{ij} - \bar{s_i}|.$$

We then normalize this value to obtain a percentage deviation:

$$e_i = E_i / |\bar{s_i}|.$$

Finally we derive the average of the deviations as a unified metric.

$$\bar{E} = \frac{1}{m} \sum_{i=1}^{m} E_i$$

$$\bar{e} = \frac{1}{m} \sum_{i=1}^{m} e_i$$

In Section 5.5.5, we refer to E_i and e_i as the absolute error and percentage error for a single test color c_i, respectively. We use $\bar{E}$ and $\bar{e}$ as overall metrics for the entire color gamut.

5.5 Experimental Results

This section presents results from several series of experiments for both the geometric alignment system and the color matching system. Section 5.5.1 compares our scalable alignment system to two existing tiled display alignment algorithms. Section 5.5.2 examines how local alignment error improves as the number of camera views is increased. Section 5.5.3 confirms that the camera homography tree algorithm remains accurate as the number of projectors in a tiled display increases. These experiments on the 24-projector display are further supported by simulation runs on very large-scale tiled displays. Section 5.5.4 compares our algorithm's running time with existing approaches. Section 5.5.5 presents the results of our color matching system compared with existing Independent Channel Balancing algorithms.

5.5.1 Comparisons with Existing Alignment Techniques

Two recent systems, Chen *et al.*'s SimAnneal [19] and Yang *et al.*'s PixelFlex [96] were selected as suitable benchmarks for tiled display alignment. We were able to obtain an implementation of the former for evaluation purposes, and were able to re-implement relevant portions of the latter algorithm from published details. The PixelFlex algorithm utilizes only a single camera view covering the entire $18' \times 8'$ rear-projection screen in our setup.[5] The SimAnneal algorithm requires detailed views of inter-projector seams, and these images were collected automatically using a pan-tilt-

[5]This camera configuration was identical to the *Cam-all* configuration in our algorithm.

zoom camera. The same camera control software (with different parameters) was used in our system to collect the *Cam-*$N \times N$ views as input to the camera homography tree algorithm. Before presenting results, we briefly describe the image processing aspects for each algorithm.

SimAnneal uses a sequence of point and line feature correspondences wherever two or more projectors share a seam. The displacement between corresponding features displayed by different projectors provides an error metric that is minimized using simulated annealing. The implementation of SimAnneal we obtained assumes for its initial conditions that the homography between adjacent projectors can be adequately approximated by a simple translation and scale. This assumption is valid only when projectors are initially well-aligned; our uncalibrated setup, as seen in Figure 5.7a, exhibits significant error from rotation and perspective effects. As a result, SimAnneal performs poorly in our experiments, rarely achieving less than 12 pixels of local error after 500K iterations. For this reason, we report results from [19], where SimAnneal was evaluated on a much smaller *Wall-*$(4, 2)$ display. We expect that with proper initialization, this value would represent a closer, but still optimistic estimate of SimAnneal's potential accuracy on a *Wall-*$(6, 4)$.

In the PixelFlex system, each projector displays an array of circles with a Gaussian intensity distribution, and the centroids of the observed Gaussians in the camera image provide a sub-pixel accurate estimate of the feature locations [96]. Our implementation of PixelFlex required straightforward modifications to the *Cam-all* version of our system; the main difference is the use of Gaussian rather than line features.

Table 5.1 summarizes the results of our experiments. In the single-view (*Cam-all*) configuration with no homography trees, our system's accuracy is comparable to the existing systems. We notice that *Cam-all* performs slightly better than PixelFlex. The reason is that, under perspective projection, lines are invariant but Gaussians

Table 5.1: Alignment Results of Various Algorithms on *Wall*-(6, 4).

System	Number of Camera Views	Local Error Avg (Max)	Global Error Avg
SimAnneal	152	1.35 (N/A)	N/A
PixelFlex	1	1.73 (3.9)	1.5
Cam-all	1	1.19 (4.1)	1.3
Cam-2×2	15	0.55 (2.3)	1.8

undergo asymmetric distortion—possibly inducing a bias in PixelFlex's estimate of feature location. When our system is able to take advantage of multiple camera views, e.g. in the *Cam*-2×2 configuration, the result improves dramatically—the average local error is reduced by half, the effect of which can be seen in Figure 5.7c.

5.5.2 Improving Alignment Accuracy with Multiple Views

This experiment investigates the trade-offs between effective camera resolution and homography chain length. Higher effective camera resolution, achieved using a tighter zoom, enables the system to locate features in the calibration slides with greater accuracy. On the other hand, the smaller field of view implies that more camera shots are needed to span the complete display. Figure 5.9 demonstrates that the camera homography tree algorithm does improve local accuracy. The single view approach, *Cam-all*, exhibits slightly more than 1 pixel error on *Wall*-(6, 4). Additional camera views improve accuracy, enabling us to achieve sub-pixel error rates on a 6000×3000 display using only a 640×480 resolution camera. The best result is achieved by *Cam*-2×2, which exhibits less than half the error of the best single camera view algorithm.

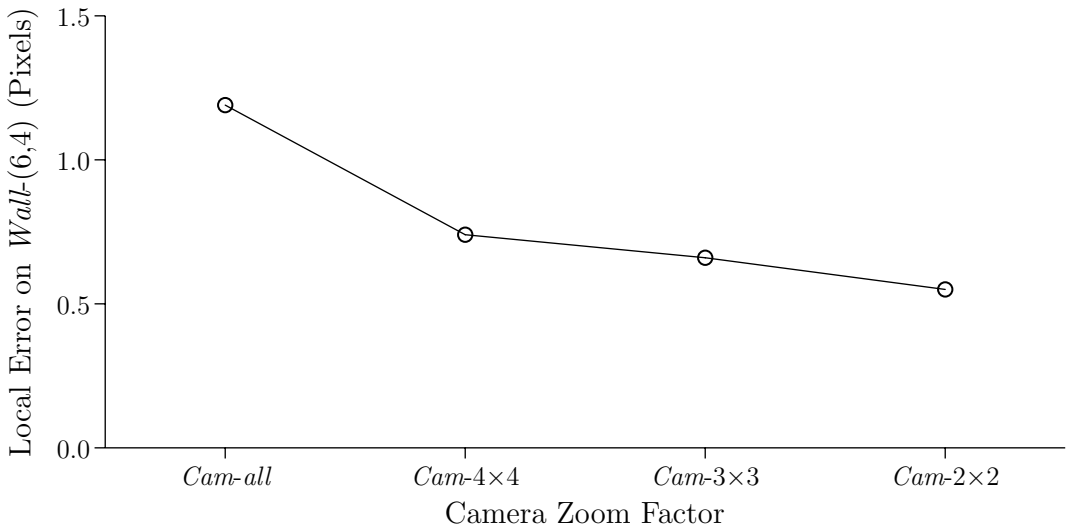


Figure 5.9: Multiple Views Improve Local Alignment Accuracy.

5.5.3 Scalability of the Alignment System

The previous experiment shows that the camera homography tree algorithm significantly improves accuracy on our current tiled display hardware. The following two experiments investigate whether it continues to outperform single-view alignment techniques across different scales of displays. The first shows that the observed behavior is also true for smaller displays and the second indicates that our algorithm scales to very large-scale tiled displays consisting of hundreds of projectors.

Scalability Results on 24-projector Display

Figure 5.10 investigates the camera homography tree algorithm's behavior on tiled displays of different sizes. These experiments were performed on the 24-projector display, using several rectangular sub-arrays of projectors ranging from *Wall*-$(2, 2)$ to the complete display. The results confirm that the multiple view approach to tiled display alignment scales well with display size.

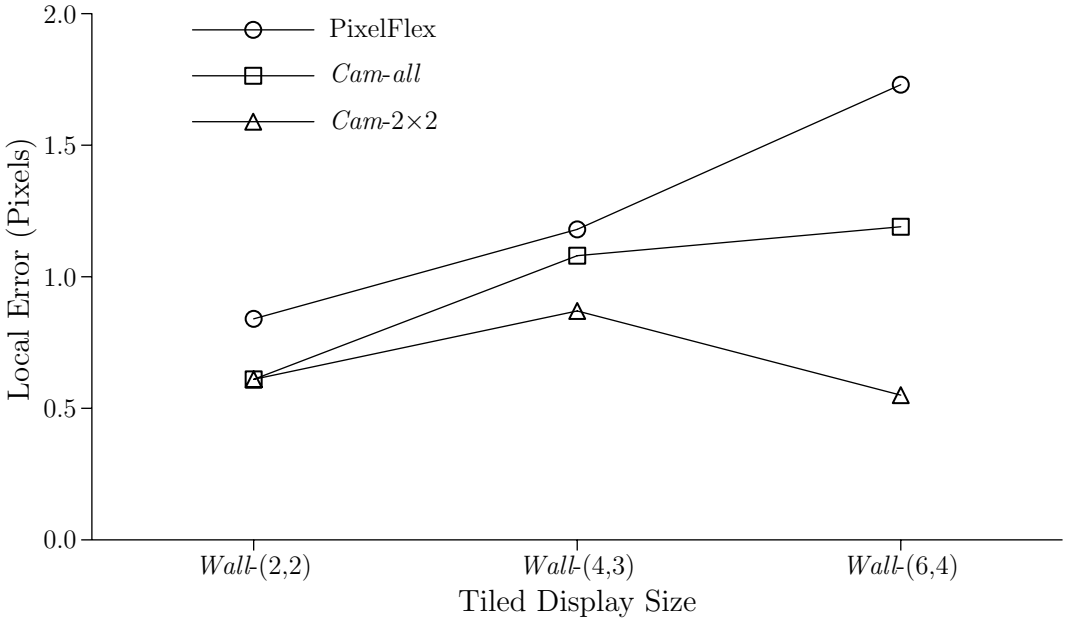


Figure 5.10: Scalability of Alignment Systems from Measured Data.

We note that local error for *Cam*-2×2 is higher than expected in the *Wall*-$(4, 3)$ scenario. We believe that this may be due to screen curvature; initial simulation results support our hypothesis. Additional experiments are being conducted.

Scalability Results on Tiled Display Alignment Simulator

To evaluate our algorithm's performance on very large displays, we run the simulator on the following tiled displays: *Wall*-(H, V), where

$$(H, V) \in \{(2, 2), (3, 2), (4, 3), (6, 4), (9, 6), (12, 8), (18, 12), (24, 16)\}.$$

Figure 5.11 shows the average local error versus the total number of projectors in a display. Each curve in the graph corresponds to a choice of *Cam*-$N \times N$, where $N \in \{2, 3, 4, 6, 9, 12, 18, 24\}$. There is no benefit to using a wider field of view camera than necessary on a given display. Therefore, each *Cam*-$N \times N$ curve starts at the

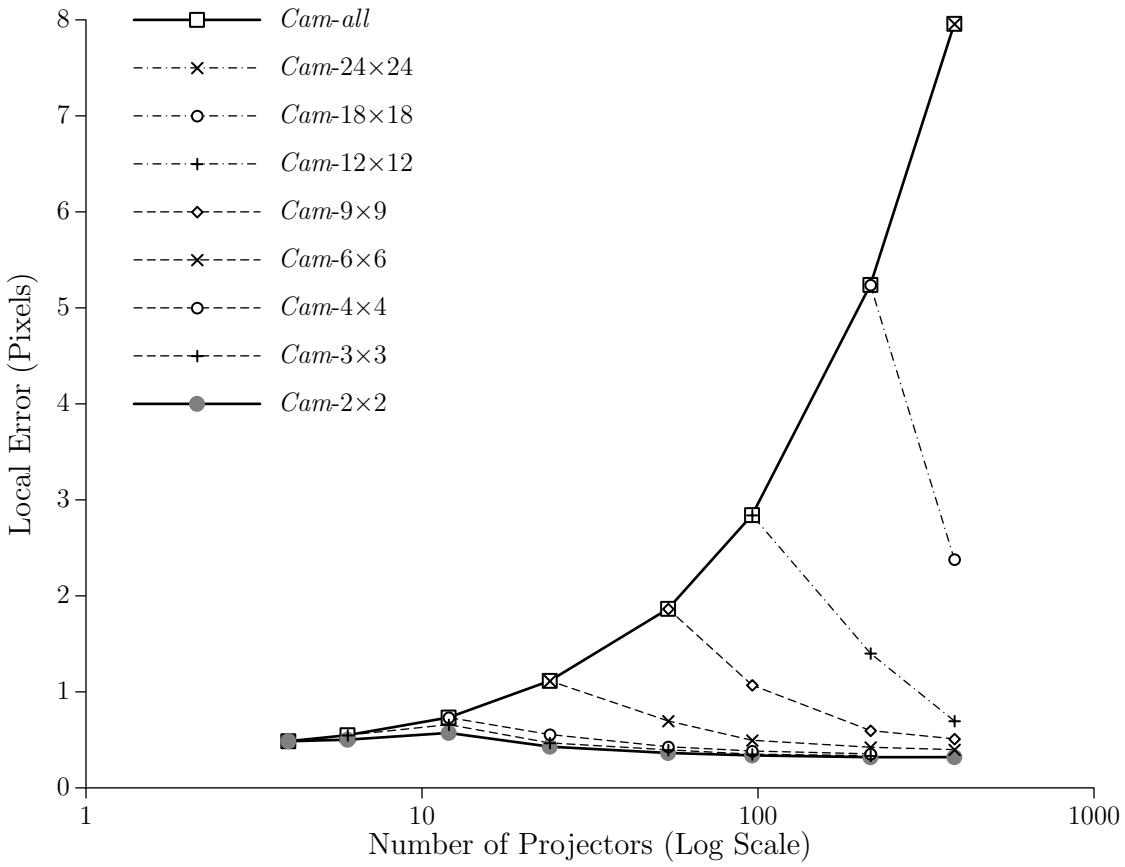


Figure 5.11: Scalability of Alignment Systems from Simulation.

largest tiled display that can be completely seen within a single view. By definition, the curve *Cam-all* simply connects the starting data point of each *Cam-*$N \times N$ curve. Simulation parameters were selected to be similar to the current physical setup: $p = 0.02$, $s = 0.1$, $c = 0.05$, $n = 1.0$, and each data point is generated by averaging the results of five runs.

As Figure 5.11 shows, the simulation data on smaller tiled displays is consistent with the experimental evidence presented before. This graph also shows simulations extended to very large scale displays. We make the following observations:

1. The alignment error for a single-view algorithm (*Cam-all*) grows almost linearly

as projectors are added. This indicates that such algorithms do not scale well.

2. The curves for the tightly-zoomed camera configurations (e.g. *Cam*-2×2) are almost flat. This validates our earlier claim that the camera homography tree algorithm scales well with display size.

3. Note that for a particular *Cam*-$N \times N$ curve, the local error decreases as the number of projectors increases. This seemingly counter-intuitive phenomenon can be explained. On a larger display, each projector appears in more camera views, according to the definition of *Cam*-$N \times N$. Our homography tree algorithm is able to utilize multiple views of point features to better refine the homographies. This in turn results in improved alignment accuracy.

4. One can derive significant benefits from the camera homography tree algorithm even with a small number of views. For instance, on *Wall*-$(24, 16)$, going from *Cam*-24×24 (i.e. *Cam-all*) to *Cam*-18×18 cuts the local alignment error from almost 8 pixels to about 2.5 pixels. In most cases, it does not require the tightest zoom setting of *Cam*-2×2 to achieve a sub-pixel accurate alignment result.

5.5.4 Running Time of Alignment System

There are two major components of the running time: the time required to collect the necessary images; and the time needed to process these images and calculate the homographies used to align the projectors. Table 5.2 presents the timing results. The "Data" column lists the time taken to collect images, and the "Comp" column is the computation time needed to calculate homographies from the data. Our system is implemented in Matlab 6.0 with a moderate amount of optimization. Since our implementation of the PixelFlex algorithm uses the same code base, its running time is

Table 5.2: Running Time of SimAnneal and Our System.

Display Size	SimAnneal			Our System		
	Setup (# steps)	Data (min)	Comp (min)	Setup	Data (min)	Comp (min)
Wall-(2, 2)	10k	10.0	15.2	*Cam-all*	0.13	0.17
Wall-(4, 3)	20k	33.0	34.5	*Cam-all*	0.53	0.25
				Cam-2×2	2.00	0.92
Wall-(6, 4)	50k	90.0	95.5	*Cam-all*	1.06	0.42
				Cam-4×4	2.52	0.92
				Cam-3×3	4.30	1.67
				Cam-2×2	5.90	2.50

almost identical to *Cam-all*. Therefore, it is not listed. SimAnneal was not evaluated on *Wall*-(4, 3); timing information reported for *Wall*-(4, 2) in [19] was used. It is clear that our system is fast: we can align *Wall*-(6, 4) in under 9 minutes.

5.5.5 Performance of the Color Matching System

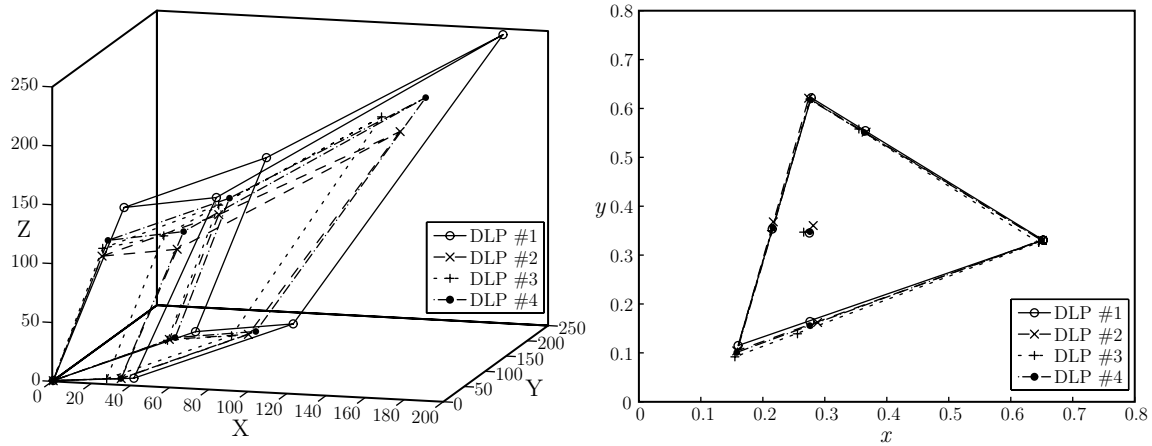
Here we compare our *Full-Gamut Color Matching* algorithm (referred to as FGCM afterwards) with the *Independent Channel Balancing* algorithm (referred to as ICB afterwards) for two test cases. In the first test case we use a uniform array of 4 DLP projectors, and in the second case we use a mixed array consisting of one DLP projector and one LCD projector. For these tests we implemented the FGCM algorithm in Matlab, and for comparison we implemented an ICB algorithm, such as that described in [62, 85]. For each test case, we perform the evaluation procedure outlined in Section 5.4.4, and calculate the color consistency metrics defined in Section 5.4.5.

Case 1: Uniform DLP Projector Array

We compare the performance of FGCM and ICB on a small tiled display consisting of four Compaq MP1800 DLP projectors. They exhibit the characteristic white enhancement non-additive gamuts as shown in Figure 5.5. These non-additive gamuts are difficult to match with an ICB approach, as the results indicate.

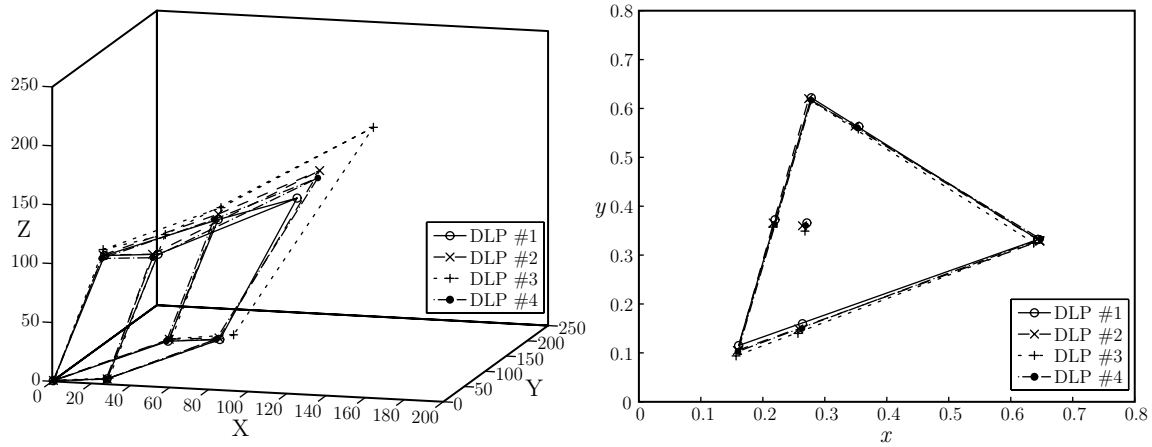
Figure 5.12a–5.12c compare the color gamuts of all four projectors before calibration, after ICB, and after FGCM, respectively. The outlines of color gamuts are shown in CIE XYZ space. The six faces of each gamut are created from the 8 sample colors (R,G,B,C,Y,M,K,W). Each of the four projectors is represented by a different line style. We notice that the white points of the uncalibrated color gamuts are stretched due to white enhancement. Further, the variation in bulb output causes the scale of the four gamuts to vary greatly, as shown in Figure 5.12a. ICB is able to equalize the luminance output of each of the RGB channels, as indicated by the well-matched R, G, and B primary colors in Figure 5.12b. However, there is still a large discrepancy in the white points due to the non-additive gamut. Finally, Figure 5.12a shows that the FGCM algorithm is able to match the entire gamut. Notice that the RGB channels match closely and the white points are well aligned. Figure 5.12d–5.12f show the corresponding CIE x-y plot for the eight colors. Figure 5.12d shows that even in an uncalibrated array, the red, green and blue chromaticity values match well among the four projectors. This is because they are of the same model and built within certain tolerances. But the CYMW colors are not well aligned due to the non-additive gamut. As can be seen in Figure 5.12e, ICB does not bring the CYMW colors into alignment. Finally, Figure 5.12f shows that FGCM is able to match all eight colors.

To quantify the color matching results, we calculate the color consistency metrics. Table 5.3 shows the absolute consistency E_i and percentage consistency e_i (in paren-



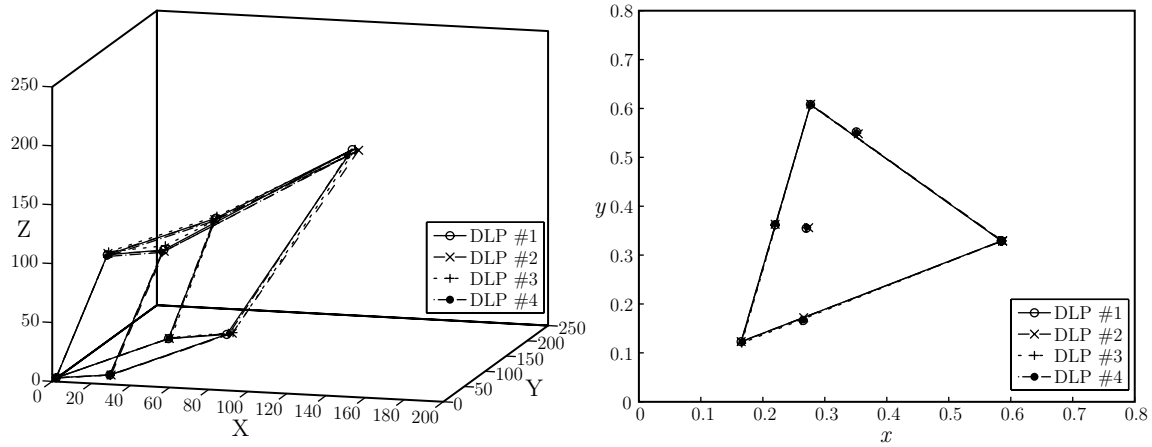
a) Uncalibrated (CIE XYZ plot)

d) Uncalibrated (CIE x-y plot)



b) After ICB (CIE XYZ plot)

e) After ICB (CIE x-y plot)



c) After FGCM (CIE XYZ plot)

f) After FGCM (CIE x-y plot)

Figure 5.12: Color Gamuts of a Tiled Display with DLP Projectors.

Table 5.3: Color Consistency of a DLP Projector Tiled Display.

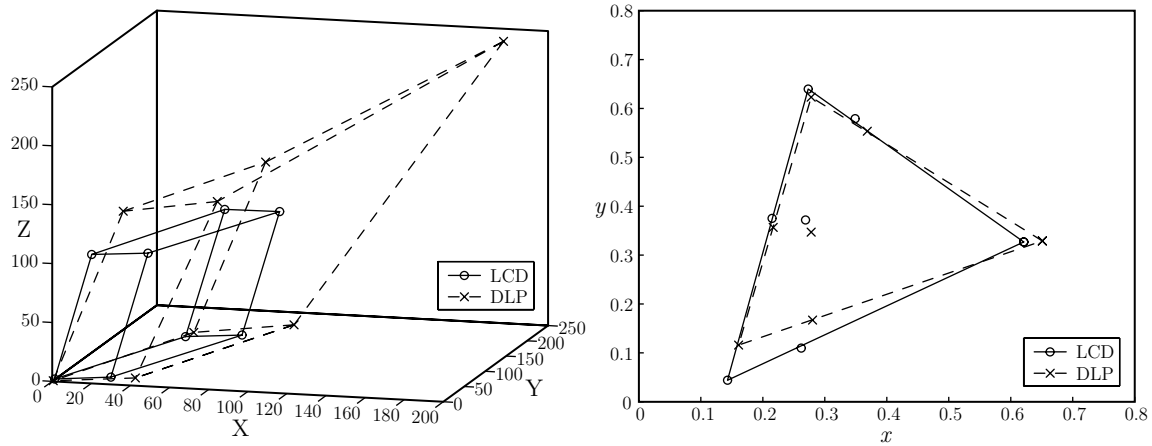
Color	Uncalibrated	After ICB	After FGCM
Red (R)	3.621 (10.22%)	0.509 (1.78%)	0.229 (0.75%)
Green (G)	7.880 (7.36%)	1.713 (1.71%)	0.770 (0.77%)
Blue (B)	13.256 (10.75%)	2.599 (2.38%)	1.313 (1.20%)
Cyan (C)	18.073 (9.54%)	4.103 (2.41%)	1.435 (0.86%)
Magenta (M)	15.875 (11.02%)	7.230 (5.95%)	1.977 (1.61%)
Yellow (Y)	12.556 (8.64%)	4.003 (3.26%)	1.453 (1.14%)
Black (K)	0.123 (15.59%)	0.207 (19.48%)	0.118 (3.26%)
White (W)	33.237 (10.40%)	21.272 (8.95%)	2.869 (1.11%)
Overall	9.710 (11.12%)	2.418 (3.40%)	0.984 (1.47%)

theses) for 8 test colors (R,G,B,C,Y,M,K,W). We also calculate the overall absolute consistency $\bar{E}$ and overall percentage consistency $\bar{e}$. As can be seen, our algorithm has a 1.47% error overall compared to 3.40% for channel balance and 11.12% for no correction. But the effect of gamut matching really becomes obvious near the white point where FGCM has a 1.11% error compared to 8.95% for ICB.

Case 2: Mixed DLP and LCD Projector Array

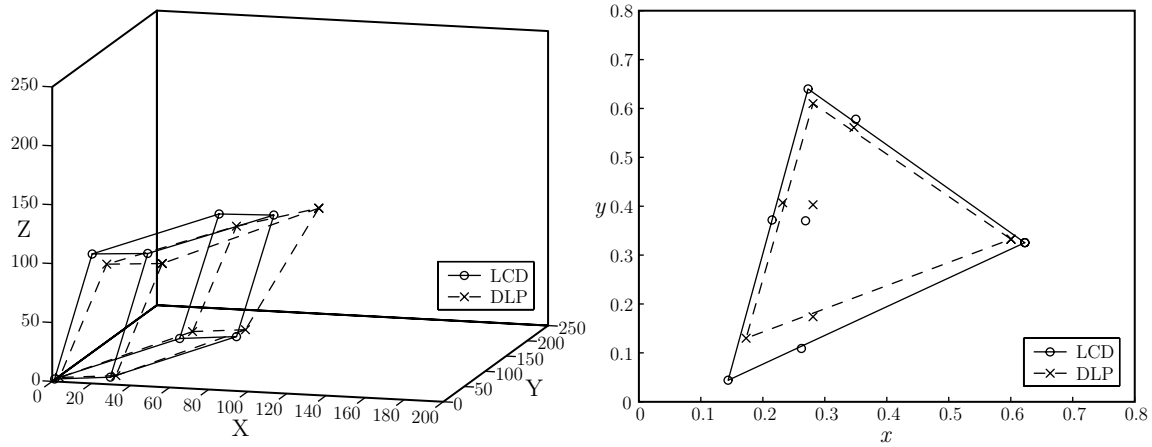
The second test case is a mixed array consisting of one Compaq MP1800 DLP projector and one Toshiba TLP511U LCD projector. We perform the same test procedure and calculate the same color consistency metrics as in the first test case. We again apply our FGCM algorithm and the ICB algorithm to this array of projectors. We present results in the same format as those presented in Section 5.5.5.

Figure 5.13a–5.13c show the gamut plots of both projectors before calibration, after ICB, and after FGCM, respectively. The corresponding CIE x-y plots for the test colors are shown in Figure 5.13d–5.13f. We notice that mixed arrays of projectors are challenging to color match because the chromaticity of the RGB primaries are likely



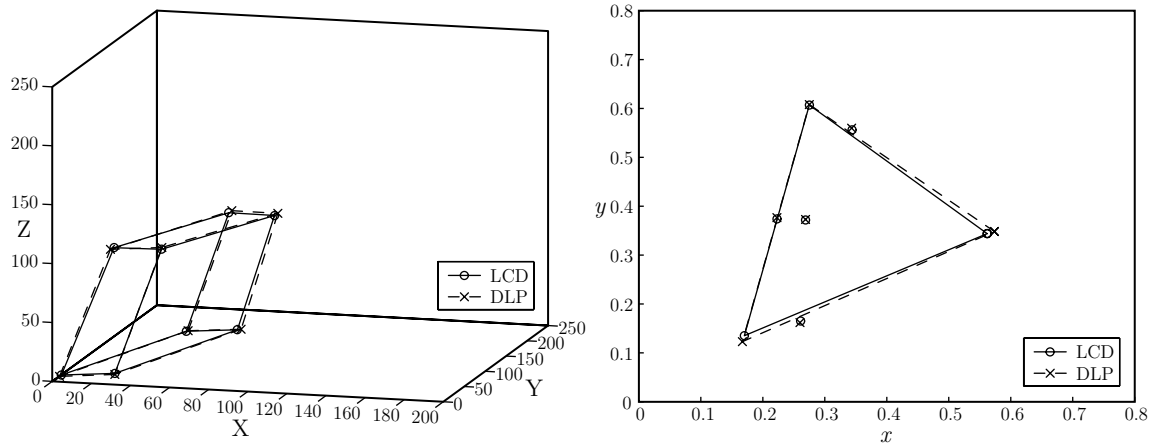
a) Uncalibrated (CIE XYZ plot)

d) Uncalibrated (CIE x-y plot)



b) After ICB (CIE XYZ plot)

e) After ICB (CIE x-y plot)



c) After FGCM (CIE XYZ plot)

f) After FGCM (CIE x-y plot)

Figure 5.13: Color Gamuts of a Tiled Display with Mixed Projectors.

Table 5.4: Color Consistency of a Mixed Projector Tiled Display.

Color	Uncalibrated	After ICB	After FGCM
Red (R)	6.269 (17.03%)	1.683 (5.28%)	0.444 (1.33%)
Green (G)	2.821 (2.37%)	4.485 (3.90%)	0.739 (0.64%)
Blue (B)	19.722 (15.42%)	8.703 (8.25%)	1.465 (1.28%)
Cyan (C)	22.419 (11.20%)	9.524 (5.49%)	1.458 (0.82%)
Magenta (M)	28.030 (19.38%)	8.587 (7.46%)	0.827 (0.67%)
Yellow (Y)	15.609 (10.09%)	3.671 (2.66%)	1.625 (1.19%)
Black (K)	0.795 (50.22%)	1.490 (40.02%)	0.836 (11.93%)
White (W)	93.586 (32.54%)	16.821 (8.28%)	1.409 (0.74%)
Overall	11.914 (12.95%)	4.435 (6.21%)	1.007 (1.27%)

to be very different, as can be seen in Figure 5.13d. As a result, the ICB algorithm produces significant error even for the RGB colors and the error becomes worse at the white point. Our FGCM algorithm is able to match the two color gamuts much better. Numerically, FGCM is able to achieve a 1.27% overall percentage consistency compared to 6.21% for ICB and 12.95% for uncalibrated, as reported in Table 5.4.

Real Time Imagery Correction Performance

We test the performance of real time imagery correction using programmable graphics hardware. Two commodity graphics cards are used: the ATI Radeon 9700 Pro and the Leadtek GeForce4 Ti4600. The GeForce4 card is installed on a PC with a 550 MHz Pentium III processor. The ATI card is installed on a second PC with a 3.06 GHz Pentium 4 processor. We use the frame rate of an image viewing application as the performance metric. To test the impact of different levels of imagery correction, we develop four pixel shaders:

A applies only the geometric alignment;

Table 5.5: Performance of Real Time Imagery Correction (fps).

Platform	A	B	C	D
550 MHz P3/GeForce4	22.9	22.6	22.3	22.1
3.06 GHz P4/Radeon	86.4	86.5	86.4	86.4

B applies the geometric alignment with an alpha mask;

C applies the geometric alignment with the color map;

D applies the geometric alignment, alpha mask and the color map as described in Section 5.3.6. This represents the full projector calibration.

Table 5.5 shows the frame rates of the image viewer using the four shaders. It is clear that there is no significant performance hit on either card from applying FGCM.

5.6 Summary

This chapter describes two key components in a projector calibration system—a scalable geometric alignment system and a full gamut color matching system. Using the homography tree algorithm, the alignment system is able to incorporate multiple camera views to achieve sub-pixel accurate alignment results for tiled displays containing up to hundreds of projectors. The color matching system uses a non-parametric color map for non-additive color gamuts of commodity DLP projectors, resulting in 1.5% color consistency among all projectors. Finally, we demonstrate that by leveraging on the latest programmable graphics hardware technology, all imagery corrections can be applied in real time with no performance impact on applications.

Our vision-based geometric alignment system is practical for large format multi-projector tiled displays. It uses a sub-pixel accurate feature detection algorithm that

simultaneously calibrates intrinsic camera parameters. It also includes an automatic vision-based system for measuring tiled display alignment accuracy. A comprehensive series of experimental tests on a 24-projector display demonstrate that our camera homography tree algorithm significantly improves local alignment accuracy by incorporating information from multiple, uncalibrated camera views. Our algorithm's accuracy exceeds that of existing solutions, and unlike those approaches, scales better as projectors are added to the display system.

We also provide an alignment simulation tool. It helps system designers examine the impact of design decisions on the expected accuracy of a tiled display. Simulation results indicate that our approach is practical even for very large-scale tiled displays. These simulations would help a designer determine the quality and number of cameras needed, and the time necessary for aligning a tiled display.

Our full gamut color matching system effectively color matches displays composed of projectors from different vendors (mixed arrays) or DLP projectors. These situations present particular challenges because of chromaticity variation in the primary colors and/or the white enhancement in DLP projectors. Our algorithm is able to achieve a measured color uniformity of 1.47% overall compared to 3.40% for an ICB algorithm on a DLP projector array. For mixed DLP/LCD projector arrays, our algorithm is able to outperform the ICB algorithm by a factor of 5, reducing the overall average error from 6.21% to 1.27%. We also demonstrate that color matching can be applied in real-time on the latest commodity graphics cards.

Our system is now in regular use at the Princeton Scalable Display Wall. The increasing size of tiled displays has warranted a new class of scalable alignment and automatic color matching solutions, such as those described here. We anticipate such solutions will increasingly be required by future displays.

One of the future work is to investigate the proper integration of luminance bal-

ancing and color matching, although luminance balancing itself is largely a solved problem. Using an alpha mask for luminance balancing implicitly requires a linear color transfer function, or 1.0 gamma. How to apply the alpha mask while preserving the projector gamma and a non-parametric color mapping still represents a challenge.

Rear projection screens, especially high gain "black" screens, have a severe brightness fall-off for off-axis view angles. Coupled with the fact that all projector pixels are not illuminated from a perpendicular direction, this causes abrupt brightness changes in the overlapping areas. Commercial cube-type projection systems solve this problem by using a Fresnel lens to bend all incoming projector light rays to be perpendicular to the screen. However, this is generally not feasible for a tiled display with haphazardly placed projectors. With proper modeling of the transmittance/reflectance characteristics of the screen material, it is possible to develop a dynamic photometric correction system that can compensate for the off-axis fall-off effect in real time. This can be especially useful for virtual reality applications with head-tracked users.

Chapter 6

Conclusions and Future Work

This dissertation provides a framework for scalable and flexible video delivery on tiled displays. To support a multitude of applications, we have classified the video sources into four types—single camera, single computer, multiple cameras, and multiple computers. We further proposed three classes of video encoding schemes—uni-stream video, tiled video, and layered video. These combinations of video sources and encodings allow a system designer to build a display system to support a wide variety of applications ranging from immersive tele-conferencing to high quality digital cinema.

We have described three major system components of a scalable video decoding system for tiled displays, namely, a high performance MPEG video decoder, a scalable parallel MPEG video decoder, and a projector calibration system for seamless video display. They can be easily extended and used as building blocks to construct any video decoding system that the framework allows. Through experiments, we have demonstrated that this decoding system is capable of satisfying the most demanding applications of the framework—it decodes and plays an IMAX resolution video at 39 frames per second on a commodity-based tiled display.

The next three sections summarize what we have learned from building this scal-

able video decoding system. The final section discusses future research directions for scalable video delivery.

6.1 High Performance MPEG Video Decoding

Scalable video delivery on a tiled display system requires a powerful parallel video decoder for the cluster architecture. We chose to base the parallel decoder on a software-based decoder core. There are two main reasons to this decision. First, unlike a hardware-based decoder or a closed source commercial software codec, an open source software decoder provides us unfettered access to the underlying modules and functions which is necessary when experimenting with different types of parallelization methods. Second, a software-based solution is cost-effective and at the same time allows the decoder performance to ride on the exponential curve of Moore's Law, both of which are critical to the success of a commodity-based video system.

Based on these reasonings, our design goal for the software decoder core is very clear—we want to achieve maximum performance through only minimum changes to a standard open source reference decoder.

After applying most known techniques to the reference decoder written by the MSSG, we obtained a decoder with a performance on par with the state-of-the-art commercial decoders. However, an in-depth analysis of this decoder revealed that it had become memory bound in two of the most time consuming modules—motion compensation and display.

We proposed three techniques to overcome the memory bottlenecks in this software decoder. First, we introduced an alternative frame buffer layout called the Interleaved Block-Order to exploit 2D cache locality present in the video decoder. Second, we explicitly prefetch reference macroblocks during the VLD phase. By doing so, we

successfully overlap the memory access latency with the computation intensive VLD. Third, we break the display granularity down to macroblock level. Smaller bursts of AGP port accesses are then interleaved with computation and main memory access. This results in an effective write bandwidth equal to the CPU write buffer. Together, all three techniques improve the SIMD instruction optimized decoder by another factor of two.

Meanwhile, because the optimizations happen at macroblock level, we are able to reuse most of the original reference code, and keep most of the decoding algorithm intact. This allows us to parallelize the decoder easily, in order to support ultra high resolution video decoding on a cluster architecture.

6.2 Scalable Parallel MPEG Video Decoding

Armed with a high performance software video decoder core, we further designed a scalable parallel video decoder for tiled displays. Our design goals of the parallel decoder were two-fold—scalable and high performance. To be scalable requires the decoder to be completely bottleneck free, in terms of both computation and communication. High performance is necessary to meet the most demanding need of the framework, that is, decoding an ultra-high resolution video on a commodity cluster.

Recognizing that functional parallelization works only on an SMP architecture and simple data parallelization does not scale, we designed a hierarchical parallel decoder. In such a system, two levels of splitters parse the video stream in tandem to keep an array of macroblock decoders running at full throttle. Because, the decoders operate at macroblock level, costly pixel redistribution is completely eliminated. Further, by using a number of second level macroblock splitters, we remove the computation bottleneck in parsing an MPEG stream at macroblock level. Parsing a stream at picture

level can be done as fast as the video can be send to the first level picture splitter. Therefore, the overall system is entirely bottleneck-free. The only potential limiting factor is the network/disk bandwidth available to the first level picture splitter.

Our experiments confirm that the system scales very well. It is able to achieve an almost linear acceleration as the number of nodes in a system increases. On the pure performance side, this parallel decoder is efficiently implemented on the high performance decoder core, and it plays an IMAX resolution video at 39 frames per second on a 4×4 tiled display driven by commodity PC's.

6.3 Seamless Video Display

Being able to decode ultra-high resolution videos at real time frame rates is not the end of the story for a scalable video decoding system on tiled displays. Due to the tiled nature of such displays, care needs to be taken in order to present a video seamlessly to the viewer. In a projector-based tiled display, three factors conspire together to ruin a perfect picture. First of all, there is the geometric misalignment in the overlapping regions between adjacent projectors which causes geometric distortion and discontinuity in the video. Second, the brightness (luminance) of the display varies both within individual projectors and across the entire screen. Finally, differences in color filters and projector bulbs cause the color characteristics to differ among the projectors. Although future display technologies might be able to ameliorate some of these situations, the projector array is the most popular and feasible way to build a tiled display. Therefore, all these problems must be solved for a scalable video decoding system.

Previous projector calibration methods address these issues with mostly static content in mind. With proper edge blending and luminance balancing, a few pixels of

alignment error with no color matching achieves satisfactory results for most images, because the artifacts mingle with the content. Video display, on the other hand, calls for a much more rigorous calibration. The human visual system is able to separate the static artifact field from the moving content very easily. A sub-pixel accurate alignment and fully matching colors are necessary for a video to appear seamless on a tiled display. And these are what we accomplished.

We first devised a scalable alignment system for tiled displays. Like previous systems, it relies on structure light and image processing to recover the geometry of each projector. What makes this system stand out is the homography tree algorithm that incorporates multiple camera views to produce a more detailed image of the screen, in essence, emulating a much higher resolution virtual camera. This allows the tiled display to scale to very large sizes. Experiments and simulations show that our algorithm is able to achieve sub-pixel accurate alignment results for displays with up to hundreds of projectors. This system is also flexible; it allows a system designer to trade the alignment accuracy with the number of camera views, which in turn determines the calibration speed. In most settings, only a few camera views are required to achieve a significantly improved alignment result. Therefore, the system is also fast, and practical for everyday use.

The second component we constructed for a projector calibration system is a full gamut color matching system. Previous color matching algorithms rely on channel constancy of the display devices, which is true for CRT and LCD projectors. However, most low-cost commodity DLP projectors today use a technique called White Enhancement to boost the light output. One undesirable consequence of this process is that channel constancy is no longer valid. This results in a non-additive color gamut which can not be mapped parametrically using a matrix transform. We designed a non-parametric full gamut color matching algorithm to address the challenges pre-

sented by DLP projectors. We use an inexpensive colorimeter to sample the color transfer function of each projector. A common standard transfer function is then derived, and a color map is calculated for each projector so that it can emulate the common color gamut. Our experiments indicate that the full gamut color matching algorithm is able to match a tiled display with DLP projectors, or even mixed DLP/LCD projectors, to about 1.5% color consistency.

Performing the geometric correction, luminance balancing, and color matching requires a large amount of computation. Fortunately, advances in programmable graphics hardware has made real time imagery correction possible. Through experiments we found that all these corrections can be applied on the graphics card without any performance impact on the applications.

6.4 Future Directions

In this dissertation, we proposed a framework for scalable video delivery on tiled displays. Further, we designed, implemented, and evaluated three major components for building a scalable decoding system. Although this is by no means a complete system, it provides a solid groundwork for the entire delivery pipeline. Here we describe some of the future research directions.

Scalable Encoding

We have only described the second half of an entire delivery pipeline. The encoding portion of it is as important and even more challenging. Similar to the decoder, a scalable encoder starts from a high performance video encoder engine. Although we would like to take the same approach and develop a software encoder, the dynamics are different for the encoder. Video compression is typically orders of magnitude more

complex and slower than the decompression. In order to build a real time streaming system, it is very likely that a hardware-based encoding solution might be needed. Software encoders can still used for offline compression of videos. Some applications of the framework generate uni-stream video from multiple sources. In this case, a parallel encoder is needed.

Novel Imaging Devices

As we have described in Chapter 1, there are many novel imaging devices that need to be built for the framework, such as a camera array, a multi-resolution camera system, or a multi-resolution CCD device. These require some ingenuity in mechanical design, optical design, and electrical engineering, which is outside the scope of computer science research. They are nonetheless an integral part of the framework and key to its overall success.

Content Creation for Scalable Videos

Last, but certainly not the least, new studies are needed to investigate how to create and present video contents efficiently and effectively on such a large scale high resolution display. Design methodologies developed for desktop monitors typically do not work well for such displays, because of the larger viewing distance, wider viewing angle, and the freedom to move in front of the screen.

New tools for creating high resolution contents are also necessary. These include not only the new imaging devices and encoding systems mentioned before, but also scalable content editing, management, and storage.

We have been living in a world with TV resolution contents and displays for the past half century. Now, finally, with advances in display technology, scalable high resolution video has become not only feasible but practical. While this dissertation only scratches the surface, we hope that research in this area will continue and flourish, and one day these videos will be as ubiquitous as computing itself, so that people can better communicate with each other and the world.

Bibliography

[1] W. Abu-Sufah, D. J. Kuck, and D. H. Lawrie. Automatic Program Transformations for Virtual Memory Computers. In *Proceedings of the National Computer Conference*, pages 969–974, June 1979.

[2] S. M. Akramullah, I. Ahmad, and M. Liou. A Data-Parallel Approach for Real-Time MPEG-2 Video Encoding. *Journal of Parallel and Distributed Computing*, 30(2):129–146, Nov 1995.

[3] P. M. Alt. Displays for Electronic Imaging. *IEEE Micro*, 18(6):42–53, Nov/Dec 1998.

[4] Y. Arai, T. Agui, and M. Nakajima. A Fast DCT-SQ Scheme for Images. In *Transactions of the IEICE*, number 11, pages 1095–1097, November 1988.

[5] J.-L. Baer and T.-F. Chen. An Effective On-chip Preloading Scheme to Reduce Data Access Penalty. In *Proceedings of the 1991 Conference on Supercomputing*, pages 176–186, 1991.

[6] A. Bala, D. Shah, U. Feng, and D. K. Panda. Experience with Software MPEG-2 Video Decompression on an SMP PC. In *Proceedings of the 1998 ICPP Workshop on Architectural and OS Support for Multimedia Applications/Flexible*

Communication Systems/Wireless Networks and Mobile Computing, pages 29–36, 1998.

[7] P. Baudisch, D. DeCarlo, A. T. Duchowski, and W. S. Geisler. Attentive User Interfaces: Focusing on the Essential: Considering Attention in Display Design. *Communications of the ACM*, 46(3), Mar 2003.

[8] A. Bavier, A. B. Montz, and L. Peterson. Predicting MPEG Execution Times. In *Proceedings of ACM SIGMETRICS*, pages 131–140, June 1998.

[9] M. Bern and D. Eppstein. Optimized Color Gamuts for Tiled Displays. In *Proceedings of the ACM Symposium on Computational Geometry*, pages 274–281, 2003.

[10] A. Bilas, J. Fritts, and J. P. Singh. Real-Time Parallel MPEG-2 Decoding in Software. In *Proceedings of the 11th International Parallel Processing Symposium*, Geneva, Switzerland, Apr 1997.

[11] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. Seitz, J. Seizovic, and W.-K. Su. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[12] D. Callahan, K. Kennedy, and A. Porterfield. Software Prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, 1991.

[13] B. Chazelle and D. P. Dobkin. Intersection of Convex Objects in Two and Three Dimensions. *Journal of the ACM*, 34(1):1–27, January 1987.

[14] H. Chen, K. Li, and B. Wei. A Parallel Ultra-High Resolution MPEG-2 Video Decoder for PC Cluster Based Tiled Display System. In *International Parallel and Distributed Processing Symposium*, Apr 2002.

[15] H. Chen, K. Li, and B. Wei. Memory Performance Optimizations for Real-Time Software HDTV Decoding. In *IEEE International Conference on Multimedia and Expo*, Aug 2002.

[16] H. Chen, R. Sukthankar, G. Wallace, and K. Li. Scalable Alignment of Large-Format Multi-Projector Displays Using Camera Homography Trees. In *IEEE Visualization*, Oct 2002.

[17] H. Chen, G. Wallace, A. Gupta, K. Li, T. Funkhouser, and P. Cook. Experiences with Scalability of Display Walls. In *Proceedings of Immersive Projection Technology Symposium*, Mar 2002.

[18] T.-F. Chen and J.-L. Baer. A Performance Study of Software and Hardware Data Prefetching Schemes. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 223–232, 1994.

[19] Y. Chen, D. Clark, A. Finkelstein, T. Housel, and K. Li. Automatic Alignment of High-Resolution Multi-Projector Display Using an Uncalibrated Camera. In *Proceedings of IEEE Visualization*, 2000.

[20] S. Coleman and K. S. McKinley. Tile Size Selection Using Cache Organization and Data Layout. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 279–290, 1995.

[21] C. Cruz-Neira, D. Sandin, and T. DeFanti. Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE. In *Proceedings of ACM SIGGRAPH*, pages 135–142, 1993.

[22] C. Cruz-Neira, D. J. Sandin, T. A. DeFanti, R. V. Kenyon, and J. C. Hart. The CAVE: audio visual experience automatic virtual environment. *Communications of the ACM*, 35(6), June 1992.

[23] R. Cucchiara, M. Piccardi, and A. Prati. Exploiting Cache in Multimedia. In *IEEE International Conference on Multimedia Computing and System*, volume 1, pages 345–350, 1999.

[24] R. Cucchiara, M. Piccardi, and A. Prati. Hardware Prefetching Techniques for Cache Memories in Multimedia Applications. In *Proceedings of the 5th IEEE International Workshop on Computer Architectures for Machine Perception*, pages 311–319, 2000.

[25] P. E. Debevec and J. Malik. Recovering High Dynamic Range Radiance Maps from Photographs. In *Proceedings of ACM SIGGRAPH*, pages 369–378, 1997.

[26] M. Deering. Geometry Compression. In *Proceedings of ACM SIGGRAPH*, pages 13–20, 1995.

[27] P. Denning. Virtual Memory. *Computing Surveys*, 2(3):169, September 1970.

[28] S. Eckart and C. E. Fogg. ISO/IEC MPEG-2 Software Video Codec. In *Proceedings of Digital Video Compression: Algorithms and Technologies 1995*, pages 100–109. SPIE, 1995.

[29] J. L. Elshoff. Some Programming Techniques for Processing Multi-Dimensional Matrices in a Paging Environment. In *Proceedings of the National Computer Conference*, 1974.

[30] A. Finkelstein, C. E. Jacobs, and D. H. Salesin. Multiresolution video. In *Proceedings of ACM SIGGRAPH*, 1996.

[31] T. Funkhouser and K. Li. Large Format Displays. *IEEE Computer Graphics and Applications*, 20(4), 2000. Guest editor introduction to special issue.

[32] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5:587–616, 1988.

[33] M. Hereld, I. Judson, J. Paris, and R. Stevens. Developing Tiled Projection Display Systems. In *Proceedings of Fourth Immersive Projection Technology Workshop*, 2000.

[34] M. D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance*. PhD thesis, Computer Science Division, University of California at Berkeley, 1987.

[35] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan. WireGL: A Scalable Graphics System for Clusters. In *Proceedings of ACM SIGGRAPH*, 2001.

[36] G. Humphreys, M. Houston, Y.-R. Ng, R. Frank, S. Ahern, P. Kirchner, and J. T. Klosowski. Chromium: A Stream Processing Framework for Interactive Graphics on Clusters. In *Proceedings of ACM SIGGRAPH*, 2002.

[37] M. Ikekawa, D. Ishii, E. Murata, K. Numata, Y. Takamizawa, and M. Tanaka. A Realtime Software MPEG-2 Decoder For Multimedia PCs. In *International Conference on Consumer Electronics, Digest of Technical Papers*, pages 2–3, 1997.

[38] IMAX Corporation. IMAX Website. <http://www.imax.com>.

[39] Intel Corporation. Intel Architecture Optimization Reference Manual. <http://developer.intel.com/design/pentiumii/manuals/245127.htm>.

[40] Intel Corporation. Intel Architecture Software Developer's Manual Volume 3: System Programming. <http://developer.intel.com/design/pentiumii/manuals/243192.htm>.

[41] Intel Corporation. Open Source Computer Vision Library. <http://www.intel.com/research/mrl/research/opencv/>.

[42] Intel Corporation. VTune Performance Analyzer. <http://developer.intel.com/software/products/vtune/>.

[43] ISO/IEC 13818-2:2000. *Information technology – Generic coding of moving pictures and associated audio information: Video*. 2nd edition, 2000.

[44] ISO/IEC 14496-2:2001. *Coding of Audio-Visual Objects—Part 2: Visual*. 2nd edition, 2001.

[45] ITU-T. *Recommendation H.263: Video Coding for Low Bitrate Communication*. ITU, 1995.

[46] ITU-T. *Recommendation H.264: Advanced Video Coding for Generic Audiovisual Services*. ITU, 2003.

[47] K. Jack. *Video Demystified: A HandBook for the Digital Engineer*. HighText Publications, 1996.

[48] N. P. Jouppi. Improving Direct-mapped Cache Performance by the Addition of a Small Fully-associative Cache Prefetch Buffers. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 364–375, 1990.

[49] E. Kang, I. Cohen, and G. Medioni. A Graph-Based Global Registration for 2D Mosaics. In *Proceedings of International Conference on Pattern Recognition*, 2000.

[50] A. C. Klaiber and H. M. Levy. An Architecture for Software-Controlled Data Prefetching. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 43–53, 1991.

[51] J. T. Klosowski, P. D. Kirchner, J. Valuyeva, G. Abram, C. J. Morris, R. H. Wolfe, and T. Jackman. Deep View: High-Resolution Reality. *IEEE Computer Graphics & Applications*, 22(3):12–15, May/June 2002.

[52] W. Kunzman and G. Pettitt. White Enhancement for Color Sequential DLP. In *SID Conference Proceedings*, 1998.

[53] M. K. Kwong, P. T. Tang, and B. Lin. A Real Time MPEG Software Decoder Using a Portable Message-Passing Library. Technical Report Preprint MCS-P506-0395, Mathematics and Computer Science Division ANL, April 1995.

[54] M. D. Lam, E. E. Rothberg, and M. E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, 1991.

[55] R. B. Lee. Realtime MPEG Video via Software Decompression on a PA-RISC Processor. In *Compcon '95, Technologies for the Information Superhighway*, pages 186–192, 1995.

[56] W. Lee, G. J. Golston, and Y. Kim. Real-time MPEG Video Codec on a Single-Chip Multiprocessor. In *Proceedings of the SPIE Conference on Digital Video Compression on Personal Computers: Algorithms and Technologies*, pages 2187:32–42, Feb 1994.

[57] D. LeGall. MPEG: A Video Compression Standard for Multimedia Applications. *Communications of the ACM*, 34(4):46–58, April 1991.

[58] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk. The Digital Michelangelo Project: 3D Scanning of Large Statues. In *Proceedings of ACM SIGGRAPH*, pages 131–144, 2000.

[59] K. Li, H. Chen, Y. Chen, D. W. Clark, P. Cook, S. Damianakis, G. Essl, A. Finkelstein, T. Funkhouser, T. Housel, A. Klein, Z. Liu, E. Praun, R. Samanta, B. Shedd, J. P. Singh, G. Tzanetakis, and J. Zheng. Building and Using a Scalable Display Wall System. *IEEE Computer Graphics and Applications*, 20(4):29–37, July/August 2000.

[60] M. Liou. Overview of the p×64 kbit/s Video Coding Standard. *Communications of the ACM*, 34(4):59–63, April 1991.

[61] L. C. Loschky and G. W. McConkie. User performance with gaze contingent multiresolutional displays. In *Proceedings of the symposium on Eye tracking research & applications 2000*, pages 97–103. ACM Press, 2000.

[62] A. Majumder, Z. He, H. Towles, and G. Welch. Achieving Color Uniformity across Multiprojector Displays. In *Proceedings of IEEE Visualization*, 2000.

[63] A. Majumder and R. Stevens. LAM: Luminance Attenuation Map for Photometric Uniformity Across a Projection Based Display. In *ACM Virtual Reality and Software Technology*, 2002.

[64] D. Mentley. State of Flat-panel Display Technology and Future Trends. *Proceedings of the IEEE*, 90(4):453–459, Apr 2002.

[65] D. Monk. Digital Light Processing: a New Image Technology for the Television of the Future. In *International Broadcasting Convention*, pages 581–586, Sep 1997.

[66] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8), April 1965. <http://download.intel.com/research/silicon/moorespaper.pdf>.

[67] T. C. Mowry. Tolerating Latency in Multiprocessors Through Compiler-inserted Prefetching. *ACM Transactions on Computer System*, 16(1):55–92, Feb 1998.

[68] T. C. Mowry, M. S. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, 1992.

[69] MPEG Requirements. *Overview of the MPEG-7 Standard*. Doc. ISO/MPEG N4509, Pattaya MPEG Meeting, 2001.

[70] Myricom Inc. GM Library Reference. <http://www.myricom.com/scs/GM/doc/gm-toc.html>.

[71] K. Patel, B. C. Smith, and L. A. Rowe. Performance of a Software MPEG Video Decoder. In *Proceedings of the 1st ACM International Conference On Multimedia*, pages 75–82, 1993.

[72] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design*. Morgan Kaufmann Publishers, second edition, 1998.

[73] A. Peleg, S. Wilkie, and U. Weiser. Intel MMX for Multimedia PCs. *Communications of the ACM*, 40(1):25–38, Jan 1997.

[74] J. Philbin, J. Edler, O. J. Anshus, C. C. Douglas, and K. Li. Thread Scheduling For Cache Locality. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 60–71, 1996.

[75] P. Ranganathan, S. Adve, and N. P. Jouppi. Performance of Image and Video Processing with General-Purpose Processors and Media ISA Extensions. In *Proceedings of International Symposium on Computer Architecture*, pages 124–135, 1999.

[76] P. Ranganathan, V. S. Pai, H. Abdel-Shafi, and S. V. Adve. The Interaction of Software Prefetching with ILP Processors in Shared-Memory Systems. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 144–156, 1997.

[77] R. Raskar, M. Brown, R. Yang, W. Chen, G. Welch, H. Towles, B. Seales, and H. Fuchs. Multi-Projector Displays using Camera-Based Registration. In *Proceedings of IEEE Visualization*, 1999.

[78] R. Raskar, G. Welch, M. Cutts, A. Lake, L. Stesin, and H. Fuchs. The Office of the Future: A Unified Approach to Image-based Modeling and Spatially Immersive Displays. In *Proceedings of the ACM SIGGRAPH*, July 1998.

[79] R. Samanta, J. Zheng, T. Funkhouser, K. Li, and J. P. Singh. Load Balancing for Multi-Projector Rendering Systems. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, August 1999.

[80] D. Schikore, R. Fischer, R. Frank, R. Gaunt, J. Hobson, and B. Whitlock. High-resolution Multiprojector Display Walls. *IEEE Computer Graphics and Applications*, 20(4):38–44, Jul/Aug 2000.

[81] P. Sederquist and M. Leeser. Optimizing the Data Cache Performance of a Software MPEG-2 Video Decoder. In *Proceedings of International Conference on Multimedia*, pages 291–301, 1997.

[82] H. Shum and R. Szeliski. Panoramic Image Mosaics. Technical Report MSR-TR-97-23, Microsoft Research, 1997.

[83] A. J. Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, Sep 1982.

[84] M. C. Stone. Color and Brightness Appearance Issues for Tiled Displays. *IEEE Computer Graphics and Applications*, September 2001.

[85] M. C. Stone. Color Balancing Experimental Projection Displays. In *9th IS&t/SID Color Imaging Conference*, April 2001.

[86] R. Sukthankar, R. Stockton, and M. Mullin. Smarter Presentations: Exploiting Homography in Camera-Projector Systems. In *Proceedings of International Conference on Computer Vision*, 2001.

[87] R. Surati. *A Scalable Self-Calibrating Technology for Seamless Large-Scale Displays*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1999.

[88] C. W. Tang and S. A. VanSlyke. Organic Electroluminescent Diodes. *Applied Physics Letters*, 51(12):913–915, September 1987.

[89] G. Taubin and J. Rossignac. Geometric Compression through Topological Surgery. *ACM Transactions on Graphics*, 17(2):84–115, April 1998.

[90] Y. Tung, C. Ho, and J. Wu. MMX-based DCT and MC Algorithms for Real-Time Pure Software MPEG Decoding. In *IEEE International Conference On Multimedia Computing and Systems*, volume 1, pages 357–362, 1999.

[91] H. Uchiike and T. Hirakawa. Color Plasma Displays. *Proceedings of the IEEE*, 90(4):533–539, Apr 2002.

[92] G. Wallace, H. Chen, , and K. Li. Color Gamut Matching for Tiled Display Walls. In *Proceedings of Immersive Projection Technology Workshop (IPT2003)*, May 2003.

[93] B. Wei, C. Silva, E. Koutsofios, S. Krishnan, and S. North. Visualization Research With Large Displays. *IEEE Computer Graphics and Applications*, 20(4):50–54, Jul/Aug 2000.

[94] B. Wilburn, M. Smulski, H.-H. K. Lee, and M. Horowitz. The Light Field Video Camera. In *Proceedings of Media Processors, SPIE Electronic Imaging*, 2002.

[95] C. Wyszecki and W. S. Stiles. *Color Science: Concepts and Methods, Quantitative Data and Formulae*. Wiley-Interscience, 2nd edition, July 2000.

[96] R. Yang, D. Gotz, J. Hensley, H. Towles, and M. Brown. PixelFlex: A Reconfigurable Multi-Projector Display System. In *Proceedings of IEEE Visualization*, 2001.

[97] W. Yang, H. Kim, M. Shin, I. Park, and C. Kyung. A Multi-Threading MPEG Processor with Variable Issue Modes. In *The 6th International Conference on VLSI and CAD*, pages 545–548, 1999.

[98] Y. Yu and D. Anastassiou. Software Implementation of MPEG-2 Video Encoding Using Socket Programming in LAN. In *Proceedings of the SPIE Conference on Digital Video Compression on Personal Computers: Algorithms and Technologies*, pages 2187:229–240, Feb 1994.

[99] C. Zhou and et al. MPEG Video Decoding with the UltraSPARC Visual Instruction Set. In *Compcon '95 Technologies for the Information Superhighway*, pages 470–477, 1995.

[100] D. F. Zucker, M. J. Flynn, and R. B. Lee. A Comparison of Hardware Prefetching Techniques for Multimedia Benchmarks. In *Proceedings of the Third IEEE International Conference on Multimedia Computing and Systems*, pages 236–244, 1996.

[101] D. F. Zucker, M. J. Flynn, and R. B. Lee. Improving Performance for Software MPEG Players. In *Compcon '96 Technologies for the Information Superhighway*, pages 327–332, 1996.

[102] D. F. Zucker, R. B. Lee, and M. J. Flynn. An Automated Method for Software Controlled Cache Prefetching. In *Proceedings of the Thirty-First Hawaii International Conference on System Sciences*, volume 7, pages 106–114, 1998.

[103] D. F. Zucker, R. B. Lee, and M. J. Flynn. Hardware and Software Cache Prefetching Techniques for MPEG Benchmarks. *IEEE Transactions on Circuits and Systems for Video Technology*, 10(5):782–796, Aug 2000.