Video over IP: An Example Reconfigurable Computing Application for a Handheld Device

by

Elina Kamenetskaya

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

[June 2003]

May 20, 2003

Author_____
Department of Electrical Engineering and Computer Science
May 20, 2003

Certified by_____
Krste Asanovic
Thesis Supervisor

Accepted by_____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Video over IP: An Example Reconfigurable Computing Application

for a Handheld Device

by

Elina Kamenetskaya

Submitted to the
Department of Electrical Engineering and Computer Science

May 20, 2003

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

As the demand for live video communication increases, the devices available to the users need to be made faster, smaller, lower power, and easily modifiable. This project demonstrates an example video over IP application using a Compaq IPaq along with a field programmable gate array used for data processing. The video data is passed directly from the camera to the network card via the gate array, thus avoiding unnecessary delay and computation requirements imposed on the CPU by regular video over IP communication software. The application achieves a display rate of 9 frames per second with 76800 byte frames, corresponding to a wireless throughput of 5.5Mbps. The CPU usage is reduced to 1/3 of that used by a standard video over IP implementation operating at the same rate.

Thesis Supervisor: Krste Asanovic
Title: Professor, MIT Laboratory for Computer Science

Acknowledgements

While working on my thesis, I have received immense help and support from students and staff at MIT, HP, as well as from my family and friends. From MIT, I would like to thank my thesis supervisor, Professor Krste Asanovic for his guidance and support, as well as Kenneth Steele, Fataneh Ghodrat and Kenneth Barr. From HP, I would like to thank Jamey Hicks, Brian Avery, Andrew Christian, and George France for always having an answer to any question and never hesitating to take time out of their day to help me. I would also like to thank my mother and father for their support and patience throughout all my years at MIT, and Anne Hunter for somehow having the time to handle over 5000 peoples' problems.

Table of Contents

List of Figures

1. Background and Motivation

1.1 Research Overview

This thesis discusses the purpose and implementation of a video over IP application for a handheld device. Uses of video communication over IP include internet telephony, video conferencing, collaborative computing, and distance learning. The actual frame rate and resolution of a video over IP application depends on the speed of the video processing hardware/software and network bandwidth. Quality applications require high processor capabilities or specific image processing hardware. This is due to the fact that video communication is generally extremely processor intensive. When the general purpose CPU is the only available resource, it must absorb video data from the camera, compress the data, sort it into packets, and then push it through to the network card for transmission. In order to free the CPU from these tasks while running video communication software, this application uses a Virtex 300E XILINX gate array to transfer the data directly from the camera to the network card.

Using a gate array for video acquisition and processing brings many advantages to the proposed application. As the demand for video communication increases, the devices available to the users need to achieve faster frame rate requirements, reduced processor allocation demands, reduced size constraints, reduced power consumption needs, and easy modification capabilities. These are the five criterions I would like to achieve in my thesis.

- **Faster Frame Rate** - Most of the processing time is wasted as the CPU attempts to process large amounts of image data, compress it, and send it out to the network. This project avoids putting large computational strains on the processor by passing live video data directly from the camera to the network card, thus bypassing the CPU.
- **Reduced Processor Allocation** - In addition to speeding up the video communication, freeing the processor adds another inherent benefit to the user by saving computational resources and thus enabling him to run other applications.
- **Reduced Size** - This application is developed on a handheld designed to host a wide variety of reconfigurable computing applications. No new hardware needs to be added to the device; a gate array simply needs to be programmed with the appropriate application.
- **Reduced Power Consumption** - Extra power is needed only to supply the gate array. This is significantly lower than running all video processing and compression directly on the CPU.
- **Easy Modification Capabilities** – It takes less then a second to program most gate arrays. Binary VHDL application files can be stored on the handheld and loaded into the gate array at will, thus enabling the user to run a wide variety of applications at fast hardware speeds.

1.2 Reconfigurable Computing

Reconfigurable computing combines the flexibility of software and the speed advantage of hardware. General-purpose processors are suitable for a wide variety of tasks. However, to tackle computational intensive problems designers have often used Application Specific Integrated Circuits (ASICs). These two extremes present a trade-off between speed and flexibility. ASICs achieve the highest speed, but provide no reconfigurability options. Microprocessors lack in speed but can be made to tackle the greatest variety of tasks. Programmable Gate Arrays (PGA's) lie in between these two extremes. They enable the designer to create application specific hardware, this achieving a high speed advantage over a processor. However they can be reconfigured to solve a completely different application specific problem, thus providing much of the flexibility of a processor.

The ease of the gate array's modification is advantageous to both the user and the designer. Wireless communication standards evolve quickly and demands for new features and services within these standards arise quite rapidly. This dynamic environment drives the equipment manufacturers towards programmable solutions that they can update in the field. The ideal is a platform that service providers can deploy and then upgrade in the field with only software changes over its operational life [8]. This is what using a gate array enables us to do. For example, in order to implement a different compression algorithm or a different packet transport scheme to keep up with the ever changing standards, the designer simply needs to modify the Hardware Description Language (HDL) code used to program gate arrays. No additional hardware needs to be purchased or installed.

1.3 Video over IP

Video communication has revolutionized the way people stay in touch with one another, discuss business, learn, and teach. Applications of video communication over IP include internet telephony, video conferencing, collaborative computing, and distance learning. Many companies are in the process of setting up or have already set up video communication equipment to conserve on time and travel costs while keeping personalized interaction with their clients and co-workers. Companies are especially attracted to video over IP applications because these will provide free unlimited video communication, as most companies already have Local Area Network (LAN) capabilities.

In addition to the usefulness of these applications in homes and offices, as the use of handhelds and the accessibility of LAN's become available to a wider audience, the range of IP telephony and video applications gets even further extended to mobile systems. LAN's are now set up not only in research institutions and businesses, but in many public areas, such as Newberry Street in Boston. The idea of having a free anytime audio and video connection with anyone in the world naturally appeals to a very vide range of users. Handhelds are already widely used in many audio over IP applications, and this is only a short step away from a full audio and video over IP connection.

2. Two Approaches to Video over IP

This thesis describes two ways to approach a Video over IP application design. The original implementation uses the iPaq StrongARM processor to read the data from

the camera device, process the data, sort it into network packets, and write it to the network card for transmission to the recipient. The enhanced approach achieves the same basic functionality and improved performance, however all video data is processed by modules in an FPGA, thus freeing the iPaq resources.

The original configuration refers to the configuration provided at the start of the project by Brian Avery and Jamey Hicks. This configuration can be thought of the standard way of implementing a video over IP application. The image data fills up the FIFO queue in the FPGA, and is then fed directly to the iPaq StrongARM processor. Upon acquiring one full frame of data, the processor breaks it up into smaller chunks and sends it to the network card for transmission. Refer to Figure 1 below for a quick overview of this process. Although the routing from the iPaq processor to the wireless card traverses the FPGA, the FPGA does not serve a practical role in this scheme – it simply routs the request and response signals directly between the iPaq PCMCIA port and the wireless card socket connection.

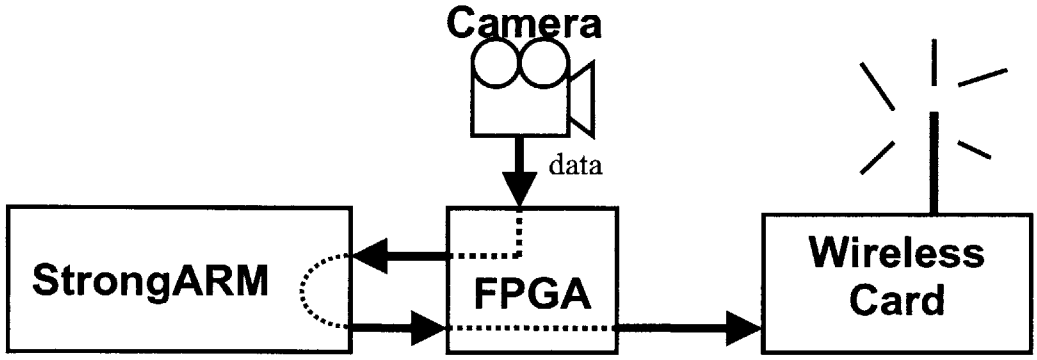

Figure 1: Original Application Overview. The StrongARM processor reads the data from the camera with the help of several modules in the FPGA. It then forwards the data through the FPGA to the wireless card.

The enhanced Video over IP application creates an alternative path for the image data. The data now travels directly from the camera to the wireless card via the FPGA, thus bypassing the processor of the iPaq. The purpose of the CPU in the iPaq shifts from being the primary resource for data manipulation to only supplying control signals to the drivers and the FPGA. When enough data has accumulated in the FIFO of the FPGA, the software drivers simply send a special packet header to the wireless card through the FPGA. The FPGA listens for this header, and upon reception, writes the FIFO data directly into the network card following the header information. Refer to Figure 2 below for an illustration of the data and header path of the enhanced application.

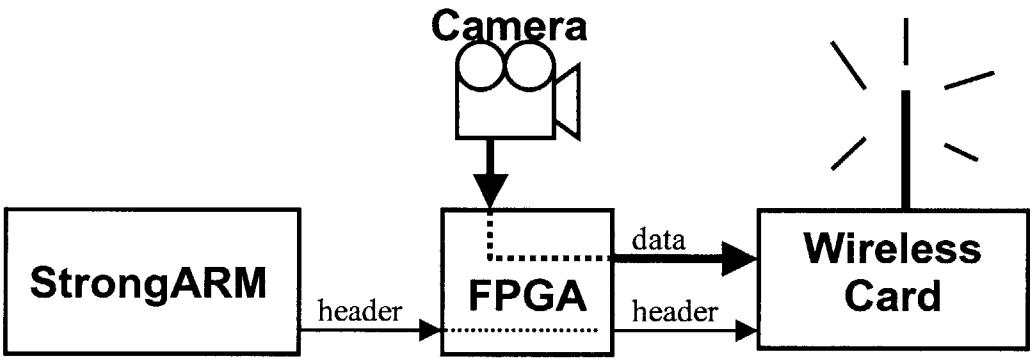

Figure 2: Enhanced Application Overview. The StrongARM processor simply sends the packet header information through the FPGA into the wireless card. The data is written by the FPGA directly into the card.

3. Devices and Drivers

In order to construct a video over IP application, several basic steps need to occur. First, the application needs to establish communication with the camera, initialize its parameters, and instruct it to start streaming image data. This data then needs to be stored

in memory until a predefined amount has accumulated. At this point, communication with the network card needs to be established, and the data needs to be written to the card for transmission. The devices and drivers used in order to achieve the above requirements are described below.

3.1 Camera

The camera used in this application was the VGA CMOS H3600 Phillips Camera. Although the resolution of CMOS cameras is smaller than that of CCD cameras, CMOS cameras are sufficient for many everyday purposes such as security and video telephony. In these applications, a high resolution is not required and the physical size and cost of the camera are the most important considerations. The driver used and modified to communicate with the camera was the h3600_backpaq_camera driver written by Andrew Christian.

The h3600_backpaq_camera driver consists of standard IO functions to initialize the camera and to read images. The open method sets basic parameters such as the frame rate and the image resolution. The frame rate is determined by a clock signal provided to the camera, and can be modified as needed to change the frame rate and thus the rate at which the camera outputs data. The image resolution is set by configuring the FPGA decimator to operate in the desired decimation mode. Please refer to section 3.3 for a description of the decimator. Finally, the open method in the camera driver instructs the camera to start filling up the FIFO in the FPGA with image data.

The method of most interest is the read method. In the original configuration, the read function takes an argument buffer corresponding to the size of the image, and

returns after a complete frame has been read from the camera and written into this buffer. In the revised configuration, the read method instructs the camera to start streaming the data directly to the network card. The method does not return until the exit of the program.

3.2 Network

The network card used in this application was the Cabletron RoamAbout 802.11 DS. This card is rated for operation at up to 11 megabits per second (11Mbps). The driver used and modified to communicate with this card was the Orinoco Network Card Driver written by David Gibson.

The top module in the driver is orinoco_cs, which contains methods for initializing the network card and provides methods to open sockets and send data. It communicates with the network card via the PCMCIA port. The orinoco_cs module depends on two other modules: orinoco and hermes. The orinoco module provides higher-level functionality to transmit and receive data, while the hermes module provides low-level functions used to write bytes to the network card.

In the original configuration, once a buffer containing a full frame has been returned by the camera read method, the data was broken up into packets and sent to the destination address using the standard methods defined in the orinoco_cs driver. In the revised configuration, several methods were added to the driver to enable the FPGA to write the data to the network card directly.

3.3 FPGA

The FPGA used in this application was the FPGA built into the backpaq of the H3600 iPaq series, the Virtex 300E XILINX. The input/output pins of the FPGA which were of interest to this application are those connected to the camera, the PCMCIA ports, and card socket ports. Modules from the FPGA can also communicate directly to the iPaq using standard read/write operations and an interrupt flag.

The relevant modules of the FPGA were written by Brian Avery are depicted in Figure 3 below. The Phillips_camera component is used to communicate with the h3600_camera_driver and initialize the camera. The data from the camera then passes through a decimator to reduce frame resolution as requested by the user program. The resulting data then proceeds into a FIFO queue. The FPGA also contains two PCMCIA_Slice modules to communicate with the two available PCMCIA slots.

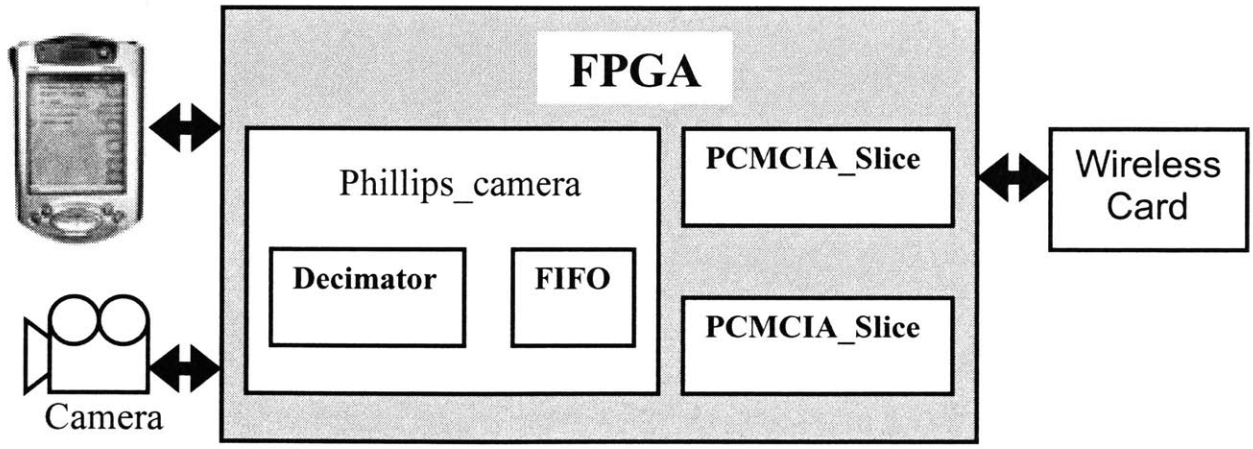

Figure 3: Connections and Components of the FPGA. The FPGA is able to communicate with the iPaq, the camera, and a wireless card connected to one of the PCMCIA ports.

Notice that the FPGA has access to both the iPaq side of the PCMCIA port and the socket side of the port which actually connects to the network card. In the original configuration, signals from the iPaq PCMCIA side were simply routed to the corresponding signals on the socket port, thus giving the iPaq a direct connection to talk to the card. However, the FPGA gives us the ability to interfere with this communication anytime we choose to do so. For example, the FPGA can write a packet directly to the card or it can modify certain fields of the packet as it they are written from the iPaq to the wireless card. The following sections of this thesis will demonstrate the video over IP application taking advantage of this setup by inserting video data directly into the payload of certain packets.

4. The Typical Video over IP Application

4.1 Typical Application Structure

In the original implementation of the Video over IP Application, the interaction of the modules described in the previous section in order to transmit one frame of video data is as follows. First, the orinoco network card driver initializes the wireless card. This is done by passing the initialization commands directly through the XILINX to the appropriate device via the Memory Input/Output (IO) communication protocol. Refer to [10] for a description of this protocol. Second, the user space application calls the open device method in the camera driver to initialize the camera parameters. After initialization, the camera starts filling the FIFO queue.

The user space application then calls the read method of the camera driver, passing it a blank buffer to be filled with exactly one frame of data. Once the FIFO has reached a High Water Mark (HWM), the FPGA sets a FIFO interrupt bit, which is received by the camera driver. The HWM value used in this application was 256 bytes. In response to this interrupt, the camera driver reads the FIFO data available, storing it into the buffer allocated for the frame data. The camera driver the waits for the next FIFO interrupt, and appends the new data to the frame buffer. This process repeats until a Vertical Blank (VBlank) interrupt is received by the camera driver, indicating the end of the frame. Once the VBlank interrupt has been received, the camera driver returns from the read call with the frame data in the buffer.

Upon reception of a complete buffer, the user space application splits the buffer into smaller chunks and sends them to the network card driver for transmission. Refer to figure 4 below for a flowchart representation of the original camera frame buffer read operation functionality.

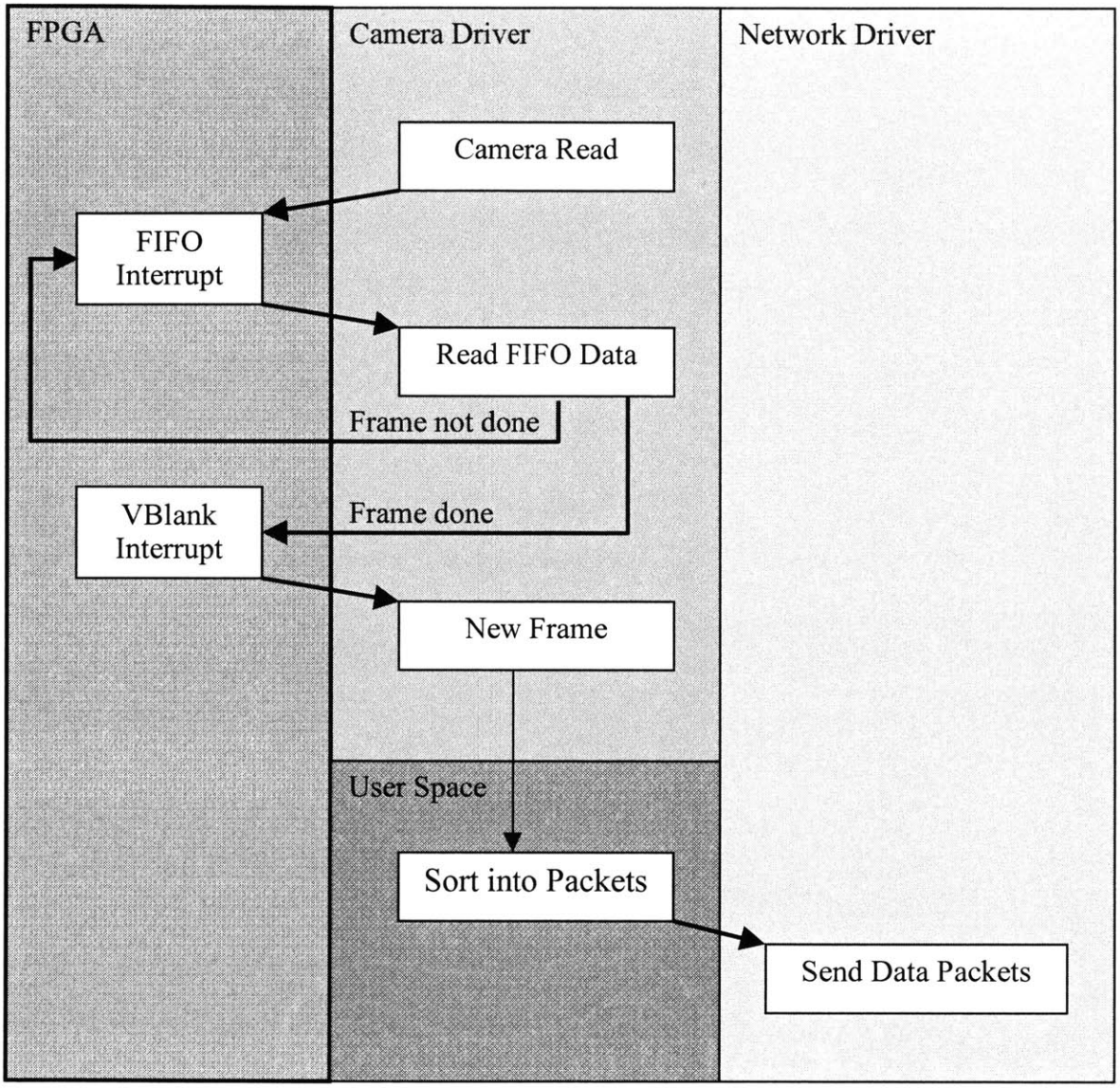

Figure 4: Flowchart of the Original Camera Read Operation. The camera driver waits for FIFO interrupts, appending data to the frame buffer each time a FIFO interrupt is received. Upon a completion of a frame, the user space applications splits the data into packets and these packets are sent to the network card for transmission.

As can be seen from the above description and flowchart, the camera driver and the network driver play the central role in data manipulation. Note that both these processors rely on the iPaq CPU for resource allocation. The FPGA plays only a minor role in this scheme, simply storing the data and setting interrupt flags. Thus, most of the video data processing is serviced by the iPaq CPU.

4.2 Video Data Path of the Typical Application

The path of the video data in the typical implementation is as follows. After leaving the camera, the FPGA decimates the data and stores it in the FIFO queue. Once enough data has accumulated in the FIFO, the FPGA generates a FIFO interrupt. The camera driver services this interrupt, and reads the data from the FIFO queue into the iPaq memory via the camera driver. The data is stored in an allocated buffer until a full frame of data has accumulated. When the camera driver returns from the read call, the user space program (in this case, camtest) manipulates the frame data and sorts it into packets. These packets are then handed over to the network card driver which writes them to the network card for transmission. The detailed path of the video data through the system established by this scheme can be seen by the thick black line in Figure 5.

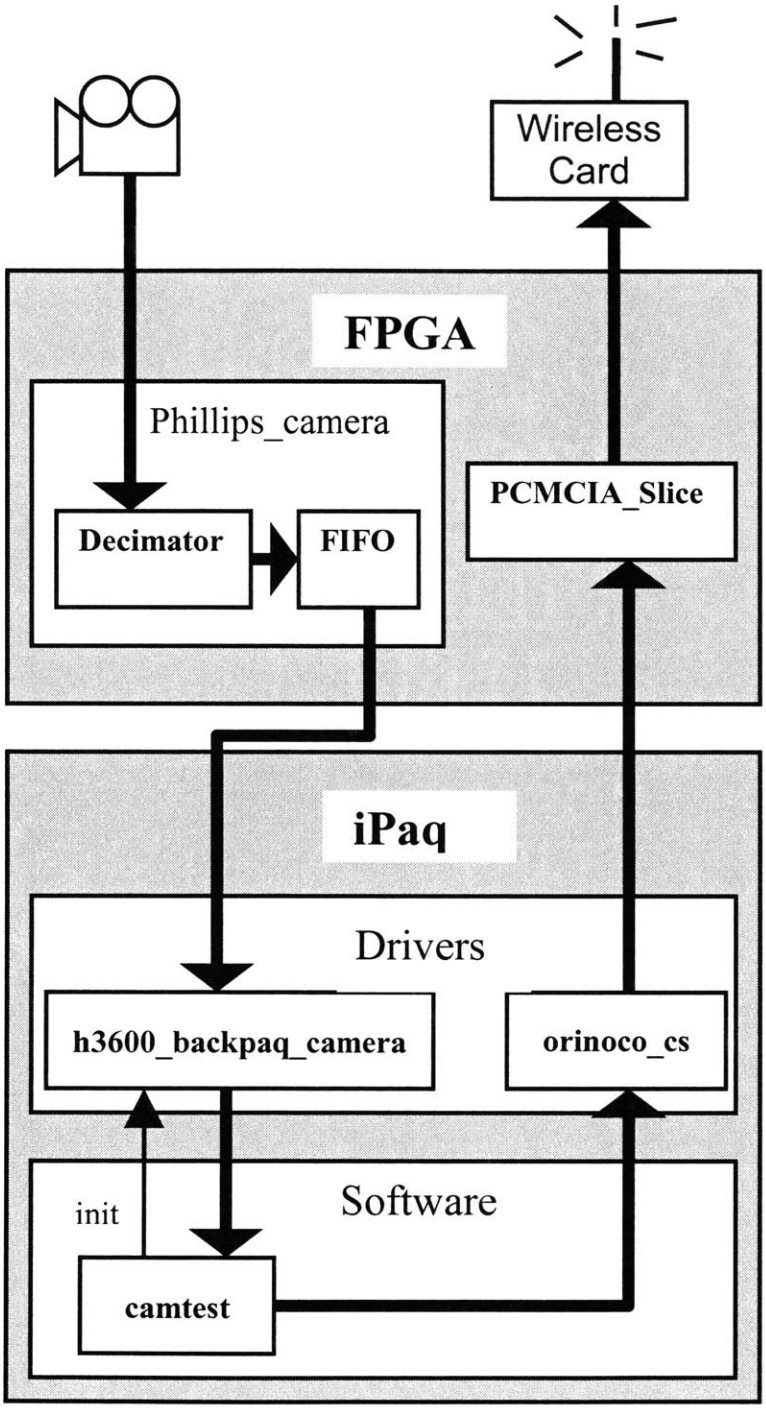

Figure 5: The Detailed Path of the Video Data in the Original Configuration. The data is generated by the camera, gets decimated, and then gets stored into a FIFO queue. The camera driver reads from this queue. Once a full frame has been read, it returns the data buffer to the software application. The software application transmits the data to the network card via the orinoco network card driver.

4.3 Disadvantages of the Typical Approach

Looking at the diagram in Figure 5, we see that the image data makes a very long route from its originating source, the camera, to its destination, the network card. The worst aspect of this approach is the requirement that all the data be directly manipulated by drivers and software. Every time the FIFO reaches a HWM, the camera driver faithfully reads all the data currently available in the FIFO from the FPGA and places it in the frame buffer provided by the read call. Every time a frame buffer is returned by the read call, the network driver splits it into packets, and feeds it byte by byte to the network card. These processes are run on CPU in the iPaq, thus constantly occupying the processor resources while video transmission is in progress.

Most video applications rely heavily on video compression schemes, thus placing additional computational strains on the CPU. With the typical approach, any compression algorithm must be implemented in software and applied to every frame buffer as it is returned by the camera driver read call.

Monopolizing all the CPU resources with video data manipulation, compression, and transmission prevents the user from using other applications on the handheld. In addition, since the resources on handhelds are generally limited, it is possible that the CPU will not be powerful enough to sustain the video application operating at the desirable frame rate and resolution.

To ameliorate the problems described above, a different structure for a Video over IP application is presented in the next section.

5. The Enhanced Video over IP Application

5.1 Enhanced Application Structure

In the enhanced Video over IP application, the interaction of the modules described in the section 3 in order to transmit one frame of video data is as follows. The orinoco network card driver initializes the wireless card exactly as described in section 4.1. The user space application then calls the open device method in the camera driver to initialize the camera parameters. Recall that after initialization, the camera starts filling the FIFO queue.

Next, the user space application calls the read method of the camera driver. Once the FIFO has reached a HWM, the FPGA sets a FIFO interrupt bit, which is received by the camera driver. The HWM value used in this application was 960 bytes to maximize the packet load to header ratio. See section 6.4 for a discussion on why this parameter was changed from the original implementation value of 256 bytes. In response to this interrupt, the camera driver schedules a packet to be sent by calling the send method of the orinoco card driver. The packet is recognized by the card driver as a camera packet due to the preset destination port field in the packet header. This port is specially allocated for the video application.

Upon reception of the special packet, the modified orinoco transmit function communicates with the FPGA to establish how much data is currently available in the FIFO. It then modifies the packet header length field and header checksum appropriately, and proceeds to write the header to the network card socket. After having written the packet header, the orinoco driver writes a word to a specially designated register in the

FPGA indicating the data length set in the packet header. The FPGA responds by reading this amount of data from the FIFO queue and immediately writing it to the network card as the packet data load. For frame synchronization purposes, two additional bytes are written into the packet which corresponds to the frame number and packet number. This is used to sort the incoming data packets into frame buffers on the receiving end. The FPGA then completes the packet transmission by completing protocol as described in the Appendix.

The camera driver waits for the next FIFO interrupt and the steps described in the previous paragraph repeat. This process continues until a VBlank interrupt is received by the camera driver, thus indicating the end of the frame. The VBlank simply causes the FPGA and the camera driver to update its frame count parameters, but does not return control over to the user space program. Instead, it proceeds directly to sending the next frame. The camera driver returns from the read call only upon an exit from the program. Refer to figure 6 below for a flowchart representation of the enhanced camera frame buffer read operation functionality.

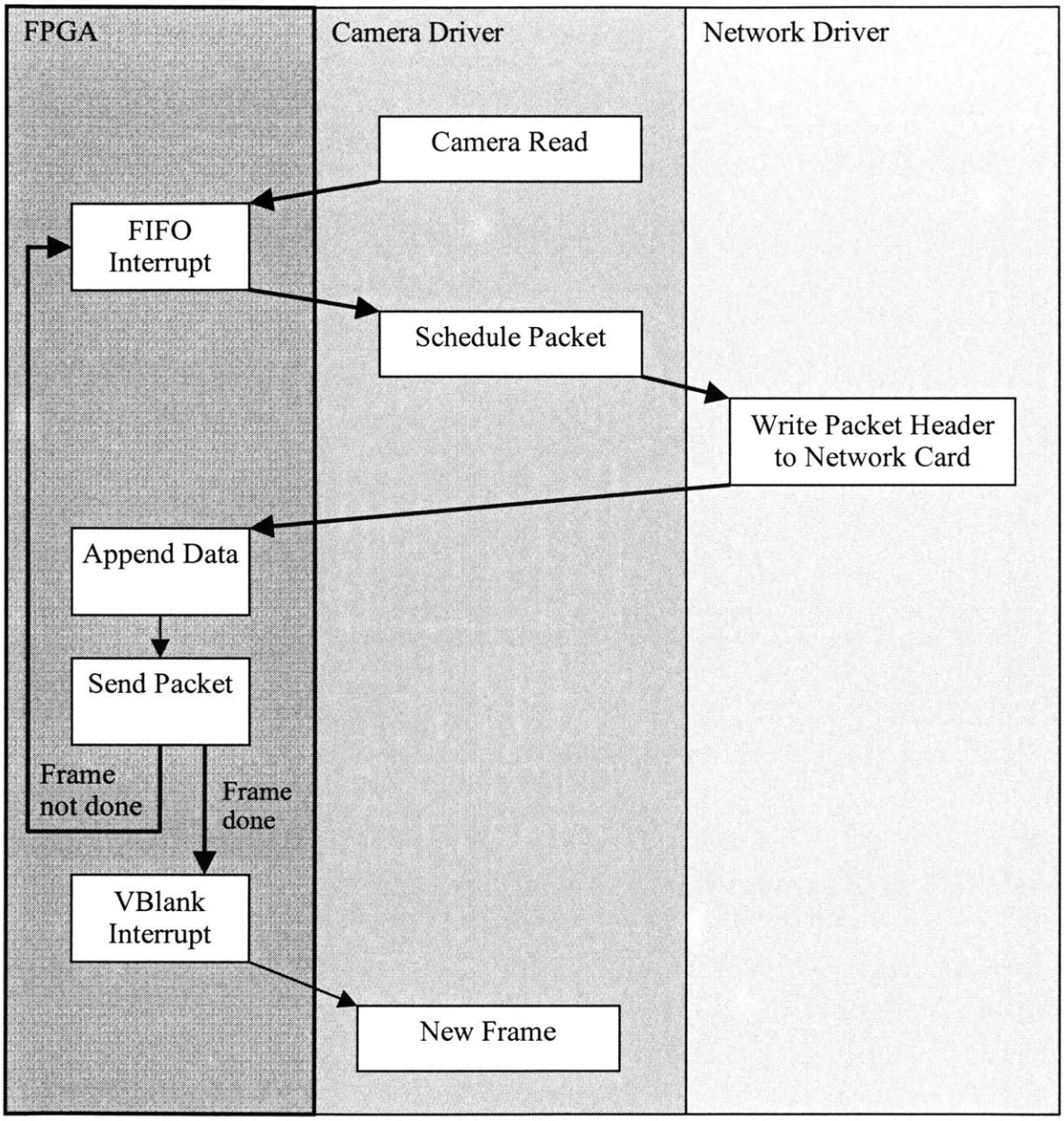

Figure 6: Flowchart of the Enhanced Camera Read Operation. Upon reception of a FIFO interrupt, the camera driver sends a command to the network driver to send a header to the wireless card. Upon completing the forwarding of this header, the FPGA then proceeds to add the data payload to the packet and internally complete the send protocol. The VBlank interrupt informs the camera driver that a complete frame has been sent.

As can be seen from the above description and flowchart, the camera driver and the network driver only serve the controlling role in the scheme. The data payload never exits the FPGA, and therefore does not require any resources allocated by the iPaq CPU. Instead, the FPGA directly handles all video data manipulation and simply waits for packet size and packet header information from the CPU.

5.2 Video Data Path of the Enhanced Application

The path of the video data in the enhanced implementation is as follows. After leaving the camera, the FPGA decimates the data and stores it in the FIFO queue. Once enough data has accumulated in the FIFO, the FPGA generates a FIFO interrupt. The camera driver services this interrupt by scheduling a send packet task in the network card driver, which in turn sends a special packet header to the wireless card. When this is completed, the PCMCIA_Slice module in the FPGA takes the data directly from the FIFO queue and writes it to the network card as the packet payload. The data thus travels directly from the camera, through the FPGA, and into the network card. The detailed path of the video data through the system established by this scheme can be seen by the thick black line in Figure 7.

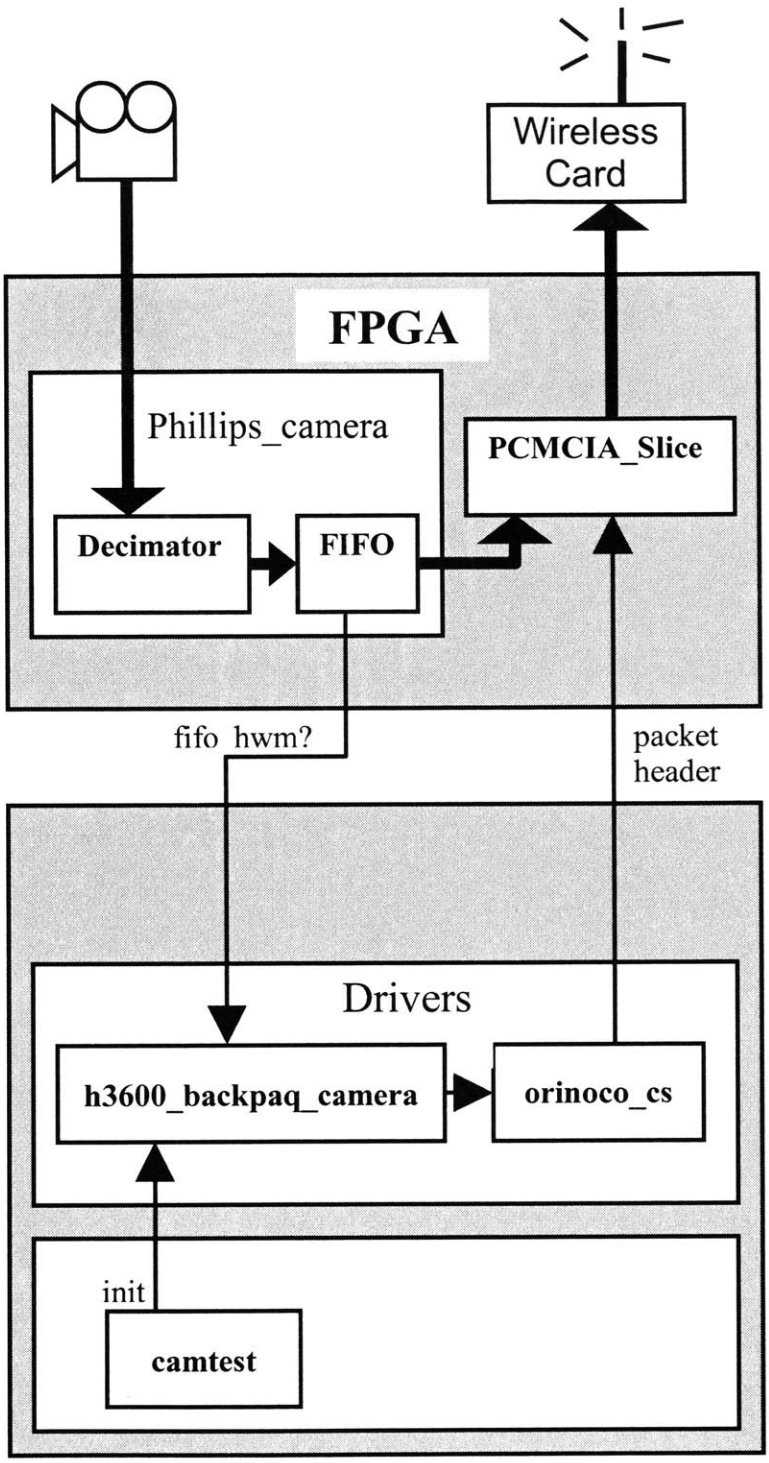

Figure 7: The Detailed Path of the Video Data in the Enhanced Configuration. The data is generated by the camera, gets decimated, and then gets stored into a FIFO queue. The camera driver then instructs the network driver to send a packet header. Upon reception of this header, the PCMCIA_slice module in the FPGA fills in the camera data into the packet payload directly from the FIFO queue.

5.3 Implementation

5.3.1 The Camera Driver Operation

The modified version of the H3600 Phillips Camera Driver operates in the following manner. During the call to the camera open command, in addition to initializing the camera parameters, the driver also opens a UDP socket which will be needed later for data transmission. There were no further modifications to the camera open method.

The new camera read method acts as follows. The driver remains idle until a FIFO or a VBlank interrupt is received. A FIFO interrupt indicates that a preset amount of data has accumulated in the FIFO and needs to be cleared. A VBlank interrupt indicates that the end of a frame has been reached. Upon receiving a VBlank interrupt, the camera driver simply updates the number of the frame to keep track of which image is being processed, and sets the value of bytes read in the current frame to zero. If the interrupt was caused by the FIFO queue, we first set then set the HowManyLinesRead register in the FPGA to zero; this is used for synchronization as will be seen in the next paragraph. A sendHeader task is then scheduled by the kernel which will actually take care of sending the packet header to the network card. An approach requiring less synchronization would involve sending the packet header directly in the interrupt handler; however this was not feasible because the kernel in interrupt mode does not allow socket communication. Although using the scheduler adds complexity to the code, it is actually beneficial because it does not lock the CPU in interrupt mode until the transmission of the packet has completed.

Once inside the sendHeader task, the sock_sendmsg function is called to send the packet header to the network card. The destination port of the packet header is set to a predefined port used exclusively by this application. We then wait until the HowManyLinesRead register is no longer zero; indicating that the orinoco driver has modified this parameter since we first set it to zero during the interrupt handler routine. As indicated in the following section, the orinoco driver modifying the HowManyLinesRead register indicates that the packet header has been successfully written to the network card and the clearing of the FIFO queue has begun. We then update the bytes_read variable to keep track of the frame progress. Refer to Figure 8 for a flowchart description of the camera read operation.

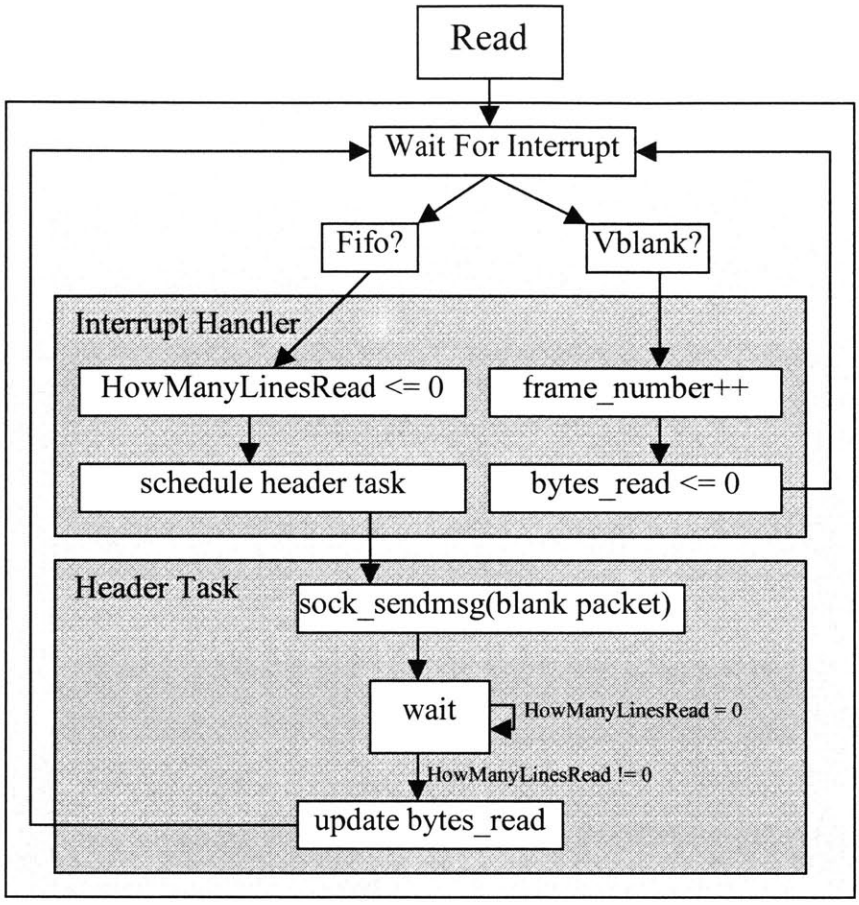

Figure 8: The Camera Open and Read Operation Flowchart. The open method initialized the camera and opens a socket for later use. The read method increments the frame number in response to a VBlank interrupt and sends a packet header in response to a FIFO interrupt.

5.3.2 The UDP Network Protocol

The packets are sent using the UDP protocol which requires less overhead bandwidth then the TCP protocol by avoiding delivery verification notifications. In order to implement hardware to communicate with a network card, it was first necessary to deduce the communication protocol used by the card. Using the Linux platform gives an advantage in accomplishing this because the network card driver code is readily available. As mentioned earlier, the driver used to deduce the protocol was the Orinoco

Network Card Driver. In addition to using the diver source code, a logic analyzer was used to observe the packet being handed over to the card by the driver, and a network analyzing program called Ethereal was used to grab the packets sent to view the actual data sent by the card. Refer to the Appendix for the description of the deduced protocol.

Several fields in the packet header needed to be modified by the new orinoco_xmit function. These fields include the UDP checksum, total length, and the header checksum. In addition, extra bytes were added at the beginning of the packet to indicate the packet number and the frame number. Since the data is appended to the packet after the header has already been written to the card, the data checksum of the UDP packets was set to zero, indicating that this checksum is not used for data verification purposes. Please refer to [13] for a description of the UDP protocol. The IP and UDP headers are also shown below in figures 9 and 10 below

.

<table>
<tr><td>4-bit version</td><td>4-bit header length</td><td>8-bit type of service</td><td colspan="2">16-bit total length</td></tr>
<tr><td colspan="3">16-bit identification</td><td>3-bit flags</td><td>13-bit fragment offset</td></tr>
<tr><td colspan="2">8-bit time to live</td><td>8-bit protocol</td><td colspan="2">16-bit header checksum</td></tr>
<tr><td colspan="5">32-bit source IP address</td></tr>
<tr><td colspan="5">32-bit destination IP address</td></tr>
<tr><td colspan="5">options (if any)</td></tr>
<tr><td colspan="5">Data</td></tr>
</table>

Figure 9: IP Header. The only field in the IP header modified by the Orinoco driver was the total length and the header checksum.

<table>
<tr><td>16-bit source port number</td><td>16-bit destination port number</td></tr>
<tr><td>16-bit UDP length</td><td>16-bit UDP checksum</td></tr>
<tr><td colspan="2">Data</td></tr>
</table>

Figure 10: UDP Header. The fields modified in the UDP header by the Orinoco driver were the UDP length and the UDP checksum.

5.3.3 The Network Driver Transmit Operation

In order to implement a special camera packet transmission routine, the orinoco_xmit method used in the Orinoco Network Card Driver was modified as follows. Before writing the received packet to the network card, the modified routine first checks weather the destination port corresponds to that specially allocated to the camera application. If the port is different, the packet transmission proceeds as usual, and the unmodified packet is written to the card. Otherwise, the camera packet is handled in the following manner.

In order to set several values in the packet header, we first need to decide how big to make the packet. In order to minimize the strain on the network card, it is ideal to send packets as large as possible provided that they do not exceed the network limit of 1500 bytes. To decide on the size of the packet, it is therefore sufficient to simply ask the FIFO how much data is currently available (data_avail) and set the packet length to the minimum of this value and the network limit. The frame_id is also read from the FPGA so that this value can be written into the packet header. The packet counter is stored directly in the special camera transmit function. Using the dava_avail and the frame_id values, several fields in the packet header are updated as indicated in the previous section.

The header buffer is then written to the network card. In a normal packet transmission operation, the packet data gets written to the card following the header. However, in the modified approach, the driver simply writes a command to the FPGA indicating that the packet header has been written and the network card is now ready to receive the image data from the FIFO to insert into the packet payload. This command is

sent by setting the HowManyLinesRead register in the FPGA to the appropriate value as was earlier defined in the packet header structure.

Remember from section 5.3.1 that this same register is set to zero by the camera driver when a FIFO interrupt occurs. When giving the command to send a camera packet, the camera driver then waits until the HowManyLinesRead register is reset to a non-zero value by the network driver in order to deduce when the clearing of the FIFO has actually began.

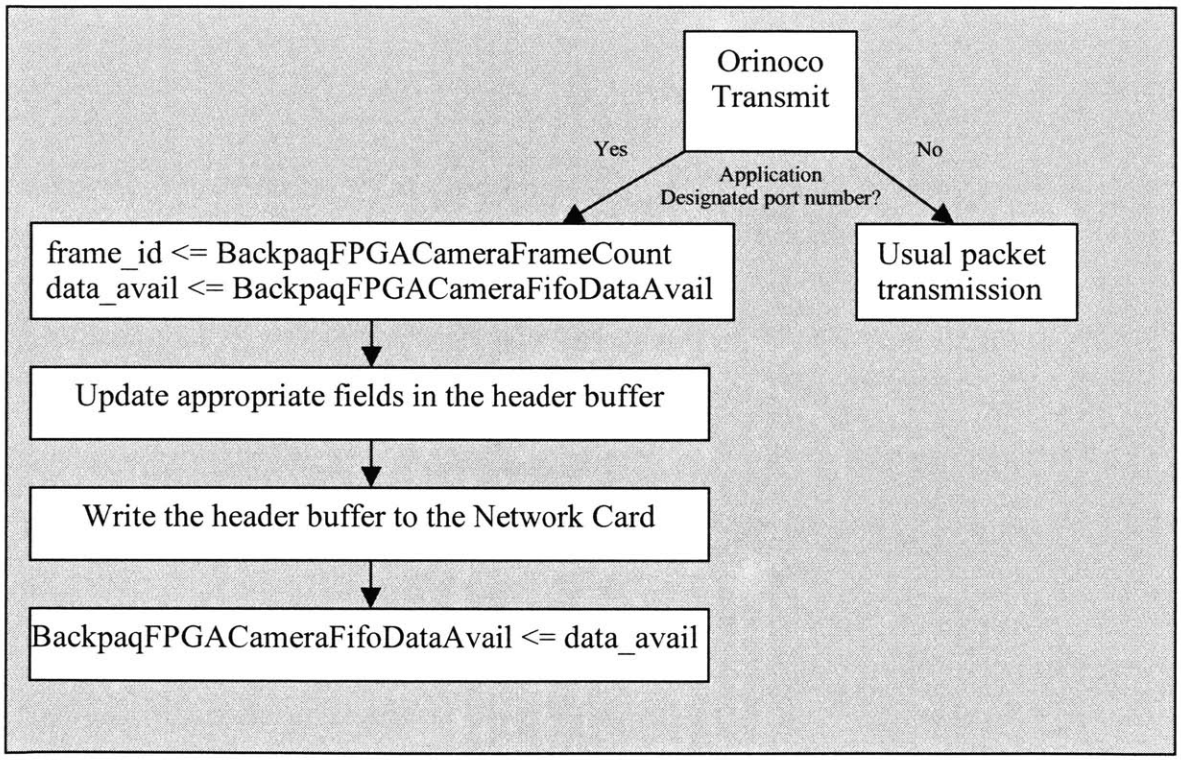

Figure 11: The Transmit Camera Packet Flowchart. When requested to transmit a special video data packet, the orinoco transmit driver reads the frame_id and the data_available values from the FPGA, updates the packet header according to this information read, writes the modified header to the network card, and sends the command to the FPGA to take over the completion of the packet.

5.3.4 The FPGA State Machines

The primary modification to the FPGA modules in order to implement the camera transmission protocol was the addition of a state machine implementing the final steps of the protocol described in the Appendix. Recall from the previous section that the FPGA takes over the action of writing the data and the transmission commands to the network card as soon as the packet transfer method in the modified orinoco driver sends the proceed command to the FPGA. This command is the last step in the orinoco transmit procedure – the writing of data_avail to the FPGA BackpaqFPGACameraFifoDataAvail register. Until this command is received, the PCMCIA_Slice module in the FPGA simply forwards the read/write requests between the camera and the network card to allow for regular communication.

The initial step necessary before passing over control of the network card to the FPGA is to disable all FIFO and VBlank interrupts to ensure that the camera driver will not fire twice on the same interrupt. Once this is ensured, the FPGA is free to start clearing the FIFO and writing the data to the network card. To implement this, the state machine simply waits for the FIFO module to signal that data is ready to be read with its prefetchAvailable signal, reads this data, and writes it to the network card. This read/write loop repeats until the amount of data written corresponds to the number of bytes to write as requested by the orinoco transmit command.

The state machine then reads a status register in the network card waiting for the value read to indicate that the data has been written to the card buffer as requested. The FIFO then writes initialization commands to three network card special parameter

registers, and the initialization register. It then reads a support and a status register, and finally writes the evacuation command to the card.

Upon completion of the packet write command the state machine simply returns to its normal operation state where it again forwards the read/write requests as normal between the iPaq and the network card.

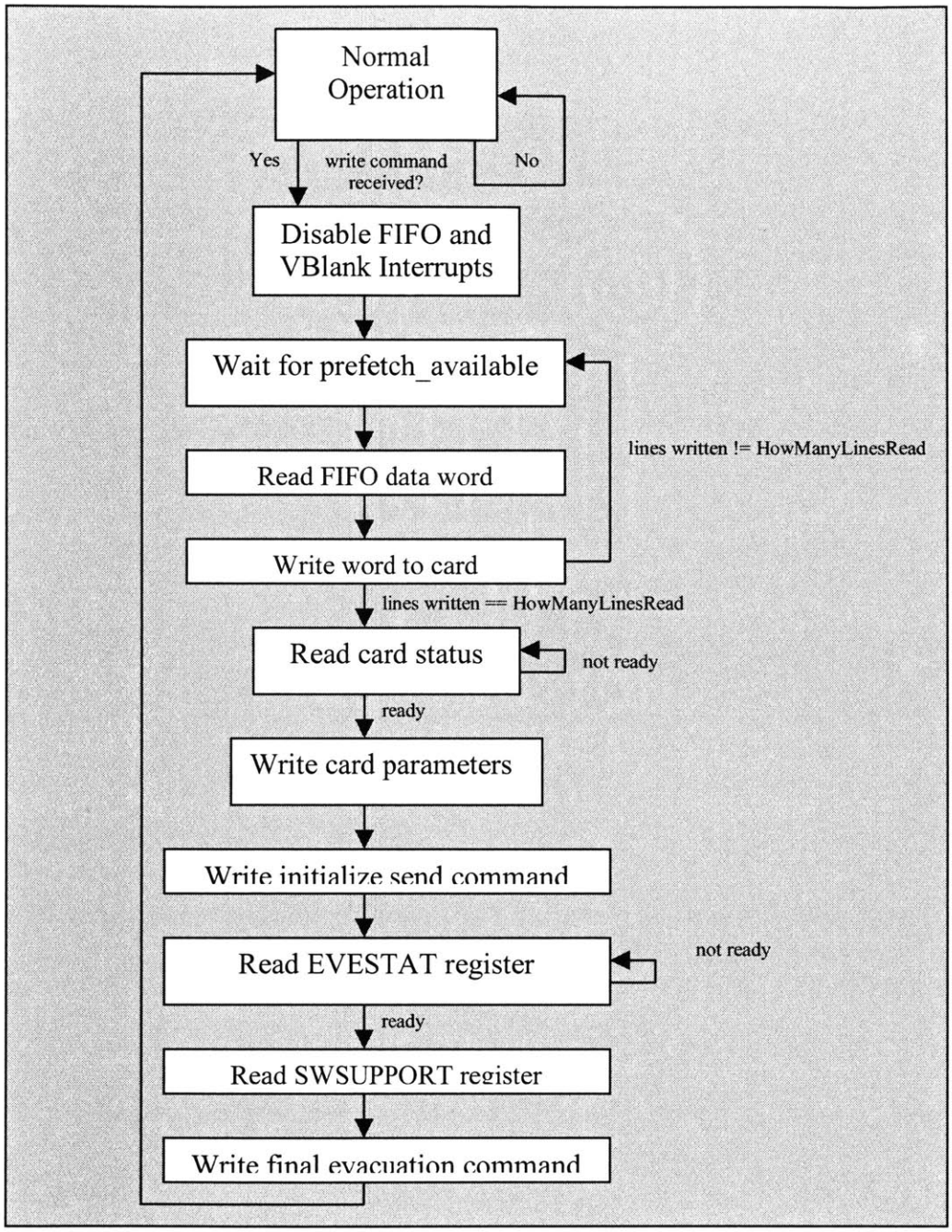

Figure 12: The Write Packet State Machine. In normal operation, the read/write requests are simply forwarded between the PCMCIA socket and the network card. Once a command has been received from the orinoco driver to complete the camera packet, the packet transmission state machine proceeds as shown above.

5.4 Design Decisions and Tradeoffs

5.4.1 Interdependencies and Synchronization

One of the most challenging aspects of designing this application was the implementation of a synchronization scheme between all the modules in the design. The algorithm of the application as designed in section 5.1 needs to occur in precisely the described order. This is challenging because the camera driver and the network driver run on separate threads and the processes on the FPGA run asynchronously of both of these processes.

The synchronization was achieved in the following manner. The FPGA write packet state machine remains idle in the initial stages of the process. The camera driver receives a VBlank interrupt to acknowledge a new frame, and then waits for a FIFO interrupt. The driver then schedules the send header procedure. As mentioned earlier, this scheduling needs to occur because this procedure can not be completed in interrupt mode. In order to ensure that no other interrupts get fired until the task has actually completed, the interrupt routing writes a zero to the HowManyLinesRead register in the FPGA before exiting and scheduling the socket send procedure.

When the send header task actually starts executing and a call to the send_msg is made from the camera driver, the network driver responds by first seeing how much data is currently available in the FPGA, and then writing the packet header according to this information. The reason the amount of data is established here is because this is the last possible place we can do this before needing to use this data directly in the packet header modification. Because the processes in the drivers run as scheduled tasks, there is no guarantee that they will be executed immediately following one another. There could be a

potentially large delay between a call to call to the send_msg function from the camera driver and the time to when the network driver actually starts executing this command. Choosing to read the data_available value in the network driver ensures that we read as much of the FIFO data as possible to clear it in the most efficient manner.

After writing the header to the card, the network driver sets the HowManyLinesRead register in the FPGA to the number corresponding to the amount of data which needs to be written to the FPGA. At this point, the camera driver is still in the send header procedure, and it is continuously looping on the HowManyLinesRead register, waiting for its value to be set by orinoco driver. The setting of this register corresponds to the orinoco driver commanding the FPGA to start clearing the FIFO. Once this register has been set, the camera driver knows that the FPGA has started passing the image data to the network card. It thus updates the number of bytes read, exits the send header function, and waits for the next interrupt.

5.4.2 Hardware Tasks vs. Software Tasks

Having the capability to control the camera and the network card in hardware through the FPGA or in software through drivers introduces many possible schemes for designing applications to communicate with these devices. Deciding to implement a certain task in software is often the simplest way to get it done. However, implementing it in hardware gives an innate speed advantage and also does not monopolize CPU resources.

The general approach to sorting tasks in this application into software or hardware task has been as follows. If a task serves a general setup or control purpose, the CPU was

left to handle it. If the task involves large amounts of data manipulation or storage, it was handled in the FPGA.

The camera driver in the enhanced application is still in charge of initializing the camera during the open method and monitoring frame progress during the read method. At any given point, the camera driver has knowledge of how much data has been sent, how much remains, and of a FIFO overrun or another error has occurred with the camera operation. The camera driver uses this information to start and stop the scheduling of the frame transmission calls as needed, and reassign the camera operation parameters as needed.

The network driver was in charge of modifying header parameters, sending the packet header to the wireless card, and instructing the FPGA to start appending the payload. A different approach could have used the camera driver to simply write the FPGA a command instructing it to send the entire packet, header and data, directly to the network card. This approach would have completely bypassed the network driver and effaced the need for the send packet scheduler in the camera driver. However, this approach would prevent other applications from using the network card since it would expect an unconditional right to write to the card at any time.

Giving the network card driver control of the packet header parameters can also allow it to handle network congestion by modify certain fields in the header according to the network needs. The approach chosen in this application does not cause significant overhead processing by the iPaq CPU because packet headers are very short relative to the data payload. As mentioned earlier, the data payload is also maximized to increase the video data to header data ratio.

5.5 Advantages of the Enhanced Approach

Implementing a Video over IP application in the manner presented in this thesis solves many problems imposed by the application in its original configuration. The primary achievement of the new setup is the ability to demonstrate higher quality performance of the video application while freeing the CPU of all computation involving data manipulation. By comparing the transmission in Figure 5 (the original configuration) to that in Figure 7 (the enhanced configuration) we clearly see a dramatic reduction in the path length the data travels as it is generated by the camera and sent out the network card. Refer to the section 6 for actual performance calculations.

6. Performance

6.1 CPU Usage

As described in the sections above, the enhanced application requires significantly less CPU usage then the original application. When running at 9 frames per second, the original configuration consumes 77% of the available CPU resources, while the enhanced configuration consumes only 24%. This corresponds to a 2/3 CPU load reduction. When running at a low speed of 1 frame per second, the CPU usage of the enhanced approach is only marginally better than that of the original approach. This is because the added use of the scheduler and inter-driver interactions in the enhanced application adds additional overhead on the CPU, thus balancing the savings from re-routing the data. Note that as the frame rate increases, the slope of the CPU requirements of the enhanced application is much smaller then that of the original application. The load on the CPU in the original

configuration is proportional to the frame rate, since the CPU directly handles every byte read from the camera and transmitted to the network. However, in the enhanced configuration, we see the slope decreasing with increasing frame rate. This is because as we increase the rate, the packets approach their maximum size, but the rate at which the packets are sent remains constant. The CPU usage is not affected by the size of the packets since the data is inserted into them directly by the XILINX.

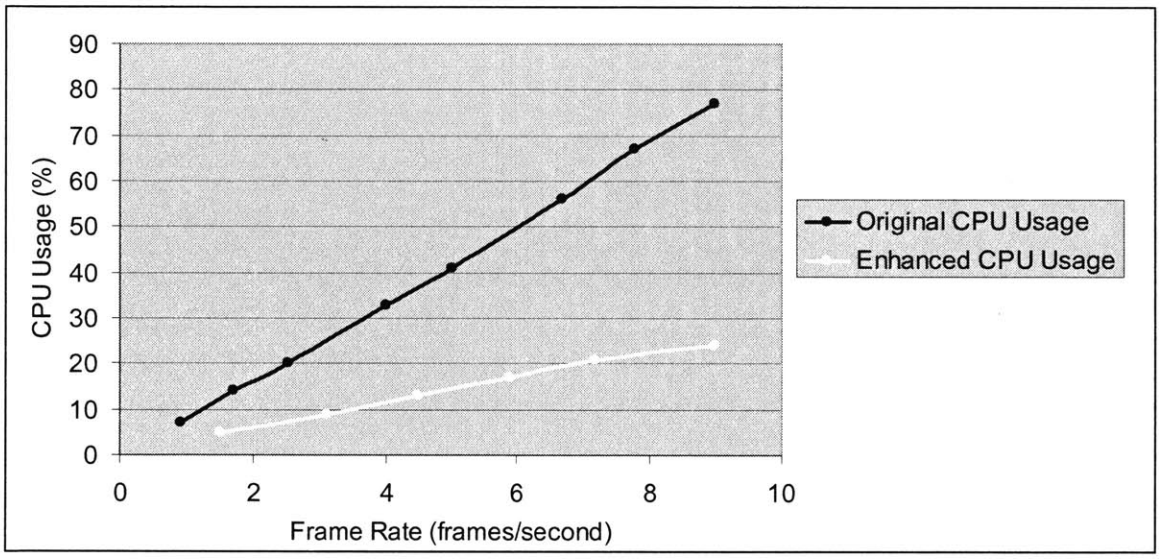

Figure 13: The Original and Enhanced Application CPU Usage. Running at 9 frames per second, the enhanced application reduces the load on the CPU to 32% of its original value.

6.2 FPGA Usage

The usage of the XILINX resources for the enhanced application is listed in Figure 14 below. Only 25% of the available RAM was used to store the data from the camera. By increasing the FIFO, up to $1/10^{th}$ of a 76800 byte frame can be stored in the gate array. This could help with the buffering problem described later in section 6.4

Number of External GCLKIOBs	1 out of 4	25%
Number of External IOBs	256 out of 312	82%
Number of LOCed External IOBs	256 out of 256	100%
Numberof BLOCKRAMs	8 out of 32	25%
Number of SLICEs	1151 out of 3072	37%
Number of DLLs	1 out of 8	12%
Number of GCLKs	1 out of 4	25%
Number of TBUFs	129 out of 3200	4%

Figure 14: FPGA Resources Usage. Note that only 25% of the available RAM is used for the FIFO implementation.

6.3 Frame Rate

The uncompressed frame rate achieved by both applications given the network limitations was 9 frames per second. The frames were 320x240 bytes, which corresponds to a data rate of 5.5Mbps. Both programs utilized the available network bandwidth to its full capacity. Although no apparent frame rate increase was seen in the enhanced application, this was simply due to the fact that the iPaq CPU is powerful enough to manage data at such a high rate. A weaker CPU would not be capable of operating at this rate in the original configuration, but would have no problems in the enhanced configuration. In addition, if compression was added to the scheme and the frame rate

was raised to maintain a throughput of 5.5Mbps, the StrongARM CPU would not be able to keep up with the load since we saw in section 6.1 that, even without compression, the CPU usage in the enhanced application was at 77%. Section 6.4 proves the above claims by noting that in the original application frame rate of 9 slows down to 6.5 when another application is sharing the CPU. The enhanced application continues to operate at 9 frames per second and at the same time shares twice as much CPU power with the other application.

6.4 Response to Disturbances

Two disturbances analyzed in this section are the response to application which require CPU usage, and the response to applications which require network card usage. Due to reduced usage of the CPU, the enhanced application performs better with applications which share the CPU. However, since no buffering of data occurs in the enhanced application, it performs worse when run in parallel with applications requiring network bandwidth.

The application used to model processor usage was simply a loop which increments a counter. When running in parallel with this program, the original application was not able to operate at above 6.5 frames per second, and allowed 7 million increments to occur per second. In contrast, the enhanced program was able to operate at 9 frames per second and allowed 13 million increments per second to take place.

Unfortunately, the enhanced application is far more sensitive to programs sharing network bandwidth than the original application. The original program reads a camera

frame, stores this data in an internal buffer, and only sends it when the network card is available. The enhanced program attempts to read the data from the camera and send it to the network card at the same time. Therefore, if the card is not available for a long enough period of time, the FIFO overflows thus corrupting the entire frame. One way to deal with this problem is by replacing bytes of the current frame with bytes of the previous frame for those frames where not many bytes have been lost by FIFO overflows. Figure 14 below shows the performance of the original and the enhanced application while running in parallel with a program attempting to send other network packets. Note that the frame rate is measured in terms of the complete number of frames transmitted, i.e. frames where zero FIFO overflows have occurred.

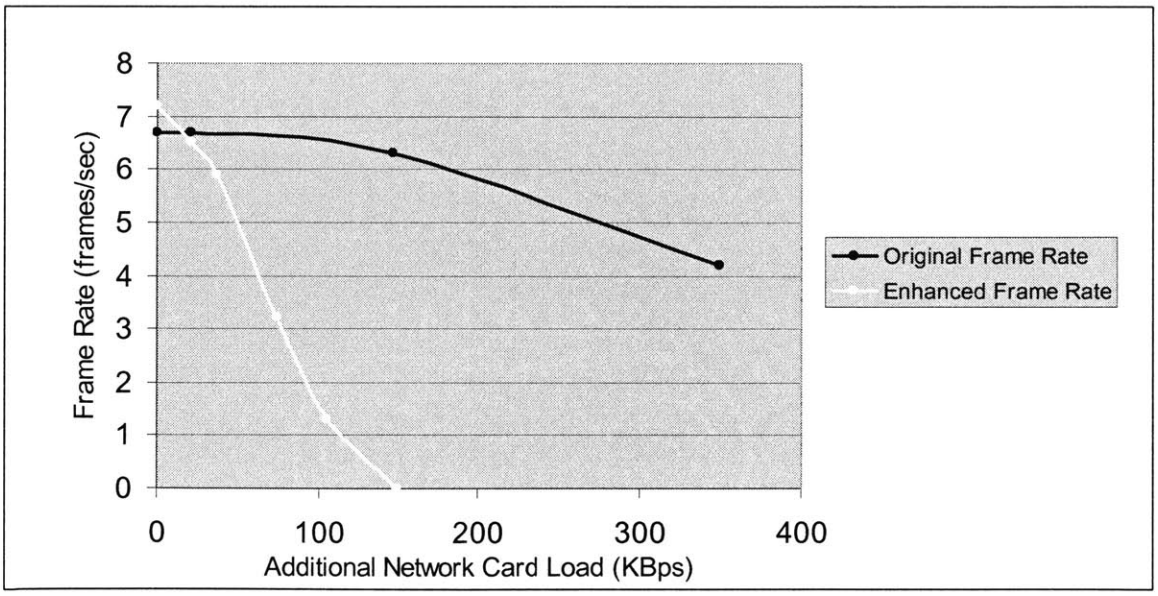

Figure 15: Response to Additional Network Load. Because the frame rate is measured in complete frames per second, the enhanced application is easily deterred by other applications sharing the network bandwidth.

6.5 Optimization

In order to maximize the transmission rate, the packets have been set to the maximum value allowed by the network used, or up to 1500 bytes whenever enough data was available in the FIFO. The FIFO interrupt is set to fire when 960 bytes have accumulated in the FIFO, with the expectation that the FIFO will contain close to 1500 bytes when the network transmit function actually determines the packet size. The packet header is only 12 bytes, and is therefore insignificant in size to the rest of the data load. Maximizing the packet size has two advantages. First, it ensures that the CPU is used as little as possible, since the number of FIFO interrupts and the number of packet headers written to the network card is minimized. Second, the network used appeared to allow the highest bandwidth when the packet size was maximized.

Much of the FPGA time is spent writing the video data to the network card. Each write cycle in the original application takes 1300ns. In order to minimize this delay, the write cycles of the data to the FPGA were made as short as the card hardware allowed without introducing errors. For the Cabletron RoamAbout 802.11 DS card used, this limit was determined to be at 800ns. This allowed a 1400 byte packet to be written to the card in 30% less time. This feature did not improve the performance of the enhanced application because the limiting factor was the transmission of data from the card to the network, not the camera to the card. However, given a higher network capacity, being able to write to the card faster would correspond to being able to transmit packets at a higher rate.

6.6 Limitations

The primary constraints in the enhanced implementation were the maximum packet size imposed by the network and the network capacity and noise. Because the CPU usage was at 24% running at even the fastest frame rate, the CPU power was no longer a limiting factor.

At 9 frames per second, the packet size is 1000 - 1400 bytes and the packets are sent every 1.8ms. This corresponds to a data rate of 5.5Mbps. Following, is an image taken from the logic analyzer showing a transmission of three packets from the XILINX to the network card.

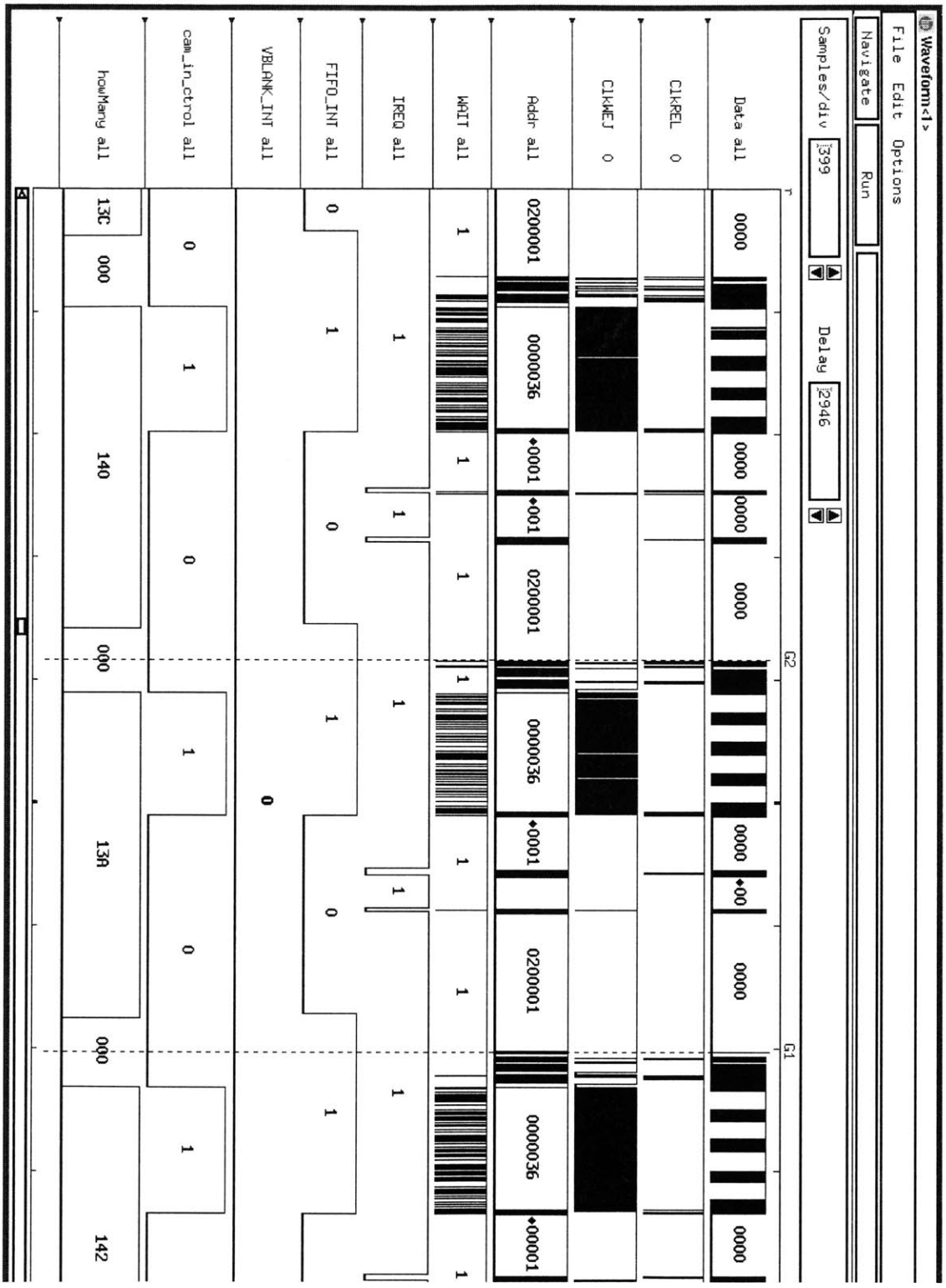

Figure 16: Writing Packets to Network Card. This image was generated by the logic analyzer as a packet was written from the XILINX to the network card. The time between the markers G1 and G2, or the time between subsequent packet transmissions is 1.8us. With 1200 byte packets, this gives us a rate of 5.5Mbps.

After the packet is written to the card, the orinoco driver waits for an interrupt from the card indicating that the packet has been sent and the buffer is now free to accept the next packet data. In the above diagram, this is the second IREQ sent by the card after a packet has been written to it. The time from which the packet data starts being written to the card to the time when the card sends this interrupt is only 1ms. Thus, at this rate, if we are sending 1300 byte packets at 1000 packets per second, our throughput will in fact be 10Mbps, which at 76800 byte frames corresponds to almost 17 frames per second. However, the noise and other limitations of the wireless network did not appear to allow transmission rates of greater then 5.5Mbs for large amounts of data. Refer to the figure below for the limitations imposed by the network.

Transmission Rate (Mbps)	Data Transmitted Until Rate Reduction (KBps)
9.2	5.4
8.4	192
7.2	384
6.3	922
5.5	No Reduction

Figure 17: Network Limitations. Although the instantaneous capacity of the network approached 10Mbps, the network did not appear to support a constant rate of more then 5.5Mbps.

7. Evaluation

The Video over IP application implemented and analyzed in this thesis demonstrates many advantages of reconfigurable computing applications. The frame rate achieved by the enhanced application was limited by the network bandwidth as opposed

to the available CPU resources. The application did not require new hardware, and utilized only a small section of the XILINX gate array situated in the iPaq backpaq. It is thus still small and portable enough to be used as a handheld device. The usage of the CPU was reduced by 69% at the highest frame rate, which corresponds to significantly smaller power consumption. Actual power measurements of the CPU and the XLINIX were not taken. The overall design is very portable and easy to change, as the VHDL in the gate array can be modified almost as fast as software. Thus any compression scheme or another protocol implementation can be done directly in the gate array.

8. Further Improvements

A large improvement in performance can be achieved by using a compression scheme. There are many possible transform, quantization and coding schemes for moving images. Adding a VHDL module implementing a compression algorithm of the image data before sending it over the network will enable the application to further improve the image resolution and frame rate.

Adding error correction to frames with lack only a small amount of data will also improve performance. If a frame is missing less then 10% of the data, pixels can be reused from the previous frame to fill in this gap. This will often be unnoticeable at high frame rates and will dramatically reduced dropped frames.

9. Conclusion

Researchers in the past have tried various attempts to solve the problem of video communication over IP. Attempts have been made to modify different standards for audio and video encoding [1] [3], increasing throughput of servers using scheduling, caching, and file structures [2], using innovative video compression schemes [5] [6], and deducing various techniques for packet loss compensation [7].

The simple scheme described in this thesis does not take advantage of any of the above methods, and yet achieves a relatively high frame rate of 9 frames per second and requires only 24% of the CPU resources available in the iPaq. The original application running at the same rate uses 77% of the CPU resources, making most compression schemes unattainable if the same throughput of 5.5Mbps is maintained. Adding compression to the enhanced architecture will not put any additional strain on the CPU since all data manipulation would be done in hardware.

10. References

1. H. Schulzrinne, "RTP Profile for Audio and Video Conferences with Minimal Control," January 1996 ftp://ftp.isi.edu/in-notes/rfc1890.txt

2. Tal Anker, Danny Dolev, Idit Keidar, "Fault Tolerant Video on Demand Services," http://theory.lcs.mit.edu/~idish/ftp/icdcs99.pdf

3. Rodrigo Rodrigues, António Grilo, Miguel Santos and Mário S. Nunes, "Native ATM Videoconferencing based on H.323"

4. Kwan Hong Lee, "Impromptu: Audio Applications for Mobile IP, " Master's Thesis, September 2001.

5. Mike Podanoffsky, "Compressing Audio and Video Over the Internet," Circuit Cellar Ink: The Computer Applications Journal, September 1997. http://world.std.com/~mikep/compressingvideo.pdf

6. Ali Saman Tosun, "Video Compression: MPEG-4 and Beyond, " http://www.cis.ohio-state.edu/~jain/cis788-99/compression/

7. Ranga S. Ramanujan, Jim A. Newhouse, Maher N. Kaddoura, Atiq Ahamad, Eric R. Chartier, and Kenneth J. Thurber, "Adaptive Streaming of MPEG Video over IP Networks1," November 1997.

8. B. Salefski and L. Caglar, "Re-configurable computing in wireless," in Design Automation Conference, Las Vegas, NV, June 2001

9. Matti Tommiska, "Reconfigurable Computing in Communications Systems," November, 1998 http://citeseer.nj.nec.com/tommiska98reconfigurable.html

10. PC Card Standard Release 8.0, published by the PCMCIA corp, 2000

11. The PCMCIA Developer's Guide Third Edition, published by Sycard Techbology, 2000

12. Linux Device Drivers, 2nd Edition, Alessandro Rubini & Jonathan Corbet, published by O'Reilly, June 2001.

13. TCP/IP Illustrated, Volume 1, Richard Stevens, published by Addison-Wesley Publishing Company, 1994.

14. IEEE 802.11 Handbook, A Designer's Companion, Bob O'Hara and Al Petrick, published by IEEE Press, Jan 2000.

Appendix: The Orinoco Driver UDP Packet Transmission Protocol

```
/*****************************************************************************************/
/ This is the protocol that describes how the Orinoco Network Card Driver communicates with the Network Card.  /
/ The data being sent in this case is "0123456789\n" This information was gathered by:                         /
/   1. Using a digital analyzer to watch the packet being handed over to the card by the driver.              /
/   2. Using Ethereal to grab the packet to see what actually got sent by the card                            /
/   3. Looking at the Orinoco driver code to see how it commands the card to send the packet. The most relevant /
/      function to do this is orinoco_xmit in orinoco.c                                                       /
/*****************************************************************************************/

/*****************************************************************************************/
/ Tell card you have a packet ready to transmit                                            /
/*****************************************************************************************/
Read from HERMES_OFFSET0(1Ch). If msb high, delay 1us & read again. Else, continue (give up after 500 tries)
  Write the Card ID (00BFh) to reg HERMES_SELECT0 (18h)
  Write the Offset (0000) to reg HERMES_OFFSET0(1Ch)
Read from HERMES_OFFSET(1Ch). While MSB or MSB-1 bits are high, delay us & read again. Else, continue
  Write 12 bytes of 0 to HERMES_DATA0(36h)
  Write 0060h to HERMES_DATA0(36h) to say that packet tx is about to start.

/*****************************************************************************************/
/ Send the packet header to the card
/*****************************************************************************************/
Read from HERMES_OFFSET0(1Ch). If msb high, delay 1us & read again. Else, continue (give up after 500 tries)
  Write the Card ID (00BFh) to reg HERMES_SELECT0 (18h)
  Write the HERMES_802_3_OFFSET (002E) to reg HERMES_OFFSET0(1Ch)
Read from HERMES_OFFSET0(1Ch). While MSB or MSB-1 bits are high, delay us & read again. Else, continue
  Write eth II destination(5000 D68B FC25) to HERMES_DATA0(36h)
  Write eth II source(0100 EEF4 F356) to HERMES_DATA0(36h)
  Write the (min(33,data_len) + 35header) = 0044h to HERMES_DATA0(36h)
  Write AAAA 0003 0000 to HERMES_DATA0(36h)
  Write eithernet type(0008h) to HERMES_DATA0(36h)

/*****************************************************************************************/
/  Send the packet body to the card
/*****************************************************************************************/
Read from HERMES_OFFSET0(1Ch). If msb high, delay 1us & read again. Else, continue (give up after 500 tries)
  Write the Card ID (00BFh) to reg HERMES_SELECT0 (18h)
  Write the (HERMES_802_3_OFFSET(002Eh) + size(hdr))=0044h to reg HERMES_OFFSET0(1Ch)
offest=(2E+size(hrd)) = 2Eh+22d=44h
Read from HERMES_OFFSET0(1Ch). If msb high, delay us & read again. Else, continue.
  Write 0045h to HERMES_DATA0(36h) (45h=header len 20 bytes, 00=some differentiated service field)
  Write total packet body lengh (2 bytes), unrounded to even. (here 11bytes + 28header = 27h)
  Write packet identification (2 bytes)
  Write fragment offset (0040h)
  Write protocol(11=UDP) in first byte, time to live in second byte(1140h)
  Write header checksum (2 bytes)
  Write from addr (4 bytes, 3AC0h CFCEh)
  Write to addr (4 bytes, 3AC0h F7CEh)
  Write from port (0504h)
  Write to port (3200h)
  Write checksum (2 bytes)
  Write the data. In this case the data was:
   w3130 w3332 w3534 w3736 w3938 wEC0A write data (extra EC to round to even - otherwise 63 bytes in packet)
```

```
/*****************************************************************************************/
/ Issue the command to send the packet
/*****************************************************************************************/
Read from HERMES_CMD_INIT(0000h). If msb high, delay us & read again. Else, continue.
  Write 0000h to HERMES_PARAM2 (0006h)
  Write 0000h to HERMES_PARAM1 (0004h)
  Write 00BFh to HERMES_PARAM0 (0002h)
  Write 010Bh to HERMES_CMD_INIT(0000h)


/*****************************************************************************************/
/ Read some status regs to see how happy it is
/*****************************************************************************************/
Read reg EVESTAT(0030h). If (0000 0000 000x 0000) x-bit high, continue. Else, delay us & read again.
Read SWSUPPORT0(0028h) to make sure card is still present (should get 7D1Fh)
Read reg STATUS(0008h) (should get 000Bh)

/*****************************************************************************************/
/ Finally, tell it to send the packet
/*****************************************************************************************/
  Write HERMES_EV_CMD(0010h) to HERMES_EVACK(34h) to make it send the packet
```