Accelerating Photoshop Applications with Reconfigurable Hardware

Guangyu Gu
Department of Electrical and Computer Engineering
Northwestern University

Abstract

In this paper, we investigate the implementation of reconfigurable computing in commercial applications. A Reconfigurable Processing Unit, the Xilinx XC6200 is introduced and our design flow with the HOTWorks development system is shown. We show the design of reconfigurable computing versions of Photoshop plug-ins, and compare various approaches of implementation. We analyze the test result and make suggestions to future designs.

1 Introduction

Introduced in the mid 1980s, the field programmable gate array (FPGA) is now widely used in different applications. The new uses of FPGAs move beyond the implementation of digital logic and the emulation of logic systems. Due to the volatility of FPGAs, they can perform as general-purpose computation machines. We can map not only standard hardware circuits, but also operations from algorithms and general computations onto FPGAs.

While these FPGA-based custom-computing machines may not beat the performance of microprocessors for all applications, they can offer extremely high performance for a wide range of computations. Although a custom hardware implementation can be more efficient than any generic programmable system for a specific problem, the fact is that few applications will ever merit the expense of creating application-specific solutions[11]. With the flexibility to be reprogrammed, FPGA-based computing machines can offer the highest realizable performance for many different applications with reasonable cost.

There are several ways in which reprogrammable logic can be added to standard computer systems: as a functional unit, as a coprocessor, as attached processing units, or as standalone processing units[11]. The functional unit approach and the coprocessor approach will generate low delay during communication with the CPU, but they are not able to handle a large computation task independently. They can beat conventional coprocessors with their flexibility. For example, in the Chimera Project at Northwestern University, FPGA functional units are integrated with a MIPS processor to form a new processor[9]. In the NAPA adaptive processing architecture, at each node, the adaptive logic processor resource is connected to a general purpose scalar processing unit through a dedicated co-processor interface[12]. Also, there is the Garp architecture from Berkeley, in which the standard processor and the reconfigurable coprocessor share a single memory hierarchy, and the coprocessor is under direct control of the main processor[13].

With the attached processing unit approach and the standalone processing unit approach, we can map complex operations onto multiple FPGAs. Since the FPGA units are connected to the CPU through certain memory and I/O interfaces, the communication can be a bottleneck for the whole system. Currently these two approaches are the most common methods, in which reprogrammable systems are put on computer add-on cards or into separate cabinets. Commercial reconfigurable computing cards for PC and for workstation are already available, such as HOTWorks PCI board from Virtual Computer Corporation, and WildFire PCI board from Annapolis Micro Systems Inc.

To introduce reconfigurable systems into commercial applications, we need a specialized compiler to distribute computation tasks to different resources. There are many new challenges for constructing this kind of compilers. In the MATCH project[10] at Northwestern University, a control processor is used to distribute different computational tasks to general processors, DSPs and FPGAs. In this case, a dedicated and efficient compiler is extremely important for a satisfactory result.

Alternatively, we can save the effort of constructing a compiler. Some commercial applications, such as Photoshop and Netscape, have a built-in mechanism called plug-in interface. This enables users to insert customized

functions into host processes. Implementing reconfigurable computing in these commercial applications depends on coordinating hardware and software with instructions in the software code.

2 The HOTWorks Development System

The HOTWorks development system is a product from Virtual Computer Corporation. It consists of a PCI board and the supporting software[1]. On the PCI board, there is a XC6200 RPU for reconfigurable computing. There are also 2 Megabytes of SRAM on the board as well as other logic devices to implement the PCI interface between the host PC and the XC6200.

The software package comprises a VHDL compiler, a place and route tool for the XC6200, and C++ classes for accessing the XC6200 from the host processor. These programs are designed for Windows 95.

The design flow for the hardware circuits on the XC6200 is shown by the figure below. Firstly, structural VHDL code can be edited in any text editor. Next, a VHDL compiler called VELAB is invoked and an EDIF file is generated. Then, a place and route tool called XACT6000, which reads the EDIF file, will be responsible for placement and routing[6]. When the routing is completed successfully, it outputs the design configuration as a CAL file, which is loaded by user programs at run-time. With the information in the CAL file, the system will know how to configure the XC6200.

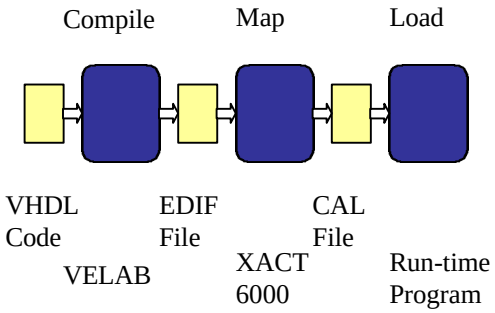


Figure 1. Hardware Design Flow

Although designs and circuits can be tested by a program called "PCITest"[1], the interaction of software and hardware and the realization of run-time reconfiguration are implemented by some C++ code. Because all of the configuration registers in the XC6200 are mapped into the memory and I/O spaces of the host PC, we can access these registers just like we read or write to memory locations. Also, the unique character of the XC6200 enables access to individual logic units in the XC6200. These operations have been encapsulized by certain C++ classes from Xilinx[1]. Thus, several low-level reads/writes can be replaced by a simple call to some member functions in certain classes.

Basically, users need to write their own C++ codes on the Visual C++ platform in their applications. Those C++ classes from Xilinx must be included and compiled along with the user codes.

3 The XC6200 RPU

The reconfigurable computing power of the HOTWorks System comes from the XC6200 Reconfigurable Processing Unit. The XC6200 is a family of fine-grain, sea-of-gates, SRAM-based FPGAs. These devices are designed to operate in co-operation with a microprocessor or microcontroller to provide an implementation of functions normally placed on an ASIC [7].

The XC6200 architecture may be viewed as a hierarchy. At the lowest level lies a large array of simple cells. Each cell contains a computational unit capable of simultaneously implementing one of a set of logic functions, a D-type register and a routing area through which inter-cell communication can take place. There is no LUT in logic cells. Instead, there are multiplexers in each cell, which are controlled by bits within the configuration memory. The design uses the fact that any function of two Boolean variables can be computed by a 2:1 multiplexer if suitable values from the inputs or their complements are chosen.

Neighbor cells are grouped into 4x4 blocks. These blocks, communicating with neighboring 4x4 blocks, form a cellular array themselves. Thus we can also have 16x16 arrays and 64x64 arrays, even 128x128 arrays. At each level, corresponding routing resources are provided. There are wires between neighboring cells, as well as wires of length 4, length 16, and so on. It appears that there are enough routing resources in the XC6200. However, because the area dedicated to routing is still limited, there is a bottleneck for routing complex designs. This also causes problems in system-level synthesis.

The unique character of the XC6200 is that it supports direct access from the processor to nodes within the

user's circuits. Thus, the functional unit output of any cell can be read and the flip-flop within any cell can be written. These accesses are carried out through the control store interface and involve no additional wiring within the user's design. In many applications, this access to internal nodes is the main path through which data is transferred to and from the processor, and in some applications it may be the only external I/O method.

4 Photoshop Plug-in Interface

Adobe Photoshop is a popular image processing package with a modular architecture. Through the plug-in interface, Photoshop plug-in modules can be added or updated independently by end users to customize Photoshop to their particular needs[2]. We can regard plug-in modules as helpers to the Photoshop host program, as shown in figure 2.

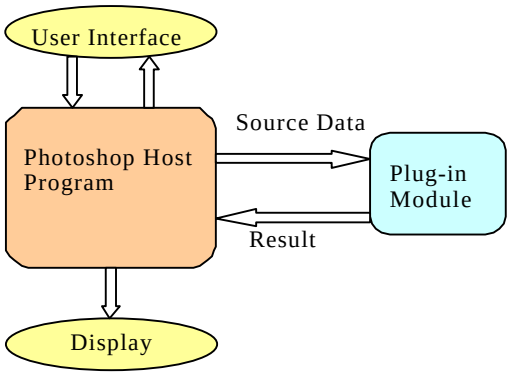


Figure 2. Communication between Photoshop Host and Plug-in

When we want to process an image in some format not supported by the standard Photoshop package, we may develop a plug-in module to read in all the information contained in that source file. All of the image information will be put into the built-in data structures in Photoshop. Thus, we are free to manipulate the image with those functions from Photoshop.

Sometimes, to create a certain visual effect, we need to process a source image in fashions not supported by Photoshop. We hope to focus on developing our own function to process a 2-D pixel array, i.e., the bit map. We do not want to bother with opening an image file, decompressing the image, and writing back the result. To achieve this, we can write a plug-in for Photoshop, which is only concerned with that kind of image processing, and the Photoshop host program will implement all the other functions.

Some plug-in modules contain computationally intensive operations. If we succeed in accelerating these modules, then the whole application has been significantly accelerated as well. We can choose better algorithms and apply efficient data structures in our plug-in module. Alternatively, we can map some computation onto reconfigurable hardware.

5 The Grayscale Problem

We have chosen Adobe Photoshop as our platform for reconfigurable computing applications. There is one category of plug-in modules called *filter*, which enables inserted code to process image data prepared by Photoshop.

Obviously, some image processing procedures consist of intensive computation, such as fix-point multiplication, coding, DCT and other transforms. These computations normally take a large amount of time on conventional processors. If we introduce the XC6200, and exploit the inherent parallelism of operations, considerable time may be saved.

In some cases, it is desired to convert a color image into a grayscale image, either for less storage size or for specific visual effects. Any color image can be converted to an RGB mode image. Usually, eight bits are used for a color, so the intensity of each color can take one of the 256 discrete values. For RGB mode images, the color of each pixel is described by 24 bits.

It is pretty straightforward to turn RGB mode images into grayscale images. Simply, we have

$$Grayscale = R/3 + G/3 + B/3 \quad [4]$$

Since human eyes have different sensitivity to the frequencies of red, green and blue, the grayscale of a pixel can be computed more accurately as

$$Grayscale = 0.299 \times R + 0.587 \times G + 0.114 \times B \quad [5]$$

When set to RGB mode, Photoshop will represent any image as an RGB pixel array, or bitmap, after it has opened that image. Once we get the information of a pixel from Photoshop, we have the intensity of red, green, and blue. With the XC6200, the computation of 0.299×R, 0.587×G, and 0.114×B can be done in parallel. It is feasible to build a pipeline for each multiplier and for the final addition stage to further improve performance.

6 Evaluation of Hardware Approaches

6.1 Direct Access

Since the XC6200 supports direct access within the whole user circuit, we may eliminate all the I/O buffers and pads in our design. Basically, the user's code gets all the input data and writes to certain columns in the XC6200, where we put input buffer registers to hold the source data. The output of those input buffer registers is fed into some complex logic network. The network output is routed to certain output buffer registers, and is then read by the host PC.

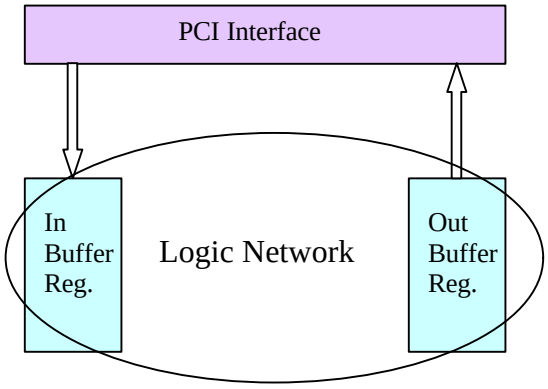


Figure 3. Direct Access Approach

In user code, there are C++ function calls to write to the input buffer registers, and to read the output buffer registers. These functions are translated into several software instructions and moreover, it will take multiple cycles to access those logic units in the XC6200. The reason is that several additional configuration registers need to be updated to make sure the reads and writes are only applied to related logic units. These operations in software take considerable time, and they result in heavy PCI traffic between the host PC and the XC6200.

With direct access, we are liberated from building input and output sub-circuits for our design. On the other hand, we have to spend more time coordinating the software operations and the hardware circuits when we are designing complex sequential circuits, because the software code visits the buffer registers during the state transitions of the sequential circuits.

In fact, a test version of our logic has been implemented with direct access approach. It takes 16 seconds to process 1×10^6 pixels. Compared with x seconds for a purely software version, this is intolerably slow.

6.2 The On-board SRAM Approach

There is two Megabytes of fast SRAM on the HOTWorks Board. The memory is organized into two banks. Each bank of the SRAM can be accessed from either the PCI Interface (controlled by PC Host) or the XC6200. The memory banks have two separate address buses and four read/write signals to control the SRAM. Thus the development system provides a flexible architecture in order to implement a wide variety of algorithms.

Obviously the on-board memory can act as a system input/output buffer. The host PC can transfer a large amount of input data in one function call, filling the SRAM with input data. When the XC6200 is working, it retrieves input data from the SRAM and also writes the result back to the SRAM. Finally, the host PC reads the content of the SRAM, which is the result from the XC6200.

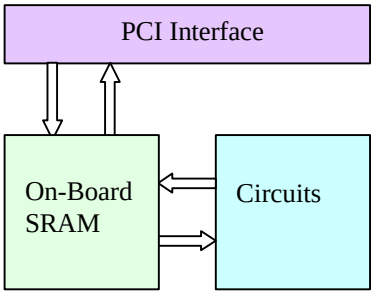


Figure 4. On-Board SRAM Approach

The SRAM is mapped transparently into a region of PCI memory address space. The function calls writing and reading SRAM through the PCI Interface consist of a large amount of PCI bus

transfers and a few accesses to configuration registers.

This approach is clearly a winner over the direct access approach when the block of traffic data is large enough. For example, if we want to transfer more than 64 bytes (16 words) of data to the XC6200, the SRAM approach takes less time.

7 Basic Design Flow of Plug-in Modules

The framework of our plug-in module is based on the C code obtained from Photoshop Plug-in SDK. In this framework, we specify how many iterations are needed to process the image data, in what order we want to process the pixels, and some other conditions.

There are several C++ classes provided by Xilinx to access and control the XC6200 from the host PC. We initialize an object of the *XC6200* class, which has full access to various resources on the XC6200 and the HOTWorks board. A pointer to that object is passed to the plug-in framework. With that pointer, we can read and write arbitrary columns in the XC6200, manipulate the on-board SRAM memory, and control the behavior of the XC6200 as well as the clock rate of the system from within the plug-in module.

The code to filter image data is located in certain functions in the plug-in framework. Basically, all the operations not suitable for hardware have been implemented in software. Computationally intensive operations are assigned to the XC6200. Software controls writing data to the on-board SRAM and then writes to certain columns in the XC6200, signaling it to start computing. While the XC6200 is working, it processes the source data in the SRAM, and writes back results. At the same time, the host PC is free and can keep on processing data. When the XC6200 finishes computing, the software code on the host PC invokes instructions to read back the results from the on-board SRAM.

Once the plug-in module finishes its task, the control will be returned to the Photoshop host program, and we are free to manipulate the image with all the functions within Photoshop.

8 Implementation of On-board SRAM Version of the Grayscale Plug-in

8.1 Design of Circuit

For each term in 0.299×R+0.587×G+0.114×B, an 8-bit by 8-bit multiplier is needed. Building three 8-bit by 8-bit multipliers will take considerable resources on the XC6200. An alternative approach is to use hardwired shift registers and add up all the partial results. For example,

$$0.299 \times R$$
$$= (1/4 + 1/32 + 1/64) \times R$$
$$= (R >> 2) + (R >> 5) + (R >> 6)$$

Several 8-bit adders and 9-bit adders are built to accumulate all the partial results from shift registers, and pipeline registers are placed between two stages to enable a high clock rate.

Since the source data is read from the SRAM and the results are sent back to the SRAM, we need some sub-circuits to take care of the interaction between the SRAM and the XC6200, since the XC6200 does not provide built-in control mechanism for memory access.

8.2 SRAM Interface

Generally speaking, to process one pixel, the R, G, and B values need to be passed to the XC6200 through the SRAM, at a rate of 24bits/pixel from the host PC to the on-board SRAM.

For our instance of grayscale conversion, the transfer rate from host PC to the on-board SRAM is 24bits/pixel, while the rate from on-board SRAM to host PC can be reduced to 8bits/pixel.

Since there are several modes of SRAM operation, we have two possible schedules for the data traffic.

Schedule 1: First, the PCI Interface has a 32-bit bandwidth to the SRAM, then it gives up the control of data bus, and lets the XC6200 RPU have a 32-bit bandwidth to the SRAM. This procedure will be repeated until all the data has been processed[1]. These two connections are shown in Figure 5.

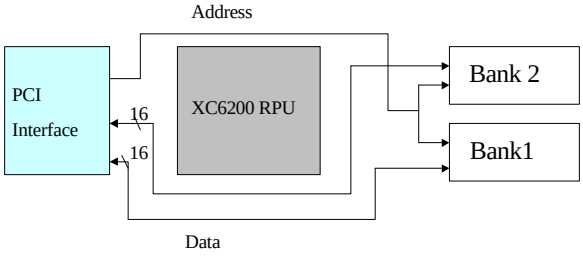
Schedule 2: First, PCI interface controls bank1, the XC6200 controls bank2, then they swap their banks, and so on[2]. These two connections are shown in Figure 6.

The benefit of schedule 1 is that we can minimize the number of PCI reads and PCI writes. On the other hand, schedule 2 overlaps computation on XC6200 with the PCI reads and writes.

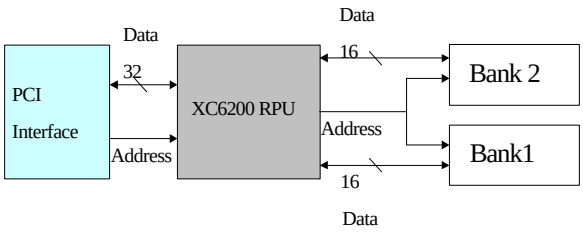
We have noticed some facts of the PCI bus traffic. It takes about 0.1 seconds for the host PC to write to 1×10^6 consecutive locations on SRAM, while it takes about 0.5 seconds to read the contents of those 1×10^6 consecutive locations. We may suppose that the XC6200 always works at 20MHz, and it always takes 1.25 cycles for the XC6200 to process one pixel(One read cycle per pixel, and one write cycle every four pixels). Therefore, it takes about 0.06sec to process the data in 1×10^6 memory locations.

Since each pixel has an 8-bit result, and 32 bits may be read back in one PCI read operation, we needs 1 PCI write, and 0.25 PCI reads for each pixel with schedule 1. Therefore it takes 0.1 + 0.06 + 0.5/4 = 0.285 seconds to process 1×10^6 pixels.

With schedule 2, two 16-bit PCI writes are needed for the 24 bits input data of one pixel, and the result from two pixels can be read back in one 16-bit PCI read. It takes MAX(2 PCI write time + 0.5 PCI read time, average time for the XC6200 to process one pixel) to process one pixel. Therefore it takes $0.1\times2 + 0.5/4\times2 = 0.45$ seconds to process 1×10^6 pixels.

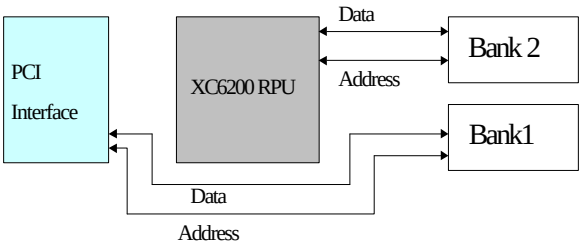


Connection 1

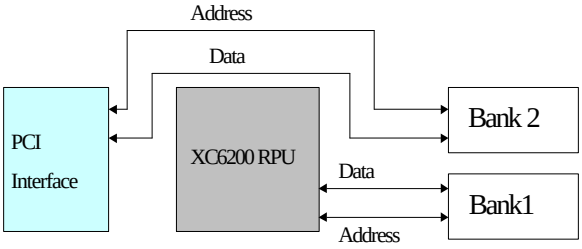


Connection 2

Figure 5. Schedule 1 for the SRAM Connections



Connection 1



Connection 2

Figure 6. Schedule 2 for the SRAM Connections

Schedule 1 has better performance than schedule 2. Therefore, in our design, both the PCI interface and the XC6200 RPU access the on-board SRAM with 32-bit bandwidth.

We can also conclude that schedule 1 is better in most of the applications for this system. Suppose that for a certain algorithm, 4×10^6 bytes of data need to be transferred to the XC6200. With schedule 1, we need at 0.1 seconds, while with schedule 2, we need 0.2 seconds. When the circuit can process 32 bits in less than 2 clock cycles, the computation will take less than 0.1 seconds. Therefore with schedule 1, we need less than 0.2 seconds for the whole process. With schedule 1, we will need 0.2 seconds in total. When we take more facts into consideration, the results will remain the same for most cases. This is because the computation time is much less than the data transferring time.

An SRAM macro design from Xilinx basically satisfies the requirement of schedule 1[3]. It is a circuit dedicated to the communication between user circuits and the SRAM module. Additional logic is added to make timing clear for the SRAM read and write operations. Two 32-bit

registers serve as the from-SRAM buffer and the to-SRAM buffer to make all the operations synchronized to a global clock signal. Once we feed the SRAM macro with RDWR, CLK, and CLK2 (it has a doubled rate when compared to CLK), the SRAM macro will generate all the control signals for the SRAM. Therefore the operations on the SRAM module have been encapsulized by the SRAM macro. Figure 7 shows the working of SRAM Macro. This SRAM macro can be reused in many other applications.

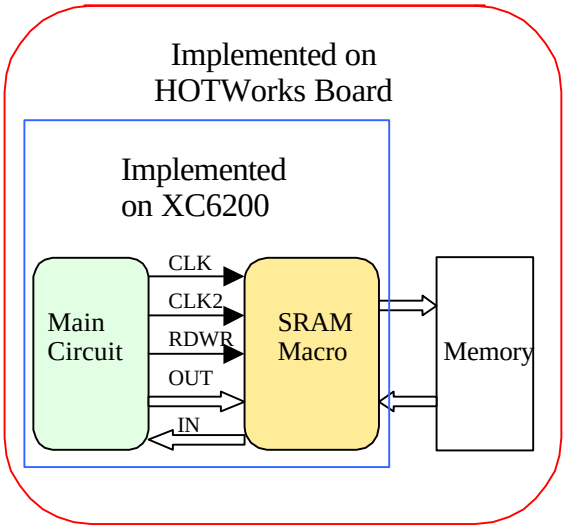


Figure 7. Interface to SRAM Macro

8.3 Finite State Machine

For the grayscale problem, R, G and B takes 24 bits, while the result for a pixel takes 8 bits. For improved performance, the results of four pixels can be transferred at the same time.

An FSM is built to guide the XC6200's access to the SRAM. The circuit is designed to perform four read operations at consecutive locations in lower address spaces, followed by one write in upper address spaces for four pixels' grayscale. Corresponding addresses and the RDWR control signals will be generated at the same time.

All the input data (R, G, and B) are located in the lower end of address space, while all the results (grayscales) are stored in the upper end of the address space. We also built a version in which the read address and the write address both start at 0. The data is safe because there are four reads along with one write, so the writing back will not erase any useful source data.

8.4 Address Generation

A counter with enable port has been built to provide addresses. Its output is used to generate the read address and write address for the SRAM macro. A 19-bit MUX controlled by the FSM selects the read address or the write address and sends the output to the address port in the SRAM macro.

8.5 Circuit Layout

The final design is in structural VHDL. The code has been compiled and the resulted EDIF file has been routed. The layout is shown in Figure 8.

In Figure 8, the blocks on the left are adders and shifters. The blocks on the right are counters, multiplexers and the finite state machine. Only parts of the SRAM macro have been shown in this figure.

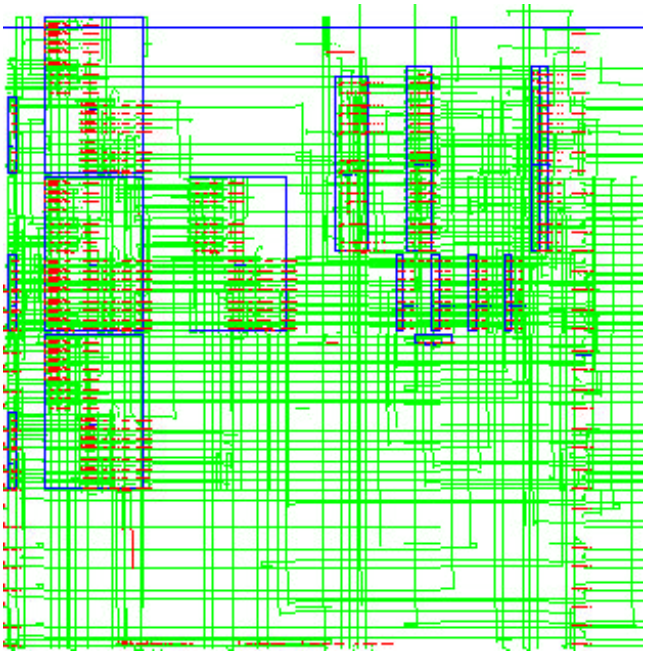


Figure 8. Circuit Layout for XC6200

The layout figure shows that only a small part of logic units has been utilized, but the routing in certain areas is intensive. It was limited routing resources that made the design more difficult.

9 Test Results

Because there is a long carry chain in the address counter and the SRAM needs good timing signals, the HOTWorks board works at 20MHz, processing 262,144 pixels in an iteration. On average, it takes 1.25 cycles to process each pixel. The computation of grayscale is nearly 10 times faster than the C++ operations in software.

To demonstrate how the computation on hardware cooperates with the PCI interface and the software in plug-in framework, an SRAM-based reconfigurable version of grayscale plug-in was completely implemented for Photoshop. An enhanced reconfigurable version was implemented, in which the host PC processes data when the XC6200 is computing. Finally, two pure software version of plug-in were developed. In one of them, we compute the grayscale with multiplication and addition as we normally do in software code. In the other, shifting and accumulation are used instead. All the plug-ins were running on our Micron PC (Pentium II 400) with a FAT HOTWorks Board, which contains an XC6200 chip with a 128x128 cell array. It takes about 0.7 to 1 seconds for each version to process a 1280x1024 color image, and the difference of run time is very small.

The grayscale values obtained in the reconfigurable computing version contain certain truncation errors introduced in shifting and addition. About 5% of the results are 1 less than the results in the software version doing multiplication. We are putting more bits into the computation circuit to reduce truncation errors.

To determine the system speedup, the grayscale filter operations were repeated 100 times, and the run time recorded in the table below. The result showed that the first reconfigurable version was 10% slower than the pure software version doing multiplication. But the second reconfigurable version is 15% faster than that software version. The software version doing shifting takes least time to finish.

Software (multiplication)	1st Reconf.	2nd Reconf.	Software (shifting)
44 sec	49 sec	37 sec	31 sec

It's very feasible to implement multipliers on the XC6200, to achieve the same high throughput rate for the computation of grayscale. Therefore we may beat the software doing multiplication. When the software simply does shifting and addition, all the operations can be completed in several clock cycles, therefore the high CPU rate helps to achieve a faster solution than the reconfigurable computing method.

It appears easy to obtain great speedup with hardware, because some operations may be simpler when implemented on hardware, and it is quite easy to exploit the inherent parallelism of applications with hardware circuits. But the difficulty arises as we need to tell the hardware when to start computing and when to stop, what input to be fed to the RPU, and what output to be read back. With the Direct Access approach, we have to perform lots of read and write operations on the control registers and configuration registers of the XC6200 for each piece of input data. This consumes a considerable amount of time by producing heavy traffic on the PCI bus. With the on-board SRAM approach, there is less PCI traffic, but the total speedup still depends on efficient implementation of PCI transfers.

10 More Analysis of Test Results

10.1 Components of Execution Time

The total execution time can be divided into the following parts.

The *preparation time*: the time elapsed when the Photoshop host program prepares data for the filter plug-in. It is the same for both the pure software plug-ins and our reconfigurable computing solutions. For large images, this time can be neglected.

The *writeRAM time*: the time needed to call the writeRAM function to move the data from a user-specified area to the on-board SRAM.

The *readRAM time*: the time needed to call the readRAM function to move the data from on-board SRAM back to a user-specified area.

The *writeRAM* and *readRAM time* is dominated by PCI traffic time. Normally, they are the largest fractions of the total execution time. For some reason, with the same traffic, *readRAM time* is much longer than *writeRAM time*. Therefore, *readRAM* is the dominant fraction of the total execution time.

The *auxiliary-moving time*: the time consumed when the data are moved to and from the Photoshop data structure. It can be saved if we

add considerable complexity to our XC6200 configuration. Normally, we need to copy the source data to a temporary location and pack them in specified order before we call the writeRAM function. The reason is that Photoshop packs the source data in its own way, which will cause inconvenience for the XC6200 to process the data.

The *software time*: the time consumed by the operations having to remain on the host PC. It can not be completely removed for some complex computations, because some functions will take too much resource to be mapped to hardware, or there is no more space to hold them in the XC6200. These functions can be regarded as part of the pure software version, which can not be parallelized or pipelined by hardware.

The *hardware time*: the time consumed by the operations on the XC6200 RPU. It is affected by the SRAM interface on the XC6200 board, since in each clock cycle, at most 32 bits can be read or written. This presents a bottleneck for our parallel processing on the board.

In Photoshop *filter* plug-ins, this restriction normally means that only one combination of R, G, and B values can be transmitted in one clock cycle. Therefore, to keep the process stable, the best case we can expect is one pixel processed in every one to two clock cycles. Figure 9 shows the major components of the execution time in Photoshop plug-in.

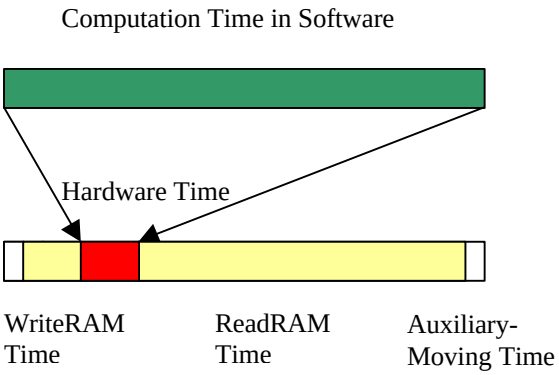


Figure 9. Major components of plug-in execution time in pure software version and reconfigurable computing version

10.2 Comparison of Reconfigurable-Computing Version to Pure Software Version

Suppose we have a software version of a Photoshop plug-in, which takes n seconds on a Pentium II 233 PC to process an image containing 1×10^6 pixels. Also, suppose that we developed a reconfigurable computing version on the XC6200.

For our grayscale application only, some simulation results are as follows. It takes 0.09 seconds to move data to the write buffer, and 0.12 seconds to call the *writeRAM* function. Also it takes 0.13 seconds to call the *readRAM* function, and 0.04 seconds to get data from the read buffer.

To make the SRAM interface work correctly, the main part of the circuit works at 20MHz. For 1×10^6 pixels, we need to perform 1.25×10^6 read/write operations. This will take 0.0625 seconds.

If we neglect the *software time* and *preparation time*, and we neglect the fact that the host PC can process data during the *hardware time*, the total execution time will be 0.09 + 0.12 + 0.13 + 0.04 + 0.06 = 0.44 seconds.

If the software version is worse than the reconfigurable computing version, it will take n seconds, in which n > 0.44 seconds. Suppose these 0.44 seconds are all used by image processing functions, then it takes at least 0.44×10^{-6} seconds to process each pixel. Therefore, on average, the software version needs at least 103 cycles to process each pixel.

Thus, we may conclude that, to beat the software version, we have to put certain computation tasks, which take at least 103 cycles on the host PC, onto the XC6200. We also need to make sure that the pipeline structure has a throughput of 1 result every 1.25 clock cycles, because we expect the FPGA take 0.0625 seconds to process 1×10^6 pixels.

10.3 Achieve More Speedup and Introduce the *Software Time*

In this example, we assume that 1/4 of the computation task can not be mapped onto the XC6200, but we still want to achieve a speedup of 2. We still suppose the pure software version takes n seconds to process 1×10^6 pixels.

For best result, we can overlap the FPGA *hardware time* with the host PC *software time*, therefore we will not considerate *the hardware time*. Furthermore, we neglect any possible *auxiliary-moving time* and the PCI traffic is as low as the case in the grayscale example. We have $n/4 + 0.12 + 0.13 < n/2$, that is, n > 1 second. This means that we need to put the computation tasks, which take 233 cycles on the host PC, onto the XC6200, and the pipeline structure must have a high throughput.

Obviously, the computation of grayscale is not so complex, and we can not exploit so much inherent parallelism. We may conclude that with the HOTWorks system, it is very difficult to enjoy a speedup of 2 while placing only 3/4 of the total computation tasks onto the XC6200 and considering all the potential overhead.

We can explain this with Amdahl's Law:

$$T_p = (a + (1-a)/p) \times T \qquad [8]$$

When we need more speedup, $a + (1-a)/p$ should be small. When *software time* can not be eliminated, we have $a > 0$. Then, we have to make *p* large enough. But *p* depends on the characters of different computation tasks.

10.4 Some Conclusions

Great speedup is only available in the cases where software versions comprise lots of computations for each pixel. Also, in order to achieve that speedup, a hardware configuration which exploits considerable parallelism and pipelining must be found. Moreover, a better PCI transfer rate can greatly reduce the time consumed by the whole process.

Fortunately, better PCI transfer solutions are available in the market. If the PCI transportation is improved, our reconfigurable-computing version plug-ins will clearly beat pure software version plug-in.

11 Summary

With the plug-in interfaces, reconfigurable hardware can be used in commercial applications. With efficient implementations, reconfigurable computing can accelerate those applications, even if the pure software version is running on fast high-end PC/workstations. To make the reconfigurable-computing approach more appealing, vendors need to improve their development system with better PCI BUS performance, and users must find suitable applications and come out with an efficient, tight circuit design.

Acknowledgements

This research was funded in part by DARPA contract DABT63-97-C0035 and NSF grants CDA-9703228 and MIP-9616572.

Special thanks to Doug Wilson, who contributed a lot to this project.

Reference

[1] VCC, *HOTWorks User's Guide*

[2] Adobe Systems Inc., *Photoshop 4.0 SDK*

[3] Xilinx Inc., *Parameterized Libraries Project*

[4] Satnum Singh and Robert Slous, *Accelerating Adobe Photoshop with Reconfigurable Logic*, IEEE Symposium on FPGAs for Custom Computing Machines, pp. 236-244, 1998

[5] Richard Jackson, Lindsay MacDonald and Ken Freeman, *Computer Generated Color*, John Wiley & Sons Ltd, Chichester, England, 1994, pp.28-29.

[6] Xilinx, *Series 6000 User Guide*, Chapter 4

[7] Xilinx, *XC6200 Field Programmable Gate Arrays*, pp. 3-7

[8] David Patterson and John Hennessy, *Computer Architecture, A Quantitative Approach*, Second Edition, Morgan Kaufmann Publishers, Inc., 1996, pp. 29-32

[9] Scott Hauck, Thomas W. Fry, Matthew M. Hosler, and Jeffrey P. Kao, *The Chimaera Reconfigurable Functional Unit*, IEEE Symposium on FPGAs for Custom Computing Machines, pp. 87-96, 1997

[10] Prithviraj Banerje, Alok Choudhary, Scott Hauck and Nagaraj Shenoy, *A MATLAB Compilation Environment for Distributed Heterogeneous Adaptive Computing Systems*, http://www.ece.nwu.edu/cpdc/Match/Match.html

[11] Scott Hauck, *The Roles of FPGAs in Reprogrammable Systems*, Proceedings of the IEEE, Vol. 86, No. 4, pp. 615-638, April, 1998

[12] Charle R. Rupp, Mark Landguth, Tim Garverick, Edson Gomersall, Harry Holt, Jeffrey M. Arnold and Maya Gokhale, *The NAPA Adaptive Processing Architecture*, IEEE Symposium on FPGAs for Custom Computing Machines, pp. 28-37, 1998

[13] J. R. Hauser and J. Wawrzynek, *Garp: A MIPS Processor with a Reconfigurable Coprocessor*, IEEE Symposium on FPGAs for Custom Computing Machines, pp. 12-21, 1997.