GATOS: A Windowing Operating System for FPGAs

D. G. Bailey, K. T. Gribbon, C. T. Johnston
Institute of Information Sciences and Technology
Massey University, Private Bag 11 222
Palmerston North, New Zealand
d.g.bailey@massey.ac.nz, k.gribbon@massey.ac.nz, c.t.johnston@massey.ac.nz

Abstract

FPGAs are increasingly being used to implement low-level vision operations in stand-alone configurations. For real-time image processing applications there is a need for several interactive operating system functions such as tuning, calibration, user interaction, and debugging which are difficult to perform when not under the control of a host operating system. This paper proposes the Gate Array Terminal Operating System (GATOS) which provides a set of high-level IP blocks that implement the graphical user interface of an interactive windowing operating system. The requirements and desired levels of functionality for GATOS are discussed and the preliminary design is presented.

1. Introduction

FPGAs are increasingly being used to implement low-level vision operations because the architecture of FPGAs is ideally suited to such vision tasks. The reprogrammable matrix of logic cells allows application specific hardware to be constructed, while at the same time maintaining the ability to change the functionality of the system with ease. Performance gains are obtained by bypassing the fetch-decode-execute overhead of serial processors and by using the inherent parallelism of digital hardware to exploit concurrency within the algorithm. As a result FPGAs may achieve the same computation with operating clock frequencies an order of magnitude lower than high-end serial processors, lowering power consumption.

Two major forms of parallelism exist in low-level image processing [1]: spatial parallelism, in which the image is divided into multiple sections and processed concurrently, and temporal parallelism, where the algorithm may be represented as a time sequence of simple concurrent operations. The inherent ability of FPGAs to support concurrent processing blocks and the large number of I/O pins for data access allows them to exploit both types of image parallelism [2].

Two system configurations are commonly used: hosted, and stand-alone. In the hosted configuration, the FPGA is utilized as a co-processor to a conventional serial computer. Software running on the computer transfers the image data being processed into memory shared between the two systems, and then issues a signal to initiate the processing by the FPGA. The results of the processing are then available again through the shared memory, or via registers constructed on the FPGA.

In the stand-alone configuration, the host computer is dispensed with, and all of the processing is performed on the FPGA. This complete shift away from using a serial computer is made possible by two recent developments: FPGAs now have sufficient processing resources available that make it practical to include the whole application on a single FPGA for non-trivial tasks [3, 4]; and the introduction of high level languages enabling an algorithmic approach to programming FPGAs [5, 6].

One limitation of the stand-alone configuration is that issues such as tuning, calibration, user interaction, and debugging become significantly more complex. In the hosted configuration, many of these operations are performed under the control of the host operating system.

1.1. Functions of operating systems

Device drivers for interfacing with peripherals: On an FPGA these correspond to intellectual property (IP) blocks or cores that implement the interface functions. These are generally at the low level, and provide the glue for configuring or communicating data with the device. At the device driver level, limited meaning or interpretation is made of the data being transferred. Examples are drivers for external memory,

0-7695-2500-8/06

keyboard, mouse, VGA display, camera, network, and so forth. On a hosted configuration, the FPGA is used primarily as a computation engine, and many of these functions are not required. On a stand-alone configuration, these functions are essential, even if it is just for the purposes of gathering data to process.

An environment for interacting with a user: These are at a higher level, and generally assign meaning or interpretation to the data being received. Examples are console windows where characters typed on a keyboard are interpreted as commands, and the response or debugging information is echoed back to the user. In a graphical environment, mouse gestures and clicks are interpreted as actions to manipulate widgets (windows, buttons, sliders, etc) on the display with the actions used to control corresponding underlying objects by adjusting parameters. In the hosted configuration, the operating system of the host provides all of the interaction with the user. These functions are normally absent from within a stand-alone FPGA configuration, and must be explicitly developed for the applications that require them.

Implementation of common functions: These are intermediate level functions, required to convert the high level data from the user interaction to the low level form required by the device drivers. Examples are character generation on the display, and conversion between different number representations (binary to decimal conversion). These may be implemented as a library of common IP blocks.

Task scheduling: Controls the scheduling of tasks or processes for execution, and allocation of resources to each process. Also facilitates and controls interprocess communication. On an FPGA, much of this is implemented directly in hardware. The FPGA is generally configured for a single function, although this may include several modules or tasks running in parallel. On the FPGA, interprocess communication is often through the use of shared memory or registers, and semaphores may be implemented directly in hardware if necessary to arbitrate between accesses to these resources.

Memory management: This sub-system controls the allocation of memory resources to processes, ensuring that each process is allocated enough memory for its execution and that it cannot run into the allocated memory space of another process. If shared memory is used, such as in interprocess communication, the memory manager arbitrates access to the memory. The types of memory available on a typical stand-alone FPGA system consist of fabric RAM, block RAM, and off-chip RAM.

1.2. Proposal

Prior work on operating systems for FPGAs has concentrated on the hosted configuration. Here the native operating system of the host is augmented with functions that enable reconfiguration of the FPGA along with scheduling of data communication and the execution of the algorithms on the FPGA [7].

In our work with stand-alone FPGA configurations for image processing applications [2, 4, 8, 9], we have identified the need for several operating system functions that are unavailable in this configuration. These all rely on interaction with the user, and fall into three main categories: algorithm debugging, algorithm tuning, and managing display real-estate.

One problem with debugging image processing algorithms is the large volume of data contained within an image. This makes conventional hardware debugging paradigms impractical. With complex algorithms, it is extremely difficult to design test vectors that test all of the functionality of the system, especially when there may be complex interactions. In image processing, the problem is even more difficult, because the algorithm may be working perfectly as designed, but it may not be appropriate or adequate for the task to which it is applied. Validation using test vectors will only verify that the algorithm is implemented correctly, and not whether the correct algorithm is actually being used. When processing video, this is exacerbated by the dynamic nature of the data being processed. Functions required are being able to freeze the input stream, have an easy way of displaying the results of intermediate processing stages, including image statistics, and have some method of displaying textual debugging information. These can be used to isolate the deficiencies in the algorithm, and allow a dataset to be captured for test vector based validation.

Many image processing algorithms require tuning or adjustment of algorithm parameters in order to achieve best results. If only a single parameter is required (for example adjusting a lens distortion factor [8]) then a relatively simple interface is required, such as using key presses to increase or decrease the parameter value. If several interacting parameters must be adjusted simultaneously (for example selecting color thresholds when segmenting a color image [4]), then some form of graphical user interaction is essential. This may require the display of auxiliary information such as intermediate images and histograms that are only required while tuning.

A number of these tasks may require simultaneous display of multiple images or data sets. Calculating these in real time is generally not a problem, as the

FPGA will have separate hardware constructed for each operation or function. However, full interaction requires being able to control which views are visible, and the views may include textual labels required for debugging purposes. The most familiar paradigm for such user interaction is a windowed environment, where each image is within a separate window, which may be moved around on the limited display space to show the regions of interest. As image processing algorithm development and debugging is often an interactive process, it is important that the FPGA enables real-time dynamic interaction, including the automatic display of contextual information relating to where the cursor is currently pointing.

In a conventional software-based image processing environment, many of these functions are facilitated by the underlying operating system. On an FPGA, however, there is no underlying operating system. Our proposal is to develop the features of a windowing operating system that will enable many of the user interaction modalities commonly encountered in modern software environments. The Gate Array Terminal Operating System (GATOS) will provide a set of high-level IP blocks that implement the graphical user interface of a windowing operating system. As each function or operation must be implemented in hardware on an FPGA, only those features that are required in any particular application will need to be instantiated. It is assumed that low-level IP blocks for any device drivers are readily available. These will not explicitly be included within GATOS as they are likely to depend significantly on the particular hardware configuration. Intermediate level functions, such as character generation, and cursor control will be considered part of GATOS.

Section 2 discusses some of the requirements, issues, and desired functionality for GATOS. In section 3, simple implementations of various display functions and an IP block for character generation are described, to demonstrate the feasibility of implementing GATOS. Section 4 concludes with a summary.

2. Requirements and desired functionality

GATOS will support the display of many widgets. These widgets could be sliders, windows, or buttons and are based upon an underlying widget model. The underlying model contains a set of properties, such as position and size that are common to all widgets. This is comparable to an object-oriented model in the software domain.

2.1. Interactive widgets

Interactive widgets such as buttons and sliders can be created using the fixed widget model by overlaying the widget at a fixed position relative to a window or the screen.

A button has simple on/off state linked to a more complex action such as controlling zoom, saving data to non-volatile memory or cycling between different views or images. These interactive widgets provide control for the display or change parameters for image processing algorithms that are being processed in the background. A button needs to react to mouse clicks and requires either a color and/or label to indicate its state or action.

Sliders provide more complex functionality. A handle is required which the user can grab and drag to specify the value of some parameter. An important consideration is the step size of the slider which is directly related to the bit width of the slider variable and must be mapped appropriately to the movement of the slider. Sliders are useful for specifying tuning thresholds or as scroll bars within windows.

2.2. Windows

When developing image processing applications it is useful to see what is occurring at different stages of an algorithm. The simplest approach is to segment the screen into multiple windows of fixed size and position. This allows input, intermediate, and processed images to be simultaneously displayed along with other useful data such as histograms. It is also relatively simple to implement. Images can be down sampled to fit the fixed window and this can be easily achieved if the factor is an integer multiple. Fixed windows can be enhanced with buttons for selecting different images or zooming.

Dynamic windows that allow repositioning and overlapping offer greater flexibility than fixed windows and help maximize display ‘real estate’ as dynamic windows may be repositioned if they are obscuring view. However they come at additional cost in terms of implementation complexity and resource requirements. Unlike fixed windows, dynamic windows require additional properties to be stored in their data structure.

For use in image processing the windows need to have the ability to be minimized or maximized. The windows require the data structure to contain a z-index to determine the relative order for display. These windows also must allow for other widgets (such as buttons) to be incorporated within them. In this case the widget’s screen coordinate will be given by its relative position within the window. The relative position

within the display can be obtained easily once this information is known. These widgets are located on the top layer of the window, but not necessarily the global top layer as Figure 1 illustrates.

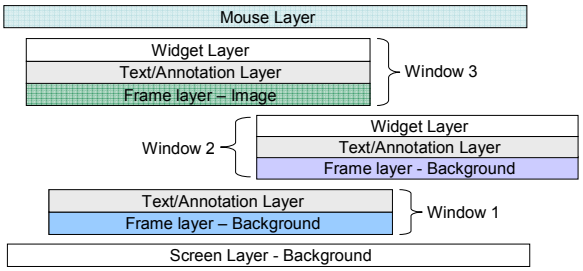

Figure 1: x-z diagram showing window layers

2.3. Overlays

Text and other graphical annotations are required for labeling and highlighting objects or regions of interest. Text and graphics overlays are usually associated with a window and appear on the top layer of window but not necessarily the global top layer as this is used for the mouse pointer. It may also be desirable to annotate areas of windows with text and graphics, so a global annotation layer can be useful. Updates to text output displayed in a window (such as printing debug information) must be slowed down because updating text at the display refresh rate would be too fast for the user to comprehend.

3. Implementation

The GATOS system comprises a number of different functions detailed in section 1. This section will discuss possible approaches to implement some of these core functions. An efficient approach for representing and displaying the widgets which form the windowed environment is an important requirement.

3.1. Data representation

Hardware must be built to implement every widget. A naïve implementation would build one hardware module for each widget present within the application. This approach is very expensive in terms of resources.

It is observed that only the display windows may be overlapped, but any controls contained within the windows should not overlap. Also, in terms of user interaction, only a single widget is being modified at a time, or the adjustment process is sufficiently slow to allow the adjustments to be multiplexed. These factors imply that it should be possible to implement only a single hardware module for each type of widget, and multiplex this hardware between the different instances of this in the application.

This requires representing each widget with an appropriate data structure, and have the hardware module that implements the widget operate on the specific data whenever the widget appears on the display, or is adjusted. While this approach is usual within a software-based system, particularly an object oriented software system, the method of implementing such a system in hardware is not as obvious.

One of the issues is that unlike the unified memory space available in operating systems in software, multiple memory spaces exist in hardware across different resources. Hardware registers (flip-flops), fabric RAM, and block RAM can be used to store data structures internally on the FPGA. External RAM is also a common addition to many FPGA-based systems. Each memory resource has its own trade offs. For example, hardware registers are plentiful on modern FPGAs but are quickly consumed when used to represent data structures because multiplexing between a large set of registers is expensive particularly if the data path is wide.

Conversely, a data representation that configures logic cells as fabric RAM or uses dedicated memory resources like block RAM or external RAM to represent data structures can reduce resource utilization because multiplexing is performed internally through the address port. However, this potentially places large amounts of data behind limited bandwidth and serialized connections, making it difficult to access all relevant parts of the data structures. This is an important consideration if on-the-fly display generation is being performed, as this can introduce timing constraints. This becomes a complex scheduling issue as widget properties must be retrieved from RAM in anticipation of the widget being displayed. This may also require some form of local buffering so that access to relevant parts of the data structure is guaranteed.

A data representation provides more flexibility and extensibility as widgets can be created and destroyed on-the-fly. However, the different data storage methods detailed above can place limitations on the number of widgets that may be represented simultaneously. For example, block RAM provides dedicated memory but is only available in limited pools.

3.2. Display system

The display system must be designed first. Alternatives are the conventional frame buffer approach, common within computers, or to calculate what is displayed for each pixel on-the-fly (as data is sent to the terminal). With either approach there are bandwidth issues that may make it difficult to access the required data when it is needed.

The frame buffer approach requires large amounts of memory in the form of off-chip RAM. In many applications, off-chip RAM is also needed for storing images during processing. Bandwidth considerations require this RAM to be independent. For systems with limited off-chip RAM it may be more pertinent to use on-the-fly display generation for the GATOS GUI.

The frame buffer approach also complicates interaction with widgets and may require specialized memory resources and architectures. As the mouse is moved we need to be able to determine which widget is being interacted with, requiring random access to the frame buffer. The difficulty arises because of a resource conflict as multiple processes attempt to access the shared frame buffer. The interaction detection process may conflict with the frame buffer update process leading to the use of specialized and expensive multi-port RAM.

For image processing applications, the ability to zoom in/out of an image window would be a useful feature. Buttons to accomplish this could be placed on the window title bar. Zooming introduces scheduling issues as line buffering is required in order that each pixel in the enlarged image is repeatedly displayed for a period of time that is proportional to the magnification factor.

3.3. Character generation

Text annotation is an essential element of GATOS and therefore an efficient approach for representation and display of text is required. Text can be stored at a low-level in bitmap (pixel data) form and read directly out of the frame buffer. However, as text may be dynamically updated during run-time it is inefficient and costly to store text as pixel data. A more efficient approach is to represent text as a series of ASCII codes that are interpreted to pixel data at the appropriate location on the screen on-the-fly.

4. Summary

FPGA-based systems configured for stand-alone operation are becoming increasingly popular for implementing real-time image processing applications. However, the absence of a host operating system makes tasks such as tuning, calibration, user interaction, and debugging more complex.

Our proposed operating system, GATOS will mitigate these problems by providing a set of high-level IP blocks that implement the graphical user interface of a windowing operating system. The described system focuses on both intuitive user requirements and efficient implementation of desired functionality.

This paper presents our preliminary design for GATOS. We have implemented fixed position windows, rudimentary buttons, slider widgets arranged in a fixed array and block-based character generation.

5. Future work

The next step is to implement an efficient method for organizing and displaying multiple overlapping and repositionable windows. The design must be refined for multiplexing widgets in arbitrary configurations and between multiple windows. Additionally, the character generator must also allow pixel level positioning rather than at the block level.

6. Acknowledgements

The authors would like to acknowledge the Celoxica University Programme for generously providing the DK3 Design Suite.

7. References

[1] A. Downton. and D. Crookes, “Parallel Architectures for Image Processing,” *IEE Electronics & Communication Engineering Journal*, vol. 10, pp. 139-151, Jun. 1998.

[2] C. T. Johnston, K. T. Gribbon, and D. G. Bailey, "Implementing Image Processing Algorithms on FPGAs," *Electronics New Zealand Conference,* Palmerston North, New Zealand, pp. 118-123, Nov. 2004.

[3] W. Bohm, R. Beveridge, B. Draper, C. Ross, M. Chawathe, and W. Najjar, "Compiling ATR probing codes for execution on FPGA hardware," *Field-Programmable Custom Computing Machines* ,pp. 301-302, 2002.

[4] C. T. Johnston, K. T. Gribbon, and D. G. Bailey, "FPGA based Remote Object Tracking for Real-time Control," To appear in *International Conference on Sensing Technology*, Palmerston North, New Zealand, 2005.

[5] I. Alston and B. Madahar, "From C to netlists: hardware engineering for software engineers?" *Electronics & Communication Engineering Journal*, vol. 14, no. 4, pp. 165-173, 2002.

[6] System Level Design, Synopsys Inc., "SystemC Overview,"2001, www.systemc.org, visited on August 2004.

[7] T. Wiangtong, P. Y. K. Cheung, and W. Luk, "Multitasking in hardware-software codesign for reconfigurable computer," *Circuits and Systems* , vol.5, pp. V-621-V-624, 2003.

[8] K. T. Gribbon, C. T. Johnston, and D. G. Bailey, "A Real-time FPGA Implementation of a Barrel Distortion Correction Algorithm with Bilinear Interpolation," *Image and Vision Computing New Zealand*, Palmerston North, New Zealand, pp. 408-413, Nov. 2003.

[9] K. T. Gribbon, D. G. Bailey, and C. T. Johnston, "Colour edge enhancement," *Image and Vision Computing New Zealand*, Christchurch, New Zealand, pp. 297-302, Nov. 2004.