Genode FX: an FPGA-based GUI with Bounded Output Latency and Guaranteed Responsiveness to User Input

Norman Feske
Genode Labs
<norman.feske@genode-labs.com>

Matthias Alles
University of Kaiserslautern
<alles@rhrk.uni-kl.de>

Abstract

This paper presents a composition of hardware and software that forms a fully fledged windowed GUI on a single FPGA plus memory chip. The solution scales with the full range of Xilinx FPGAs. But even on the low-cost Xilinx Spartan3A FPGA, the GUI operating at a 16bit resolution of 640x480 is able to guarantee a bounded output latency of only 220 ms and a guaranteed response time to user input of less then 20 ms. The paper covers both an overview of our custom hardware design and our software algorithms that make these guarantees possible.

1 Introduction

Field-programmable gate arrays (FPGA) are becoming increasingly popular in embedded controlling devices and high-end measurement devices. In contrast to conventional digital signal processors and micro controllers, FPGAs allow for creating application-specific system-on-chip (SoC) solutions at moderate costs. Such SoC solutions are able to consolidate signal processing, peripheral components, glue logic, and even soft-core CPUs into one chip. Because FPGAs can be arbitrarily reconfigured, the functionality of the device can be updated without changing the physical hardware. This clears the way for the cooperative design of special-function hardware and software, and enables the implementation of sophisticated functionality into the embedded device, for example complex online analysis in addition to the aggregation, storage, and monitoring of measurement data.

The trend towards more complex embedded functionality motivates the use of graphical user interfaces (GUI) and touch screens instead of text-based LCD displays and buttons. For operating full-color pixel displays, there are discrete graphics solutions available, for example 32-bit micro controllers with an on-chip display controller. A natural alternative approach to placing such a discrete chip onto the PCB of an FPGA-based device is the integration of the GUI into the FPGA. With Genode FX, this paper describes one possible realization of this approach. Thereby, we have to deal with two major challenges. First, there are real-time requirements demanded by the applications. Controlling and monitoring applications often rely on the timeliness of displayed information. Also, an immediate response to user input is desired, in particular when using touch screens that provides no tactile feedback to the user. The second challenge is the severe constraint of resources such as FPGA slices, memory, and processing time.

The solution described in this paper addresses these challenges. Section 2 introduces our custom SoC design for implementing interactive graphics into an FPGA. The rationale behind the design of our real-time GUI software stack is detailed in Section 3. Section 4 describes the software implementation, validates the design, and reports the achieved performance on our custom SoC platform. It is followed by a revision of related work concerning quality of service on the GUI level in Section 5.

2 A system on chip for handling interactive graphics

The hardware part of Genode FX is based on Xilinx' Embedded Development Kit, which offers the designer a large degree of freedom when designing embedded systems on Xilinx FPGAs. Ready-to-use or custom cores can be parametrized and plugged together to form the system that best meets the designer's requirements. Such cores can be processors, interface controllers, timers, or embedded memory. Figure 1 gives a coarse overview of a Genode FX system. The system consists of

MicroBlaze CPU The MicroBlaze is a RISC-like soft-core CPU that can be customized at design time. It is possible to configure cache sizes, pipeline depth, floating-

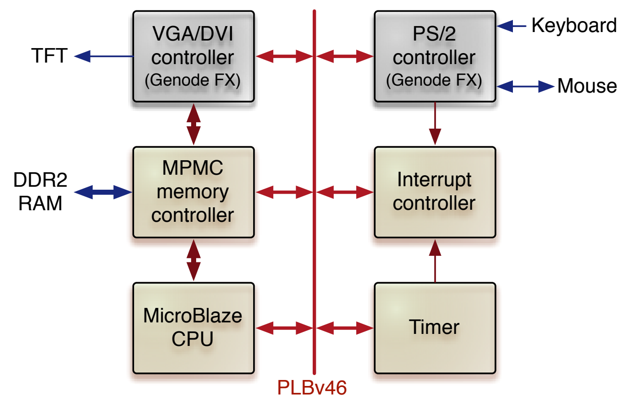

Figure 1: Relationship between the Genode FX hardware components.

point instruction support, and many other parameters. For MicroBlaze software development, the GNU compiler collection is available.

Timer The timer allows to periodically interrupt the processor.

PS/2 controller This custom controller is used to detect user input from the mouse or the keyboard. Whenever an event is detected, an interrupt is generated such that the CPU can process the PS/2 packet.

Interrupt controller The interrupt controller collects the interrupts from all interrupt sources in the system and reports them to the CPU. In Genode FX, the interrupt source is either the timer or the PS/2 controller.

Display controller Our custom VGA/DVI display controller offers the required frame buffer capabilities. It allows for a programmable video timing. The pixel clock, however, has to be chosen at design time. Pixel clocks of more than 100 MHz are possible, yielding high resolutions (e. g., 1280x1024) at a 16 bit color depth.

Multi-Port Memory Controller (MPMC) This core is the interface to the DDR2 main memory of the system. It allows multiple sources to access the main memory.

Processor Local Bus (PLB) All components are built around the PLB, which allows the MicroBlaze to configure the other cores and to access the main memory.

The MPMC allows multiple sources to access the main memory. In Figure 1, it is connected to the PLB, to the MicroBlaze, and to the custom display controller. The connection to the MicroBlaze is used to fill its cache lines in burst transfers without using the slower PLB. The display controller uses the connection to the MPMC to fetch pixel data in burst transfers. Arbitration between the memory-access requests is performed within the MPMC and can be customized. In Genode FX, the display controller has the highest priority to prevent the pixel-fetch buffer from becoming empty. All other requests are performed round robin.

Note that Genode FX is not limited to a specific FPGA family. The low-cost Spartan3 series as well as high-end Virtex4 or Virtex5 FPGAs are supported. The MicroBlaze processor can also be replaced by the hardwired high-performance PowerPC processors that is available in some Virtex FPGAs. Furthermore, Genode FX is not required to use DDR2 SDRAM via the MPMC. For early versions, we used a custom dual-port SRAM controller.

3 Designing the GUI as resource scheduler

This section presents the design of our quality-of-service-aware GUI software and explains the employed and novel key techniques. The description is not specific for Genode FX but meant to be equally applicable in the context of general-purpose operating systems. Therefore, this section fosters the use of the terminology found on such systems. We speak of the GUI server as the part of the software stack that manages and arbitrates the access to physical resources such as processing time, bus bandwidth, and input devices. It serves potentially many GUI clients, which represent concurrently running applications. The GUI server should manage physical resources in such a way that the temporal requirements of all GUI clients are respected. Furthermore, it must ensure a low response latency to user input. To meet these requirements, we need to model the GUI server as a real-time process that schedules and executes redrawing jobs.

To successfully plan ahead of time, a scheduler relies on the knowledge of scheduling parameters, in particular the execution time of each job, in advance of execution.

The classical approach for performing redrawing jobs, however, relies on a tight interplay of the GUI server with its GUI clients. To update a screen portion, the GUI server determines the set of GUI clients that are visible at the screen region and instructs each GUI client to redraw its visible portion. In turn, each GUI client responds to the request by invoking a sequence of graphical primitives composing the client's pixel representation at the GUI server. The set of graphical primitives as supported by the GUI server comprises for example the drawing of lines, the output of text strings, and the filling of polygonal shapes. Consequently, the time needed to update a screen portion depends on the number and type of graphical primitives as selected by each GUI client to produce its pixel representation. Hence, the GUI server cannot reason about the time needed to execute the involved graphical primitives ahead of execution. From the GUI server's point of view, the execution time of each redraw job is unbounded. Major commodity GUIs such as the Windows XP GUI, Mac OS (until version 9), and the X window system employ such a protocol.

3.1 Making worst-case execution times of redrawing jobs predictable

A method to dissolve the dependency of the GUI server from its clients during screen updates is to move each GUI client's graphical representation into the GUI server and thereby enable the GUI server to autonomously reproduce pixels out of the now server-locally stored model of the client representation. This model can be based on raw pixel data or on a higher-level abstraction such as a widget set including buttons, menus, and other basic GUI elements. Performing the transformation of each client's GUI model to pixels locally enables the GUI server to predict all graphical primitives that are needed for any screen update and, as a consequence, to estimate overall execution times of redrawing jobs in advance of execution.

The sequence of graphical primitives is a function of both the known transformation of the GUI model to pixels and the actual window layout. The latter, however, may have a significant influence on the required graphical primitives but is not known at admission time of a GUI client. This problem is best illustrated by the *painter's algorithm* as used by the Windows Vista's Desktop Window Manager, the Quartz window manager of Mac OS X, and the composite extension of the X window system.

Deficiency of the painter's algorithm As a painter, the algorithm produces the final image by painting all objects ordered by their distance from the viewer, starting with the rearmost (background) and finishing with the foremost (top window). In the final image, window portions that are covered by other windows get correctly over-painted and are no longer visible. Combined with the use of alpha channels, which is comparable with applying layers of watercolor with different translucencies to a canvas, this algorithm provides maximum flexibility with regard to the shapes of windows and their opacity. With regard to predicting redraw-execution times however, this algorithm is not well suited. Even though the sequence of graphical primitives used by the Painter's Algorithm can be determined immediately prior execution, we cannot predict a realistic redraw execution time for a specific GUI client at its admission time because the actual costs depend on the other GUI clients and on the window layout, which is not fixed during the lifetime of the GUI client. Therefore, the admission of new GUI clients is based on an overly pessimistic worst-case redraw-execution time assuming that all windows are covering each other and thus, must be painted for each redraw operation.

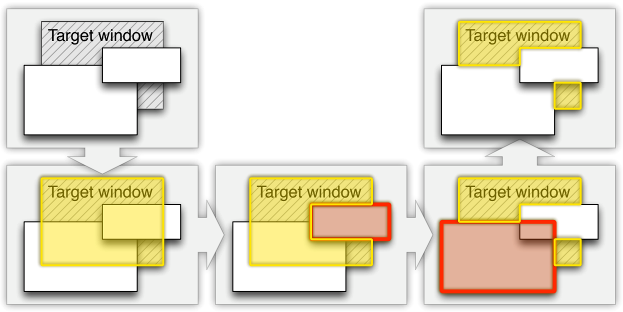

Figure 2: Determining the visible portion of a window by successively clipping the window's shape against each overlapping window.

Decoupling redraw-execution times of different clients To dissolve the inter-dependency of windows from each other during the redraw of one particular window (target window), the painting algorithm should limit its operation to only the target window but should not paint any other windows. This can be achieved by preceding the painting operation by a geometric analysis that computes the target window's visible portion. The visible portion is determined by successively cutting out the shape of each window in front of the target window from the target window's shape. Such a technique was originally employed by most window systems but current-generation commodity GUIs discarded this approach in favour of executing the painters algorithm via hardware-accelerated graphics.

Figure 2 illustrates this procedure. The resulting shape is then used as clipping boundary while painting the target window to mask out all pixels that are covered by other windows. The worst-case redraw-execution time for each window corresponds to painting the window when fully exposed and it is invariant toward the presence of other windows and the window layout (ignoring the computational overhead for the geometric analysis at this point). With known temporal characteristics of the single graphical primitives, this technique enables us to predict redraw-execution times prior execution and, therefore, base the admission of GUI clients on realistic worst-case execution times.

3.2 Local scheduling of redrawing jobs

With the satisfied precondition of known worst-case redraw-execution times, the construction of the GUI server as a periodic process clears the way for deploying the full variety of well-understood admission and scheduling strategies for such processes. In addition, the redraw-job scheduler can take the different characteristics of planned and spontaneous redrawing jobs into account. Once admitted for a defined window size and a fixed update interval, a planned job as used by real-time media applications corresponds to a classical real-time job with its deadline being implied by the update interval. A valid schedule for a set of planned jobs can be obtained by using a standard algorithm such as Earliest Deadline First (EDF).

In contrast, spontaneous jobs can be induced at arbitrary times by any GUI client posting an update of its GUI model or by user interaction. The occurrence of spontaneous jobs is not predictable. Once triggered, such a job does not have a deadline assigned but it should be processed as soon as possible (best effort) without affecting planned jobs. A classical best-effort GUI server handles spontaneous jobs only and executes each job when it arrives at a blocking synchronous client interface shared among all GUI clients. In contrast, our real-time GUI server receives spontaneous jobs via an asynchronous client interface, enqueues the incoming redraw-job requests into a redraw queue, and processes redraw-queue elements when the planned schedule permits execution. Because the GUI models for all clients are locally known to the GUI server, each redraw-queue element contains only the information about the corresponding window and the window portion to be redrawn but does not include graphical primitives.

3.2.1 Constraining priority inversion through artificial preemption points

Executing (low-priority) spontaneous jobs during the time left in the schedule of (high-priority) planned jobs, however, raises a priory-inversion problem because once started, a long-taking spontaneous job must first be completed before the next planned job can be executed. This delay introduces jitter and may corrupt the schedule of planned jobs. Thanks to the locally known model of all client's GUI representations as described in Section 3.1, we can apply a custom technique called redraw splitting to overcome this problem. Prior to the execution of a spontaneous job, we first estimate its execution time based on the knowledge of the graphical primitives needed for transforming the GUI model to pixels. If the spare time slot in the schedule is not sufficient to execute the spontaneous job completely, the job gets subdivided into smaller jobs addressing distinct screen portions of the original job in a way that each sub job's execution time fits nicely into a spare time slot in the schedule.

3.2.2 Redraw queueing

Due to the unpredictability of spontaneous jobs that can be issued by any GUI client at any time, the GUI server can be confronted with overload situations. For example, a terminal application may generate a large number of spontaneous jobs when scrolling through large text output. If the GUI server is not able to process graphics operations fast enough, subsequent jobs will stack up at the redraw queue of the GUI server and render the GUI inaccessible until all pending redraw operations are executed. Because this delay is unbounded and depends on the behaviour of the GUI clients, a malicious GUI client would be able to impose the denial of service of the GUI server.

The key for tackling an unbounded population of the redraw queue lies in the characteristics of redrawing jobs. If there exist multiple jobs in the redraw queue that refer to the same screen region, only the computational result of the most recent job is important whereas the intermediate states as produced by the other jobs get successively repainted. Consequently, such intermediate jobs can be discarded. Video players employ a similar approach for dealing with situations for which the available processing time is not sufficient for decoding all frames of a video stream. In such situations, intermediate frames get dropped to yield the remaining processing time to the most current frame. This technique provides quality of service by trading the smoothness of the video playback for the timeliness of the presented information and thereby prevents unbounded overload situations.

For the GUI server, we introduce an analogous technique that we call redraw dropping. In contrast to a video player that performs frame dropping at the spatial granularity of the whole video frame, a window system composes the screen of a number of potentially overlapping windows, for which redraw dropping can be applied individually. For each incoming redraw job, the GUI server searches for a pending job in the redraw queue that refers to the same window. If such a pending job exists in the queue, this job gets replaced by the compound of the existing job and the incoming job. If both jobs refer to distinct regions of the window, the resulting job will refer to the bounding box of the original job and the incoming job. This way, a once enqueued job for a particular window can successively be enlarged by incoming jobs while staying in the redraw queue. The maximum extent of the enqueued job, however, is limited by the size of its corresponding window. Consequently, the redraw queue's size is bounded by the number of windows present on the screen, which prevents the queue from overrunning. Furthermore, all redraw-queue elements refer to different windows and thus to distinct screen regions. Thanks to successive clipping as described in Section 3.1, the actual execution time of each job correlates to the visible portion of its window. The sum of the execution times of all enqueued jobs is bounded by the number of pixels on screen. Therefore, our algorithm enables the GUI server to inherently avoid overload situations and to guarantee a bounded worst-case latency for any graphical output on screen. This worst-case latency is the time needed to perform the redraw of the whole screen.

Our combination of redraw splitting with redraw dropping enables the GUI server to streamline redraw operations and input handling into one periodic process. It provides response-time guarantees for user input including visual focus feedback and processes the redraw of all GUI clients. The scheduling of redraw jobs is local to the GUI server and thereby enables the use of a wide range of scheduling strategies, for example by considering multi-threaded versus single-threaded operation.

3.3 Dealing with user interaction

The previous section presented how the characteristics of redraw jobs enable the scheduler to apply a specially tailored scheduling strategy leading to the prevention of overload situations by design. We can apply a similar technique to handling user input.

Pointer devices such as mouses or tablets sample user input at high rates (e. g., 16K bits per second for PS/2) and generate a flood of motion events during mouse or stylus movements. Each motion event is a spontaneous job that requires event handling in the GUI server. This includes translating device-specific coordinates to screen coordinates, moving the mouse cursor, determining the GUI element under the mouse cursor by traversing meta data, visually changing the GUI element on changed mouse-focus, and the routing of the event to the referred GUI client. Due to the cost of these operations, a steady supply of user input events at such a high rate can induce a high load to the GUI server and its clients. The user, however, is only able to perceive the resulting visual changes at a rate lower than 100 Hz. Consequently, for each perceived GUI state, the GUI server may have underwent intermediate states that are ignored by the user[1] but produce system load.

By turning event handling into a periodic mode of operation, the overhead for handling high-rate user input can be significantly reduced. Analogous to the redraw handling, the first step is the decoupling of job submission (an input device interrupt occurs) and execution (the GUI interprets the event) by introducing a first-in-first-out queue. Each time, an input event is generated by the input device, the interrupt handler enqueues the event into a device-event queue. Therefore, the insertion of device events happens aperiodic but at a known maximum rate, which dictates the required queue size.

At a low rate of 100 Hz, the periodic event-processing thread of the GUI server interprets the batch of device events currently stored in the queue. Due to the characteristics of motion events, the batch contains large sequences of motion-only events that can be merged to only one event by accumulating the motion vectors of successive motion events. Consequently, the resulting number of input events to be executed by the GUI server is bounded by the rate on which the user can supply non-motion events such as button press or release events. Typically, this rate is not higher than 100 Hz such that for each period, the GUI server must handle only a few (empirically ca. 0 to 3) input events that imply only negligible computational costs.

[1] The high temporal resolution of input events as supplied by pointer devices is required by only a few applications such as paint programs to accurately digitize brush strokes. For such applications, the GUI server should provide the raw stream of input-device events via a dedicated interface.

4 GUI software stack

The rationale as described in the previous section is the result of our extensive practical experiments using the DOpE real-time window server [4, 5]. The core of this window server forms the basis for the Genode FX GUI software stack. This section describes its most interesting properties and reports on the practical experiences made.

4.1 Widgets as server-side client representation

Section 3.1 highlighted the need for server-side client representations to enable the GUI server to process redrawing jobs independent from its clients and thereby make the job execution times predictable. The design space for a server-side model of a client representation ranges from pixel-based representations to high-level descriptions of the GUI elements (widgets).

By using a pixel-based representation shared between the GUI client and GUI server, the redraw functions in the GUI server are simple pixel-copy operations whereas the GUI client can freely express its visual appearance. This approach is used for example by the Mac OS X Quartz engine and the minimal-complexity Nitpicker GUI server [7]. The great flexibility for GUI clients and the simplicity of the GUI server, however, comes at the cost of a high memory usage. Each window requires an equally sized pixel buffer to store the representation, even when the window is fully covered by other windows. Furthermore, this approach requires a tight interplay between the GUI server and its clients for providing visual feedback to user interactions. For example, to highlight the GUI element under the mouse cursor, the GUI server has to provide mouse motion events to the GUI client, which, in turn, determines the GUI element at the mouse position, updates the corresponding part of the pixel buffer, and then notifies the GUI server to refresh the changed pixels on screen.

In contrast, when the GUI server implements the widget set, functionality such as mouse-over focus and window resizing can be handled locally in the GUI server without involving the GUI client. With regard to memory-resource usage, a server-side widget set is significantly more efficient because a typical semantic description of a widget consumes only a few bytes regardless of the actual size on screen. For example, for representing a button widget, the GUI server needs to store only its position, size, state, and the button text, which consumes significantly less memory than the corresponding pixel-based representation. The GUI server produces the pixel representation from the semantic model only if the widget is visible on screen and therefore provides a large potential for performance optimizations based on window layout. For example, if a client updates the text of a button, it pushes the new button property to the GUI server, which stores it locally. The transformation to pixels, however, is only performed if the button is not completely covered by other windows. If partly covered, the transformation costs are proportionally related to the visible portion. Further arguments in favour of a server-side widget implementation are fostered consistency and interoperability between GUI clients because the GUI server facilitates one common look and feel for all GUI clients. However, as proven by the GNOME and KDE projects, such properties can be provided by client-side libraries as well.

With DOpE, we explored the design range by providing a fully functional server-side widget set that also facilitates the use of pixel-based client representations by the means of a special widget type. The widget set consists of layout widgets, which organize a number of child widgets according to geometric rules, and leaf widgets, which represent the actual state of the GUI client. Therefore, the GUI model used for each GUI client is a tree of widgets of the following types. *Window* widgets consist of standard window controls such as a title bar and resize elements and manage exactly one child widget as its content. *Grid* widgets can arrange child widgets in rows and columns, whereas the size of the rows and columns are determined based on the geometric constraints of the child widgets and client-defined constraints (weights or fixes sizes). *Container* widgets enable the GUI client to freely position child widgets via pixel coordinates. *Frame* widgets can hold one arbitrarily-sized content widget, which can be larger than the frame's dimensions. In this case, the frame provides scrollbars to let the user freely choose the view port on the content widget. For expressing actual client state, DOpE provides labels, buttons, text entry fields, load displays, numeric scales, and scrollbars as leaf widgets.

DOpE's widget set is designed to enable GUI clients to realize more complex GUI elements by composing these basic widget types. For example, a tree widget can be realized by combining nested grids with leaf widgets. In addition to the already mentioned leaf widgets, DOpE provides a widget type called *vscreen* that enables GUI clients to use pixel-based representations shared with DOpE. Each

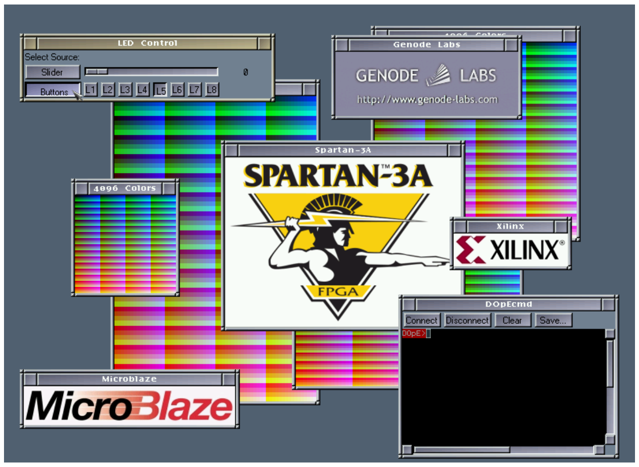

Figure 3: Screenshot of a demo application implemented with our custom GUI software stack.

vscreen widget has an associated pixel buffer. The normal mode of operation is that a GUI client writes pixels to the vscreen buffer and then notifies DOpE to update the changed part of the buffer on screen. The screenshot shown in Figure 3 displays the use of several widget types including windows, buttons, scrollbars, and scalable vscreens.

4.2 Application programming interface

The decision of using widgets as server-side GUI model was taken to broaden our experimental playground as much as possible and thus, to enable the exploration of a number of GUI-server-related problems beyond quality of service. One particular field of interest was the application of a domain-specific language as client API for a GUI server. In our previous work on GUIs, we observed that the use of high-level script languages such as Tcl/Tk can drastically reduce the GUI-related code complexity compared to the use of binary interfaces. Using such a high-level abstraction as client API of a GUI server raised interesting questions regarding the performance overhead on parsing textual commands, the costs of communicating textual strings instead of binary data between client and server, the complexity of the server-side support code, and the utility value and convenience of a language as API. DOpE clients communicate with the DOpE server by using textual such as

```
grid = new Grid()
button_1 = new Button(-text "OK")
button_2 = new Button(-text "Cancel")
g.place(button_1, -row 1 -column 1)
g.place(button_2, -row 1 -column 2)
win = new Window(-content grid)
win.open()
```

This sequence creates a window presenting two buttons arranged horizontally within a grid layout. Note that the GUIdescribing code is principally generic and does not contain resolution-dependent physical pixel values, font parameters, or style attributes. It is up to the GUI server to translate this semantic description of widgets and their topology to physical pixels in a way that fits the target device and the needs of the user best, for example by adhering the look and feel as configured by the user. Each command is handled by DOpE as a nonblocking atomic operation that returns immediately with either an error code or a success indication. In contrast to Tcl/Tk, which is a general-purpose script language, the DOpE command language does not handle control flow or conditional execution. This design facilitates DOpE's simple program logic with regard to its client API and keeps the complexity of the server-side command interpreter and the widget-support code for the textual commands at less than 1,500 lines of source code (SLOC). Unlike the X protocol, which transports a potentially high number of graphical primitives and pixel data from the client to the server on each redraw operation, DOpE's client API requires only few messages to communicate updates of its GUI model to the GUI server and effectively decouples the client and the server for the most of the time. After creating a window as done in the example above, the further handling of its rearrangement on screen (move, top, resize) and the management of mouse highlighting and keyboard focus are locally handled by DOpE and do not require any interaction with the client. The client gets involved only when an event occurs for which the client signalled interest beforehand. For example, to respond to the activation of a button by the user.

4.3 Resource scheduling

Each redraw request undergoes three stages that correspond to different abstraction levels. First, a redraw request is triggered by a client request (e. g., a client places a button into a window) or by user input (e. g., the user moves a window). This request refers to the targeted window and gets enqueued into the redraw queue by applying the redraw-merging technique described in Section 3.2. If multiple redraw requests targeting the same window are issued at a high rate, these requests get merged and reside as one request in the redraw queue. At the second stage, an independent redraw process transforms the window referenced by the redraw request to pixels. Enabled by the local knowledge of the GUI model for any window on screen, DOpE is able to generate the sequence of graphical primitives required to create the pixel representation. These graphical primitives are essentially the drawing of scaled images, the drawing of vertical or horizontal lines, and the output of text. DOpE provides these graphical primitives via software rendering that operates on a pixel back buffer in main memory. Its widget rendering engine is designed such that most pixels get touched only once during one transformation. However, there are cases for which one pixel gets subsequently written multiple times. For example, when drawing text on a button, the background of the button is drawn first and then partially overwritten by the textual label. During the third stage, the result of the transformation gets transferred from the pixel buffer to the frame buffer and thereby becomes visible on screen.

On Genode FX, the costs for transforming the widget-based GUI model to pixels is dominated by the memory bandwidth. Thanks to the clipping and optimization techniques of the widget-rendering engine, the number of memory accesses is roughly the same for each pixel (paint to back buffer, copy back buffer to front buffer). Therefore, we use a simple cost model as the basis for redraw scheduling, which derrives the costs of a redraw request from the amount of pixels that must be produced (request size) and the bandwidth at which pixels can be generated:

$$processing\ time = \frac{redraw\ request\ size}{bandwidth} \tag{1}$$

A further consequence of the approach is the inherent double buffering of graphical output that completely avoids displaying inconsistent GUI states that occur during the transformation of the GUI model to pixels. Because the mouse cursor handled in the second stage, it moves always smoothly at the rate of the periodic redraw process and is free from flickering artifacts. No hardware mouse cursor is needed.

4.4 Control flow between GUI client and GUI server

The original version of DOpE was used on a multi-tasking OS where multiple GUI clients interact with the GUI server via message-based IPC communication. This version used to employ two threads, an RPC interface thread for serving client requests and a redraw-processing thread. The RPC interface thread dispatches incoming client requests by operating on the server-side GUI model. Such operations perform only little updates of data structures and return immediately. As side effect of these operations, the redraw queue gets populated with screen regions that need to be updated. In contrast to the interface thread, the redraw-processing thread is executed strictly periodic. In each period, this thread handles user-input and processes redraw operations. Because DOpE employs the user input handling as described in Section 3.3, user-input handling causes only negligible costs. The redraw processing, however, is the primary consumer of both processing time and bus bandwidth and therefore, is subject to redraw scheduling within DOpE.

For Genode FX, we have turned the interaction between client and server into a single control flow such that the GUI server code is now a plain library to be linked with the GUI client. Because both the GUI server code and the client code are executed by the same thread, a Genode FX application can be implemented without the need for an underlying multi-threading kernel. But multi-threading can be used if available, for example via the xil kernel.

Each iteration of the main loop involves the following steps. First, the input-event queue as filled by the interrupt handler get processed. Each processed input event may influence the GUI model and thereby populate the redraw-request queue. But thanks to the user-input handling as described in Section 3.3, the worst-case execution time (WCET) of input handling is negligible. In the second step, the client's application code is executed via registered event-callback functions. Thereby, the application code may generate any amount of requests to update its GUI model. Because these updates are nonblocking and fast operations, however, the WCET of the client code does not depend on the redraw performance of the GUI server and can be estimated individually. The third step performs the expensive transformation of the GUI model to pixels. But thanks to our redraw-queueing technique, we have fine control over the WCET to spend for each iteration. For example, if we have a pixel throughput of 1 million pixels/second, we can limit the redraw processing to 10,000 pixels per iteration and thereby keep the WCET of the redraw processing below 10 ms. Thereby, the responsiveness to user input can be deduced from the sum of both the WCET of the application callbacks and the predefined WCET of the redraw processing.

4.5 Advanced features

In addition to the previously described redraw scheduling, DOpE's widget-based GUI model enables the implementation of advanced features such as partially translucent windows, drop shadows, and arbitrarily-sized windows while providing a worse but still bounded worst-case redraw processing time.

A straight implementation of such features would employ the painter's algorithm by drawing windows from back to front and properly incorporating each window's translucency values for painting pixels (alpha blending). Therefore, the processing time for such a redraw operation would correlate with the number of overlapping windows and is unbounded. DOpE's redraw engine functions differently by prepending the actual redraw operation with a geometric visibility analysis. For each pixel on screen, DOpE can determine the front-most window that contributes to its color value. Based on this information, it subdivides each redraw request into a set of fully exposed window areas and propagates a redraw request to each of these windows. The window, in turn, decides if the background of the window contributes to the window's pixel (alpha value is smaller than 1.0). If so, the window first issues a redraw operation for its used screen area to the windows that are visible through it as part of the window's background and then paints its foreground. Consequently, the redraw engine always paints from front to back and lets the actual widget for each layer decide to process another background layer (if the widget is at least partially translucent) or not (if the widget is opaque) before applying its foreground colors. As a consequence of this strategy, the costs of processing a redraw request comprising a number of translucent layers depends on the policy of each incorporated widget but it can also be bounded by limiting the recursion-depth of the background redraw processing. Imposing such a limit results in depth-limited translucency and bounded redrawing costs. Figure 4 displays the result of the depth-limited translucency algorithm for a limit of two translucent layers.

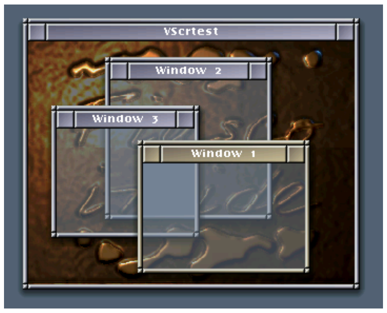

Figure 4: Depth-limited translucency and drop shadows.

4.6 Evaluation

As presented in Section 4.3, the use of software rendering with the data path to the frame buffer as the most significant performance constraint suggests a proportional relationship between the number of pixels to redraw and the redraw-execution time. To validate this claim, we conducted two experiments.

For both experiments, we use the Genode FX hardware described in Section 2 instantiated on the Xilinx Spartan3A Starter Kit. It requires 7,512 of 11,776 (63%) look-up-tables and 6,183 of 11,776 (52%) flip-flops available in the Spartan3A-700. The DDR2-SDRAM is connected to the MPMC using 16-bit data width and a clock of 100 MHz. The CPU core and the processor-local bus are clocked at 50 MHz. The display controller uses a pixel clock of 50 MHz. We have configured the MicroBlaze CPU with 4KB data and instruction caches. With enabled display, the memory throughput gained with a simple for loop operating on 32-bit values is 16.21 MB/sec for reading, 23.68 MB/sec for writing, and 13.24 MB/sec for copying.

The first experiment analyses how the pixel throughput is influenced by local GUI features. For each screen po-

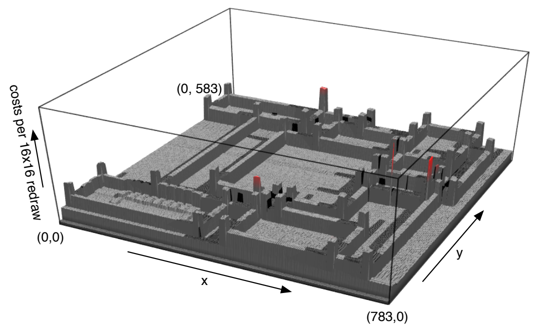

Figure 5: Influence of local GUI features on the pixel throughput. Each value is the average of four consecutive measurements. (800x600, redraw-operation size of 16x16 pixels, drop shadows disabled)

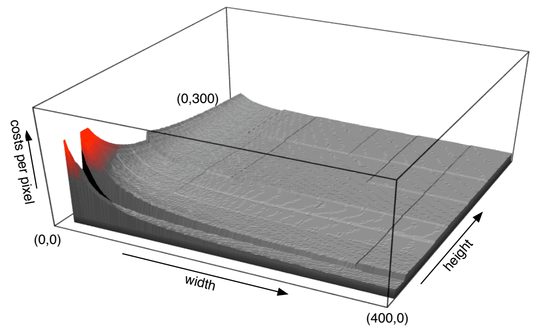

Figure 6: The pixel throughput as a function of the width and height of redraw operations. The data corresponds to the top-left quarter of the screen as depicted in Figure 3. Values for requests smaller than 8x8 pixels are not shown.

sition of the scenario displayed in Figure 3, we measured the time needed to perform a redraw operation of the surrounding 16x16 pixels. Ideally, we would observe the same costs for each screen position. In Figure 5, we can observe two effects. First, the costs for executing graphical primitives within windows is almost constant, regardless of the window's content. For example, the region at the left corner corresponds to the LED control panel whereas most of the other windows display a scaled image. The second observation is the overhead for the geometric analysis and for clipping. On window boundaries, multiple windows must be revisited, which increases the costs. In practice however, most redraw requests span regions than contain both hot spots (window corners) and cheap areas (window content) such that the overhead of the geometric analysis remains moderate. Still, we could construct pathological window configurations that will break our overly simple cost model.

The second experiment addresses the claim that the pixel throughput is invariant to the redraw-request size. For each position (x,y) of the top-left quarter of the GUI scenario, we performed a redraw operation of the rectangular area between the top-left screen corner and respective position (x,y) and measured the execution time divided by the redraw-request size x*y. The important observation of this benchmark is the nonuniform pixel throughput for very thin redraw operations as displayed in Figure 6. For such requests, the overhead for traversing the widget representation clearly dominates the costs. The second interesting observation are the clearly visible cache effects for redraw operations with a height smaller than 14 pixels. Although these anomalies hint at possible refinements of the temporal model, in practice, the simple model turned out to be effective for taking scheduling decisions. Thanks to our redraw merging technique, the corner cases of thin redraw operations occur mostly combined with bigger redraw jobs and do not dominate the overall performance.

On our hardware platform, we observe an average pixel throughput of 1.4 million pixels per second. Because the upper limit of pending pixels to redraw is limited by the size of the screen, the maximum latency of serving any GUI request is the time needed to update the whole screen. For a 800x600 screen, we achieve a worst-case latency of 344 msec (480,000 pixels / 1.4 million pixels per second). For a 640x480 screen, the worst-case latency is 220 msec. With the known pixel throughput, we can further choose an appropriate pixel-limit parameter for the redraw processing. To achive a responsiveness to user input of 20 msec, we limit the redraw processing to produce no more than 28,000 pixels per iteration (20 msec * pixel throughput). The pixel-limit parameter can either be hard wired or obtained by measuring the pixel throughput at runtime.

5 Related Work

There exists surprisingly sparse work by the real-time community taking the special characteristics of graphics operations on GUIs into account. We consider Artifact [9] as the most significant contribution in this domain. Artifact is a real-time window system built in 1995 on RT Mach. It differentiates between real-time clients and non-real-time clients. Real-time clients can only use graphical operations with known execution times and must provide a client model for the use of these primitives. In the GUI server, real-time and non-real-time requests are processed by independent threads, which concurrently access the frame buffer and are executed at different priorities. Artifact has no server-side knowledge about client representations but immediately reacts upon graphical commands supplied through client-interface invocations. Consequently, the Artifact GUI server cannot perform redraw-dropping techniques. In contrast to Artifact, our approach does not require temporal models of client behaviour because it performs the transformation from client representations to pixels locally.

The traditional approach to achieve fluent media playback on general-purpose PC hardware is the use of hardware overlays that effectively remove the GUI server from the latency-critical data path between the application and the hardware. A generalization of this technique that moves the composition of the screen from multiple independent pixel buffers into the hardware is described in [3]. The proposed hardware introduces a programmable per-pixel indirection for pixel-read operations performed by the output unit of the graphics device. For each pixel to be displayed in screen, a *Frame-Selection-Vector* table, exclusively accessible by the GUI server, contains an offset value to be added to the current pixel address when outputting the corresponding pixel. With such a hardware in place, the scheduling policy of the underlying operating system would be directly applicable to graphics. To our knowledge however, this technique was never implemented.

In 1995, another approach for creating custom display hardware with QoS support was conducted in the context of the Nemesis project. Nemesis [6] is an OS architecture specialized for distributed multimedia applications. The DAN Framestore [8] is a frame-buffer device that is capable of arbitrating the access to the physical frame buffer for up to 256 ATM-based stream connections with individual QoS properties. To spatially isolate the different clients on screen, the DAN Framestore performs key-based clipping protection. For each pixel, the device maintains an additional tag value. In contrast to the frame buffer, the tag buffer is only writable by a privileged software component. Each client stream has a unique stream ID. When a client stream issues a write operation to a particular pixel, the DAN Framestore performs the pixel-write operation only if the client ID equals the tag value of the targeted pixel.

Both frame-selector vectors and tagged pixels share the idea of storing the arbitration policy per pixel and thereby ease the multiplexing of different clients based on their pixel-based representation. In contrast, our work takes the transformation of the client's GUI elements (widgets) to pixels into account.

6 Conclusion

This paper presented a custom hardware and software to realize windowed GUIs on low-cost FPGAs. Thereby, we payed special attention to the constraints of such platforms, in particular the memory bandwidth, processing time, and the simplicity of the hardware. Our key contribution is the decoupling of the application code from the redraw processing. The application code operates on a high-level model of the GUI whereas the redraw process translates this model to pixels in a well-controlled manner and can thereby apply optimizations and scheduling. This design clears the way for both bounded output latency and guaranteed responsiveness to user input.

Both our hardware designs and our GUI software stack are freely available as open source [1] and can be deployed on a wide range of FPGA platforms including Spartan3, Virtex4, and Virtex5. For example, Genode FX is used on the Virtex5-based rapid prototyping platform described in [2]. There are many future challenges we want to pursue. We expect a significant performance gain from micro-optimizing the low-level pixel operations. Furthermore, we consider implementing hot-spot functions into our SoC hardware.

References

[1] Genode FX website. URL: `http://fpga-graphics.org`.

[2] M. Alles, T. Lehnigk-Emden, C. Brehm, and N. Wehn. A Rapid Prototyping Environment for ASIP Validation in Wireless Systems. In *Accepted for publication, eda-Workshop09, Dresden, Germany*, May 2009.

[3] J. Epstein. A High-Performance Hardware-Based High Assurance Trusted Windowing System. In *Proceedings of the 19th National Information Systems Security Conference*, Oct. 1996.

[4] N. Feske and H. Härtig. Demonstration of DOpE — a Window Server for Real-Time and Embedded Systems. In *24th IEEE Real-Time Systems Symposium (RTSS)*, pages 74–77, Cancun, Mexico, Dec. 2003.

[5] N. Feske and H. Härtig. DOpE — a Window Server for Real-Time and Embedded Systems. Technical Report TUD-FI03-10-September-2003, TU Dresden, 2003.

[6] D. M. Ian Leslie, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas in Communications, 14(7)*, Sept. 1996.

[7] Norman Feske and Christian Helmuth. A Nitpicker's guide to a minimal-complexity secure GUI. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*, 2005.

[8] I. Pratt. User-Safe Devices for True End-to-End QoS. Proceedings of the 7th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 97), 1997.

[9] J. E. Sasinowski and J. K. Strosnider. ARTIFACT: A platform for evaluating real-time window system designs. In *IEEE Real-Time Systems Symposium*, pages 342–352, 1995.