The Journal of Supercomputing, 26, 239–257, 2003
© 2003 Kluwer Academic Publishers. Manufactured in The Netherlands.

Context Switching in a Run-Time Reconfigurable System

KIRAN PUTTEGOWDA kiran@vt.edu
Bradley Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061-0111

DAVID I. LEHN dlehn@vt.edu
Bradley Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061-0111

JAE H. PARK jhpark@vt.edu
Bradley Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061-0111

PETER ATHANAS athanas@vt.edu
Bradley Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061-0111

MARK JONES mtj@vt.edu
Bradley Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061-0111

Abstract. A distinguishing feature of reconfigurable computing over rapid prototyping is its ability to configure the computational fabric on-line while an application is running. Conventional reconfigurable computing platforms utilize commodity FPGAs, which typically have relatively long configuration times. Shrinking the configuration time down to the nanosecond region opens possibilities for rapid context switching and virtualizing the computational resources. An experimental context-switching FPGA, called the CSRC, has been created by BAE Systems, and gives researchers the opportunity to explore context-switching applications. This paper presents results obtained from constructing both control-driven and data-driven context switching applications on the CSRC device, along with unique properties of the run-time and compile-time environment.

Keywords: configurable computing, FPGA, run-time reconfiguration, virtual hardware, context switching, multi-context

1. Introduction

The configuration time for an FPGA is the amount of time required to configure the device. Traditional FPGAs have relatively long configuration times [1], and applications that use run-time reconfiguration (RTR) on FPGAs often suffer from this effect. For applications that depend upon on-line modifications, any configuration delay increases the overall computation time. At any instant of application execution the computational logic implemented for the application may

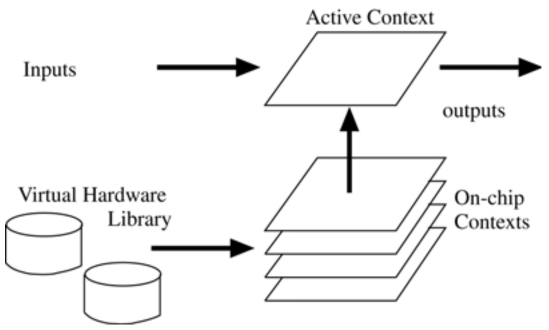

Figure 1. Concept of context switching.

not be used completely. In an application that requires only a part of the computational logic at a given time, there is the possibility of time sharing computational resources.

The devices used to share hardware store multiple configurations in different sets of internal RAM as shown on Figure 1. Such configurations are called Contexts. Each programmable part of the device is controlled by multiple RAM bits, and only one RAM bit configures the programmable unit at a given time. So the device holds more than one configurations on the chip and closer to the programmable unit. Global context lines act as addresses for the RAM bits to activate a particular context. A single on-chip context is activated at a given time. Any other context can be activated in as fast as one clock cycle. This internal configuration activation process is known as context switching. Devices with these properties are known as multi-context devices. Systems with these properties have a number of advantages over traditional programmable logic like ability to implement efficient virtual hardware, simplified routing, and higher data bandwidth.

The experimental platform used in this research is the reconfigurable computing module (RCM) that enables the technique of context switching reconfigurable computing. A detailed description of the platform is given in Section 2.1. This board has two experimental multi-context devices called the context switching reconfigurable computer (CSRC).

The RCM board is controlled by the host via a basic application programming interface (API). This API communicates with a basic operating system running on an onboard PowerPC, which handles communication with the host API and off loads some of the processing work from the host to the RCM board. To provide control for context switching applications without host intervention it is necessary to put control logic closer to the context switching hardware. On the RCM there is a Xilinx XC4085 FPGA [1] that can be used for this purpose. The FPGA is used for miscellaneous control functions for the system and contains a fully host-programmable table-based finite state machine (FSM) for controlling the applications.

Reconfiguration in an RTR system can be data-driven or host-driven. Dynamic recon- figuration driven by the data generated while processing online is called data-driven RTR. Static reconfiguration by the host machine controlling the reconfigurable hardware is known as host-driven RTR.

To demonstrate the application of context switching in such RTR systems three applications will be presented. The first application shows host-driven context switching with a demonstration of basic video processing. The second application uses hardware-driven context switching. This can be used to switch between known contexts sequentially or used in a data-driven reconfigurable application. A motion detection algorithm is discussed which can be broken into parts to be loaded on different contexts. These are activated in sequence as virtual hardware by a hardware based programmable state machine. The third application shows purely data driven context switching. It uses an implementation of the Enigma encryption algorithm to process data streams. Each stream is tagged for a certain user and each context contains optimized configurations for processing each user stream.

A brief background on context switching hardware and its implementation used here will be discussed in Section 2. Section 3 will be an explanation of the run-time environment used to support the use of context switching hardware. Sample applications will then be discussed in Section 4. The concepts for evaluating the performance is presented in Section 5, the results of which will be discussed in Section 6. Conclusions arrived upon based on these results will be presented in Section 7.

2. Background

Hardware systems that swap configurations into and out of a programmable device during runtime are termed virtual hardware. Approaches to words achieving this with minimum latency has been primarily through partial reconfiguration and context switching. Xilinx developed XC6200 FPGA [2] a device in which the configuration registers are memory-mapped. This provided many features such as partial reconfigurability and the ability to configure bus-mapped registers on the array. This device was used to develop a number of RTR systems using the virtual hardware model. Brebner introduced the swappable logic unit (SLU) [3] for virtual hardware that was analogous to pages or segments in virtual memory systems. In terms of hardware the SLU is a logic unit with fixed size and fixed interfacing signals. Software would view the SLU as a function, sub-routine or operations provided by an object class. The operating system supplies the routing between SLUs. This work proposes to make the capabilities of SLUs available as library functions that can be called from software programs.

The concept of virtual hardware was actually introduced by Ling and Amano [4]. The work was based on a multi-context device. The terms hardware page for the stored context and preloading for the process of loading a context before it is actually used for computation were introduced by Ling and Amano [4] in this work. WASMII [4] is a data driven computational system that uses virtual hardware concepts based on a multi-context device to visualize an infinite hardware for applications. WASMII was later implemented on a multi-context device called the dynamically reconfigurable logic engine (DRLE) developed by NEC [5]. During the same period the concept of a dynamically configurable gate array (DPGA) [6] was proposed by Bolotski et al. [19] who also discussed context swapping within an FPGA. DPGA [6] was a similar approach towards rapid RTR. The DPGA system programs several different sets of configuration RAMs for each logic function. The appropriate configuration set could be selected at run-time using global signal wires to select the appropriate context at any given time. Logic values for the function would be taken from a small RAM, and the global context control wires would act as the address used for that RAM. In a later paper, reconfigurable accelerators built by placing DPGAs on the same die as a normal processor [7] were presented. Xilinx [8, 9] filed a patent on the multi-context programmable device in 1995 and presented the work as a time-multiplexed FPGA [10]. The patented device has an architecture similar to the Xilinx XC4000E [1] with multiple configuration planes. The reconfigurable communication processor [11] developed by Chameleon Systems, Inc. has a reconfigurable fabric with two configurable planes; one for executing while the other configures the next part of the application. Scalera and Vásquez presented the context switching reconfigurable computing device in Scalera and Vázquez [12]. This device has been used to implement a prototype research platform called the reconfigurable computing module.

2.1. *Reconfigurable computing module (RCM)*

The RCM board designed by Sanders (now BAE Systems) is a PCI card that houses the CSRC chips and other support hardware used to demonstrate context switching. The basic architecture is shown in Figure 2. It consists of a Power PC 750 microprocessor.

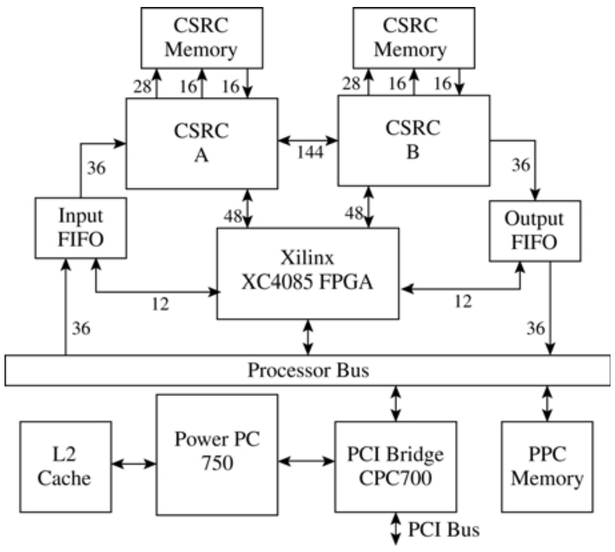

Figure 2. The RCM board block diagram.

The RCM board contains two CSRC devices with private memory. The two CSRC devices are directly connected to each other with 144 wires and there are 48 wires from each CSRC to a support FPGA. The processor to CSRC communication is primarily through the processor bus connected to 36-bit wide FIFOs. The assumed data flow is from the processor through the CSRC A to CSRC B and back to the processor. The Xilinx XC4085 support FPGA is intended to provide a variety of support functions. The FPGA contains the ability to receive interrupt requests from the host processor, manage FIFO control flags, program the CSRC devices, and serve as a DMA controller to move data to and from the CSRC devices.

2.2. *CSRC architecture*

The context switching reconfigurable resources available on the RCM board is the context switching reconfigurable computer [12]. FPGAs approach higher speedups when implementing algorithms with deep pipelines. However, generating pipeline control signals, implementing state machines and interfacing with external RAM or other integrated circuits require bit-wise programmability. The CSRC device architecture has a 4-bit DSP dataflow engine that is also capable of implementing control logic efficiently.

Figure 3 taken from Scalera and Vázquez [12] shows the microarchitecture of CSRC, which consists of 16-bit wide data pipes. Each pipe consists of context switching logic arrays (CSLAs). A single CSLA consists of context switching logic cells (CSLC) and is capable of processing two 16-bit words and producing a 16-bit result. The result of one CSLA is available as input to two adjacent CSLAs in the

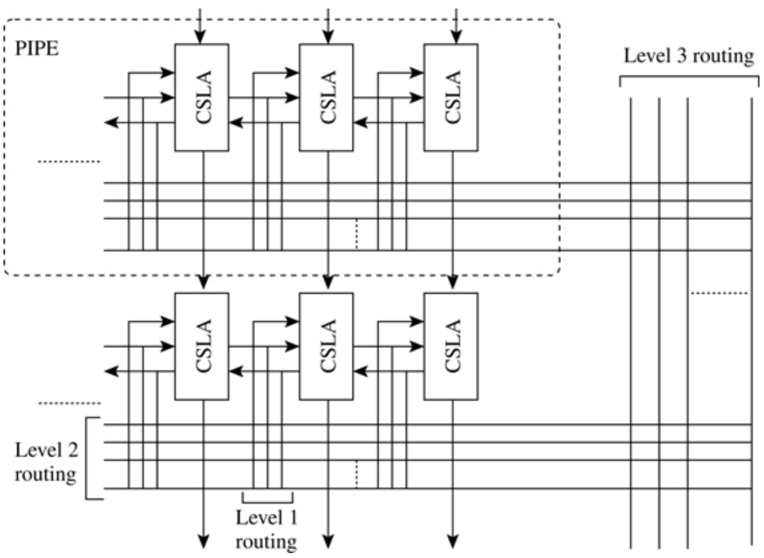

Figure 3. CSRC architecture.

pipe. Thus, a pipe can be used as a data path in both directions. This is particularly useful when a part of the algorithm is implemented as a pipe in one context where data flows from left to right; and the next context can take the result from the previous context and processes it from right to left to implement the next part of the algorithm. The advantage of such a data flow mechanism is that it eliminates the need to reroute data from its physical origin in one context to its physical input in the subsequent context.

The CSLC is composed of a four input lookup table (CSLUT), a context switching flip-flop (CSFF) a tri-state buffer and the carry logic. The carry logic is such that the carry chain can be connected, disconnected or fed a logic zero or one every four bits. This enables a pipeline granularity of 4-bits. Each configurable resource in the CSLC, along with each routing resource, has four configuration bits among which a single bit is selected as the current configuration; thus achieving four configuration planes. The CSRAM implements the global sharing scheme used for sharing data between contexts. Another way of sharing data between contexts is by the private/public addressable registers. Each CSLC has a private register for each context and a public register. During a context switch the CSLC value is stored in public register if it is to be shared or kept in a private register if not. When a context is activated it can select between its previous value in the private register and the shared public register value.

3. System

Extensive run-time support is required to access the hardware resources on the RCM platform. The RCM platform has a run-time environment that is specific to its hardware [13]. The system environment for run-time support is described in considerable detail in this section. The run-time environment consists of a layered stack of software. This allows the user to develop applications using the context switching features of the system. Figure 4 illustrates this run-time stack.

Memory mapped reads and writes over the PCI interface is used for communication between the RCM platform and the host. Application interfaces were built on top of the memory-based communication.

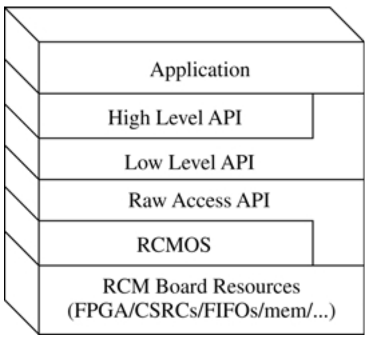

Figure 4. Run-time environment stack.

3.1. Host APIs

Hardware resources are accessed by the application either through a high-level application programming interface (API), low-level API, or a combination of both. The low-level API is implemented using the "Raw Access API" which provides only basic services such as opening/closing the board, and reading/writing memory-mapped areas.

The raw API has no knowledge of the hardware configuration and programmable resources on the board. It does not access special features of the software running on PowerPC or any features of configurations loaded in the CSRCs or the FPGA. The initial setup of software on the PowerPC is done with this raw API, accessed through a higher level API, and board firmware. The memory controller on the board is configured to map memory ranges to various devices. This is used to provide some direct access to the FIFOs and FPGA. Complex communication and control is implemented with a memory-based handshaking protocol with software on the Power PC. The "Low-level API" uses this protocol to provide transparent access to the hardware from the host application. The features of this API include PowerPC configuration, FPGA configuration, CSRC configuration (basic and caching), CSRC context switching, and data streaming.

This low-level API is implemented through a basic C API and the ACS API [14]. The C API is suitable for high speed direct access. The ACS API is an interface for distributed adaptive computing systems. It allows the board to be accessed in a distributed dynamic network of heterogeneous reconfigurable hardware. In Figure 4 both these APIs are represented by the "Low-Level API" layer. The Xilinx FPGA is used to control the signals on the board. It manages the CSRC clock, reset, context switching control, programming control and FIFO status and control. Many CSRC connections also go through this FPGA. The low-level API does not have information about how these control is implemented. It simply maps the FPGA with address, data and control lines. Specific control can be implemented either by the application programmer or another layer of API using the lowlevel API calls.

The high-level API provides an object-oriented view of the board's physical parts. This layer is based on specific features of the FPGA configuration as well as the low level API features implemented in the RCMOS. This API includes the functionality needed to take full advantage of the hardware. This provides a high level view of the system such that applications using it are isolated from many of the details of register access and bit manipulation. Thus, complex functionality involving many low level API calls can be wrapped up into an easier to use interface.

3.2. RCMOS

The PowerPC on the board allows flexible programs to be run close to the context switching hardware. It is used to implement a lot of functionality in the system. All this is achieved through a mini-operating system called the RCM operating system (RCMOS). It is used to implement many of the low level API functions in an efficient manner. Programming the configurable hardware resources would involve bit wise data writes which can be slow over the PCI bus. It is achieved in an efficient way by loading the configurations into the board memory and letting software in the powerPC do the work. Many of the CSRC control operations such as control-driven context switching and clocking are also efficiently implemented as software in the PowerPC so that it runs closer to the hardware than across the PCI bus.

3.3. Configuration caching

To achieve effective run-time reconfiguration fast switching between configurations is required. In traditional configurable systems, configuration switching latency is large due to long configuration time and communication latency over the system bus. The CSRC device with four on-chip contexts provides fast configuration switching. For applications requiring more contexts than the number of contexts present on the device, the device is reconfigured during application execution. The average reconfiguration time in such a system is discussed in Section 5. To eliminate the communication latency over the system bus configurations are stored on the PowerPC memory. But it has limited capacity when compared to configuration storage on system memory. Configuration caching was implemented as a virtual hardware hierarchy similar to the concept of memory hierarchy. The intention was to exploit temporal locality in the configuration sequence.

Figure 5 illustrates this configuration caching hierarchy. The on-chip device configurations are considered as the high speed and low capacity level. It is on top of the hierarchy. The lowest level of the hierarchy is the configurations stored in mass storage. This storage can be a host hard disk or remote storage accessed over a network. Access speed increases and capacity decreases as the configurations move from mass storage to host memory to RCM board memory to the device contexts. If a particular level cannot hold the required number of contexts, they are stored in the next higher capacity level and are loaded on demand. The host uses an API that stores configurations on board via the RCMOS for board level configuration caching. The RCMOS keeps track of configurations loaded into contexts on the device. The application gives a high level request for a particular configuration to be loaded. If the requested configuration is currently loaded in a context, then switching is fast. If the requested configuration is not currently on a device context, then

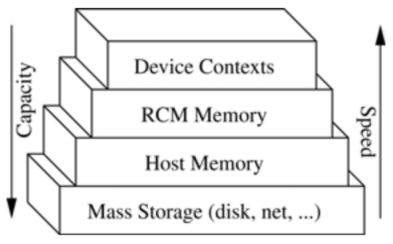

Figure 5. Configuration caching hierarchy.

RCMOS uses standard replacement algorithms to determine the context to be replaced with the requested configuration.

4. Applications

The context switching hardware system can be utilized in a variety of ways for building applications. The hardware system, along with the run-time environment, can efficiently implement applications that are either data-driven or control-driven run-time reconfigurable. Customized applications can make full use of the available resources. Application framework environments [13] can also target such a system.

4.1. Application frameworks

The infinite virtual hardware characteristics of context switching reconfigurable hardware make it attractive for users of some types of application frameworks. One such framework is Janus [15]. This framework takes a high level description of an application as input. Along with a description of the target hardware the application is temporally partitioned. A run-time environment then executes the application pieces in hardware. Janus can partition the application in any optimal way to adapt to the hardware. It must only respect ordering and resource requirements specified in the application description.

This approach works well on reconfigurable computing because the hardware can be reconfigured for each piece of the application. However, there is a significant performance degradation due to reconfiguration time. Depending on the application framework run-time implementation, hardware characteristics, and application properties; this can range from insignificant to overwhelming computational cost [15, 16]. Context switching platforms can accelerate application frameworks by improving performance of their virtual hardware characteristics.

4.2. Application control

Application on the CSRC need specific control logic which is implemented on the RCM board by logic programd on the Xilinx XC4085. This interface logic can be embedded as an application specific logic programd into the Xilinx XC4085 FPGA. This approach leads to a specific programmable bitstream for each application. Control logic needs to be implemented in a more generic and flexible way by making it controllable from the host application. One such approach is a host programmable finite state machine (FSM). It is implemented as a state table-based design as shown in Figure 6. The state table is implemented as a memory, with the present state represented by the address for the data which stores the next state variables and the output values. Combination of this state variable and the inputs provides the address for the table. The output bits can be programd to be mapped to any line interfacing with the CSRC. An abstract description of the FSM is converted into this memory-

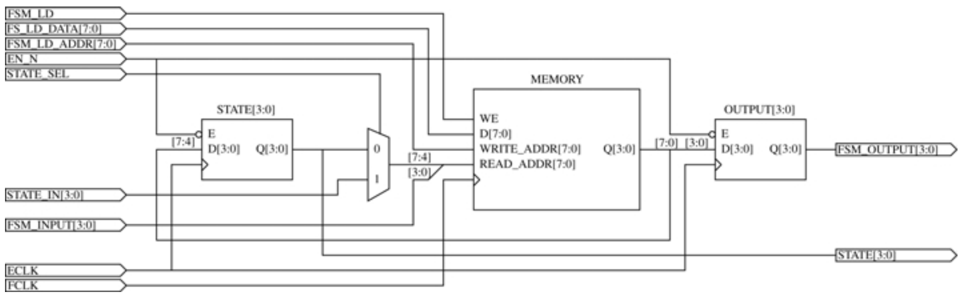

Figure 6. Finite state machine circuit.

based table format by the high level API. This is then loaded into the FPGA with a protocol of low level register writes. The prototype CSRC parts have limited logic and routing resources which require external control logic for some applications. The host programmable FSM allows flexible application control using high speed hardware available closer to the CSRC devices.

4.3. Context switching control

Context switching is controlled from a number of points in the system. Figure 7 illustrates the path on the RCM board through which the switching control signals pass. The control signals can be initiated at any of these points. The actual route is application specific. Applications requiring user input need the full control path from host to CSRC. Some applications can improve performance by moving this control closer to the CSRCs. The latency involved in a context switch is directly related to the number of layers that signals have to pass through. Latency for control from the host is large. For debugging this communication is used to stall the CSRC logic. Using such a control mechanism the context switching mechanisms implemented are discussed in the following sections.

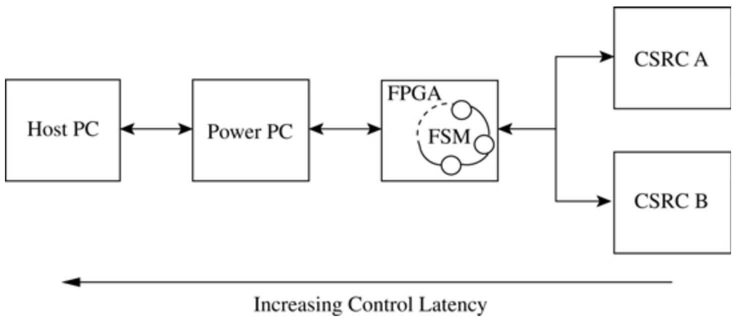

Figure 7. Context switching control routes.

4.4. Host-driven context switching

A continuous video stream is processed by filters stored in various contexts. A user request from the host causes the context to switch. This enables single-cycle algorithm changes. If more filters are needed than the available contexts on the device, then the configuration cache is used to store them before configuring them. Configuration of the inactive contexts takes place while another filter is running in the active context. Simple filters have been implemented on the CSRCs that include basic pass through, the motion detection difference filter and a delay filter.

4.5. FSM-driven context switching

In applications requiring external logic to achieve context switching, the host programmable FSM is programd to control the contexts on the CSRCs. Motion detection algorithm was implemented to demonstrate this FSM-driven control.

4.5.1. Motion detection algorithm. The motion detection algorithm [17] basically consists of capturing an image, processing it and sending out a cropped part of the image where there is motion. Such an application is useful for power critical remote sensing motion detection. The algorithm mapped onto the system is shown in Figure 8. The image is captured by the host machine and passed to the RCM board through the FIFOs. The four parts of the algorithm are implemented as four different contexts inside the CSRC. The FSM controls the switching of active contexts. The binary image generated by the CSRC is used to generate the cropped image by the host machine.

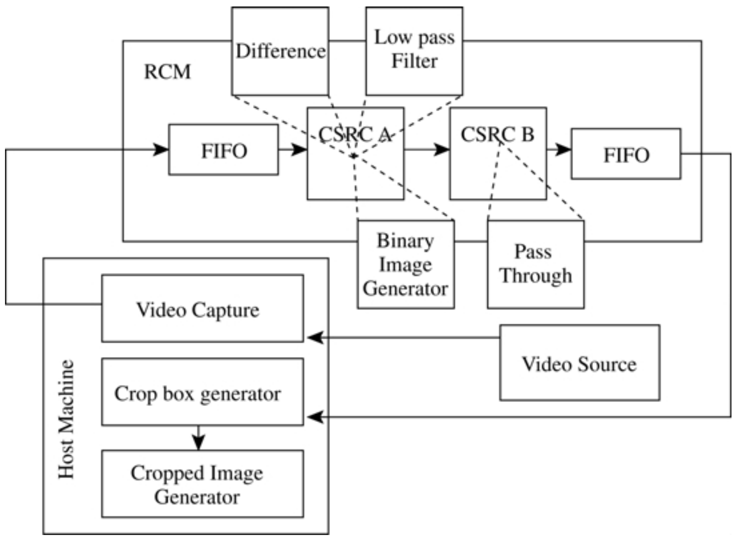

Figure 8. Motion detection application.

A video capturing card is used for generating a video stream. The video stream consists of 160×120 8-bit gray scale image sequences. The image data is stored in the sharable memory between the host PC and the PowerPC on the RCM board. The image is transferred into the CSRC through the FIFO with two pixels per word.

The difference block enhances the portions of the frame that have changed due to moving objects. It generates a difference image from two sequential images. Denote each image frame of the video stream as I_i, where i is the sequence index of the video. The previous image, I_{i-1}, is stored in the CSRC A memory. The current image, I_i, is streamed through the input FIFOs. The difference image, $|I_i - I_{i-1}|$, is stored back to the CSRC A memory. The CSRC memory shares the data between the contexts.

Ideally, the difference image should not have any spot noise and only the moving object on the image should be highlighted as long as the background remains constant. However, there are many factors that generate small background changes, such as wind, ground vibration, etc. A low-pass filter eliminates this spot noise and smoothes out the image. The low-pass filter is implemented as a 5×5 averaging filter that reads the differenced image from the CSRC memory.

If the intensity from one image to the next changes, the difference block will also produce non-zero pixels throughout the difference image. The pixel differences are compared with a threshold and non-zero pixels due to intensity variations are detected. Ideally a dynamic threshold value should have been generated. Due to the lack of resources on the prototype CSRC, a static hard-wired threshold is used instead of calculating the threshold. With enough resources, this block can be implemented as a separate context.

The binary image generator block compares the filtered image with the threshold and generates a binary image with a "1" for pixels above the threshold and a "0" for those below the threshold. It uses the filtered image stored in the CSRC memory, generates the binary image and, streams it out through the output FIFO. This binary image has only the objects that are moved without the spot noise and disturbances due to intensity variations.

This image can be used to clip only the part of the original image frame where there is movement.

4.6. Data-driven context switching

Data-driven context switching control is achieved by the data being processed. A network encryption application where the data packet to be processed contains the context number to be made active is implemented to demonstrate data-driven control.

4.6.1. Enigma encryption. The application is basically a demonstration of simple encryption and decryption of network traffic for multiple channels. As the resources on the prototype CSRC are limited, a simple Enigma-like encryption scheme was selected.

The Enigma Machine [18], built in the 1920s by the Germans and used in World War II, was based on a system of three rotors that substituted cipher text letters for plain text letters. The rotors spin in conjunction with each other, performing varying substitutions. A letter typed on the keyboard of the machine is sent through the first rotor, which shifts the letter according to its present setting. The new letter passes through the second and third rotor, where it is replaced by a substitution according to present settings of the second and third rotor. This new letter is bounced off of a reflector, and back through the three rotors in reverse order. As the plain text letter passes through the first rotor, the first rotor rotates one position. The other two rotors remain stationary until the first rotor rotates 26 times (one full rotation). Then the second rotor rotates one position. After the second rotor rotates 26 times, the third rotor would rotate one position. This principle of the shifting rotors allows $26 \times 26 \times 26 = 17{,}576$ possible positions of the rotors. To decode the message, the rotors are set to the initial settings, and then the cipher text is put through the machine. This gives the plain text back.

An encryptor/decryptor for bytes was built based on this concept. Each rotor has $2^8 = 256$ slots. The key and the shifting effects of the rotor are realized by adding the key and offset to the byte and obtaining its modulus for 256. The encryptor implementation is pipelined, so data returning through the rotors is implemented by repeating the rotors in the reverse order. The shifting of the repeated rotors is timed accordingly to match the shifting of the original rotor. The obtained byte is then scrambled nibble-wise using two 4×4 tables. The table entries represent the actual rotor settings.

Using the available RCM prototype board, the following system was implemented. FIFOs are used to stream data into the processing element. The processing elements consist of the two CSRCs and the Xilinx support FPGA. Output FIFOs are used to stream data out of the processing elements. The system design partition is shown in Figure 9.

Each of the four contexts on the CSRC B contains an enigma encryptor with a particular rotor configuration and key. The key can be made programmable using the data in the header. The CSRC A has a controller context on it that reads the header of the packet, determines the particular encryptor to be used, and the number of bytes in the packet. This number is stored in a down-counter register. It signals the support FPGA to set the particular context on the CSRC B. CSRC A's controller context now acts as a pass through and down-counts the number of bytes. After the

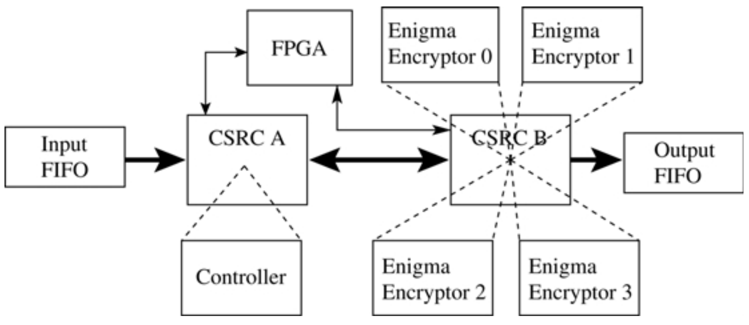

Figure 9. The Enigma application.

counter reaches zero, the controller context resets its controller and waits for the next packet to arrive.

5. Performance analysis

This section discusses the concepts for performance evaluation of the system. Further evaluations based on the applications are presented in Section 6.

5.1. *Average reconfiguration time*

Applications that require more contexts than the available number of contexts in the device store context configurations in external memory and load them on the device when required. The context switch time in such a system would have a significant latency if the context is not available on the device. Assuming that each context for the application has an equal probability of being requested during any context switch, an expression for the average reconfiguration time, which is the average time required to switch application contexts, is given by:

$$t_{\text{avg}} = \begin{cases} t_s & \text{if } 1 < n \leq k \\ p_s t_s + p_c t_c & \text{if } n > k, \end{cases} \tag{1}$$

where, t_{avg} = average switching time, t_s = context switch time, t_c = context configuration time, k = number of device contexts, n = number of application contexts, p_s = probability that context is on the device, $k - 1/n - 1$ and p_c = probability of a reconfigure, $1 - k - 1/n - 1$.

Figure 10 is a plot of the average reconfiguration time against the number of application contexts for different number of device contexts. The plot shows that for an increase in the number of device contexts(k) from one to four and from four to eight there is a significant reduction in the average reconfiguration time (t_{avg}). Adding a new context would involve adding new RAM cells for context storage, multiplexers and the necessary routing for each configurable resource. The combined area of these resources exceeds the area of the programmable part itself. So the VLSI area increases linearly with the number of contexts on the device. Hence there is diminishing returns of reduced reconfiguration time for more device contexts. It can also be seen that a match between the number of application contexts(n) and the number of device contexts(k) results in a shorter average switching time(t_{avg}). This is intuitive as most of the times an application context requested will be in the device in such a case. For very high number of application contexts the curves come closer. This implies that when there are many application contexts the advantage of having more device contexts to reduce average switching time is smaller. This plot illustrates a worst case situation with no locality-of-reference in the requested context sequence. With such a switch request, the probability that the context requested is on the device will be higher than $k - 1/n - 1$; thus the average context switch time would be lesser

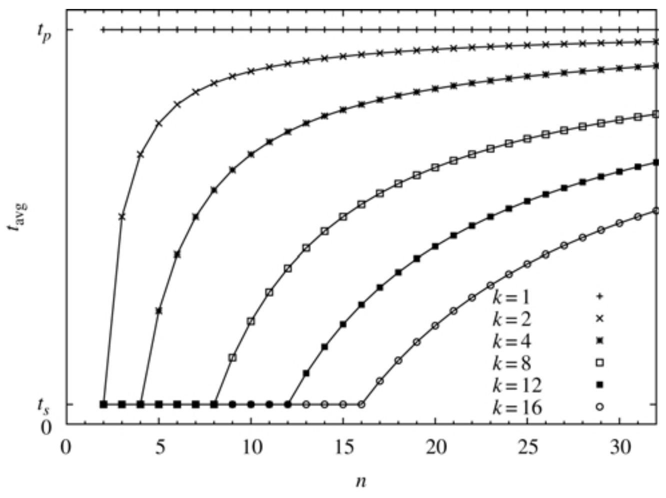

Figure 10. Plot of average reconfiguration time.

than that observed in the plot. This locality-of-reference depends on the applications and the way they are programmed.

A context can also be configured in the background while another context is processing data. This would reduce the effective context program(configuration) time and hence the average reconfiguration time. The new configuration time is given by Equation (1)

$$t_{\text{ac}} = \begin{cases} t_s & \text{if } t_c \leq t_{\text{exc}} \\ t_c - t_{\text{exc}} & \text{if } t_c > t_{\text{exc}}, \end{cases} \qquad (2)$$

where, t_{ac} = configuration time adjusted for background configuration, t_s = context switch time, t_c = context configuration time and t_{exc} = context execution time after the next required context is determined

For applications in which the next active context cannot be determined until the present context is executed, t_{exc} will be zero and Equation (2) reduces to $t_{\text{ac}} = t_c$. In applications with a known sequence of contexts to be made active t_{exc} will be the actual context execution time. If this is less than t_c, the average reconfiguration time is just t_s. This capability can be utilized in applications that can anticipate the next context required. With these techniques the reduced average reconfiguration time will help the total application execution time to approach the time required for processing data alone.

6. Results

An analysis of costs and benefits in terms of area and bitstream size for multi-context approach for run-time reconfigurable systems was done. This section discusses the

Table 1. Area requirement for each application. Second column shows the area requirement for each context of application in a multi-context device

Application	4-context implementation		1-context implementation
Motion	Difference	783.0	
Detection	Filter	1,188.8	2,055.0
Algorithm	Bin. image gen.	253.3	
	Enigma0	1,331.8	
Enigma	Enigma1	1,331.8	5,245.0
Encryption	Enigma2	1,331.8	
	Enigma3	1,331.8	

results of the analysis based on the applications mapped to the experimental context switching platform.

6.1. Area

Reusability of area for emulating infinite hardware is one of the strongest argument in favor of multi-context devices. To assess this claim, a study of area requirement for the applications was performed.

The gate equivalent areas obtained from Synplify for the applications implemented in a single-context and in the four-context device are shown in Table 1. The table shows the area counts in each context for the application in the four-context implementation. For implementing an application in a multi-context device, each context is expected to have an area count of the highest number among its individual context area requirement. Motion detection application can be implemented in a three-context device with a minimum of 1,188.8 equivalent gates without incurring any delay in reconfiguration. But with only a single context, the application requires a device with 2,055.0 equivalent gates. For the Enigma encryption we see that in a four-context implementation we require at least 1,331.8 equivalent gates for each of the Enigma engines. In a single-context implementation of all four Enigma engines bound together the application requires at least a 5,245.0 equivalent gate device.

Based on these results it can be inferred that for applications that can be partitioned equally among all the available contexts resource requirement will scale well with the number of contexts on the device. It should be noted that in this study the silicon real estate overhead of adding more contexts is not considered quantitatively.

6.2. Bitstream size

This section presents an analysis of the effect of context switching strategies on the bitstream sizes for the application. This directly affects the storage resource

Table 2. Area requirement for motion detection application for different number of application contexts

3-context implementation		2-context implementation	
Difference	783	Diff & filter	1,973.1
Filter	1,188.8		
Bin. img. gen.	253.3	Bin. img. gen.	253.3

requirements and configuration time. Table 1 shows that a motion detection algorithm implemented in three-contexts will have bitstreams for three contexts of a 1,188.8 equivalent gate device and the single context will have bitstream for a single 2,055.0 equivalent gate context. In the Enigma encryption application, storage requirements for the bitstreams are much more similar and scale well with the number of contexts. With this observation it might seem that context switching on a multi-context device leads to larger bitstreams. But if we consider larger applications that require run-time reconfiguration, the multi-context approach is scalable better than a single context device of the same capacity. This is observed by mapping the motion detection algorithm on devices with less than three device contexts. In a two context device the inactive context can be loaded with the configuration for the next part of the algorithm to achieve seamless execution. A similar capacity single context device would hold two contexts of the algorithm in its single available context. These two approaches are indicated in Table 2.

Table 2 shows equivalent gate requirements for the motion detection algorithm implemented in different number of application contexts. As bitstream size is dependent directly on the area, these numbers indicate the bitstream size required for each context. The first implementation shows a three context implementation with a highest area of 1,188.8 equivalent gates, so if the application is implemented in a device with 1,188.8 equivalent gates per context, the total bitstream size required will be for around $1{,}188.8 \times 3 = 3{,}566.4$ equivalent gates. The second implementation requires a 1,973.1 equivalent gate per context device and the total bitstream size required will be for $1{,}973.1 \times 2 = 3{,}946.2$, which is much higher than in the first implementation. Supposing more contexts are added to the image processing algorithm implementation, bitstream length scales at 1,188.8 equivalent gate steps in a multi-context device rather than 1,973.1 equivalent gate steps. This shows that having a greater number of smaller contexts is better in terms of performance if the application can afford the higher effective reconfiguration time.

7. Conclusions

Context switching is advantageous for applications that require only a part of the hardware at a particular time. Applications which efficiently map to such a device should also partition into parts that require almost the same area in terms of gate equivalents. To run such applications seamlessly, context switching must occur quickly. The CSRC device has the capability of switching context in a single clock cycle. The system achieves context switching with various degrees of speed and controllability. The system also supports the loading of a particular context on the system when a different context is processing data. In the case of run-time reconfiguration, this means that enough time is available to load contexts in the background. This feature, along with the hardware cache, implements an effective hierarchy of virtual hardware. Control of all these requires complex external support such as that provided by the Xilinx support FPGA in the RCM system.

The analysis presented here indicates that a multi-context approach for run-time recon- figuration helps effective virtual hardware implementation. Although the approach does not indicate significant savings in terms of silicon real estate when compared to conventional RTR approaches, it increases the application scalability of the RTR system. This can effectively help in development of more generalized RTR systems that can implement a variety of applications with reduced application development effort. This approach is advantageous when the application is divided efficiently for the resources.

Application are implemented in an effectively smaller area on a multi-context device than on a single context device. This has several positive effects on the overall system design. The routing inside the chip is reduced, because the effective area to which a design is mapped is smaller; thus interconnect delays are smaller. It would be interesting to study the reduction in computational load for a routing tool when the application is divided into smaller contexts. The possibility of performing partial reconfiguration on multi-context devices will add another dimension to the scalability of the system. It could be an interesting and a worthwhile research approach to explore in future.

References

1. Xilinx Inc. *The Programmable Logic Data Book*, San Jose, CA, 1999.
2. Xilinx, Inc. XC6200 Advance product information, 1996.
3. G. Brebner. A virtual hardware operating system for the Xilinx XC6200. In *Proceedings of 6th International Workshop on Field Programmable Logic and Applications*, pp. 327–336. Springer, 1996.
4. X.-P. Ling and H. Amano. WASMII: A data driven computer on a virtual hardware. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, April 1993.
5. Y. Shibata, M. Uno, H. Amano, K. Furuta, T. Fujii, and M. Motomura. A virtual hardware system on a dynamically reconfigurable logic device. In *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, April 2000.
6. A. DeHon. DPGA utilization and application. In *MIT Artificial Intelligence Laboratory, Transit Note 129*, September 1995.
7. A. DeHon. DPGA-coupled microprocessors: Commodity ICs for the early 21st century. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 31–39, 1994.
8. Xilinx. Time multiplexed programmable logic device. Patent no. 5646545, July 1997.
9. Xilinx. Configuration modes for a time multiplexed programmable logic device. Patent no. 5600263, February 1997.
10. S. Trimberger, D. Carberry, A. Johnson, and J. Wong. A time-multiplexed FPGA. In *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, April 1997.
11. Chameleon Systems, Inc. CS2000 reconfigurable processor. CS2000 Advance product information, 2000.
12. S. M. Scalera and J. R. Vázquez. The design and implementation of a context switching FPGA. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1998.

13. D. I. Lehn. Application framework for a context switching runtime reconfigurable system. M.S. thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia USA, April 2002.
14. M. Jones, L. Scharf, J. Scott, C. Twaddle, M. Yaconis, K. Yao, P. Athanas, and B. Schott. Implementing an API for distributed adaptive computing systems. In *Proceedings of the IEEE International Conference on Communications*, pp. 222–230, April 1999.
15. R. D. Hudson. *Architecture-Independent Design for Run-Time Reconfigurable Custom Computing Machines*, Ph.D. thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia USA, August 2000.
16. R. D. Hudson, D. I. Lehn, and P. M. Athanas. A run-time reconfigurable engine for image interpolation. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 88–95, April 1998.
17. N. Vaswani and R. Chellappa. Best view selection and compression of moving objects in IR sequences. In *Proceedings of International Conference on Acoustics, Speech, and Signal Processing Proceedings*, Salt Lake City, Utah, 2001.
18. Deutsches Museum. Enigma encryption machine. http://www.deutsches-museumbonn.de/ausstellungen/meisterwerke/2_3enigma/ enigma_e.html.
19. M. Bolotski, A. DeHon, and T. Knight. Unifying FPGAs and SIMD Arrays. In *Proceedings of the 1994 IEEE Workshop on Field Programmable Gate Arrays, IEEE Computer Society Press, Napa California*, February 1994.