# Evaluation of Rapid Context Switching on a CSRC Device

David I. Lehn, Kiran Puttegowda, Jae H. Park, Peter Athanas, and Mark Jones
Bradley Department of Electrical and Computer Engineering
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061-0111

## Abstract

*One property that distinguishes reconfigurable computing from rapid prototyping is the 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.*

## 1 Introduction

Fast FPGA configuration switching has emerged as a possible solution to a number of reconfigurable computing issues. The particular FPGA-like devices store multiple configurations called *contexts* in different sets of internal RAM. Each programmable part of the device is controlled by multiple RAM units. Global context lines act as addresses for these RAM units to activate a context. Any one device context can be active at a time and activating another context can be accomplished as fast as one clock cycle. This configuration activation process is known as *context switching*. This technique enables a number of possibilities not achievable with traditional reconfigurable logic including virtual hardware, simplified routing and lower power dissipation.

Commodity FPGAs have relatively long configuration times [1], and applications that use Run-Time Reconfiguration (RTR) on FPGAs often suffer from this. Any configuration delay comes at the cost of valuable computation time. Context switching can dramatically reduce this time by overlapping configuration loading with computation.

In an application that requires only a part of the computational logic at a time, there is the possibility of sharing the computational resources. In a context switching device routing is shared between contexts; hence routing can be simpler. Additionally access to input and output pins may be simplified when there is no need to route around currently unused logic. The need for I/O multiplexing may also be eliminated in applications sharing I/O pins among modules that are mapped to different contexts. The BAE-Systems Context Switching Reconfigurable Computer (CSRC) [2] device, an experimental context switching FPGA, will serve as the device exercised in this paper.

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 results of which will be discussed in Section 5. Conclusions arrived upon based on these results will be presented in Section 6.

## 2 Background

WASMII [3] was the first system proposed that was conceptualized as a multi-context device. It is a data driven computational system that uses the concept of *virtual hardware* to visualize an infinite hardware for applications. WASMII was later implemented [4] on Dynamically Reconfigurable Logic Engine (DRLE) a multi-context device developed by NEC. During the same period the concept of Dynamically Configurable Gate Array (DPGA) [5] was proposed by Bolotski, et. al. which also talked about context swapping within an FPGA. In a later paper, DeHon proposed placing DPGAs on the same die as a normal processor to act as a reconfigurable accelerator [6]. Xilinx filed a patent on the multi-context programmable device in 1995 [7][8]. The patented device has an architecture similar to the Xilinx XC4000E [1] with multiple configuration planes. The reconfigurable communication processor [9] 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 Comput-

ing (CSRC) device in [2]. This device has been used to implement a prototype research platform called the Reconfigurable Computing Module (RCM).

Above contributions do not give a detailed study of context switching from a system perspective. This paper discusses the application level and system level issues in a multi-context programmable system. The work presented here provides a study of context switching in various modes of operation from control oriented switching to data oriented switching between contexts.

## 2.1 CSRC micro architecture

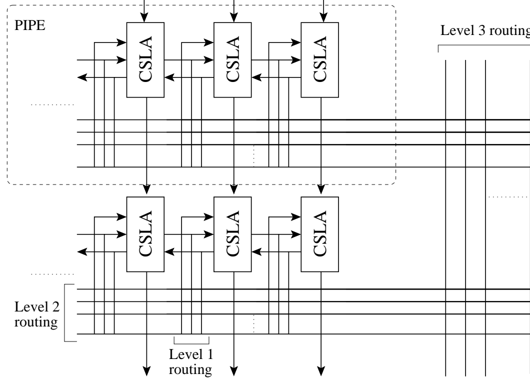


Figure 1: CSRC architecture

Figure 1 taken from [2], shows the architecture of the device. The CSRC device consists of 16-bit wide data pipes each consisting 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 to produce a 16-bit result. The result of one CSLA is available as input to two adjacent CSLAs in the pipe. Thus, a pipe can be used as a data path, where data flows in both directions.

The CSLC is the heart of computation for the CSRC device. It consists of a four-input lookup table (CSLUT), a context switching flip-flop (CSFF), a tri-state buffer and the carry logic. Each configurable resource in the CSLC along with each routing resource has four configuration bits; a single bit is selected as the current configuration. Each CSLC has a private register for each context and a public register. During context switch, the CSLC value is stored in public register if it is to be shared, else kept in a private register.

The prototype CSRC consists of eight pipes stacked one above the other with eight CSLAs each. Each CSLA has sixteen CSLCs; for a total of 1k CSLCs. The bitstreams for the CSRC FPGAs are downloaded serially. The user is required to specify which context is being loaded and then supply a clock and the configuration data. A context can be programmed when another context is active.

## 2.2 Reconfigurable Computing Module (RCM)

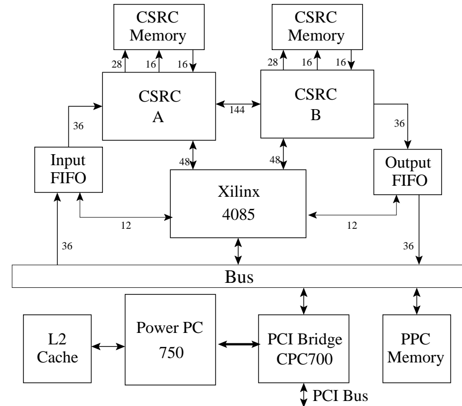


Figure 2: RCM platform architecture

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. An IBM CPC700 device provides connection between the Host Machine and the processor through the PCI bus. It also provides the secondary cache and memory control.

The RCM contains two CSRC devices with private memory. The two CSRC devices are directly connected with 144 lines and there are 48 lines 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 status flags of the FIFOs are available as inputs to the FPGA that makes it available to the processor and the CSRCs. The Xilinx XC4085 support FPGA is intended to provide a variety of support functions. The FPGA contains the ability to at least 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.

# 3 Runtime Environment

Support for different context switching applications requires extensive runtime support. The RCM board has a runtime environment that is specific to its hardware. The PowerPC on the board allows flexible programs to be run close to the context switching hardware. Figure 3 shows the software levels used for the RCM runtime environment.

Two methods of communicating with the board are available: a low bandwidth serial line, used mainly for debugging, and memory mapped reads and writes over

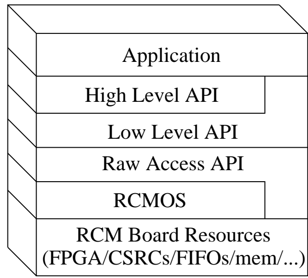


Figure 3: Runtime environment stack

the PCI interface. Application interfaces were built on top of the memory-based communication.

### 3.1 Host Application Programming Interfaces (API)

As shown in Figure 3 an application accesses the hardware resources through either a high level API, low level API, or a combination of each. The low level API is implemented using the "Raw Access API" which provides only basic services:

- open/close board, and
- read/write memory mapped areas

The raw API makes no assumptions regarding the hardware configuration. It does not access special features of any software running on the PowerPC or any features of configurations loaded in the CSRCs or FPGA. The initial setup of software on the PowerPC must be done with this 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.

All complex communication and functionality is implemented with a memory-based handshaking protocol with software on the PowerPC (described in Section 3.2). The "Low Level API", as shown in Figure 3, uses this protocol to provide transparent access to the hardware from the host. It consists of a number of features:

- PowerPC configuration,
- FPGA configuration,
- CSRC configuration (basic and caching),
- CSRC context switching, and
- data streaming.

For the RCM board this low level API exists in two forms. One is a basic C API. The other is through the ACS API [10]. The C API is suitable for high speed direct access. The ACS API allows the board to be accessed in the same uniform method as other ACS supported devices in a distributed dynamic network of heterogeneous reconfigurable hardware. In Figure 3 both these APIs are represented by the "Low Level API" layer.

The Xilinx FPGA is used to control many signals on the board. The CSRCs clock and reset, FIFO status and control, context switching control, programming control, and many CSRC connections all go through this FPGA. The low level API makes very few assumptions about how all this control is implemented; it simply memory maps the FPGA with address, data, and control lines. Specific control is up to either the application programmer or another layer of API.

Applications can also take advantage of a "High Level API". This API provides an object oriented view of the board representative of its physical parts. This API includes much of the functionality needed to take full advantage of the hardware. This layer is based on specific features of the FPGA configuration as well as the low level API features implemented in the basic Operating System (OS) running on the PowerPC called the RCMOS. Applications using this higher level API are isolated from many of the details of register access and bit manipulation. 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 RCM board is used to implement many of the low level API functions in an efficient manner. Programming of the configurable hardware resources involves bit manipulation that would be too slow over the PCI bus. Loading the configurations into the board memory and letting software in the PowerPC do the work is more efficient. Many of the CSRC operations such as control-driven context switching and clocking are also more efficiently implemented closer to the hardware than across the PCI bus.

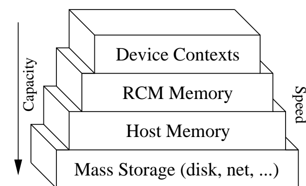


Figure 4: Configuration caching hierarchy

Context switching can be seen as part of a configuration caching hierarchy as shown in Figure 4. At the lowest level, configurations are stored in mass storage. This storage can be persistent physical media such as 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 hardware contexts. If any level cannot hold the number of configurations required for an application they are stored in the next higher capacity level and loaded on demand. For such applications caching affects the average switching time.

For board level configuration caching, the host uses an API that stores configurations on the board via RCMOS. The application uses high level functionality which requests a certain configuration to be loaded. The RCMOS handles the details of keeping track of which configurations are loaded into which contexts in the device. If the requested configuration is currently loaded in a context, then switching is a high speed hardware operation. If the requested configuration is not currently in a context, then the RCMOS uses standard replacement algorithms to determine which context it will replace with the requested configuration. This could be improved by giving hints to what future requests may be so that RCMOS could load contexts as a background task before they are needed.

### 3.3 Hardware

The Xilinx XC4085 on the RCM board is used to implement control logic for applications. This logic can be either hard coded for a specific task or have some flexibility controllable from the host application. One flexible approach is a host programmable Finite State Machine (FSM). This is easily implemented as a table based design. An abstract description of the FSM is converted by the high level API into a memory based table format. This is then loaded into the FPGA with a protocol of low level register writes. This allows flexible application control with the advantage of moving logic to high speed hardware. The prototype CSRC parts have limited logic and routing resources which require some applications to depend on this external control logic.

## 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 can also target such a system. One such framework is Janus [11]

### 4.1 Application Control

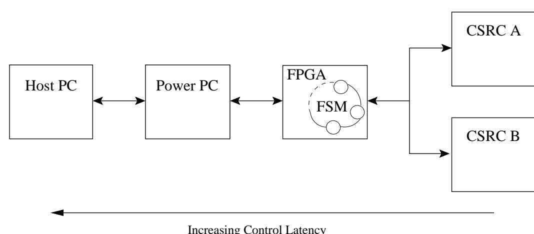


Figure 5: Context switching control routes

Figure 5 shows the paths on the RCM board used for context switching control. The actual route is application specific. Signals from the CSRCs may need to travel back to logic on the host or anywhere in between. Applications with user input require the full control path from host to CSRC. Applications can improve performance by moving appropriate logic closer to the CSRCs. One possibility is to locate FSM structures on the FPGA. Data are input from the CSRCs to the FSM which will control switching signals. It is also possible that the CSRCs can control their own context switching and not require any external logic.

The latency involved in a context switch is directly related to the number of layers that signals have to pass through. If the CSRCs require host processing for a decision the latency could be large. For debugging this communication can be used to stall the CSRC logic. If a switching decision is made in logic close to the CSRCs, then the switch is done fairly fast. On the RCM board this switching time is as low as 1 clock cycle for internal CSRC switching and 2 clocks if external logic is required.

Examples of applications making context switching decisions in different locations are given in the following sections.

### 4.2 Motion Detection Algorithm

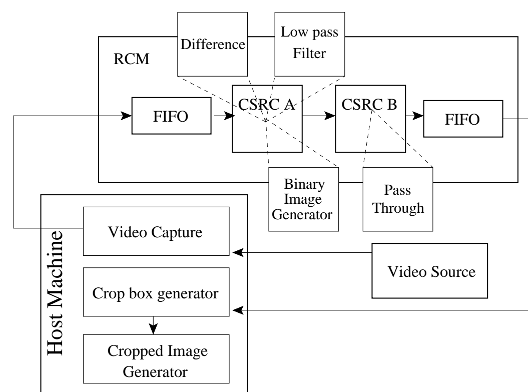


Figure 6: Image processing application

The motion detection algorithm implementation demonstrates the control of CSRC contexts through the host programmable Finite State Machine. The motion detection algorithm [12] consists of capturing an image, processing it and sending out a cropped 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 6. 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 in the CSRC. The FSM controls the switching of active contexts.

The video stream consists of $160 \times 120$ 8-bit gray scale image sequences. The image data is stored in

the sharable memory between the host PC and the PowerPC and 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, many factors like wind, ground vibration, etc generate small background changes. A low-pass filter implemented as a $4 \times 4$ averaging filter eliminates this spot noise and smooths out the image.

The difference block will also produce non-zero pixels throughout the difference image due to intensity variations between frames. 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. 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 streams out the binary image 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 part of the original image frame where there is movement.

## 4.3 Video Filter

A variation on the motion detection application is a host-driven video processing application. A continuous video stream is processed by filters stored in contexts. A user request from the host causes the context to switch. This enables single-cycle algorithm changes. If more filters are needed than available contexts, then some cache manipulation may need to take place. The loading of new filters into contexts that are currently inactive take place while another filter is running. This allows unlimited "virtual hardware" filters. Simple filters have been implemented on the CSRCs that includes basic passthrough, the motion detection difference filter and a delay filter.

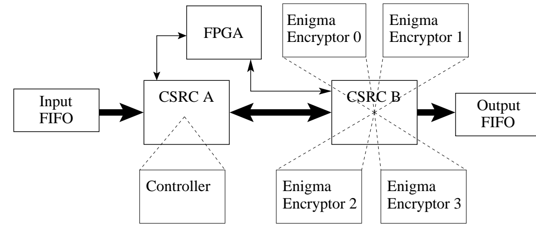


Figure 7: The enigma application

## 4.4 Enigma Encryptor

This application is a demonstration of the CSRC for data-driven network processing in which the context to be made active is dependent upon the data being processed. The application is a demonstration of simple encryption and decryption of network packet for multiple channels. As the resources on the prototype CSRC are limited, a simple Enigma-like encryption scheme was selected.

The Enigma Machine [13] was based on a system of three rotors that substituted cipher text letters for plain text letters. A letter to be processed is sent through the first rotor, shifting it according to its present setting. The new letter passes through the second and third rotor, where it would be substituted according to their settings. This letter is bounced off a reflector, and back through the three rotors in reverse order. As the plain text letter passes through the first rotor, the first rotor would rotate one position. The other two rotors would remain stationary until the first rotor rotates 26 times (one full rotation). Then the second rotor would rotate one position. The third rotor would rotate one position when the second completes a full rotation. To decode the message, the rotors are set to the initial settings, and then the cipher text is put through the machine.

This concept was used to build an encryptor for bytes. 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 returning data 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 scrambled nibble-wise using two $4 \times 4$ tables. The table entries represent the actual rotor settings.

Using the available RCM board, the following system was implemented. FIFOs are used to stream data into the processing elements. The processing elements consist of the two CSRCs and the support FPGA. Output FIFOs are used to stream data out of the processing elements. The system design partition is shown in Figure 7.

Each of the four contexts on the CSRC B contain an enigma encryptor with a particular rotor configuration and key. Each of these individual rotor configurations

is used for each channel. The CSRC A has a controller context that reads the header of the packet, determines the intended channel, and the packet length. The data in the header indicates the intended channel and signals the support FPGA to set the particular context on the CSRC B. CSRC A's controller context now acts as a passthrough and down-counts the number of bytes. When the counter reaches zero the controller context resets its controller and waits for the next packet to arrive.

If each of the contexts process data for a particular channel, this application demonstrates the data-driven virtual hardware implementation for a network with four channels. With the hardware cache it can be expanded to any number of channels.

## 5 Results

The run-time environment and the applications were used to analyze the costs and benefits of context switching for run-time reconfiguration. The results of the analysis are presented in the following sections.

### 5.1 Area

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

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

| *Application* | *4-context impl.* | | *1-context impl.* |
|---|---|---|---|
| Motion Detection Algorithm | Difference | 783.0 | 2055.0 |
| | Filter | 1188.8 | |
| | Bin. img. gen. | 253.3 | |
| Enigma Encryption | Enigma0 | 1331.8 | 5245.0 |
| | Enigma1 | 1331.8 | |
| | Enigma2 | 1331.8 | |
| | Enigma3 | 1331.8 | |

Table 1 shows the gate equivalent areas obtained from Synplify™for each contexts of the applications implemented in a single context and a four-context device. A device with an equivalent gate count of the highest number will be required for that particular application. Motion detection application can be implemented in a three context device with a minimum of 1188.8 equivalent gates without incurring reconfiguration delay. But with a single context, the application requires a device with 2055.0 equivalent gates. For the Enigma encryption a four-context implementation requires at least 1331.8 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 5245.0 equivalent gate device.

With such an approach the multi-context implementation of the motion detection algorithm will have bitstreams for three contexts of a 1188.8 equivalent gate device and the single context will have bitstream for a single 2055.0 equivalent gate context. In the enigma encryption application storage requirement for the bitstreams scale well with the number of contexts. With this observation it seems that context switching might lead to larger bitstreams. But for bigger applications that require run-time reconfiguration, the multi-context approach is scalable better than a single context approach. Suppose more contexts are added to the image processing algorithm implementation, bitstream size scales at 1188.8 equivalent gate steps in a multi-context device rather than 2055.0 equivalent gate steps.

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.

### 5.2 Switching Time

For applications requiring more than the available number of contexts in the device, the CSRC configuration cache, discussed in Section 3.2, can be used to store additional contexts. This, along with the host memory, implements an effective virtual hardware environment. 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 application context has an equal probability of being requested, an expression for the average context switching time was derived:

$$t_{\mathrm{avg}} = \begin{cases} t_{\mathrm{s}} & \text{if } 1 < n \leq k \\ p_{\mathrm{s}} t_{\mathrm{s}} + p_{\mathrm{c}} t_{\mathrm{p}} & \text{if } n > k \end{cases} \quad (1)$$

where;

| | |
|---|---|
| $t_{\mathrm{avg}}$ | average switching time |
| $t_{\mathrm{s}}$ | context switch time |
| $t_{\mathrm{p}}$ | context program time |
| $k$ | number of device contexts |
| $n$ | number of application contexts |
| $p_{\mathrm{s}}$ | probability that context is on device, $\frac{k-1}{n-1}$ |
| $p_{\mathrm{c}}$ | probability of a reconfigure, $1 - \frac{k-1}{n-1}$ |

The variation of the average context switch time with numbers of application contexts for different number of device contexts is plotted in Figure 8. The plot shows that for an increase in number of device contexts($k$) from one to four and four to eight there is a tremendous improvement in the average reconfiguration time($t_{\mathrm{avg}}$). Adding a new context would involve adding new RAM for context storage, multiplexers and the necessary routing for each configurable resource. Out of these RAM takes up more area. So the VLSI area in adding new contexts increases linearly. For more device contexts diminishing returns

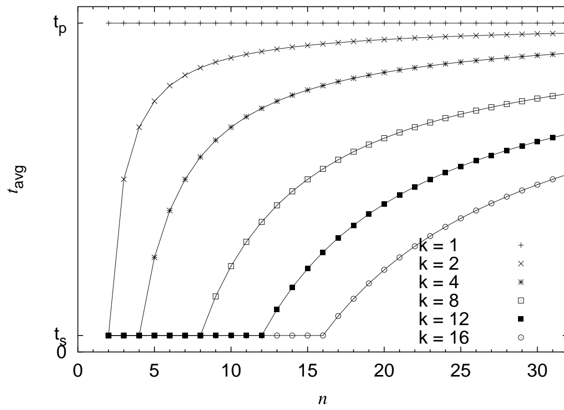

Figure 8: Average context switch time

in terms of reduced reconfiguration time is obtained. It can also be seen that a match between $n$ and $k$ results in shorter average reconfiguration time($t_{\text{avg}}$). For high number of application contexts the curves come closer. This implies that for a lot of application contexts the advantage of having more device contexts to reduce average switching time is smaller.

With locality-of-reference for the switch request, the probability that the context requested is on the device will be higher; thus the average reconfiguration time would be smaller than that observed in the plot. This locality-of-reference depends on the applications and the way they are programmed. There is also the possibility of configuring a context in the background while another context is processing data. This would reduce the effective context program time and hence the average reconfiguration time in applications that can anticipate the next context required.

## 6 Conclusions

The described system achieves context switching with various degrees of speed and controllability. 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 system.

This type of system allows a particular implementation in an effectively smaller area on the FPGA than a system with a single context. This has several positive effects on the overall system design. The routing inside the chip is reduced as the effective area to which a design is mapped is smaller; thus interconnect delays are smaller. It also saves power as only the active part of the design is running, while rest of the design is simply stored on the SRAM bits and memory.

## References


[1] Xilinx Inc., *The Programmable Logic Data Book*, San Jose, CA, 1999.

[2] S. M. Scalera and Jóse R. Vázquez, "The design and implementation of a context switching FPGA," in *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1998.

[3] Xiao-ping 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.

[4] Y. Shibata, et. al., "A virtual hardware system on a dynamically reconfigurable logic device," in *Proceedings of IEEE symposium on FPGAs for custom computing machines*, April 2000.

[5] A. DeHon, "DPGA Utilization and Application," in *MIT Artificial Intelligence Laboratory, Transit Note 129*, September 1995.

[6] A. DeHon, "DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century," in *Proceedings of IEEE Workshop on FPGAs for custom computing machines*, 1994, pp. 31–39.

[7] Xilinx, "Time multiplexed programmable logic device," July 1997, Patent no. 5646545.

[8] 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.

[9] Chameleon Systems, Inc., "CS2000 Reconfigurable Processor," 2000, CS2000 Advance product information.

[10] J. Scott C. Twaddle M. Yaconis K. Yao P. Athanas M. Jones, L. Scharf and B. Schott, "Implementing an API for distributed adaptive computing systems," in *Proceedings of the IEEE International Conference on Communications*, April 1999, pp. 222–230.

[11] R. D. Hudson, *Architecture-Independent Design for Run-Time Reconfigurable Custom Computing Machines*, Ph.D. thesis, Virginia Polytechnic Institute and State University, Blacksburg, VA USA, August 2000.

[12] 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.

[13] D. Museum, "Enigma encryption machine," http://www.deutsches-museum-bonn.de/ausstellungen/meisterwerke/2_3enigma/enigma_e.html.