Generic Software Framework for Adaptive Applications on FPGAs

Suhaib A. Fahmy*, Jörg Lotze*, Juanjo Noguera†, Linda Doyle* and Robert Esser†

*CTVR, Trinity College, Dublin, Ireland
Email: {suhaib.fahmy, jlotze, linda.doyle}@tcd.ie

†Xilinx Research Labs
Email: {juanjo.noguera, robert.esser}@xilinx.com

Abstract

Adaptive systems are set to become more mainstream, as numerous practical applications in the communications domain emerge. FPGAs offer an ideal implementation platform, combining high performance with flexibility. While significant research has been undertaken in the area of FPGA partial reconfiguration, it has focussed primarily on low-level architecture-specific implementations. Building upon this previous work, we present a system model and software architecture for implementing runtime adaptive applications on FPGAs, separating the control and processing planes and abstracting away the details of hardware reconfiguration from the system designer. Hardware processing components appear as software components in the runtime system, enabling their inclusion in adaptive applications. We present an adaptive wireless application, demonstrating the use of the model and software architecture.

1 Introduction

Runtime adaptive systems are able to respond to events by modifying their behaviour at runtime according to some reasoning process. As real applications that depend on adaptive behaviour become more mainstream, the demand for a practical implementation framework increases. Applications such as cognitive radios [1] and recent advances in autonomous self-configuring networks [2], rely wholly on adaptive capability. Such systems have thus far relied on the flexibility afforded by general purpose processors (GPPs) and Application Specific Standard Parts (ASSPs). However as standards and algorithms become more complex, hardware platforms that can handle high throughput will become an essential enabler of such systems. FPGAs, combining the performance advantages of hardware with some of the flexibility of software, offer an ideal platform for the implementation of adaptive applications.

Much effort within the research community has been focussed, thus far, at facilitating FPGA partial reconfiguration through novel architectures and low-level design tool advancements. However, hardware reconfiguration is but a piece of the adaptive systems jigsaw, and how to harness this capability when developing adaptive applications is an area in need of attention. While the ability to reconfigure an FPGA, even partially, is essential to the development of adaptive hardware applications, the mapping between this and application level adaptation requires further investigation. A framework that enables the design of adaptive applications for those not specialised in the minutiae of FPGA architecture features will accelerate the adoption of such technologies by application designers. The aim of this work is to de-couple the implementation of adaptation, as viewed at the application level, from the details of reconfiguration, as typically discussed at the hardware level.

We present a framework for implementing adaptive applications on FPGAs, comprising three key contributions:

- A system model that separates the control and processing planes of adaptive applications and defines a generic interface between them.
- A software architecture that enables self-adaptation while abstracting away low-level hardware details from the control plane, thus narrowing the gap between application level adaptation and hardware reconfiguration.
- An adaptive wireless video demonstrator that is built upon this framework.

We present some related work in Section 2, then describe our system model in Section 3 including the software architecture and the hardware interface. Section 4 presents an application of our system framework to adaptive video transmission and reception. Finally in Section 5, we draw conclusions and discuss future work.

978-0-7695-3716-0/09 $25.00 © 2009 IEEE
DOI 10.1109/FCCM.2009.6

2 Previous Work

Adaptive systems, when referred to in the literature, can mean a variety of things, from low-level partial reconfiguration techniques to adaptive implementations. Other researchers consider adaptive systems as being those where tasks can be mapped to different processing resources at run-time [3]. We have chosen to use the term *adaptive applications* to refer to the class of systems where application behaviour changes dynamically at runtime. This adaptation is not defined as part of a protocol, rather occurs on rare occasions in response to events.

Partial Reconfiguration with Xilinx FPGAs has been receiving increased attention thanks to recent improvements in the capabilities of modern devices such as the Virtex-5 but also thanks to enhanced low-level design tools [4]. Xilinx FPGAs have the capability to modify part of a design, while the rest remains operational, making them attractive for implementing adaptive systems.

Much of the effort recently undertaken within this area has focussed on low-level implementation. Examples include 1- and 2-dimensional slot-based reconfigurable architectures [5] and the online placement and routing of adaptive systems on FPGAs [6, 7]. These efforts provide a foundation for building adaptive systems, however, they do not tackle the application level view of adaptation, where a system is able to reason about events and respond by modifying its behaviour at runtime; this is something we tackle in this paper.

Numerous adaptive applications have been implemented on FPGAs. These include automotive [8] and networking applications to improve performance [9] and reduce power consumption [10]. Some of the closest work we have found to that proposed here is in [11], where the authors design an autonomous system for interference avoidance. They use a Bayesian network to select one of three mitigation filters depending on an estimate of the noise. The key differences between our work and theirs is that they require the system designer to have hardware knowledge, they also explicitly incorporate hardware reconfiguration within the control software. In our work, we have abstracted away hardware reconfiguration, allowing a system designer with no FPGA experience to program an adaptive system in C++, while making use of optimised hardware components.

It is clear from the literature, that previous solutions to the design of adaptive applications have generally been adhoc and lack the definition of a general solution or framework. Our work provides a general framework that can be used for designing adaptive applications at the application level, and which separates this view from the details of hardware implementation, while taking full advantage of the capabilities that exist in modern devices.

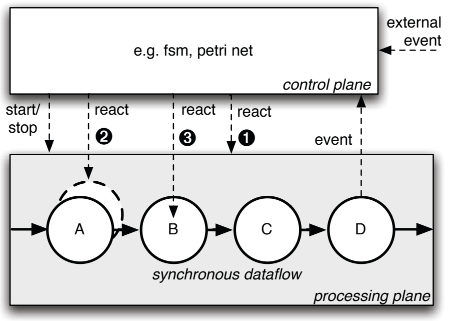

Figure 1. Adaptive System Model.

3 Adaptive Systems on FPGAs

3.1 System Model

We focus in this work on data-driven applications such as signal processing which are typically modelled using the Synchronous Data Flow (SDF) paradigm. In this model of computation, a directed graph represents the system; this contains nodes, each representing a function, and arcs connecting them, representing the flow of data. It can become cumbersome to introduce control within such a paradigm, and hence we define a separate control plane that interfaces with this processing plane. This also allows for the control plane of an application to be implemented in the most suitable way, whether using Petri nets, state machines, or any other design construct. This system model is shown in Figure 1.

The core process in an adaptive system, that represents the interface between these two planes, can be summarised in the "sense, reason, react" cycle. Events are triggered by components within the processing plane, or externally. The control plane senses these events, and reasons to determine the required response(s), before selecting a response to apply to the processing plane. We define components within the processing plane as having parameters, to which the control plane has access. The control plane also has access to the top level of the processing plane. Since the two planes are distinct, their implementations are independent.

The manner by which the control plane can effect a change in the processing plane is defined at three levels (illustrated by the numbers 1, 2, and 3 in Figure 1):

1. *Functional reconfiguration*[1] involves completely overhauling the system defined in the processing plane and replacing it with a new system, e.g., a radio switching from a passive receiver to a transceiver.
2. *Structural reconfiguration* involves replacing, removing or introducing new components in the processing plane, e.g., a radio changing the modulation scheme.

[1]Note that *reconfiguration* in this section does not refer to FPGA reconfiguration.

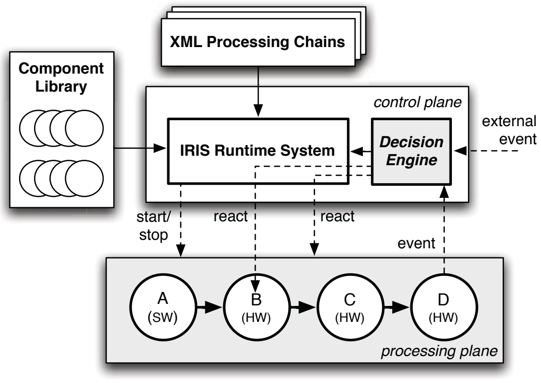

Figure 2. The Runtime Software Architecture.

3. *Parametric reconfiguration* means modifying a parameter of one of the components within the processing plane, e.g., changing the gain of a scaling component.

These levels of reconfiguration are as viewed at the application level.

This system model maps intuitively to the architecture of modern FPGAs, where the processing plane can be implemented in the logic fabric, while the control plane can be implemented on embedded processors.

3.2 Software Architecture

The software architecture which realises the system model described in the previous section, builds on the software radio application, IRIS, which has been extended to run under Linux on the FPGA and support hardware components in the logic fabric [12]. The architecture is illustrated in Figure 2. The graph of the processing plane of the system, containing the processing components, which are selected from a pre-existent library, and the flow of data between them, is described in an XML file. Each component has a set of parameters and an interface to the user-created *Decision Engine*, which allows for re-use in different applications. The *Decision Engine* can subscribe to events triggered by processing components. It is responsible for reasoning and reacting to these events. Reactions include parametric, structural or functional reconfigurations of the processing chain, as defined above, and are effected through the *Runtime System*. The *Decision Engine* is implemented using C++, with a simple interface to allow for rapid prototyping of adaptive applications.

To allow the execution of IRIS on FPGA hardware, we have adopted a Linux on FPGA implementation for embedded PowerPCs, as offered by Xilinx [13]. While some work has been done on designing operating systems for FPGA reconfiguration [14], we believe that adaptive applications are best served by managing reconfiguration in the user space.

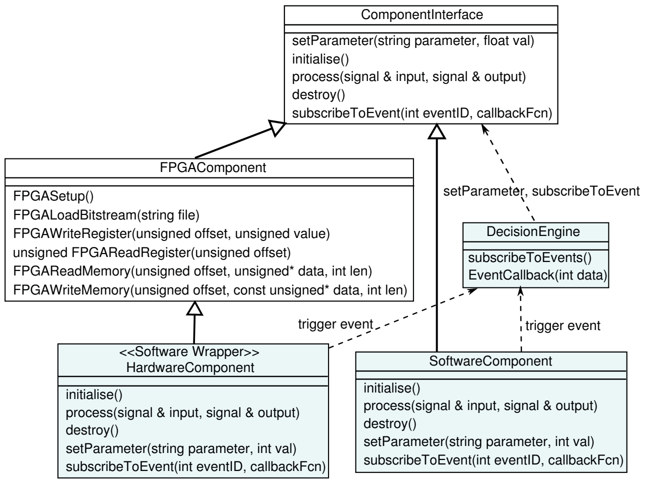
ComponentInterface
setParameter(string parameter, float val)
initialise()
process(signal & input, signal & output)
destroy()
subscribeToEvent(int eventID, callbackFcn)
FPGAComponent
FPGASetup()
FPGALoadBitstream(string file)
FPGAWriteRegister(unsigned offset, unsigned value)
unsigned FPGAReadRegister(unsigned offset)
FPGAReadMemory(unsigned offset, unsigned* data, int len)
FPGAWriteMemory(unsigned offset, const unsigned* data, int len)
setParameter, subscribeToEvent
DecisionEngine
subscribeToEvents()
EventCallback(int data)
trigger event
trigger event
<<Software Wrapper>>
HardwareComponent
initialise()
process(signal & input, signal & output)
destroy()
setParameter(string parameter, int val)
subscribeToEvent(int eventID, callbackFcn)
SoftwareComponent
initialise()
process(signal & input, signal & output)
destroy()
setParameter(string parameter, int val)
subscribeToEvent(int eventID, callbackFcn)

Figure 3. Component class hierarchy.

An adaptive application designer wishes to determine, in some predictable way, how the system will adapt to various events, and what reasoning method the system uses to determine the type of adaptation. Hence, the adaptation is determined by the designer as required for the application, and not by low level OS criteria. Adopting this approach also means that the system is abstracted from the underlying hardware, and at the same time, is able to take full advantage of standard operating system features.

System execution is managed by the *IRIS Runtime System*, which parses the specified XML file, instantiates and connects the required components, loads the *Decision Engine*, and controls and monitors system execution. Thus, both the *Decision Engine* and the *Runtime System* together form the control plane, as shown in Figure 2.

Based on this software architecture, the only two inputs that the user needs to specify are: the XML radio chains; and the C++ *Decision Engine*. The user's implementation of the *Decision Engine* does not need to consider implementation details of the components it is controlling (i.e., software or hardware implementation), since it is only accessing the *ComponentInterface* (Figure 3).

Processing components can be described in software using C++, or in hardware using a hardware description language. We do not deal with the high-level design of components, as this can be done using existing tools. Any tool that can generate HDL output is suitable. For example, Xilinx System Generator allows components to be designed using Simulink. To allow a common interface to both types of components from the control plane, one or more hardware components are wrapped in a *software wrapper* with an identical interface to standard software components. The class *FPGAComponent*, shown in Figure 3, provides a set of FPGA operations for software wrappers, such as methods for accessing registers and embedded memories (BRAMs).

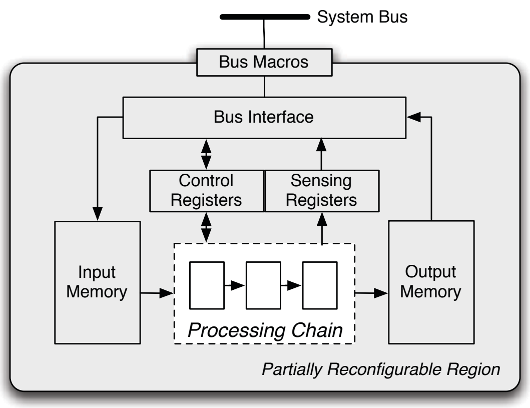

Figure 4. The partially reconfigurable region.

3.3 Integrating Hardware

Figure 4 shows the interface of a hardware chain, as loaded into the partially reconfigurable region. This *hardware wrapper*, containing one or more components, provides the common interface which is connected to the rest of the system (i.e., static design) using the system bus.

In order to transfer data into and out of the partially reconfigurable region, the first and last components in a chain are connected to BRAMs. A set of registers, accessible from software, is used for controlling hardware execution, accessing component parameters and sensing events.

When the run-time system initialises the software wrapper, the input and output memories as well as the register addresses are mapped into the Linux virtual memory space (*FPGASetup* operation in Figure 3). A bitstream is loaded into the partially reconfigurable region of the FPGA using the Linux ICAP (Internal Configuration Access Port) driver (*FPGALoadBitstream*). When the execution reaches the software wrapper, data is transferred to the hardware input memory, then the hardware is enabled using the register interface. Once execution has completed, data can be read back from the output memory by the software wrapper. The software wrapper is written such that it allows for processing of variable sized blocks of data (i.e., multiple iterations are completed if the amount of data to process is larger than the size of the shared memories).

Events can be triggered from the hardware in the partial region by setting a value in a sensing register. The wrapper checks these registers regularly and triggers a software event to the *Decision Engine*.

It is important to distinguish between application level adaptation and its manifestation in terms of hardware reconfiguration. Clearly, *functional reconfiguration* involves the whole processing chain being replaced. This simply maps to a hardware reconfiguration of the whole region, which is achieved by calling the *FPGALoadBitstream* method in the implementation of the *initialise* method.

Structural reconfiguration involves the replacement of components or the introduction of new components. In an ideal scenario, this could map to a partial reconfiguration of just the necessary part of the chain. If one partially reconfigurable region holds multiple components, represented by one wrapper from the software perspective, the whole region must be reconfigured (a call to *FPGALoadBitstream*). While it is possible to implement a hardware chain that contains multiple alternatives of a component and uses a multiplexer to select between them, this clearly wastes resources, while precluding the need for, and foregoing the benefits of, partial reconfiguration.

Parametric reconfiguration (the *setParameter* method in the software wrapper of a hardware component) is less straightforward. Parameters can include the gain of a scaling component or the constraint length of a convolutional code, amongst other possibilities. In some cases, where changing the parameter has minimal impact on the computational circuitry, setting a parameter can simply map to a write into a software accessible register (a call to *FPGAWriteRegister*). An example is a scaling component where the gain is reconfigurable. By introducing a register that holds the gain to be used by the hardware, a reconfiguration of this parameter can be made without the need for an FPGA reconfiguration. However, if the parameter to be changed implies considerable changes in the hardware circuitry, e.g. constraint length of a Viterbi decoder, it maps to an FPGA reconfiguration (a call to *FPGALoadBitstream*).

3.4 Abstracting from Hardware Implementation

To abstract away hardware implementation details, we use a design-time *Composer* tool which automatically converts the high-level (i.e., functional) XML description of a processing chain to a low-level XML representation, mapping components to the available partially reconfigurable regions and processors. It connects the hardware components in the chain, adds the hardware wrapper shown in Figure 4, generates bitstreams, and creates the software wrappers. The new *composed chain* is represented by an automatically-generated merged XML description.

The different types of reconfiguration are dealt with in the following manner. In functional reconfiguration, we assume that the *Decision Engine* invokes a new XML chain and so this chain is explicitly defined, separately, and the *Composer* creates its corresponding bitstream. For structural reconfiguration, a separately defined chain is also used, complete with its bitstream as produced by the *Composer*. For parametric reconfiguration, the *Composer* tracks which parameters map to register names of the corresponding component. In such cases, a reconfiguration does not require a change of bitstream, and so the change in value in the merged XML maps correctly to a change in parameter in the hardware component. In the case where different hardware implementations are required for a change in parameter, the designer specifies the mapping of parameter values to implementations in a text file within the hardware component tree. The *Composer* modifies the merged XML to facilitate the loading of the alternative bitstream by the software wrapper, using this file to determine the mapping.

A mapping from the original high-level processing chain to the low-level version is generated by the *Composer*, and this is used by the IRIS Runtime System to abstract away the low-level details from the *Decision Engine*.

4 An Adaptive Wireless Application

In this section, we present an application that demonstrates the use of our software architecture. We show how adaptation is incorporated into the system and how the designer can specify the adaptive behaviour without concerning himself with the details of reconfiguration.

4.1 Overview

An application consisting of an adaptive wireless video transmitter and receiver has been implemented and demonstrated. The implementation platform is the Xilinx University Program (XUP) Board which hosts a Xilinx Virtex II Pro FPGA. We use the Universal Software Radio Peripheral (USRP) radio frontend [15] which we connect to through a TCP socket via a PC bridge.

The transmit chain receives UDP packets, through ethernet, containing video data, streamed from a PC using the VideoLAN VLC Player. These are processed through the transmission chain on the FPGA, then transmitted by the USRP. At the receiver, the signal is processed by the reception chain in the FPGA, then passed, through UDP, to a VLC Player window on a PC which displays the video.

The adaptation in this application is implemented by allowing coding to be used. A wireless channel with poor signal-to-noise ratio (SNR) can cause high Bit Error Rates (BER) at the receiver. Using forward error correction reduces these errors. A convolutional coder, with variable constraint length can *optionally* be inserted prior to the modulator in the transmitter, and a matching Viterbi decoder after the demodulator in the receiver (shown by dashed boxes in Figure 5). Adding coding and increasing constraint length increases the robustness of the communication. A BER estimate produced by the Viterbi decoder or Deframer at the receiver is used by the Decision Engine to decide when to switch constraint lengths. A control channel instructs the transmitter which constraint length to use.

The radio chains for this application are shown in Figure 5. Components implemented in the FPGA fabric are

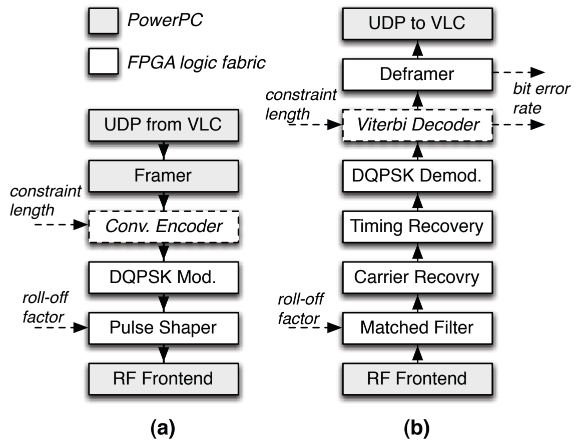

Figure 5. Transmitter (a), and receiver (b) radio chains.

shown in white, while components executed in the embedded PowerPC are shown in grey. Data is passed from the video source to the transmission chain via UDP. The *Framer* adds a 64-bit frame access code to each frame of data to identify it, and appends a checksum. The *Differential QPSK Modulator* maps information dibits (2-bit units) into phase changes of the complex base band signal. The *Pulse Shaper* (an upsampler and FIR square-root raised-cosine filter) is used to modify the signal in order to control the transmission bandwidth and SNR. Data is transmitted to the receiver by the *RF Frontend*. For forward error correction we use the Xilinx *Convolutional Coder* core, which can be added into the transmission chain as part of system adaptation. We use codes with constraint lengths 6 and 8 with a $1/2$ code rate.

At the receiver, data is obtained from the USRP radio frontend. It passes through the *Matched Filter*, which increases the SNR. When a modulated signal is received, the frequency of the local oscillator is typically off by some variable margin, which is corrected by the *Carrier Recovery* block. *Timing Recovery* finds the correct sampling points for each symbol, eliminating the effects of timing shifts and jitter. The *DQPSK Demodulator* is then used to map the complex signal samples back into information dibits. The *Viterbi Decoder* also uses a Xilinx core, with codes to match the convolutional coder at the transmitter. The *Deframer* correlates the fixed 64-bit frame access code inserted at the transmitter with the streaming data, in order to identify the start of a frame of data. It extracts the data from the frames and performs checksum verification.

4.2 Adaptation Algorithm

As can be seen in Figure 6, the BER at the receiver depends significantly on the coding scheme used. On the other hand, lower constraint length codes or no coding re-

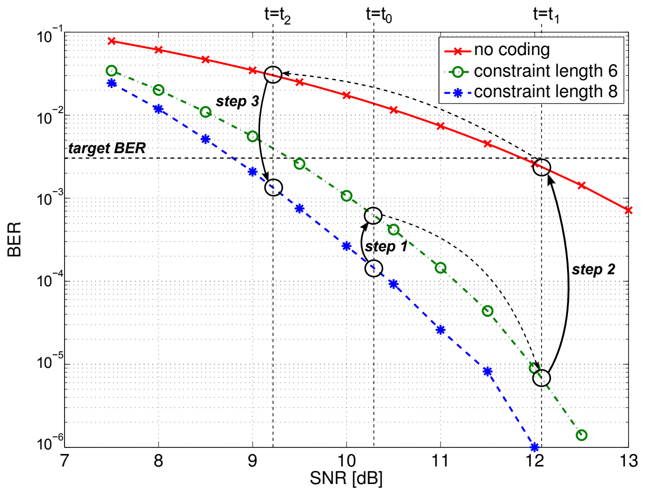

Figure 6. BER vs. SNR for different codes, and an example code switching sequence.

sults in considerable area and power savings on the FPGA, as will be discussed in more detail in Section 4.3. To resolve this trade-off we have designed an adaptive application which changes the coding scheme depending on the perceived BER.

Setting a target BER, as shown in Figure 6, the adaptation algorithm implemented in the Decision Engine at the receiver decides when a switch of the coding scheme is required[2]. Decisions are based on the simulated BER versus SNR curves shown in the figure, which are programmed into the Decision Engine using a lookup table. Based on this table, the Decision Engine can predict how the BER changes when a different constraint length is used. If the current BER is higher than the target, a reconfiguration to a higher constraint length code is triggered, representing a parametric reconfiguration. If the BER is low enough that a less complex code is sufficient to achieve the target, the Decision Engine lowers the constraint length or switches the coding off completely, representing a structural reconfiguration. To allow the transmitter to respond to an event at the receiver and reconfigure to the same constraint length, a control channel is needed. In this implementation ethernet is used, though this could be done wirelessly.

An example adaptation sequence is shown in Figure 6. The system starts in the most robust coding state (CL $= 8$) at time $t = t_0$. The Decision Engine monitors the BER output of the Viterbi decoder at the receiver. The hardware reports this value to the software wrapper using the register interface (see Figure 4), which in turn triggers an event to the Decision Engine. The Decision Engine estimates the BER with less robust coding (CL $= 6$) using the lookup table, and sees that it remains below the target BER, so it

[2]The target BER in Figure 6 is chosen for illustrative purposes only.

	PSS[a]	Tx			Rx		
CL[b]		none	6	8	none	6	8
Slices	6715	1426	1433	1435	2594	3693	6555
(% region[c])	98%	21%	21%	21%	38%	54%	95%
BRAMs	13	8	8	8	10	12	14
Mults	0	24	24	24	41	41	41

[a]Processor Subsystem, basic hardware to run Linux, in static region.
[b]Constraint length of convolutional code in radio chain (partial region).
[c]Percentage of static/partial region, respectively (6848 slices each).

Table 1. FPGA resource utilisation.

initiates a parametric reconfiguration by setting the parameter *constraintLength* of the Encoder and Decoder blocks to 6 (step 1 in Figure 6). The runtime system maps this parameter to the software wrapper of the hardware chain (as described in Section 3.3) which performs the necessary hardware reconfiguration.

At time $t = t_1$, the signal quality increases. When it is high enough that the target BER can be achieved without coding, the Decision Engine applies a structural reconfiguration to the radio to remove coding (step 2 in Figure 6). To estimate the BER without coding, we use the frame access code correlation value determined by the Deframer. It knows the sequence inserted by the Framer exactly and can therefore estimate the BER by averaging the correlation value over a predetermined number of frames. At time $t = t_2$, the channel conditions have worsened considerably and the BER has risen above the target, so the Encoder and Decoder are reconfigured through the parameter to the constraint length 8 code (step 3 in Figure 6).

In this manner, the system continues to use the most suitable type of coding for the current channel conditions.

4.3 Hardware Implementation

The FPGA implementation consists of a static region (i.e., processor subsystem), which includes the required modules to run Linux on the embedded PowerPC, such as the memory controller, the ethernet core and the ICAP module, used for reconfiguring the FPGA. The partially reconfigurable (PR) region is attached to the processor subsystem via the OPB bus and is within the address space of the PowerPC (see Figure 4). The PR region contains the processing chain and is reconfigured by the software wrapper through the Linux ICAP driver.

We decided to use a single PR region for this implementation since it is a more general solution. Since the sizes of the PR regions are fixed and must be large enough to accommodate the largest module, using multiple PR regions was deemed too inflexible and application-specific at present. As tool support is expanded, improved low-level technologies could be exploited within this framework.

Table 1 shows the resource utilisation of the processor

	Sampling Rate[a] [Msamples/s]		Bitrate[b] [Mbps]	
Coding CL	6 or 8	none	6 or 8	none
Tx HW max[c]	100	100	25	50
Rx HW max[c]	25	25	6.25	12.5
Demonstrator[d]	2	1	0.5	0.5

[a]Rate of 32-bit samples (16-bit complex) sent to the USRP.
[b]Rate at the Framer/Deframer (PHY layer bitrate).
[c]Maximum of FPGA hardware chain (from FPGA simulation).
[d]Rate chosen for video demonstrator.

Table 2. Demonstrator performance.

subsystem and various chain configurations. All chains meet the 100MHz bus frequency requirement. While the resource requirements of the transmission chain are almost constant, the area savings when reducing the constraint length at the receiver are clear. The hardware wrapper area overhead consists of the two memories used to transfer data, along with the register interface and minimal logic used to interface to the OPB bus.

We should also consider the fact that implementing all three receiver processing possibilities on a single device without partial reconfiguration would require a larger device for the receiver. We can see that in this case, the receive chain with constraint length 8 Viterbi decoder is close to filling the capacity of the partially reconfigurable region.

Table 2 shows both the maximum performance of the signal processing hardware and the sampling and bit rates we chose for the demonstrator setup. The maximum performance is the rate achievable by the hardware chain on its own, neglecting the interface to the software and to the radio frontend. In our demonstrator we are primarily limited by the bandwidth of the ethernet connection used to connect to the radio frontend. This limits the maximum achievable bit rate to 500Kbps[3]. Clearly if the demonstrator is executed on a more advanced development board with built-in radio frontend, throughput can be drastically improved, up to a maximum of 6.25Mbps. Since all radio components are clocked at the same speed, throughput is identical for all chains.

In order to quantify the power benefits of adaptation, we have measured the *average* power consumption of the three receiver radio chain configurations (i.e., no coding and coding with constraint lengths 6 and 8). In our experimental set-up, we connected a power supply with integrated ammeter to the FPGA internal core voltage (i.e., $V_{\text{ccInt}} = 1.5\text{V}$). Figure 7 shows the power consumption when increasing the throughput for the complete FPGA (i.e., processor subsystem and specific receiver chain in the PR region).

[3]Since each complex sample is represented by two 16-bit numbers, a 2Msps sampling rate represents 64Mbps of data over the ethernet. With 4 samples per symbol and 1/2 code rate, the 2Msps gives a bitrate of 500Kbps at the Framer/Deframer.

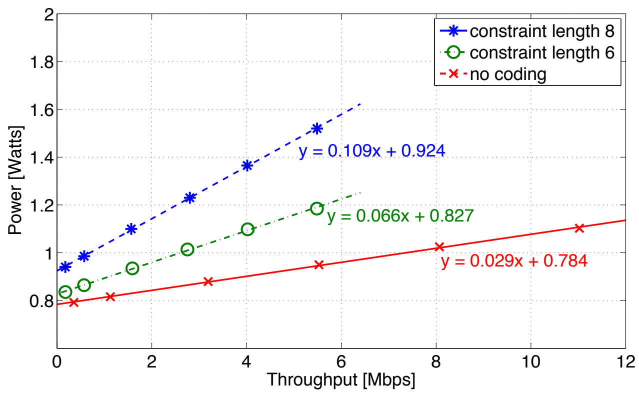

Figure 7. Measured power consumption with linear regression curves.

It is apparent that the power consumption increases linearly with throughput. Additionally, the graph shows the large impact constraint length has on power consumption given the increased amount of FPGA resources required.

4.4 Discussion

In this application, coding might be required when there is a noisy channel, yet when channel conditions are more favourable, switching off coding offers a power saving at the receiver. For applications that require a constant bitrate, such as video streaming, this means we can halve the sampling rate at the radio frontend when no coding is used (due to the $1/2$ code rate of the coder). This results in less radio spectrum occupancy and therefore reduces noise and possible interference in the signal. So it is preferable to use non-coded transmission when possible. Hence adaptation offers both power and performance advantages in this scenario.

From Figure 7, we can quantify the power reduction benefits of run-time adaptation to channel conditions (i.e., SNR) using an illustrative scenario. Assume that in order to provide the required BER in a high noise environment, we are using a radio chain with coding using constraint length 8. As we can see in the figure, at a throughput of 6Mbps, the constraint length 8 implementation consumes approximately 1.6W. If the received SNR improves to the point where a non-coding radio chain implementation can provide the required BER, an adaptive reconfiguration can reduce power consumption to under 1W, a saving of over 60%.

In this figure, we have only considered the resultant *dynamic* power reduction through adaptation at the application level. It must be also noted that using FPGA partial reconfiguration also contributes to reducing the *static* power consumption, given that we can time-multiplex functionality on the same physical resources, enabling the use of a smaller FPGA, hence reducing the static power.

Reconfiguration of the FPGA, when required, takes a time in the order of tens of milliseconds [4]. However, the adaptive applications we consider are designed to adapt over longer terms of minutes or hours, just as channel conditions are slow-changing, so this is not considered a significant overhead. During reconfiguration, the partial region is disabled, and thus consumes no power. The energy required to write the new configuration is minimal as it is done serially via the ICAP port.

This application presents a possible use of our system model and software architecture. The architecture is applicable well beyond the example presented here, which only serves as a proof of concept, highlighting both the benefits of using partial reconfiguration for adaptive systems, and the benefits of abstracting adaptation from reconfiguration.

5 Conclusions and Future Work

We have presented a system model and software architecture for implementing adaptive applications, harnessing the performance and reconfigurability advantages of modern FPGAs. By adopting an application level approach to adaptation, we enable system designers to implement self-adaptive applications in which the computation can exploit the performance of hardware, while the adaptation is managed in software code that can be modified easily, and use reasoning algorithms not suited to hardware implementation. This enables us to narrow the gap between application level adaptation and FPGA reconfiguration. We showed an example application that has been fully tested and demonstrated.

Our main aim at present is to continue work on the *Composer*, to improve its level of automation. We also intend to explore methods by which to exploit some of the more recent advances in FPGA partial reconfiguration. Finally, we are exploring the use of more advanced cognitive algorithms within the control plane, aided by the fact that these only need to be implemented in software.

Acknowledgements

The authors acknowledge Stephen Neuendorffer and Kees Vissers of Xilinx Labs for their support in providing the Linux on FPGA implementation used in this work.

This research project IP20060367, is funded by Enterprise Ireland under its Innovation Partnership scheme.

References

[1] S. Haykin, “Cognitive radio: Brain-empowered wireless communications,” *IEEE J. Sel. Areas Commun.*, vol. 23, no. 2, pp. 201–220, Feb. 2005.

[2] F.J. Mullany et al., “Self-deployment, self-configuration: Critical future paradigms for wireless access networks,” in *Workshop on Autonomic Communication (WAC)*, 2004.

[3] R. Lysecky and F. Vahid, “A configurable logic architecture for dynamic hardware/software partitioning,” *Design, Automation and Test in Europe (DATE)*, 2004.

[4] P. Lysaght et al., “Enhanced architectures, design methodologies and CAD tools for dynamic reconfiguration of Xilinx FPGAs,” in *International Conference on Field Programmable Logic and Applications (FPL)*, 2006.

[5] M. Hübner et al., “New 2-dimensional partial dynamic reconfiguration techniques for real-time adaptive microelectronic circuits,” in *International Symposium on Emerging VLSI Technologies and Architectures (ISVLSI)*, 2006.

[6] K. Paulsson et al., “On-line routing of reconfigurable functions for future self-adaptive systems - investigations within the æther project,” in *International Conference on Field Programmable Logic and Applications (FPL)*, Aug. 2007.

[7] N. Steiner and P. M. Athanas, “Autonomous computing systems: A proof-of-concept,” in *Engineering of Reconfigurable Systems & Algorithms (ERSA)*, 2007.

[8] C. Claus et al., “Using partial-run-time reconfigurable hardware to accelerate video processing in driver assistance system,” in *Design, Automation and Test in Europe (DATE)*, 2007.

[9] C. Kachris and S. Vassiliadis, “Performance evaluation of an adaptive FPGA for network applications,” in *Workshop on Rapid System Prototyping*, 2006.

[10] R. Tessier, S. Swaminathan, R. Ramaswamy, D. Goeckel, and W. Burleson, “A reconfigurable, power-efficient adaptive viterbi decoder,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 4, pp. 484–488, 2005.

[11] M. French, E. Anderson, and D. Kang, “Autonomous system on a chip adaptation through partial runtime reconfiguration,” in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Apr. 2008.

[12] J. Lotze, S.A. Fahmy, J. Noguera, L. Doyle, and R. Esser, “An FPGA-based cognitive radio framework,” in *Irish Signals and Systems Conference (ISSC)*, 2008.

[13] S. Neuendorffer and C. Epifanio, “Generic partially reconfigured processor systems applied to software defined radio,” in *SDR Forum Technical Conference*, 2007.

[14] G. Wrigley et al., “ReConfigME: A detailed implementation of an operating system for reconfigurable computing,” in *Parallel and Distributed Processing (PDP)*, 2006.

[15] Ettus Research LLC, Mountain View, California, USA, *Universal Software Radio Peripheral – The Foundation for Complete Software Radio Systems*, Nov. 2006.