COMMA: A Communications Methodology for Dynamic Module-based Reconfiguration of FPGAs

Shannon Koh and Oliver Diessel

School of Computer Science & Engineering
The University of New South Wales
&
Embedded Real-Time, and Operating Systems (ERTOS) Program
National ICT Australia
Kensington Laboratory
Sydney, Australia

THE UNIVERSITY OF
NEW SOUTH WALES

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Abstract

On-going improvements in the scaling of FPGA device sizes and time-to-market pressures motivate the adoption of a module-oriented design flow for the development of applications. At the same time, economic factors encourage the reuse of smaller devices for high performance computational tasks. Like other researchers, we therefore envisage a need for dynamic reconfiguration of FPGAs at the module level. However, proposals to date have not focussed on communications issues, have advocated use of specific protocols, cannot be readily implemented, and/or do not support current device architectures. This paper proposes a methodology for the rapid deployment of a communications infrastructure that efficiently supports the communications needs of a collection of dynamic modules when these are known at design time. The methodology also provides a degree of flexibility to allow a range of unknown communication requirements to be met at run time. Our aim is to support new tiled dynamically reconfigurable architectures such as Virtex-4, as well as mature device families. We assess a prototype of the communications infrastructure and outline opportunities for automating the design flow.

1   Introduction

Modern FPGAs are large, complex, heterogeneous devices that present many challenges to designers wishing to fully utilise device capabilities. Design complexity, verification, and time-to-market pressures encourage reuse of components and designs that are tried and proven. Module-based design methodologies form a class of higher-level design methods that focus on implementing a design that is specified or described in terms of its constituent modules [3][6]. As such they enhance reuse, allow designs to be rapidly developed, tested and deployed, and complement the engineering/marketing advantages of FPGAs as system components.

FPGAs are most commonly included as components of embedded systems in which performance requirements demand hardware support, yet cost, time-to-market, the need to provide alternative hardware components, or the need to revise hardware-based components over time preclude the use of ASICs, ASIPs, or processors alone. More often than not, that part of the system design that is targeted at the FPGA is static, or doesn't change while the system is operating. However, several applications for FPGAs have been identified in which the FPGA user circuitry is reconfigured while the embedding system remains active.

To date the most significant use of such dynamic reconfiguration is to support so-called hardware virtualisation, several flavours of which can be identified. In order to make do with insufficient FPGA area, a large FPGA circuit might be temporally partitioning into components that are swapped over time in order to map a large circuit into a smaller device [12]. When the computation can be temporally partitioned, the decision to virtualise may be made without the need to do so. Instead of mapping the design to a large device, a smaller one can be used in order to reduce part cost and power consumption. Virtualisation also allows application circuits to be specialised for one application instead of another, e.g. to implement 3G multi-standard wireless basestations [9].

As device sizes continue to scale, we envisage an alternative use of dynamic reconfiguration that may eventually dominate. Conceivably, complex systems involving multiple subsystems will be able to share expensive (in terms of chip area and power) and underutilised (in terms of functional density) resources such as FPGAs. This will give rise to the desire to multitask FPGA devices. Another factor likely to influence the emergence of multitasked FPGAs is that devices are scaling much faster than IP size. Multitasking offers a means of utilising the available resources and further reducing system part counts.

Allowing system designers to exploit dynamic reconfiguration raises many challenges. Of foremost concern is that we need tools to help a designer decompose the application into modules that are potentially reconfigured, or swapped while the system in which the FPGA is embedded is active. Such tools will, as a minimum, need to define the interfaces of modules as well as their timing characteristics. The partitioning of the system into modules cannot be done without reference to the resources available to implement the modules, including, necessarily, the communications infrastructure available to connect the modules together. The modules themselves may be sourced externally from IP vendors or developed in-house, which raises questions about consistency of definitions and descriptions of interfaces and behaviours. Simulation of the module communication needs to include timing. Place and route tools need to operate at the module level, and assuming the required communications can be provided or codesigned, not perform global design or optimisation steps.

The designer is also likely to want some form of runtime or operating system to be generated or provided that can hide the burden of managing the device. The OS defines policies and strategies for utilising and sharing resources, which must be fed back to the design system for it to be able to partition and adapt the implementation accordingly. Design iteration may be needed to accommodate the constraints of the device and its runtime management. Overheads should be minimised so as not to lose the benefit of hardware implementation. Moreover the penalty for designing at a higher level of abstraction and supporting virtualisation should not be large. With respect to dynamic module placement, it is desirable that modules be relocatable in order to support schedules that are not known at design time. This places a significant burden on the communications infrastructure to connect, when required, specific IO pins with module ports whose locations are not known until runtime. Inter-module connections involving dynamically placed modules may also need to be provided at runtime. A conflict thus arises between the need to provide fast routes of adequate width for the sake of performance and the constraint of finding and setting these at runtime, conceivably with little laxity in the task.

Perhaps the greatest challenge researchers and designers currently face is a lack of vendor support for their enquiries into or use of dynamic reconfiguration. Lack of support takes two forms: sub-optimal device architectures and inadequate tool support. While the concepts underlying support for dynamic modules in a general manner have been investigated since Brebner's Swappable Logic Units [2], little real progress has been made in developing widely applicable solutions with sufficient design tool support. With the release of new architectures [16], which give more recognition to the

scalability of reconfiguration mechanisms, the time seems right to move forwards on practical and effective methods for harnessing reconfiguration at the module level.

It is our contention that effective module-based reconfiguration of FPGAs requires at its foundation a communications infrastructure that can support the varying communications needs of the runtime configured modules. Invariably, these dynamically placed modules will need to connect with other modules on the FPGA and to the IO pins of the device. Since the interfaces of the modules and their runtime placement will in general vary, the communications infrastructure needs to support runtime reconfiguration of the routing, or provide an indirect mechanism for connecting ports with pins. This paper proposes a methodology for the deployment of a communications infrastructure that efficiently supports the communications needs of a collection of dynamic modules when these are known at design time. The methodology also provides a degree of flexibility to allow a range of unknown requirements to be met at run time.

In the following section, we outline a classification of dynamically reconfigurable systems from a module orientation that provides a framework for discussing related work and our contribution. In Section 3, we discuss related contributions. Our methodology and the target architecture are outlined in Section 4. We describe our progress to date and our assessment of a prototype of the methodology in Section 5. Further work, including our plans for supporting the methodology with automated tools is presented in Section 6. We conclude the paper with a summary of the main points in Section 7.

2   A Hierarchy of Dynamically Reconfigurable Systems

System designs that target FPGAs may consist of a static collection of sub-components or modules. That is, the design process results in a static circuit that is configured onto the FPGA until a system reset occurs. We define the FPGA design to be static since it is not altered while the system is active, each configuration is completely defined at design time, and there is sufficient time between resets to carry out the design to completion. Design methodologies for this type of FPGA use are relatively well understood and supported by FPGA tools, although a commonly accepted module-based orientation to the design of large-scale static designs may still be lacking.

By virtue of their reconfigurability, systems including FPGAs may exhibit a degree of flexibility and dynamism. This may take a number of forms and we propose a hierarchy based on the constraints imposed on the reconfigurable system in order to provide a framework for subsequent discussion.

First, we distinguish between complete and partial reconfiguration. In our work, we are concerned with supporting partial reconfiguration, in which part of a FPGA device is reconfigured to support new functionality. Moreover, we are interested in supporting dynamic reconfiguration as much as it may be permitted with commercial devices. This means allowing part of a device to be reconfigured while the system in which it is embedded, or indeed, the parts of the FPGA device that are not being reconfigured, remain active. We consider complete reconfiguration of the device to be an incarnation of the static design case described above. It is our intention to support static designs as a special case within our methodology.

Second, the thrust of this work is to support module-based or core-based reconfiguration, such as the replacement of one video codec with another, rather than fine-grained reconfiguration, such as the specialisation of a constant coefficient multiplier. Our approach is not intended to preclude use of fine-grained reconfiguration, rather our focus is on how to support modular design techniques and in particular the communications issues raised by dynamically reconfiguring design modules.

Apart from static designs, three types of dynamically reconfigurable systems, herein denoted DR1, DR2 and DR3 respectively, can be distinguished. DR1 is the most constrained of these and applies to systems in which we know at design time which sub-component of a design may be swapped with a given (set of) alternative(s) and, crucially, which region of the placed and routed circuit is to be reconfigured with a given dynamic module (see Figure 1 for a diagrammatic comparison of this situation with the static case). Part of the FPGA circuit design is static and another part of the circuit is to be swapped at run time. Many examples of such systems have been reported. For example a bluetooth baseband bitstream processor is reconfigured after processing the header packet to process the payload in [4]. In this class, the communication needs (bit- and band-widths), and thus the interface of the module alternatives are known in advance. Connecting a dynamically placed module to the surrounding circuitry and device pins therefore involves selecting from a number of preconfigured channels. It is worth mentioning that for DR1, the triggers for reconfiguration and all possible schedules are known at design time.
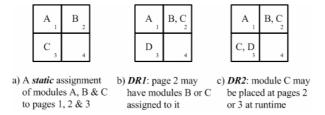
a) A *static* assignment of modules A, B & C to pages 1, 2 & 3

b) *DR1*: page 2 may have modules B or C assigned to it

c) *DR2*: module C may be placed at pages 2 or 3 at runtime

Figure 1: Classification of module-based reconfigurable systems

The class DR2 includes those systems for which the static components and the set of possible dynamic modules is known at design time, but their placement onto the FPGA fabric at runtime is not known. A representative example is depicted in Figure 1 (c). It is possible for this situation to occur when we do not know the order in which dynamic modules are called for at runtime. A possible scenario exemplifying this case is a robotic explorer that dynamically partitions real-time tasks, perhaps including vision, navigation, scientific sampling, and communications onto hardware and software subsystems according to time, area and power constraints. This situation gives rise to the need to support flexibility in the communications infrastructure in order to support the necessary interconnection between a dynamic module and other on- and off-chip components. However, given the interfaces of the dynamic components are known at design time, it is possible to engineer an inter-module communications infrastructure that supports expected requirements.

For the sake of a complete classification, we anticipate scenarios in which the set of modules for which communications needs to be supported is not known at design time. Such systems are classified as DR3 in our scheme. An example of such a system may be one in which modules arrive dynamically at runtime, perhaps in response to a user choosing to run an application downloaded from the internet. Since we consider this scenario futuristic, our thoughts on how to support such systems are not well formed. We anticipate providing communications infrastructure that is parameterised on maximum bit- and band-widths that can be supported. Applications that comprise modules requiring more resources would be rejected.

3   Related Work

Previous research into support for dynamic modules has considered many issues. Few researchers have considered the communications problem in detail or centred their proposals on the mechanisms needed to support the communications requirements of dynamic modules. The following is a representative selection of the variety of systems proposed to date.

Xilinx Application Note 290 [14] is the first and currently only commercial modular partial reconfiguration methodology known to the authors. All communication between reconfigurable modules and reconfigurable or fixed modules goes through three-state bus macros as shown in Figure 2.
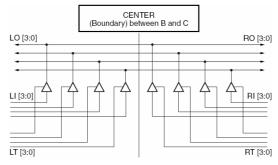

Figure 2: XAPP290 Bus Macros

Inter-module communications are limited to those passing through the bus macros and only abutting modules may communicate with one another. Thus if modules A, B and C are placed in order left-to-right, A must rely on B to send signals through to C if it wishes to communicate with it. This is unacceptable to a designer of module B if there is no knowledge of this requirement *a priori*, and also interferes with B's internal routing thus possibly affecting performance and area. The methodology works best with fixed sized slots into which to place modules and constant interfaces between the alternative modules. XAPP290 is essentially a module version-swapping methodology that can support DR1, but is likely to be too unwieldy and limiting for implementing DR2 or DR3 systems. Finally it does not support the newer Virtex-4 devices, as they do not have tri-state lines.

In [7], Kalte et al. also describe a reconfigurable system implementation, depicted in Figure 3, in which all inter-module communication is performed through tri-state lines. However, modules may have arbitrary width, which can reduce fragmentation and thereby help to boost utilisation in high load situations. In their system, all modules have to implement the AMBA bus protocol, which may limit the types of modules that can be used. The overheads of managing this bus system can be relatively high. Virtex-4 devices are not supported due to their lack of tri-state lines.
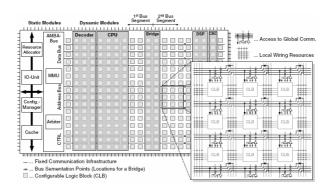

Figure 3: 1-D Relocatable Modules

Bobda et al. have described a dynamic network-on-chip that allows for modules to be plugged into a network of routers as shown in Figure 4 [1]. However, they do not describe how the modules communicate through this network, what messages are sent, and how addressing is performed. The module-router interface is not specified and it seems that at this point each module has to be tailor-made for this network, again limiting the use of currently available IP.

Marescaux et al. [10] proposed a network-on-chip utilising the XAPP290 methodology and wormhole routing as shown in Figure 5. This approach not only inherits the limitations in XAPP290 but also requires that some form of network interface be provided in order for IP to be used.
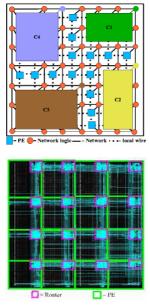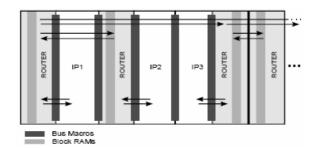

Figure 4: Dynamic NoC

Figure 5: Wormhole Routing

Horta et al. described a dynamic hardware plugin platform as shown in Figure 6 [5]. Their approach uses a separate FPGA to implement the communications network. All communication has to bear the overheads of communicating off-chip to the NID FPGA, which may be quite unacceptable or unnecessary.
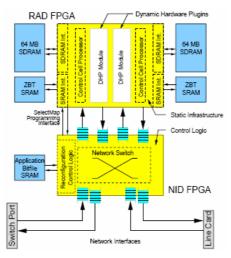

Figure 6: Dynamic Hardware Plugins

4   The COMMA Approach

The approach we chose to follow is founded on the following principles:

1.   We acknowledge the need to provide practical and efficient methodologies that minimise constraints on users.
2.   We need to support design with tools.
3.   For tools to have significant benefit they should look forwards to the capabilities of future devices.
4.   We will focus on the communications needs of dynamic modules.

We propose to develop a tool-suite to automatically generate on-chip communications infrastructure for dynamic modules. Our approach aims to provide infrastructure that can be implemented on a range of devices and platforms and at the same time is optimised for the hardware, application requirements and available design-time knowledge.

4.1   Reference Target Device

The reference device family we will use in order to describe and demonstrate our approach is the new Xilinx Virtex-4 family of FPGAs.

The frames in Virtex-4 devices are unique in that they are of a fixed-length of 41 quad-byte words, each spaning 16 CLB rows. The frames are tiled about the device into "pages" that span half the width of the device and 16 CLB rows, as shown in Figure 7, which depicts a Virtex-4 FX12 device with 64 rows and 24 columns of CLBs (and includes one hard PowerPC core, depicted as a white rectangle in Figure 7). This is the smallest device available in the FX family. The external I/O banks are also located on the left, in the middle and on the right of the device (shown in Figure 7 as black bars), rather than around the periphery in predecessor families. In addition, the left and right banks in some Virtex-4 device/package combinations (e.g. XC4VFX100-FF1517-10) are located 6 CLBs away from the edges of the device.

4.2   Module Placement Strategy

4.2.1   *Paged Module Placement*

Since each frame spans 16 CLBs vertically and the minimum unit of reconfiguration is one frame, it can be seen that we should be able to reconfigure any of the 8 pages shown in Figure 7 independently of every other. Were modules mapped to these pages, they could be dynamically swapped while other modules are left running.

We propose that each page should accommodate at most one reconfigurable hardware module. While it is possible to reconfigure less than a page, the overheads (such as placement and defragmentation) of micro-managing modules (e.g. arbitrary placement akin to [7]) are likely to be unacceptable and unnecessary. However, there may be reconfiguration delay savings if the module size can be reduced.
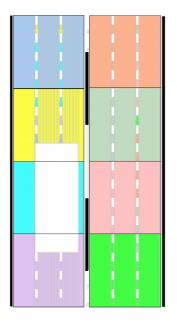

Figure 7: XC4VFX12 Pages and I/O Banks

Another advantage to this approach is that each clock region on the device corresponds to the pages shown in Figure 7, thus each module can be clocked independently, and with the new support for dynamic reconfiguration of functional blocks [15], DCMs may be dynamically reconfigured to adjust the clock frequency in each page independently.

4.2.2   *Page Aggregation*

This placement strategy opens up possibilities of placing a module in two or more adjacent pages such that larger modules can be accommodated.

Since the centre column effectively divides the FPGA resources into two halves it is preferable to aggregate pages vertically instead of horizontally. Since the ratio of rows to columns in Virtex-4 devices is always large (more than twice the rows than columns on average) it is natural to do so. This arrangement also allows carry chains to be expanded.

4.2.3   *Horizontal Page Division*

Since the minimum unit of reconfiguration is one frame it is possible to divide each page into sub-pages at the granularity of 1 CLB. This may be desired if it is known that there will be very small modules available and that some space savings are necessary.

The sample module placement in Figure 8 shows modules M1 and M2 being placed in a divided page, M3 in three aggregated pages and M4 in a page of its own.
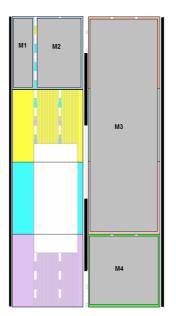
Figure 8: Page Aggregation and Division

It should be noted that division and aggregation does not need to be permanent. It is possible to place a module utilising the entire page that M1 and M2 takes up when they are removed. Similarly we are able to place a smaller module into any of the three pages that M3 occupies when it is removed.

4.2.4   *Predecessor Device Families*

The above concepts can also be implemented on predecessor device families if the entire device is viewed as a single page. The modules are then placed into horizontally divided subpages.
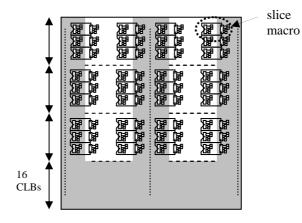
4.3   **Pin Virtualisation**

4.3.1   *Communications Infrastructure*

One of the unresolved issues with respect to current solutions is the lack of access to module and/or external I/O Pins. This limits bandwidth, placement flexibility, and restricts implementation to particular devices and/or platforms.

Our approach allows virtualisation of any module and external I/O pin in the fabric to allow any module to access any other module's pins and any external I/O pin.

This can be implemented by designing a communications infrastructure that envelops the external I/O pins used by the system for communication.

The stylised layout of a possible maximal configuration of the communications infrastructure is shown in Figure 9. The grey area, where the infrastructure resides, envelops every external IOB. Each module area shown in the diagram above occupies a little less than one page and is connected to the infrastructure via 8-bit LUT-based slice macros [11] (see Figure 10 below). Note: the number of and size of macros in Figure 9 is not illustrative of actual implementation scenarios.

Figure 9: Trident Layout

The "trident" layout has homogeneous module areas, thus allowing for simple means of relocation utilising major-address modification techniques such as REPLICA [8]. However, the critical path from the top-left to the top-right corner is long.

Slice macros are used in place of XAPP290-like tri-state bus macros in order to provide connection to the communications channels that are routed via the infrastructure. Any combination of inputs and outputs (up to 8 bits) for two vertically or horizontally adjoining CLBs is possible. Figure 10 depicts the infrastructure at the left side of a module; it is equally possible to have the module on the left side of the infrastructure.
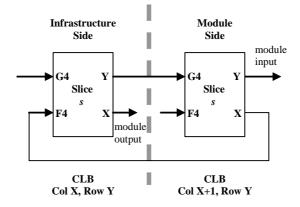

Figure 10: Macro depicting one input and one output to/from a module within the one slice

The "double-ring" layout depicted in Figure 11 overcomes the critical delay of the "trident" layout at the cost of slightly greater relocation complexity.
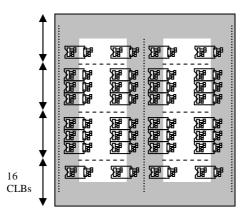
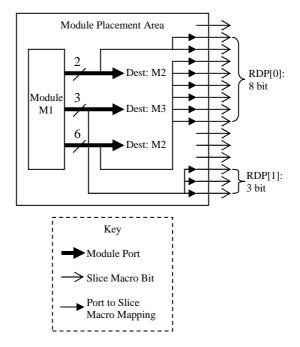

Figure 11: "Double-Ring" Layout

This layout provides better routing opportunities but three different types of module placement areas now exist (the two at the top, the four in the middle and the two at the bottom). As reconfiguration should be glitchless in Virtex-4 devices reconfiguring modules in the top and bottom module areas should not be an issue, but only modules of the same type can be relocated.

It is important to note that these observations simply imply that the communications infrastructure should be free form and optimised to the needs of the application. For example, if not all the external I/O pins are required the communications infrastructure need not envelop the pins that are not required.

4.3.2   *Reconfigurable Data Ports*

Each module area is capable of connecting a large number of bits to the communications infrastructure (a maximum possible 8×2×16=256 bits of communication for a module using all CLBs in its leftmost and rightmost columns for implementing slice macros.).

We introduce Reconfigurable Data Ports (RDPs) as a means mapping module ports to slice macros. In Figure 12 the two output ports destined for module M2 are mapped to the one 8-bit RDP (because they can be routed together) whilst the 3-bit port has its own 3-bit RDP. The purpose of the RDPs is to map the module's interface to the slice macros.

Figure 12: RDP Mapping

The definition of RDPs allows communication channels to be set up in the infrastructure. Channels of varying bitwidths can be defined to ease routing complexity (it is less complex to route a group of bits because there are fewer possible destinations than if each bit were individually routed).

4.3.3   *Implementation and Optimisation*

The communications infrastructure can be optimised depending on how much knowledge is known *a priori* about the modules, and any user-imposed constraints.

The following implementation details refer to the reconfigurable classification hierarchy in Section 2:
- *DR3:* The user specifies a fixed number of channels and their bitwidths, the necessary external I/O pins, and appropriate number of slice macros to provide per module so that the infrastructure can be optimised.
- *DR2:* Modules are available *a priori* and appropriate tools can take over some tasks of specifying the communications requirements, including tailoring of the module areas (i.e. size, location and slice macros) for optimal use of the module set.
- *DR1:* Further optimisations are possible in addition to those specified in DR2 as the routing is now entirely static since the placement is fixed.
- *Static:* This will be similar to a standard modular design flow and is simply a more restricted case of DR1 where there is only one possible module in each area.

4.3.4   Management Issues

The management of modules arriving and leaving and the setup of channels and routes may be performed by on- or off-chip agents.

The essential requirements to be fulfilled for such agents (with respect to communication) are:
- *Module registration:* The registration of module IDs in order to locate peer modules when modules request communication.
- *RDP registration:* The registration of module interfaces for channel setups.
- *Channel registration:* The communications infrastructure should be informed as to what routes should be set up.

The requirements may be performed by a host controller or by an embedded CPU core such as the PowerPC block in Virtex-4 FX devices.

4.4   Design Automation and Tool Support

In order for this infrastructure to be readily implemented we propose the development of tools to perform two tasks – the automated generation of the communications infrastructure, and the preparation of modules (e.g. RDP definition) for placement. Figure 13 depicts the complete design and tool flow for communications synthesis. The flow includes three tools that we intend to create, and an incarnation of a partial reconfiguration tool flow (see Section 5.3).

This pre-processing tool (probably with a GUI) uses Xilinx-supplied device information with user-supplied parameters to create a Chip and Communications Configuration (CCC) file, which will serve as the input for the next tool.

The user may specify a layout of the communications infrastructure and channel, slice macro and slot preferences to assist the infrastructure generation tool in optimisation. The IO pad parameters are mandatory and are used to place and optimise the infrastructure.

The infrastructure generator analyses the CCC file created by the configurator and generates an optimised communications infrastructure consisting of HDL, constraints, slice macros and accessory macros.
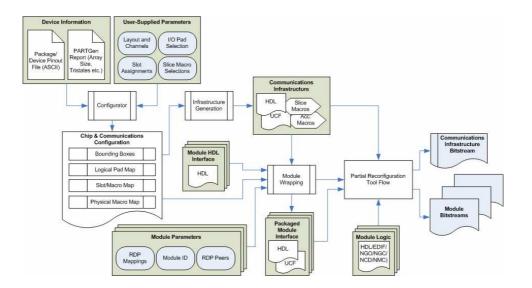

Figure 13: Design and Tool Flow for Communications Infrastructure Synthesis

Once the infrastructure has been generated, modules can be wrapped to enable placement into the infrastructure at any time thereafter (even after deployment) by using the wrapper tool to generate RDP interfaces for the modules.

When the infrastructure and an initial set of modules (which may be blank fillers) are available, they should be synthesised using an available partial reconfiguration flow (e.g. the XAPP290 flow, which will support Virtex-4 devices in ISE 8.1i, or a difference-based flow), and the bitstreams can be generated.

The system can then be deployed. Additional modules that may be added in the future can be wrapped for use as long as the CCC and infrastructure files are present.

5 The development so far, and its assessment

As the development of the tool is in its early stages, the experimental work done to date has been to determine methods to design, place and automate the generation of a communications infrastructure for a chip.

To date we have synthesised and placed a trident-layout communications infrastructure on a Virtex-4 FX12 chip with 4 module areas having 8 inputs and 8 outputs each, and 16 external I/O pins (8 inputs and 8 outputs).
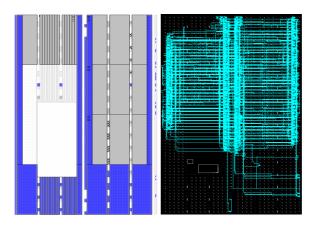
5.1 Simple Multiplexed Inputs

So far, we have implemented a circuit-switched prototype using LUT-based multiplexers, to route each bit coming into the infrastructure onto every outgoing bit. The Virtex-4 CLB architecture allows the implementation of a 2:1 multiplexer with one LUT, a 4:1 with one slice, an 8:1 with two slices and a 16:1 with four slices (or one CLB). The delay of a 4:1 MUX is 0.35ns, an 8:1 is 0.55ns and a 16:1 is 0.75ns.

A DR3 implementation with m inputs and n outputs will require approximately $n\lceil \log_{16} m \rceil$ CLBs. Thus our implementation occupies about 40 CLBs in total (1 for each module input and each external pin output).

5.2 Host Control

The control of the communications infrastructure is implemented with a simple interface supporting the minimal instruction set of 6 instructions listed in Table 1 to set the multiplexer selectors in each of the module inputs or external pin outputs.

Figure 14: Constraints and Actual Placement

This instruction set utilises SMC (Slice Macro Connector) numbers, which are logical numbers mapped to actual slice macro positions (corresponding to the physical macro map in the CCC file).

Instruction	Description
SSMC <smc_num>	Sets the active target SMC bit number.
SEIO <eio_num>	Sets the active target external I/O pin bit number.
DTYP <mod/eio>	Signifies if a module or external I/O pin is connected to this SMC.
ASRC <log_port>	Assigns a source port number (either a corresponding logical SMC number of a logical external I/O input) as the source peer of the

	current target SMC or external I/O pin.	
SLEN <slot_num>	Enables the module at the specified slot number.	
SLDS <slot_num>	Disables the module at the specified slot number.	

Table 1: Controller Instruction Set

Before placing a module's bitstream onto the device, the host executes the following instruction sequence per bit of communication:

1. SSMC or SEIO to select a particular bit of communication.
2. DTYP to specify if this bit connects to another SMC or to external I/O.
3. ASRC to specify the source of communication (another SMC or external I/O pin).

The host can then place the bitstream onto the device and execute SLEN to enable the slot and start the module running.

To remove a module the host executes SLDS to disable the slot, and then places a blank module bitstream onto the device to clear the configuration memory.

5.3   Implementation

5.3.1   *Unconstrained Routing Issue*

The communications infrastructure is implemented in HDL with the use of appropriate primitives for fast multiplexing, and synthesised with slice macros and modular placement constraints similar to those of XAPP290 as shown on the left in Figure 14.

Due to the lack of support for partial reconfiguration of Virtex-4 in ISE 7.1i, the internal routing of modules ignore the area constraints specified, as can be seen on the right in Figure 14. However, the placement is bound to the constraints. The router simply chooses the best paths while crossing module boundaries.

Since there are routes going through module boundaries special care must be taken to prevent internal module routing from using the infrastructure routes.

In order to do this we have adopted a special design flow to implement the modules. This flow, depicted in Figure 15, implements the "Partial Reconfiguration Tool Flow" process of the complete design flow outlined in Section 4.4 and depicted in Figure 13.

This special design flow executes multiple iterations of the Xilinx Modular Design Flow [13]. The first part of the flow uses the generated communications infrastructure as a module and executes the initial budgeting and active module phases of the modular design flow, resulting in a placed and routed circuit (NCD) of the communications infrastructure.

At this point the NCD design is converted to a hard macro (NMC) and is reinstantiated into the top-level, but as a hard macro this time instead of as a module. After the second initial budgeting phase, the Implementable Top Level NGC will contain the infrastructure as well (instead of just the slice macros the first time around). Blank modules (simple dummy modules that are never enabled) can then be inserted to generate an initial top-level bitstream with final assembly.
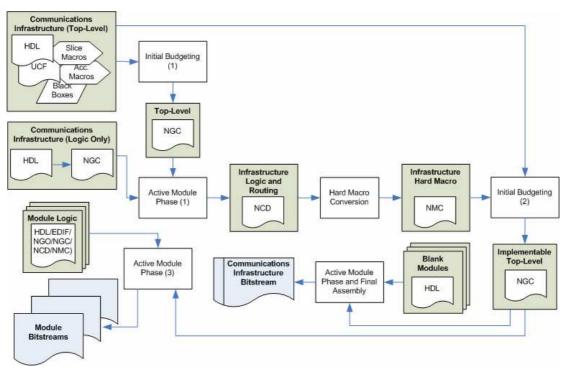
Figure 15: Partial Reconfiguration Tool Flow

Multiple iterations of the active module phase can then be executed to generate individual partial module bitstreams that are ready to be loaded onto the device.

5.3.2 Performance and Area

The maximum clock speed of the communications infrastructure after place and route (i.e. the hard macro) as per the tool flow above is 358MHz (speed grade –10 i.e. the slowest available), which should be sufficient for most modules.

The statistics of this implementation are as follows:

- *Module Placement Area (defined as a "Slot" since the infrastructure occupies some of the page area):* 576 slices (9 columns × 16 rows × 4 slices) per slot; 4 slots in total.
- *Infrastructure Area Consumption:* 572 slices (~10.5% of available chip area).
- *Infrastructure Area Overhead per page:* 3 CLB columns.

The following area and performance statistics are provided for comparison:

- 8-bit Divider: 104 slices, 234 MHz
- DES Core: 476 slices, 182 MHz
- Triple-DES Core: 623 slices, 132 MHz
- MicroBlaze Processor: 988 slices, 200 MHz
- OpenRISC CPU Core: 9172 slices, 28.5 MHz

The FX12 thus accommodates a DES core with room to spare in one slot and can fit a MicroBlaze processor in two slots.

It should be noted at this point that this implementation utilised the smallest FX-family device in the Virtex-4 range (also the one with the least number of slices). Table 2 shows hypothetical but reasonable module areas based on available chip area.

Device	CLB Array	Module Width	Module Slices	Num Mods	Load Time (ms)
LX15	64 x 24	9 (-3)	576	6	0.32
LX25	96 x 28	10 (-4)	640	10	0.37
LX40	128 x 36	13 (-5)	832	14	0.47
LX60	128 x 52	21 (-5)	1344	14	0.76
LX80	160 x 56	21 (-7)	1344	18	0.76

LX100	192 x 64	24 (-8)	1536	22	0.87
LX160	192 x 88	36 (-8)	2304	22	1.3
LX200	192 x 116	50 (-8)	3200	22	1.8
SX25	64 x 40	16 (-4)	1024	6	0.58
SX35	96 x 40	15 (-5)	960	10	0.54
SX55	128 x 48	18 (-6)	1152	14	0.65
FX12	64 x 24	9 (-3)	576	6	0.32
FX20	64 x 36	15 (-3)	960	6	0.54
FX40	96 x 44	18 (-4)	1152	10	0.65
FX60	128 x 52	21 (-5)	1344	14	0.76
FX100	160 x 68	27 (-7)	1728	18	0.97
FX140	192 x 84	34 (-8)	2176	22	1.23

Table 2: Module Area Projection

The individual module width is calculated by dividing the array width by 2 and subtracting what is projected to be a reasonable infrastructure overhead width (in parentheses) given the device provides over 128 wires per CLB column. Table 2 also lists the number of modules supported by each device and the time in ms to load the configuration for a complete module into a slot.

6 Further Work

6.1 Automation

Since the methodology and tool flow have been determined, the next step is to implement the tools identified in Section 4. We plan to implement the *Configurator*, *Module Wrapper* and *Partial Reconfiguration Tool Flow* and combine them to form a development environment.

6.2 Optimisation

Optimisation of the routes connecting slice macros and external I/O pins relies on knowledge of the communications requirements at design time. Methods for determining efficient routing infrastructures are to be determined and incorporated into the *Infrastructure Generation* tool.

6.3 Analysis of Different Layouts

We have introduced two layouts in this paper – the trident and double ring. These layouts are not optimal but are suggested as possibilities. Once the tools are complete, an analysis of alternative layouts can be performed to determine routing strategies.

7 Conclusion

In this paper we motivated the need for a module oriented methodology to cope with design pressures and expected device scaling. We discussed support needed for dynamic reconfiguration at the module level and argued that the provision of a flexible communications infrastructure that affords a degree of pin address indirection is desirable. We then presented the COMMA approach to supporting communications for new tiled FPGA architectures and outlined design flows to enable its rapid deployment. A prototype was implemented on a Virtex-4 FX12 device and the resulting design was assessed. Our plans for further work were outlined.

References

[1] C. Bobda, A. Ahmadinia, M. Majer, J. Teich, S. Fekete, and J.v.d. Veen. DYNOC: A dynamic infrastructure for communication in dynamically reconfigurable devices. In International Conference on Field Programmable Logic and Applications, 2005, pp. 153 – 158.
[2] G. Brebner. The Swappable Logic Unit: A paradigm for virtual hardware. In IEEE Symposium on FPGAs for Custom Computing Machines, 1997 pp. 77 – 86.
[3] L. Chaouat, S. Garin, A. Vachoux, and D. Mlynek. Rapid prototyping of hardware systems via model reuse. In 8th IEEE International Workshop on Rapid System Prototyping, 1997, pp. 150 – 156.
[4] J. Esquiagola, G. Ozari, M. Teruya, M. Strum, and W. Chau. A dynamically reconfigurable Bluetooth baseband unit.

In International Conference on Field Programmable Logic and Applications, 2005, pp. 148 – 152.

[5] E.L. Horta, J.W. Lockwood, and D. Parlour. Dynamic Hardware Plugins in an FPGA with partial run-time reconfiguration. In 39th Design Automation Conference, 2002, pp. 343 – 348.

[6] G. Janac, T. Poltronetti, A. Herbert, and D. Rudusky. IP supply chain-the design reuse paradigm comes of age. Integrated System Design, Vol. 13, No. 141, Mar. 2001, pp. 66 – 70.

[7] H. Kalte, M. Porrmann, and U. Ruckert. System-on-programmable-chip approach enabling online fine-grained 1D-placement. In 18th International Parallel and Distributed Processing Symposium, 2004 p. 141.

[8] H. Kalte, G. Lee, M. Porrmann, and U. Ruckert. REPLICA: A bitstream manipulation filter for module relocation in partial reconfigurable systems. In 19th IEEE International Parallel and Distributed Processing Symposium, 2005, 8 pp.

[9] J. Kim, Dong Sam Ha and J.H. Reed. A new reconfigurable modem architecture for 3G multi-standard wireless communication systems. In IEEE International Symposium on Circuits and Systems, 2005, pp. 1051 – 1054.

[10] T. Marescaux, A. Bartic, D. Verkest, S. Vernalde, and R. Lauwereins. Interconnection networks enable fine-grain dynamic multi-tasking on FPGAs. In 12th International Conference on Field-Programmable Logic and Applications 2002, 795 – 805.

[11] P. Sedcole, B. Blodget, J. Anderson, P. Lysaght, and T. Becker. Modular partial reconfiguration in Virtex FPGAs. In International Conference on Field Programmable Logic and Applications, 2005, pp. 211 - 216.

[12] J. Villasenor, C. Jones, and B. Schoner. Video communications using rapidly reconfigurable hardware. IEEE Transactions on Circuits and Systems for Video Technology, Vol. 5, No. 6, Dec. 1995, pp 565 – 567.

[13] Xilinx Inc. Xilinx ISE 7.1 Development System Reference Guide, 2005.

[14] Xilinx Inc. Two flows for partial reconfiguration: module based or difference based. Xilinx Application Note XAPP290 Sep., 2004.

[15] Xilinx Inc. Virtex-4 Configuration Guide. User Guide UG071, Aug., 2005.

[16] Xilinx Inc. Xilinx Virtex-4 Family Overview. Preliminary Product Specification DS112, Jun., 2005.