Demand Driven Assembly of FPGA Configurations Using Partial Reconfiguration, Ubuntu Linux, and PYNQ

Jeffrey Goeders, *Member, IEEE*, Tanner Gaskin, *Student Member, IEEE* and Brad Hutchings, *Fellow, IEEE*
NSF Center for Space, High-performance, and Resilient Computing (SHREC)
Dept. of Electrical and Computer Eng., Brigham Young University

***Abstract*—The PYNQ system (Python Productivity for Zynq) is notable for combining a monolithic preconfigured bitstream, Ubuntu Linux, Python, and Jupyter notebooks to form an FPGA-based system that is far more accessible to non-FPGA experts than previous systems. In this work, the monolithic preconfigured PYNQ bitstream is replaced with a combination of a simple base bitstream containing several partial reconfiguration regions and a library of partial bitstreams that implement a variety of hardware interfaces such as: GPIO, UART, Timer, IIC, SPI, Real-Time Clock, etc., that interface to various Pmod-based peripherals. When peripherals are plugged into a Pmod socket at run-time, corresponding partial reconfigurations and standard device drivers can be automatically loaded into the Ubuntu kernel using device-tree overlays. This demand-driven, partially-reconfigured approach is found to be advantageous to the monolithic bitstream because: 1) it provides similar functionality to the monolithic bitstream while consuming less area, 2) it provides a way for users to modify or augment hardware functionality without requiring the user to develop a new monolithic bitstream, 3) run-time demand loading of partial bitstreams makes the system more responsive to changing conditions, and 4) implementation issues such as timing-closure, etc., are simplified because the base bitstream circuitry is smaller and less complex.**

***Keywords*-FPGA; Partial Reconfiguration; Linux; PYNQ**

I. Introduction

The PYNQ system (Python Productivity for Zynq) provides an accessible FPGA platform that consists of a preconfigured bitstream containing built-in support for a wide variety of internal and external peripherals [1]. This built-in support for a wide variety of peripherals is one of the main reasons that software programmers with no hardware expertise can immediately utilize PYNQ. Unfortunately, if you need hardware support that is not already built in to the preconfigured bitstream, you must modify the existing bitstream. This is a tall order that requires an experienced hardware designer.

As a solution, this paper presents a demand-driven approach where users can construct hardware configurations as needed, without the need for experience with the FPGA design tools. Our basic approach (and demonstration system) takes the IO circuitry that supports the PMOD pins and most of the Arduino shield pins on PYNQ, removes it from the base bitstream, and moves it into a library of partially-reconfigurable bitstreams (each partial bitstream currently supports a single IO standard). At run-time, the user requests support for some IO standard (SPI, I2C, SPI, UART, etc.) via a simple software (Python) API. The software API partially reconfigures the requested IO standard into PR regions now contained in the base bitstream and loads the necessary Linux drivers via Linux Device Tree Overlays, functionality that is standard in recent Linux kernels.

This work was supported by SHREC industry and agency members and by the IUCRC Program of the National Science Foundation under Grant No. CNS-1738550.

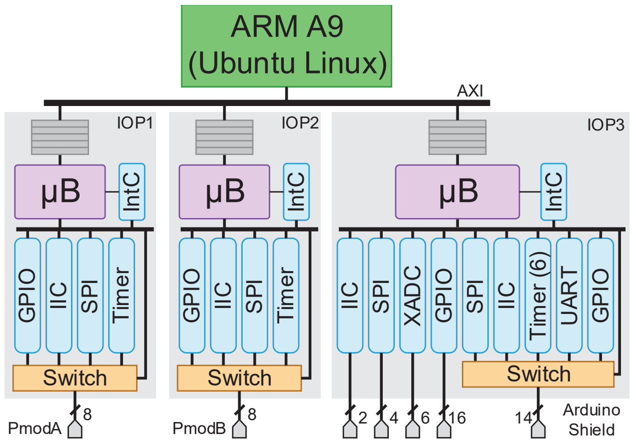
We believe that this approach offers several advantages over a standard monolithic PYNQ configuration. First, it offers greater flexibility, as the user can dictate both the type and quantity of modules they want in their system. Second, it offers greater scalability, allowing for tens or even hundreds of different possible modules, which would not be possible with an "all-in-one" approach. Third, should users venture into learning about hardware design, it offers a modularized system where each individual module is simple to understand and modify in isolation - much simpler than attempting to modify the existing base bitstream, for example. Finally, all of these benefits are provided while also reducing the logic footprint.

In adopting a platform where the hardware can be dynamically reconfigured, it is essential to also consider the impacts on the software model, particularly for a Linux system like PYNQ, where the kernel is expecting hardware to remain static. In the paper we consider both the hardware and software aspects of our proposed technique, including how the Linux Device Tree is modified at run-time in order to allow for demand driven hardware configurations.

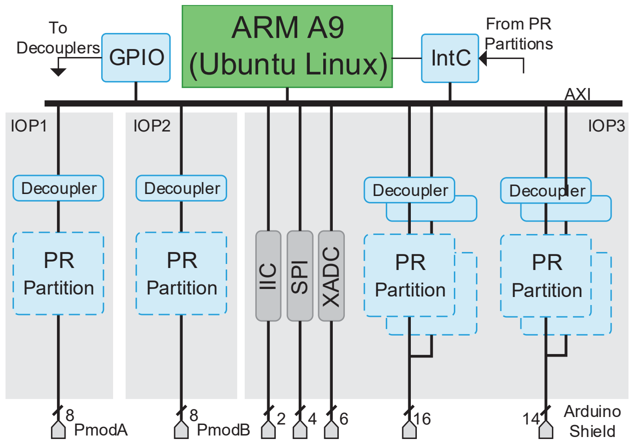
The major contributions of this paper include the following:

- The hardware architecture and software organization, particularly in the context of a Linux system, needed to support demand-driven hardware configurations.
- A publicly released demonstration of the demand-driven approach, implemented on PYNQ. This is available on Github at github.com/byuccl/byu-pynq.
- A comparison of the flexibility, performance, and area requirements of the demand-driven approach against PYNQ's standard monolithic bitstream.

The demand-driven assembly of hardware configurations,



(a) The base PYNQ External IO System.



(b) Our proposed PYNQ External IO System.

Figure 1: The PYNQ External IO system. The left-hand figure shows the Xilinx-produced base system configuration for PYNQ. This illustrates only hardware for dealing with the external IO pins on the PYNQ board (there are serveral hardware cores to deal with other I/O (audio, video, etc.) that are not the focus of this work and are not shown here). The right-hand figure shows our proposed hardware modifications to enable a demand driven configuration framework.

based upon partial reconfiguration is very scalable and will have application far beyond PYNQ.

This paper is organized as follows: Section II gives background on the PYNQ I/O system, Section III presents our hardware architecture for supporting demand-driven configurations. Section IV describes the Linux-based software organization required to support such a system. Section V provides an evaluation of our demonstration platform. A description of related work follows and Section VI concludes.

II. Background

A. The PYNQ IO System

The PYNQ system runs Ubuntu Linux on the ARM A9 CPU, and includes several IP cores implemented in the FPGA fabric to interact with both on-board and external peripherals. In PYNQ terminology, the FPGA configuration is referred to as an *overlay*; however, in this paper we use the term *configuration* to prevent confusion with other types of FPGA overlays that may not be central to this discussion.

A large portion of the resources in the base configuration are allocated to the external IO system. Figure 1a illustrates the base PYNQ configuration, where there are three separate IO systems (IOP1, IOP2, and IOP3). IOP1 and IOP2 manage the two Pmod connectors on the board, and IOP3 manages the Arduino shield pins. Each of these systems include several IP cores to implement different protocols on the IO pins (SPI, IIC, etc.), with a controllable switch to control which core has access to specific physical IO pins.

Included with each IOP system is a Microblaze soft processor which is used to execute the Xilinx bare-metal drivers for these I/O cores. The software executing on the Microblaze can communicate with the main ARM system processor via a shared memory, although for anything beyond the simple applications, the user is responsible for writing this communication software. Relative to the ARM processor, the Microblaze cores are relatively inaccessible, require the use of the embedded Xilinx Software Development Kit (SDK), and are more difficult to debug. While the Microblaze cores may be beneficial for real-time applications where latency is key, we believe the significant burden of learning the Xilinx SDK software, writing code for two different processors, and writing a communication program will often not be worth the benefit.

Overall, this monolithic *"all-in-one"* architecture requires a substantial number of resources; in total, the I/O systems consume 13000 LUTs (24% of the FPGA). Despite the high resource cost, this approach is mostly successful in its purpose. It provides designers with access to many different I/O protocols without requiring any hardware design expertise. Still, it is far from ideal. If new I/O controllers are needed, or even if the user simply needs to modify a compile-time configuration option, the entire hardware needs to be modified and recompiled, requiring an understanding of the FPGA flow. For example, in the PYNQ system, if a designer needed to change the UART baud rate or wanted to move the SPI controller to a different set of I/O pins, he or she would need to compile a new hardware configuration. Although we could add more and more I/O controllers into the configuration to provide more functionality out-of-the-box, it would require even more resources. This approach is not scalable beyond a handful of I/O configurations.

In the next section we discuss our alternative architecture that uses partial reconfiguration to load modules only when necessary.

I/O Controller	Bus	# I/O Pins	Size (LUTs)
GPIO	AXI4-Lite	1–32	25–249
UART (Uartlite)	AXI4-Lite	2	100–119
SPI (Quad SPI)	AXI4-Lite	4+	372–412
I^2C	AXI4-Lite	2	314–317
Timer/PWM	AXI4-Lite	4–6	96–256

Table I: Implemented and Tested Xilinx I/O Controllers.

III. Reconfigurable I/O Framework

A. The Hardware Architecture

Figure 1b illustrates our proposed architecture for leveraging partial reconfiguration to support user-defined configurations. Where the base PYNQ configuration (Figure 1a) includes tens of I/O controllers, we instead propose the inclusion of a handful of PR regions. Each of these regions are identical: they contain an AXI connection to the main processor bus (with a fixed memory address), a connection to a set of physical I/O pins, and an interrupt output that connects to an interrupt controller.

This architecture works very well for I/O controllers as they tend to all fit this template. Table I lists the various Xilinx I/O controllers available on the PYNQ system, all of which we have tested in our modified PYNQ system. Each of these follows the same structure of an AXI4-Lite bus connection, an interrupt output, and a few I/O pins. In addition, these cores are on the same order for size, requiring 25–412 LUTs. By sizing our PR partitions to each include 200 slices (800 LUTs) and 6–8 I/O pins, we can implement any of the I/O controller modules in any PR partition.

B. Our Demonstration PYNQ Architecture

In our demonstration PYNQ system we provide 6 PR regions, as well as modules that can be implemented in these regions for each of the IO cores listed in Table I. The floorplan of our design is shown in Figure 2.

The first PR region replaces the entirety of the IOP1 system for the PmodA connector. This replacement removes the GPIO, IIC, SPI and Timer cores, and replaces them with a single PR region. This not only reduces the area cost, but actually provides even more flexibility than the original system. For example, the PR region can be configured to implement a UART core, which was not available in the base configuration. We do the same replacement for the other Pmod, replacing the IOP2 system with a single PR region.

For IOP3, which manages the Arduino shield interface, we replace most of the IP cores with a set of four PR regions. Some pins, as shown in Figure 1, have fixed interfaces (IIC, SPI, XADC) so we leave these be. The 14 pins on the outer connector that were controlled by the IO switch and collection of IO cores are now managed by two PR regions. To provide even greater flexibility to the user, the 16 pins on the inner GPIO header that were previously only tied to a GPIO controller are now controlled by two of the PR regions, adding new flexibility to these pins. Overall this provides much higher flexibility, at again a lower area cost. It is now possible to, for example, implement multiple IIC, SPI or UART controllers, move controllers between different pins, or program the PR regions with entirely different PR modules than the cores provided in the base configuration.

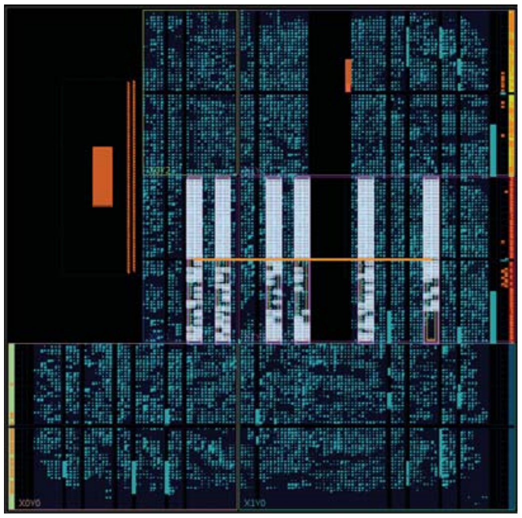


Figure 2: Floorplan our demonstration system

Although the base PYNQ configuration attached the IO controllers to separate Microblaze processors, we have elected to attach them directly to the ARM processor. In our opinion, the benefits provided by a separate bare-metal IO processor (fixed latency, some additional processing power), are outweighed by the additional complexity of introducing the SDK into the software development process, and the need to program two different processors, plus communication software. Operating the devices within Linux does introduce some complexity to the software organization, especially because the kernel needs to be aware of the dynamic reconfiguration that can take place; however, it will be seen that this is largely a non-issue with a modern Linux kernel (see Section IV).

Note that, even though our demonstration system attaches the PR regions to the ARM, it would be straight forward to instead attach them to a separate Microblaze, if desired. For example, one could build a system with both the ARM and a Microblaze as masters on the same bus, allowing the user to choose which processor to run the I/O software on at run-time. Alternatively, a larger PR region could be provided (resources permitting) that could accommodate a Microblaze processor and IO core.

C. The Compilation Flow

By electing to use PR, the hardware compilation flow is naturally more complex than the standard Vivado flow; however, the changes are actually quite minimal. The overall PYNQ system remains a Vivado block design project that

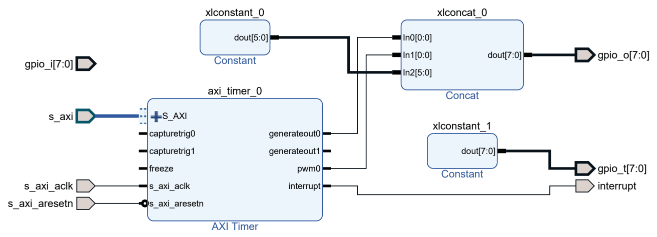


Figure 3: Block diagram for Timer/PWM module

can easily be manipulated by a user familiar with FPGA hardware design. We have only modified the external IO portion of the PYNQ design (Figure 1), the rest of the project remains intact. The PR partitions are represented as black-boxes in the design, and the user can perform *Synthesis* from Vivado as usual. To complete the *Implementation* phase and generate a bitstream we provide a single TCL script for the user to run. This script first performs Implementation on the static portion of the design, then iteratively implements each PR module, generating all necessary bitstreams.

D. A Modular, Scalable I/O System

Each module that can be implemented in a PR region is stored as its own isolated Vivado project. Figure 3 shows the block diagram for the Timer/PWM PR module. As seen in the image, the design contains the AXI interface pins, the interrupt pin and the tristate I/O pins. If the user wishes to add a new I/O module, or modify a compile-time IP configuration, they can simply copy an existing I/O project and make the desired changes. As long as the input/output pins remain unchanged, the user can place any logic in between, provided it fits within the resource constraints. This modular design means users can introduce new I/O configurations without having to interact with or recompile the full system. This has several benefits that are especially helpful for an educational/hobbyist platform: faster compile times, a much simpler design to work with, and no risk that recompiling a small module will break timing on the full system design. For example, on our workstation, the full compilation for the Timer/PWM module shown above takes nine minutes, compared to 84 minutes for the full PYNQ system.

Furthermore, this system would allow a board vendor to provide nearly limitless hardware I/O configurations without requiring the user to perform *any* hardware design. For example, a vendor could distribute several different bitstream variants of a SPI module, covering all of the common hardware configurations. While it may be insensible to expect a *user* to generate hundreds of different bitstreams, it would be completely feasible for a *vendor* to do so. In our architecture on the Zynq FPGA, these partial bistreams with 200 slices are only 149KB each.

By leveraging reconfigurability, such a system could support a nearly limitless number of different peripherals, all without requiring any hardware redesign by the end-user. This would allow the end-user to actually take advantage of the reconfigurable nature of FPGAs, and would set the system apart from fixed hardware systems like Arduino or Raspberry PI.

IV. Software Development for Reconfig. I/O

Introducing our reconfigurable architecture doesn't have a big affect on the software development process. Assuming we provide the user with a library function to reconfigure a region with a chosen bitstream, the rest can be taken care of by the user. When the user program reconfigures a PR region, they are responsible for stopping the driver for the old hardware core, and initializing the driver for the new core. Straightforward procedures to achieve this will be discussed in the following sections.

A. Linux Devices and Drivers

In Linux systems, the *Device Tree* is responsible for enumerating the hardware devices in the system and their properties, such as base address, IRQ number, driver name, etc. Figure 4 provides a snippet from a device tree file; in this case, these lines declare a few of our PR partitions. Of particular interest is the `compatible` field, which lists the kernel device driver that handles the device. On kernel boot the device tree is loaded, and all of the device drivers are loaded into the kernel and are notified of the hardware devices they are responsible for. This poses a challenge for partial reconfiguration-based systems, as traditionally the device tree is static. How we handle PR and the Linux Device Tree depends on whether the user elects to use kernel or user space device drivers. Although kernel drivers are the norm, user space drivers are still often used. User space drivers are attractive as they offer a much easier learning curve, simpler debugging, and a more stable API; however, if the driver needs to make frequent system calls, the cost of context switching may affect performance. We support both kernel and user space drivers in our framework, and have examples of both in our demonstration system.

B. Supporting Linux User Space Drivers

The Linux kernel contains the built-in Userspace IO (UIO) driver [2], which hands off low-level access of a device to user space. Any devices that use the UIO driver will be given a unique device file (`/dev/uioX`). From user programs, this file can be accessed using `mmap()`, providing memory-mapped access to the device registers. To access device interrupts, the user performs a `read()` on the UIO device file, which will block until an interrupt occurs (`select()` and `poll()` offer non-blocking alternatives).

For our reconfigurable I/O system, each partition is given an entry in the Linux Device Tree, indicating that the UIO driver will be used (Figure 4). The same UIO driver be used

```
  ...
  pr0: pr0@41A10000 {
    compatible = "generic-uio";
    reg = <0x41A10000 0x10000>;
    interrupt-parent = <&xil_intc>;
    interrupts = <0x0 0x00>;
  };
  pr1: pr1@41A20000 {
    compatible = "generic-uio";
    reg = <0x41A20000 0x10000>;
    interrupt-parent = <&xil_intc>;
    interrupts = <0x1 0x00>;
  };
  pr2: pr2@41A30000 {
  ...
```

Figure 4: PR partitions using the UIO driver

```
  /{
   compatible = "xlnx,zynq-7000";
   fragment@0 {
     target = <&amba>;
     __overlay__ {
       pr0@41A10000 {
         compatible = "xlnx,axi-uartlite";
         reg = < 0x41A10000 0x10000 >;
         interrupt-parent = <&xil_intc>;
         interrupts = < 0 4>;
       };
     };
   };
};
```

Figure 5: Linux Device Tree Overlay

for any module configured into a PR partition; thus, the device tree can remain static throughout partial reconfiguration. The user space drivers will need to be loaded and unloaded on reconfiguration. User-space drivers are managed by the user, so loading and unloading is performed by the user's application.

C. *Supporting Linux Kernel Drivers*

The use of Linux kernel drivers is a more interesting challenge for our PR architecture. The typical Linux behaviour is to load the device tree once at boot time, after which it remains unchanged. This introduces a problem since we need device drivers (which are dictated by the device tree), to be loaded and unloaded automatically when the user reconfigures a PR partition.

Fortunately, over the last couple years the Linux kernel has been modified to introduce the concept of *Device Tree Overlays*, which are device tree augmentations that can be added to, and removed from the device tree at runtime. Figure 5 provides an example of a device tree overlay file that indicates a UART will be added at address 0x41A10000 (PR region 0). When the overlay is loaded into the active device tree, the kernel will be notified of the new device, and will in turn load the Xilinx UART driver (`"xlnx,axi-uartlite"`). If the user wants to reconfigure the region to a different module, the old overlay is first removed, notifying the kernel, and thus the kernel driver, that the previous device has been removed.

Multiple device tree overlays can be applied and removed independently. This allows us to gracefully reconfigure partitions with new hardware, loading the driver for that device, while other partitions continue to operate concurrently.

Device tree overlay support is still fairly new in the Linux kernel, and in the version of the kernel used by the PYNQ 2.0 system (4.6) there is no way to add and remove device tree overlays from user space; it is expected that user space would not be responsible for changing the hardware of a system. Of course for this proposed work this is exactly what we want to do; the user application is in control of which hardware is present in the PR partition, and can swap it out at will. To provide this ability to user space we developed a kernel module, accessible from user space, that can add and remove device tree overlays. This *Reconfigurable Partition Manager* can be accessed through the `/dev/rp_mgr` file using IOCTL commands.

In our demonstration PYNQ system we provide device tree overlay files for each of the possible I/O controllers in Table I, allowing the user to use the existing Xilinx kernel drivers if they desire. If the user wants to use different kernel drivers, perhaps for new hardware they are adding to the system, it is a simple process to create new overlays. An existing overlay file (Figure 5) is copied and the `compatible` string is updated to the new driver name.

D. *User Programming Interface*

So far in this section we have described the low-level mechanisms for managing devices from user software. However, much of this can be abstracted from the user, and instead a very simple API can be provided.

To demonstrate this in our system, we created a set of Python classes that provide the user's application with a very simple API to reconfigure PR regions, and interact with the implemented hardware devices. These functions are provided as a Python package called `io_pr`.

At the heart of this package is the `IoPr` class, which provides access to the peripherals on the PYNQ system, including the PR regions. Contained within this class are member objects for the six reconfigurable partitions (`rp0`..`rp5`). These objects are instances of the `PrRegion` class, which provides access to functions to reconfigure the partition, read and write to registers in the RP's address space, and check for interrupts.

Using these classes, the user can write very simple programs to configure the hardware and control the devices; Figure 6 provides an example user program. The first step (line 7) is to instantiate the `IoPR` class which gives the user access to the hardware system. Next, on line 10, the first reconfigurable partition, RP0, is configured to implement

```
1  from pynq.overlays.io_pr import IoPr
2  from pynq.overlays.io_pr.drivers.uart import Uart
3  from pynq.overlays.io_pr.drivers.gpio import GPIO
4
5  # Instantiate IoPr class to interact with overlay
6  #   - Use user space drivers
7  overlay = io_pr.IoPr(driverModeKernel = False)
8
9  # Program RP0 with a UART
10 overlay.rp0.configure("uart")
11 uart = Uart(overlay.rp0)
12
13 # Sent data out over the UART
14 uart0.sendData([0xDE,0xAD,0xBE,0xEF])
15
16 # Program RP2 with GPIO
17 overlay.rp2.configure("gpio")
18 gpio = GPIO(overlay.rp2)
19
20 # Set GPIO as outputs and turn on LED0
21 gpio.setDirection(GPIO.ALL_OUTPUTS)
22 gpio.setValue(0x01)
```

Figure 6: Example user code for configuring PR partitions and using user space drivers.

a UART core. Behind the scenes this will perform reconfiguration by activating the PR decoupler, reprogramming using the appropriate partial bitstream, and deactivating the decoupler. If kernel drivers were being used, the appropriate Linux device tree overlay would automatically be applied by sending a command to the Reconfigurable Partition Manager, which would in turn trigger Linux to load the appropriate kernel driver.[1] However, in this example we are using user space drivers (`driverModeKernel = False`), so this step will not occur.

Once the partition is configured, the user can allocate a `Uart` object, a simple user space Uart driver we provide in our demonstration system. The RP object is passed to the Uart driver so that it has abstracted access to the device. This, for example, allows the UART code to read and write to hardware registers, or wait for interrupts, without needing the base address or IRQ number of the partition. The rest of Figure 6 (lines 16-22) shows configuring RP2 with a GPIO controller, and turning on an LED. If the user were using kernel space drivers, the RP reconfiguration would still take place in the same way as Figure 6. However, instead of using the `Uart` or `GPIO` user space drivers, they would interact directly with the device file in `/dev/`.

Our demonstration system is publicly available on Github (link in Introduction), complete with step-by-step tutorials on how to modify an existing PYNQ setup to run our system. Jupyter notebook pages are provided with demonstrations

[1]An infrastructure to handle the process of decoupling, partial reconfiguration, and applying a device tree overlay automatically from the kernel is now available in very recent versions of the Linux Kernel. This is known as the *FPGA Manager* [3], which is a joint development by Intel and Xilinx. Unfortunately this in not available in the 4.6 Linux kernel used by PYNQ, which necessitated providing our own user code for these steps.

System Components	**Base PYNQ Config.**	**Our System**	**Diff**
ARM AXI Bus	3554	4409	+855
IntC, GPIO for PR Regions	-	174	+174
IOP1 Subsystem	3291	*805	-2486
IOP2 Subsystem	3295	*805	-2490
IOP3 Subsystem	6414	*4124	-2290
Rest of PYNQ design	15468	15172	-296
Total LUTs	32019	*25487	-6532

Table II: Area comparison (LUTs) of the base PYNQ configuration with our 6 PR region system. The * values depend on what is implemented in the PR regions; the worst-case (all 800 LUTs per PR region used) is assumed. There are a total of 53200 LUTs available on the XC7Z020 FPGA.

of reconfiguring regions and interacting with devices. To demonstrate user space drivers, we created drivers in Python to manage the AXI GPIO and UART cores. In addition, we ported the Xilinx bare-metal IIC driver to Python, also as a user space driver. For kernel drivers we provide a sample demonstration of interacting with the Xilinx IIC driver. In addition, we include a new module, a real-time clock handler, which was not available in the base PYNQ configuration. This allows us to plug a real-time clock board into any GPIO pins associated with a PR partition, providing Linux with a persistent clock.

V. Evaluating our Demonstration Platform

In this section we compare our proposed demand driven configuration architecture against the base configuration provided with PYNQ.

A. Flexibility

As discussed already in the paper, the foremost benefit of this system is improved flexibility that retains the original ease of use. The user has access to six PR regions that can each implement *any* of the IO protocols present in the original PYNQ configuration. Furthermore, adding additional I/O protocols and peripherals is much more straightforward. Additional protocols can be added by creating a new PR module (this is much, much easier than implementing a new base bitstream) and additional peripherals can use standard Linux device drivers. For example, we added a real-time clock peripheral to PYNQ simply by connecting it to a PMOD connector, loading the existing I2C I/O module and loading the standard Linux driver. Accessing the peripherals directly by the ARM processor makes it much easier to use standard drivers.

B. Area

By utilizing partial reconfiguration this huge increase in flexibility can be provided while also saving logic resources. Table II provides an area comparison between the two architectures. For each of the IO systems (IOP1, IOP2, IOP3

– see Figure 1), we remove the Microblaze processor, the IO cores that feed into the IO switch, and the IO switch itself. This logic is replaced with PR regions, reducing the resource requirement by 2290–2490 LUTs per region. Attaching these regions directly to the ARM bus, increases the bus logic by 855 LUTs, and providing an interrupt controller and GPIO to control decoupling costs another 174 LUTs. For the full system, the savings is 6532 LUTs, representing over a 20% reduction in logic resources.

The PR approach provides significant area savings, while offering greater flexibility to the user. One may note that we elected to remove the Microblaze processors, which contributed to a large part of the area savings. We explained the reasoning for this choice in Section III-B; however, if a dedicated I/O processor was essential to a user's application, a Microblaze could be added for roughly 1500 LUTs.

C. Performance

In general we found that our approach does not affect the performance of the user's application. In our testing, it requires approximately 10-12ms to configure a single PR region. This is done using the process explained in Section IV-D, where a Python module interacts with the underlying hardware and drivers to decouple the PR region and perform the partial reconfiguration.

Another area of performance we wanted to explore was the affect of moving our I/O controllers off of a dedicated Microblaze bare-metal processor and onto the ARM system processor running Linux. We anticipated this change would introduce additional latencies into the device drivers, as a Linux system is of course much more complex than a simple bare metal application.

To understand how device latencies would be affected we performed a simple experiment where we measured the latency to handling an interrupt. We used our reconfigurable region to attach a GPIO controller to a Microblaze running bare-metal software on the PYNQ board, as well as to the ARM processor running Linux. We wired an output GPIO pin to an input GPIO pin, which would trigger an interrupt when the input toggled. We then created a program which would repeatedly toggle the GPIO output, and measure the latency to the ISR beginning execution. For the Linux platform the GPIO output was toggled in user space, and we tested both an ISR in the kernel, or an ISR in user space using the UIO (Section IV-B). The results of 1000 ISR invocations are shown in Table III.

The Linux kernel ISR actually has lower average latency than the Microblaze, and even the user space ISR is not much worse than the Microblaze. This was our primary justification electing to use the ARM processor for our I/O controllers. In our opinion this shows that for the PYNQ system, attaching I/O devices directly to the ARM is probably the preferred choice, and having to deal with the extra software complexity of programming multiple processors is probably not worth it, except for the strictest real-time applications.

Configuration	Interrupt Latency	
	Mean	Range
Microblaze Bare-metal	6.57 μs	6.57–6.57 μs
ARM A9 Linux (Userspace ISR)	9.81 μs	6.46–32.11 μs
ARM A9 Linux (Kernel ISR)	4.64 μs	3.41–16.26 μs

Table III: Interrupt Latencies

D. Demo Platform

As evidence of the flexibility of our architecture, we created a physical demonstration that consists of a series of breakout boards that plug into the GPIO pins controlled by our PR partitions. We have a breakout board with a battery powered real-time clock, an IIC controlled 7-segment display, GPIO controlled LED array, and more in the works.

The magic of the system is that a user can plug a board into any of the GPIO slots, and a Python program that is monitoring a couple of identification pins will detect what type of board was plugged in, automatically reconfigure the PR partition for for the appropriate protocol (IIC, SPI, etc.), which in turn loads the kernel driver. Then a demo program is automatically run on the board, utilizing the standard Linux driver. The user can add and remove boards, or swap positions, and the demos programs will continue to run.

All of this is done with a straightforward Python program running on the ARM, with no hardware design expertise required.

VI. Conclusions and Future Work

In general, in spite of an overall increase in system complexity, the partially-reconfigured system is easy to use and is more flexible and economical than the monolithic bitstream used by the default PYNQ platform. It will also scale to much larger devices such as Xilinx's UltraScale+ MPSOC where a monolithic bitstream is likely to be ungainly and to many take hours to place and route. From a hardware perspective, we find it much more convenient to compile smaller PR circuits when adding new IO capability. These circuits compile rapidly and generally have no impact on timing closure. The ARM processor is powerful enough to handle the IO directly and the system can be modified to attach a Microblaze processor to the AXI bus along with the ARM processor (larger PR regions could also support demand loading of a Microblaze if desired).

This system is a small baby step toward a larger vision - espoused by many in the FPGA community - where FPGA resources are primarily managed by a standard OS and controlled by programmers (see the Related Work Section). In this vision, partial bitstreams are not limited strictly to IO cores but also include large hardware-based accelerators and other related computing modules.

In the future, we plan to extend this research beyond IO to include support for demand-loaded accelerators. In addition, we plan to investigate demand-driven approaches for larger systems such as the MPSOC.

VII. Related Work

Related research efforts include those that 1) employ standard or custom OS's to manage FPGA resources, 2) study software/hardware FPGA interfacing, and 3) use partial reconfiguration for run-time instrumentation.

FUSE, for example, provides an abstraction for hardware acceleration that is transparent to software designers and that supports easy integration of hardware accelerators. It provides an API for POSIX threads within embedded Linux and has demonstrated speeds of 6.4-37X [4]. BORPH executes hardware designs as normal UNIX processes so that they have access to standard OS services. The hardware and software parts of user designs communicate as processes under BORPH's runtime environment [5]. Hthreads uses an operating system that supports a range of computational models and eases development with an intermediate representation along with support for high level languages [6]. ReconOS provides a multithreaded programming model with OS-level thread support for reconfigurable hardware. Hardware and software interact using semaphores, mutexes, message queue, and other standard OS mechanisms [7].

DyRACT is a partially reconfigurable FPGA system that can load and operate partial reconfigurations at run-time. PR management is part of the static region and multiple accelerators loaded by a high-level API [8]. Authors of the Python pynqpartial package extended the pynq package to simplify management of partial bitstreams on PYNQ overlays designed with static regions and PR regions [9]. RAMPSoc (Runtime Adaptive Multi-Processor System on Chip) adapts a Linux kernel to support the Message Passing Interface (MPI) and to integrate software/hardware drivers to provide message transfer over a reconfigurable/heterogeneous Network on Chip (NOC) [10].

A recent effort by Kadi et al. uses Ubuntu Linux to load, control and communicate with a partially-reconfigurable peripheral. Reconfiguration is performed via the Processor Configuration Access Port (PCAP) and a generic UIO driver added to the Linux device tree (uio_pdrv) provides direct access to the address space for the reconfigurable peripheral [11]. In contrast, the work described in this paper uses Linux device-tree overlays and automatically loads in multiple device-specific drivers on demand. Finally, partial reconfiguration and embedded Linux are used to automatically detect the identity of a sensor that is attached to a partially reconfigurable region; once the identify is confirmed, the corresponding partial reconfiguration is loaded [12].

References

[1] B. Hutchings and M. Wirthlin, "Rapid implementation of a partially reconfigurable video system with PYNQ," in *International Conference on Field Programmable Logic and Applications*, 2017, pp. 1–8.

[2] *The Userspace I/O HOWTO*. [Online]. Available: https://www.kernel.org/doc/html/v4.12/driver-api/uio-howto.html.

[3] U. Langenbach, S. Wiehler, and E. Schubert, "Evaluation of a declarative Linux kernel FPGA manager for dynamic partial reconfiguration," in *2017 International Conference on FPGA Reconfiguration for General-Purpose Computing*, May 2017, pp. 13–18.

[4] A. Ismail and L. Shannon, "FUSE: Front-End User Framework for O/S Abstraction of Hardware Accelerators," in *International Symposium on Field-Programmable Custom Computing Machines*, May 2011, pp. 170–177.

[5] H. K.-H. So and R. Brodersen, "A Unified Hardware/Software Runtime Environment for FPGA-based Reconfigurable Computers Using BORPH," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 2, 14:1–14:28, Jan. 2008, ISSN: 1539-9087.

[6] W. Peck, E. Anderson, J. Agron, J. Stevens, F. Baijot, and D. Andrews, "Hthreads: A Computational Model for Reconfigurable Devices," in *International Conference on Field Programmable Logic and Applications*, Aug. 2006, pp. 1–4.

[7] A. Agne, M. Happe, A. Keller, E. Lübbers, B. Plattner, M. Platzner, and C. Plessl, "ReconOS: An Operating System Approach for Reconfigurable Computing," *IEEE Micro*, vol. 34, no. 1, pp. 60–71, Jan. 2014, ISSN: 0272-1732.

[8] K. Vipin and S. A. Fahmy, "DyRACT: A partial reconfiguration enabled accelerator and test platform," in *International Conference on Field Programmable Logic and Applications*, Sep. 2014, pp. 1–7.

[9] B. Janßen, P. Zimprich, and M. Hübner, "A dynamic partial reconfigurable overlay concept for PYNQ," in *International Conference on Field Programmable Logic and Applications*, Sep. 2017, pp. 1–4.

[10] D. Gohringer, S. Werner, M. Hubner, and J. Becker, "RAMPSoCVM: Runtime Support and Hardware Virtualization for a Runtime Adaptive MPSoC," in *International Conference on Field Programmable Logic and Applications*, Sep. 2011, pp. 181–184.

[11] M. A. Kadi, P. Rudolph, D. Gohringer, and M. Hubner, "Dynamic and partial reconfiguration of Zynq 7000 under Linux," in *International Conference on Reconfigurable Computing and FPGAs*, Dec. 2013, pp. 1–5.

[12] A. A. Prince and V. Kartha, "A framework for remote and adaptive partial reconfiguration of SoC based data acquisition systems under Linux," in *International Symposium on Reconfigurable Communication-centric Systems-on-Chip*, Jun. 2015, pp. 1–5.

[13] K. Vipin and S. A. Fahmy, "ZyCAP: Efficient Partial Reconfiguration Management on the Xilinx Zynq," *IEEE Embedded Systems Letters*, vol. 6, no. 3, pp. 41–44, Sep. 2014, ISSN: 1943-0663.

[14] ——, "Automated Partial Reconfiguration Design for Adaptive Systems with CoPR for Zynq," in *International Symposium on Field-Programmable Custom Computing Machines*, May 2014, pp. 202–205.

[15] T. Kalb and D. Göhringer, "Enabling dynamic and partial reconfiguration in Xilinx SDSoC," in *International Conference on ReConFigurable Computing and FPGAs*, Nov. 2016, pp. 1–7.

[16] K. Vipin and S. A. Fahmy, "Mapping adaptive hardware systems with partial reconfiguration using CoPR for Zynq," in *NASA/ESA Conference on Adaptive Hardware and Systems*, Jun. 2015, pp. 1–8.

[17] P. Garcia, K. Compton, M. Schulte, E. Blem, and W. Fu, "An Overview of Reconfigurable Hardware in Embedded Systems," *EURASIP J. Embedded Syst.*, vol. 2006, no. 1, pp. 13–13, Jan. 2006, ISSN: 1687-3955.

[18] J. G. Reis, L. Wanner, and A. A. Fröhlich, in *2015 Euromicro Conference on Digital System Design*, Aug. 2015, pp. 255–258.