Dynamic Partial Reconfiguration on an IOT device

Android on zynq featuring updateable FPGA-accellerated image processing

Andreas Dejmek

Christoph Grafl

Florian Muttenthaler

Benedikt Tutzer

Winter-Term 2018/2019

Supervised by:
Nahla El-Araby
and
Nima Taherinejad

Abstract

Hardware products are hard to update after they have been shipped. Reconfigurable devices such as FPGAs or CPLDs can be updated, but to update a whole system, the system needs to be shut down. Often, only parts of a system need updating. This is possible with dynamic partial reconfiguration. The designer has to partition the given design and can then update partitions during runtime. Applying dynamic partial reconfiguration to high level designs is not trivial. We show how this can be done using the example of a Zedboard running the Android Operating System on it's integrated ARM cores and some image processing filters on the programmable logic. We found that many customizations have to be made to the Linux kernel and the Android operating system. The choice of hardware restricted us to the use of outdated tools that made developement difficult. The methods can be applied to more state of the art chips to obtain powerfull consumer multimedia devices with updatable hardware accellerators.

Submitted in the "System on Chip Design"
Laboratory at TU Wien

Contents

1 Introduction

The purpose of this project is to implement a partial reconfigurable *Internet of Things (IoT)* device, which can exchange parts of the synthesized hardware during runtime for a suitable application. Android is set up to run on the Zynq software processing unit on board of the Digilent ZedBoard. An application is presented that applies filters to given images. These filters may be subject to updates in the future and the app is able to download such updates and apply them to the *Programmable Logic (PL)* using dynamic partial reconfiguration.

2 Methodology

The Digilent ZedBoard is a developement Platform for the Xilinx Zynq *system on chip (SoC)*. It features a dual-core ARM Cortex-A9 MPCore with a clock-frequency of up to $866Mhz$, external memory support, USB and Gigabit Ethernet interfaces as *Programmable Software (PS)* and PL based on the Xilinx Artix-7 *field-programmable gate array (FPGA)* series. The PL can be used stand-alone, but the PS is dependent on the PL.

For this project, the PL needs to be configured with peripherals for the processor that are needed by the Linux kernel as well as the hardware accelerators. The resulting system must then be partitioned for the dynamic partial reconfiguration. This is discussed in Section 2.1.

The PS needs to be configured with all the software that Android needs to boot, to interact with the hardware accelerators and to dynamically reconfigure the PL. This is done in Section 2.2.

The complete design is available as a git repository at [**?**]. File paths within this document may refer to the root directory of the git repository by using *<repo>*. To interact with the ZedBoard, multiple interfaces are used:

- A command line interface is available over an *universal synchronous receiver/transmitter (UART)* interface
- Ethernet is used to access the internet
- Graphical output is available over a *High Definition Multimedia Interface (HDMI)*
- Peripherals can be attached via *Universal Serial Bus On The Go (USB-OTG)*

To achieve the best android experience, a touchscreen is attached via HDMI (for graphics) and USB-OTG (for the *Human Interface Device (HID)*).

To automate the extensive build process, two scripts are available in our git repository. *<repo>/bootimage/generate_without_android.sh* compiles the bitstream, *First Stage Boot Loader (FSBL)*, u-boot, the linux kernel and the kernel modules. Android is built seperately, since it requires a special build

environment that not all members of the team had access to. To build android, the script *<repo>/bootimage/generate_including_android.sh* can be executed.

Both scripts rely on makefiles of the underlying components they are building. This means that components that were not changed since the last build are not going to be recompiled. This saves build time during development.

2.1 Zynq Hardware Design

Digilent provides a reference design [**?**] that includes all the peripherals that are needed by the linux kernel to run on the ZedBoard. This is the design that is used to build the boot-image flashed onto the SD card that comes with the ZedBoard. It contains a *Xilinx Platform Studio (XPS)* Project that can be opened and built using the corresponding *xps* tool that is included in the Xilinx ISE Design Suite. Digilent claims that they used version *14.4*. Unfortunately, that version is missing the *axi_vdma* core in revision *v5_01_a*. That particular revision was removed in *ISE 14.4* but is available in *ISE 14.1* and replace with a newer version. Without the knowledge of the old version, the Xilinx tools are unable to upgrade the core to the new revision.

To mitigate this issue, both versions of *ISE* were installed and the core was imported from the old version to the new version using a symbolic link. This allowed us to build the project in *ISE 14.4*. Since the support for *Dynamic Partial Reconfiguration (DPR)* is better in *ISE 14.7*, we chose to use that version.

To import the old *axi_vdma* core into the new version of *ISE*, the command in Listing 1 can be used, assuming *ISE* was installed in the default directory */opt*.

Listing 1: Link *axi_vdma* core from *ISE 14.1* to *14.4*

```
1  ln -s /opt/Xilinx/14.1/ISE_DS/EDK/hw/
       XilinxProcessorIPLib/pcores/axi_vdma_v5_01_a /opt/
       Xilinx/14.7/ISE_DS/EDK/hw/XilinxProcessorIPLib/
       pcores/axi_vdma_v5_01_a
```

This design can then be adapted to include our logic needed for the hardware accelerators. Digilent published a tutorial [**?**] on how to do this on a simple example. The basic steps to do this are:

- Create new peripheral by choosing 'Hardware', 'Create or Import Peripheral...'
- The new peripheral can now be found in the 'Project Local PCores' under the 'USER' registry. Add it to the design by right clicking on it and choosing 'Add IP'
- Add it to the Zynq Processing System when prompted to do so
- Assign an address and memory size in the 'Addresses' tab

For the last step, choose a free address. The address-table for the included devices is configured as follows:

Instance	Base Address	High Address	Size
processing_system7_0	0x00000000	0x1FFFFFFF	512M
axi_dma_0	0x40400000	0x4040FFFF	64K
axi_iic_0	0x41600000	0x4160FFFF	64K
axi_vdma_0	0x43000000	0x4300FFFF	64K
axi_hdmi_tx_16b_0	0x70E00000	0x70E0FFFF	64K
axi_spdif_tx_0	0x75C00000	0x75C0FFFF	64K
axi_clkgen_0	0x79000000	0x7900FFFF	64K

The custom logic cores can be assigned addresses starting from $0x7E400000$.

XPS generates two VHDL files for the core. One is called 'user_logic.vhd' and the other carries the name of the core itself. The first one contains the actual logic. The tool generates a simple interface so that the logic can react to register reads and writes. The second file maps that simple interface to the *Advanced Microcontroller Bus Architecture (AXI)* interface so that it can be connected to Zynq's AXI bus.

The complete hardware design is available as a XPS project in the git repo at *<repo>/hardware_design/system.xmp*.

The logic design of the two hardware accelerators is discussed in the following subsections.

2.1.1 Blake2B Module

Blake2 [?] is a cryptographic hash and *Message Authentication Code (MAC)*. It is faster than *MD5*, *SHA-1*, *SHA-2* and *SHA-3*, but is at least as secure as the latest standard *SHA-3*. It comes in two flavors:

- *Blake2B* is optimized for 64-bit platforms and produces digests of any size between 1 and 64 bytes
- *Blake2S* is optimized for 8- to 32-bit platforms and produces digests of any size between 1 and 32 bytes.

A hardware implementation was created by Benedikt Tutzer and Dinka Milovancev as part of the *Digital Integrated Circuits* Laboratory at TU Wien [?]. Even though we are working on a 32-bit platform, we chose to implement Blake2B, since at hardware level we can chose what bit-width to use.

The core is added to the system design as described in Section 2.1 and given the address $0x7E410000$. A size of $64K$ is sufficient. XPS generates the needed files in *<repo>hardware_design/pcores/blake2b_v1_00_a*. The files from [?] are then added to the vhdl-source directory of the core, *<repo>/hardware_design/pcores/blake2b_v1_00_a/hdl/vhdl*. XPS needs to

be made aware of the additional files, otherwise they will be left out of the synthesis flow. To include them, the *Peripheral Analysis Order (PAO)* file of the core, found in *<repo>/hardware_design/pcores/blake2b_v1_00_a/data/blake2b_v2_1_0.pao*, needs to be adapted accordingly (Lines 3 and 4):

Listing 2: PAO file of the blake2b core

```
1 lib proc_common_v3_00_a all
2 lib axi_lite_ipif_v1_01_a all
3 lib blake2b_v1_00_a blake2 vhdl
4 lib blake2b_v1_00_a blake2b_wrapper vhdl
5 lib blake2b_v1_00_a user_logic vhdl
6 lib blake2b_v1_00_a blake2b vhdl
```

The entity from [?] needs to be wrapped in the user_logic.vhd file. It was configured to have 4 software accessible registers, each 32-bit wide:

Address	Name
base	task_reg
base + 4	message_reg
base + 8	status_reg
base + 16	hash_reg

With interface generated by the Xilinx tools, the core is only able to react to register reads or -writes from the software. It cannot send interrupt to the software. To add this functionality, an additional port, of type *std_logic* is added and routed through the wrapper in *blake2b.vhd* so that it is visible as an output port of the peripheral. It is called *Interrupt*. To have it act as an interrupt signal, it has to be declared as such in the cores *Microprocessor Peripheral Definition (MPD)* file, *<repo>/hardware_design/pcores/blake2b_v1_00_a/data/blake2b_v2_1_0.mpd*. This is done with the following lines:

Listing 3: Configure output port as interrupt

```
1 PARAMETER C_INTERRUPT_PRESENT = 1, DT = INTEGER, RANGE =
      (0,1)
2 PORT Interrupt = "", DIR = O, SIGIS = INTERRUPT,
      SENSITIVITY = EDGE_RISING, INTERRUPT_PRIORITY = MEDIUM
      , ISVALID = (C_INTERRUPT_PRESENT == 1)
```

The port then shows up as an interrupt port. To connect it to the *ARM Generic Interrupt Controller (GIC)*, one must select the Zynq tab and click on the *IRQ* table as seen in Figure 4.

On the upcoming dialog, the interrupt signal is shown as unconnected interrupt in the left column. By selecting it and clicking the arrow that points to the right it can be moved to the right and is then connected, as seen in Figure 2.

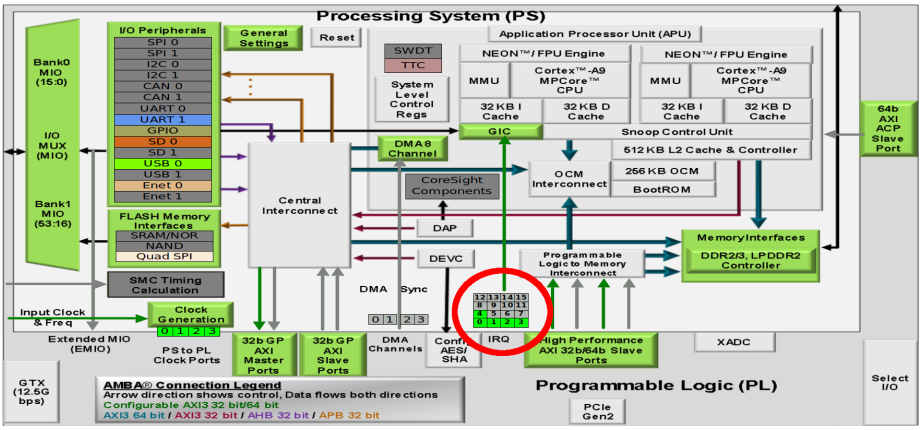

Figure 1: IRQ table in the Zynq tab

The number shown on the left side of the right column, 87 in this example, is the interrupt number that the device driver will have to connect to. This can be seen in Section 2.2.3.

The protocol how this device communicates with the device driver is simple. To hash data, it has to be split into chunks of 32-bits as the registers are only 32 bits wide. The following steps need to be followed to hash data:

1. Driver writes number of bytes to be hashed to the task register
2. If all the data was sent, go to Item 6
3. Device raises interrupt to signal that more data is needed
4. Driver catches interrupt and writes a chunk of data to the message register
5. Go to Item 2
6. Device raises interrupt to signal that hashing is done

The hash is always 64-bytes long, so to read it back to the driver it has to be split into 16 individual 32-bit chunks. To read the hash from the device, the following steps are done:

1. Driver iterates over 16 hash chunks
   (a) Driver writes index of hash-chunk to status register
   (b) Devices places the according chunk onto the hash register
   (c) Driver reads hash register
2. Driver concatenates hash chunks

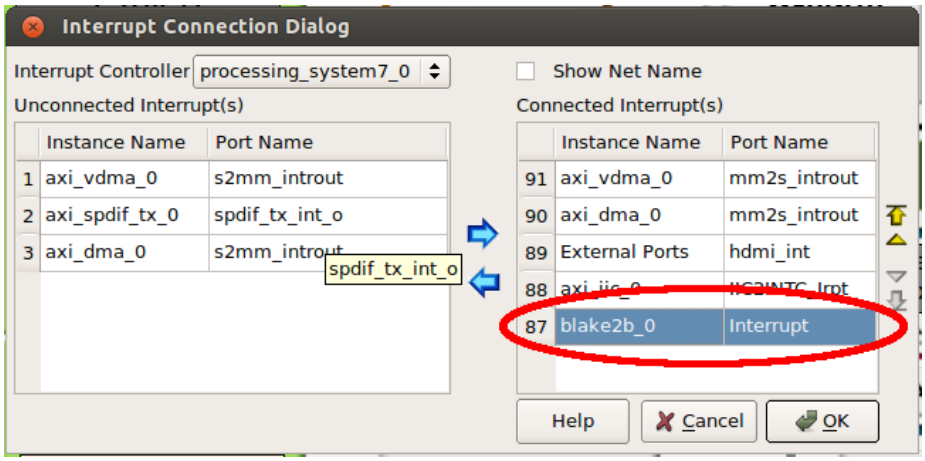

Figure 2: Interrupt Connection Dialog

2.1.2 Image Processing Module

The following three types of image filters have been designed:

- Red-Filter
- Green-Filter
- Blue-Filter

These simple filters read pixel data from an AXI interface and filter out all channels except one. The red filter filters out everything but the red channel.... Figure 3 describes the principle of the filter functionality. The logic of these filters is connected to the AXI-Lite Bus. All three filters share a common Wrapper-Interface and are accessible by a common Linux-Device-Driver.

The different cores are added to the system design via the concept described in section 2.1.3 at the address $0x7E430000$ with the size of $64K$. The logic behind the common wrapper uses 2 software accesible registers, each 32-bit wide:

Address	Name
base	write_reg
base + 4	read_reg

The value of the write_register is processed by the synthesized filter logic and the result is written to the read_register. The related Linux device driver (2.2.3) iterates over the raw *Red/Green/Blue (RGB)* data written to the device driver file and stores the filtered data which can then be read back from the device driver file.

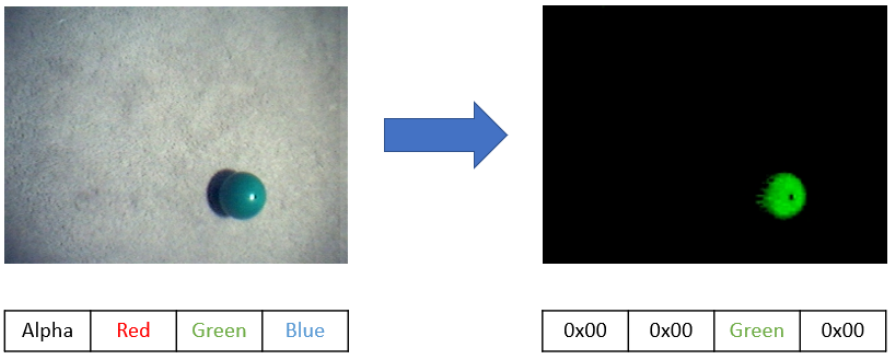

Figure 3: Green Filter example

2.1.3 Partial Reconfiguration Setup

Partial reconfiguration was done using the planAhead tool included in the Xilinx ISE Design Suite. Compared to the Vivado Design Suite, where DPR works with the VHDL files, the planAhead tool only works with the synthesized netlists.

For DPR a Bottom-Up Synthesis is required. The static logic is synthesized with a black box module definition for each DPR module.

The synthesis is done inside the script *<repo>/bootimage/generate_without_android.sh*. Since the netlists for the different filter logics are not synthesized with the flow, they must be generated separately. Both is done with the following lines:

Listing 4: Synthesis for project

```
1 make -f system.make netlist
2 # generate netlists for filter logic
3 xst -ifn synth_filter_logic.xst
```

Xilinx provides a tutorial [**?**] that describes how the DPR is done with the planAhead tool. First, the *Graphical User Interface (GUI)* based design was used to create the bitstreams. Afterwards a *Tool Command Language (TCL)* script, *<repo>/hardware_design/planAhead.tcl*, was implemented with the included TCL console. This script automatically generates the bitstreams used for partial reconfiguration. Listing 5 shows how the planAhead tool can be started in TCL mode and start the script.

Listing 5: Run planAhead in TCL mode

```
1 planAhead -mode tcl -source planAhead.tcl
```

In our project 6 bitstreams are generated:

- 3 full bitstreams with different filter logic
- 3 partial bitstreams with different filter logic

Before the partial bitstreams can be used, they need to be transformed into an other format. With the following lines the bitstreams are converted:

Listing 6: Convert bitstreams into binary files

```
1 promgen -b -w -p bin -data_width 32 -u 0 ./planAhead/
      partial_reconfiguration/partial_reconfiguration.runs/
      config_1/
      config_1_simple_filter_0_simple_filter_0_USER_LOGIC_I_filter_logic_0_red_filter_
      .bit -o ./planAhead/generated_Bitstreams/red_filter.
      bin
2 promgen -b -w -p bin -data_width 32 -u 0 ./planAhead/
      partial_reconfiguration/partial_reconfiguration.runs/
      config_2/
      config_2_simple_filter_0_simple_filter_0_USER_LOGIC_I_filter_logic_0_green_filte
      .bit -o ./planAhead/generated_Bitstreams/green_filter.
      bin
3 promgen -b -w -p bin -data_width 32 -u 0 ./planAhead/
      partial_reconfiguration/partial_reconfiguration.runs/
      config_3/
      config_3_simple_filter_0_simple_filter_0_USER_LOGIC_I_filter_logic_0_blue_filter
      .bit -o ./planAhead/generated_Bitstreams/blue_filter.
      bin
```

2.2 Zynq Software Design

The Zynq Software Design consists of the following components:

- The FSBL
- The *u-boot* utility
- A linux kernel
- The Android Operating System
- Linux kernel modules
- The image processing app

Setting up the FSBL and the u-boot utility is described in detail in [?]. The only important thing to note is that u-boot needs to be built from source code revision *b55d4b1*, as the support for the ZedBoard changed afterwards. [?] also

provides instructions on how to build the linux kernel, but since we need it to be able to boot Android quite some modifications were needed so it is explained in detail in Section 2.2.1. The Android setup itself is discussed in Section 2.2.2, while Section 2.2.3 talks about the kernel modules and Section 2.3 details the Android app. Finally, Section 2.4.1 talks about how we set up the software system to be able to do DPR.

2.2.1 Linux on ZedBoard

Digilent provides a branch of the linux kernel that is compatible with most of their boards, including the ZedBoard [?]. Source revision *06b3889* was found to be the most recent to include android support.

To build linux from [?] for the ZedBoard with Android support, the following steps need to be followed:

1. *Fix the device tree* by copying the device tree from [?] to *<kernel>/arch/arm/boot/dts/digilent-zed.dts*
2. *Setup build environment to use Xilinx tools* by setting environment variables as seen in Listing 7
3. *Generate default configuration for the ZedBoard* by running *make digilent_zed_defconfig*
4. *Add Android and touchscreen related configurations* by running *make menuconfig*
   (a) Navigate to 'Device Drivers' and enable 'Staging Drivers'
   (b) Navigate to 'Device Drivers' 'Staging Drivers', 'Android' and enable all entries
   (c) Navigate to 'Device drivers', 'Input device support' and enable 'Touchscreens'
   (d) Navigate to 'Device drivers', 'Input device support', 'Touchscreens' and enable
      - 'Ilitek ILI210X based touchscreen'
      - 'USB Touchscreen Driver'
   (e) Navigate to 'Device drivers', 'HID support', 'Special HID drivers' and enable 'HID Multitouch panels'
5. *Build the kernel* by running *make*
6. *Build the device tree* by running *./scripts/dtc/dtc -I dts -O dtb -o devicetree.dtb arch/arm/boot/dts/digilent-zed.dts*

The compiled image will be available in *arch/arm/boot/zImage*. To boot it requires a ramdisk, available from [?]. To have the linux kernel to load our kernel modules and start Android when the boot was successul, the ramdisk

Listing 7: Environment setup to build the linux kernel

```
1 export CCOMPILER=arm-xilinx-linux-gnueabi-gcc
2 export ARCH=arm
3 export CROSS_COMPILE=arm-xilinx-linux-gnueabi-
4 export PATH=$PATH:/opt/Xilinx/14.7/ISE_DS/EDK/gnu/arm/lin/
      bin/
```

has to be adapted. The easiest way to do this is to make the startup script in *etc/init.d/rcS* execute our own startup scripts from the SD card after everything else is done.

We created two startup scripts, one to load the kernel modules (*<repo>/linux-files/startup.sh*) and one to start android (*<repo>/linux-files/startup_ android.sh*).

2.2.2 Android on ZedBoard

Android Gingerbread on ZedBoard was officially supported by Xilinx partner company Iveia, unfortunately it is not any more and they do not longer provide android sources. Thankfully, a github user made the Iveia sources available by uploading them [**?**].

Android Gingerbread requires a very delicate build environment:

- Ubuntu 12.04 LTS host operating system
- make version 3.81
- gcc, g++ and cpp version 4.4
- Java JDK 1.6 from Oracle (the OpenJDK version is not suitable)
- The following packages available from the Ubuntu repositories: apt-get install libgtk2.0-0:i386 libxtst6:i386 gtk2-engines-murrine:i386 lib32stdc++6 libxt6:i386 libdbus-glib-1-2:i386 libasound2:i386 fakeroot build-essential crash kexec-tools makedumpfile kernel-wedge git-core libncurses5 libncurses5-dev libelf-dev asciidoc binutils-dev curl gcc-4.4 g++-4.4 cpp-4.4 gcc-4.4-multilib g++-4.4-multilib cpp-4.4 ia32-libs openjdk-6-jdk:i386 vim gnupg flex bison gperf build-essential zip curl libc6-dev x11proto-core-dev libx11-dev:i386 libreadline6-dev:i386 libgl1-mesa-glx:i386 libgl1-mesa-dev g++-multilib mingw32 tofrodos python-markdown libxml2-utils xsltproc zlib1g-dev:i386 gnupg flex bison gperf build-essential zip curl libc6-dev lib32ncurses5-dev x11proto-core-dev libx11-dev:i386 libreadline6-dev:i386 libgl1-mesa-glx:i386 libgl1-mesa-dev g++-multilib mingw32 tofrodos python-markdown libxml2-utils xsltproc zlib1g-dev:i386 genext2fs lib32z1-dev
- The repo tool (a wrapper around git), available from `http://commondatastorage.googleapis.com/git-repo-downloads/repo`

Listing 8: Retrieve and build android

```
1 repo init -u git://github.com/aimeemikaelac/xilinx-android
      -manifest.git -b android-zynq-1.0
2 repo sync
3
4 . build/envsetup.sh
5 lunch generic-eng
6
7
8 make -j32
9 make -f Makefile.zynq
10
11 mkdir -p root
12 sudo mount root.img -o loop,rw,sync root
13 sudo mkdir -p root/system/usr/idc/
14 sudo cp <repo>linux-files/touchscreen_config.idc root/
      system/usr/idc/Vendor_222a_Product_0001.idc
15 #sudo cp <repo>/linux-files/touchscreen_config.idc "root/
      system/usr/idc/ILITEK ILITEK-TP.idc"
16 sudo chmod 664 root/system/usr/idc/*.idc
17 sleep 1
18 sudo umount root
19 sleep 1
20 rmdir root
```

The build process can be seen in Listing 8. In Lines 1 to 2 the source code is retrieved. Lines 4 to 5 setup the environment and Lines 8 to 9 builds the system. The result is a bootable image called 'root.img'.

This image then needs to be mounted, so we can copy the touchscreen configuration file to it. This is done in Lines 11 to 20.

2.2.3 Linux Kernel Modules

Blake2b Driver:
The blake2b device driver sits between the userspace application that desires to compute the hash of a file and the PL implementation of Blake2B.
The communication with the PL was discussed in detail in Section 2.1.1.
The interface to the userspace application is simple. The driver creates a device file */proc/blake2b*. The userspace application can then write a file path to that file. This triggers the start of the hashing function. Afterwards, the application can read 64 bytes from the device file, these bytes will contain the hash.
The blake2b driver code is available at *<repo>/drivers/blake2b/blake2b.c*. The code is quite self explanatory and therefore not discussed in detail here. One caveat is that one has to call the function *wmb* between subsequent reads or writes to the device. This function will impede the compiler to rearrange the calls during optimization and therefore guarantees that the reads and writes are

executed in specified order.

Image Filter Driver:
The Image Filter device driver writes raw RGB data that needs to be processed to the write register address of the programmed filter logic. The processing of the value is then triggered automatically. Afterwards the filter data is read from the read register address by the driver. The interface to the userspace application is implemented by writing and reading from a related *.bin* file. The absolute path of this file has to be communicated to the device driver.

The communication with the PL is discussed in section 2.1.2. The communication with the application is discussed in section 2.3. The developed driver creates a device file */proc/simple_filters*, which can be triggered by the user application. The driver code itself is available at
<repo>/drivers/simple_filters/simple_filters.c.

2.3 Image Processing App

The Image processing app splits into two parts. An external server which provides the infrastructure to store and distribute bitstreams and the android app (client) which interacts with the server and the hardware on the ZedBoard.

2.3.1 Client application

The client is a native android application written in Android Studio version 3.2.1 for Android API 10 (Gingerbread 2.3.3). Unfortunately todays Android SDK (API 28) does not support Gingerbread anymore, version 25 was the last SDK that officially supports Gingerbread. To compile for Gingerbread set compileSdkVersion to 25 with the targetSdk set to 10 as seen in Listing 9.

Listing 9: App build setup

```
1 apply plugin: 'com.android.application'
2 android {
3     compileSdkVersion 25
4     defaultConfig {
5         applicationId "com.lab.soc.client"
6         minSdkVersion 10
7         targetSdkVersion 10
8         versionCode 1
9         versionName "1.0"
10        testInstrumentationRunner "android.support.test.
              runner.AndroidJUnitRunner"
11    }
12    buildTypes {
13        release {
14            minifyEnabled false
15            proguardFiles getDefaultProguardFile('proguard
                  -android.txt'),
16            'proguard-rules.pro'
```

```
17          }
18      }
19      buildToolsVersion '28.0.3'
20  }
21
22  dependencies {
23      implementation fileTree(include: ['*.jar'], dir: 'libs
            ')
24      implementation 'com.android.support:appcompat-v7
            :25.4.0'
25      implementation 'com.android.support.constraint:
            constraint-layout:1.1.3'
26      implementation 'com.android.support:support-v4:25.4.0'
27      testImplementation 'junit:junit:4.12'
28      androidTestImplementation 'com.android.support.test:
            runner:1.0.2'
29      androidTestImplementation 'com.android.support.test.
            espresso:espresso-core:3.0.2'
30  }
```

This way it is possible to use Google's support libraries, which deliver backward compatibility for newer Android classes (e.g fragments) and allows to run the app on all Android versions from Gingerbread to Pie without version specific modifications. Whenever possible Androids own APIs has been used, if not available on Gingerbread they were replaced with own implementations.

The architecture is following the single responsibility principle, separating the concerns into several classes:

- MainActivity - Android specific (UI, Events, Life cycle)
- NetworkManager - Network operations
- MsgProcessor - JSON processing
- FabricManager - PL Fabric interaction
- AppExecutors - Multi threading
- Util - Utility (Image preprocessing)
- Repository - Data format

The Network Manager spawns a new thread and opens a connection to the server. Then requests new information using the REST API (see server section). The received information is packed inside a JSON object (JavaScript Object Notation) and forwarded to the MsgProcessor. Where they are unpacked, compared and saved.

Example JSON object:

```
{
    "Index": "SOC-LAB-IOT",
    "Title": "IOT Image Processing",
    "Version": "002",
    "Description": "",
    "Changelog": ["3.11.2018 Primary release",
                  "4.11.2018 Bug fixing",
                  "6.11.2018 added stuff"],
    "File": "filter_0_0_4.bin",
    "Date": "6.11.2018",
    "Checksum": "fdg851dfg654dfg6541dfg65514dfghdfg45534terg"
}
```

If there is a new version available, the Network Manager will download the new bitstream from the sever using the REST API (see server section). The Fabric Manager will then calculate the hash from the bitstream file on the programmable logic. If they match, the PL fabric will be reconfigured with the downloaded bitstream. In a real world scenario, the whole process would run in the background without user interference. For demonstration purposes a simple user interface was implemented to trigger the download and apply filters on a test picture.

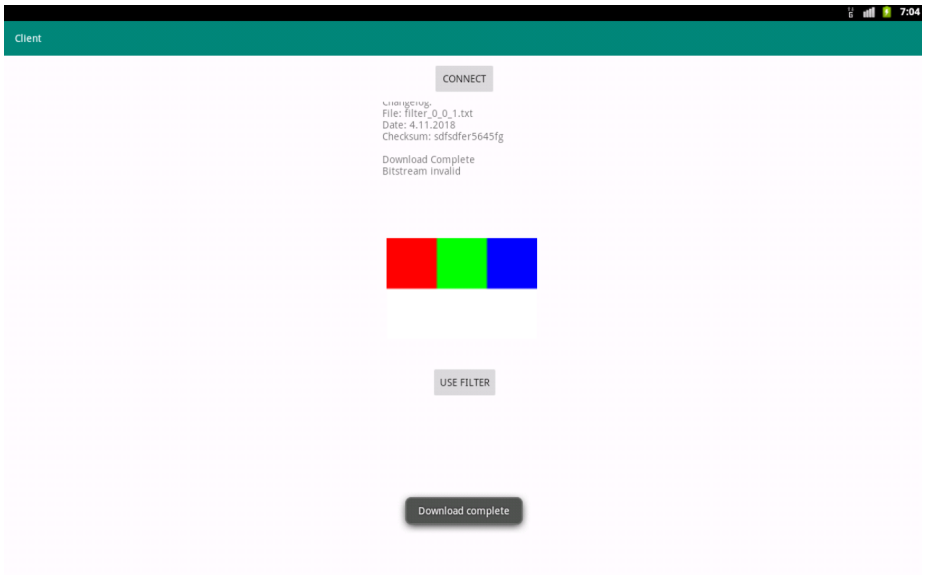

Figure 4: user interface

To apply filters the app has to pre process the image for the device driver to the following specification: 32 bit integer ALPHA | RED |GREEN | BLUE (8bit|8bit|8bit|8bit) e.g full opaque red would be (hex notation) FF FF 00 00. Every pixel will be read out and saved together to a new binary file. The path

to this file will be written to the device driver, soon after the filtered data will be read back into the UI from the same file.

2.3.2 Server

A tiny webserver with a REST API (Representational State Transfer), written with python 3.7 using Flask and its extension FlaskRESTful, therefore fully WSGI compliant (Webserver Gateway Interface). However Flask internal development server is used for the demonstration and for simplicity.

To start the server, open a terminal in the server directory. Type in following command:

```
Flask run -h *your ip* -p 5000
```

Open http://*your ip*:5000 to see if the server is reachable

2.3.3 Server API

Insomnia, a free open source REST client available for Mac,Windows and Linux, was used for API testing and server communication.

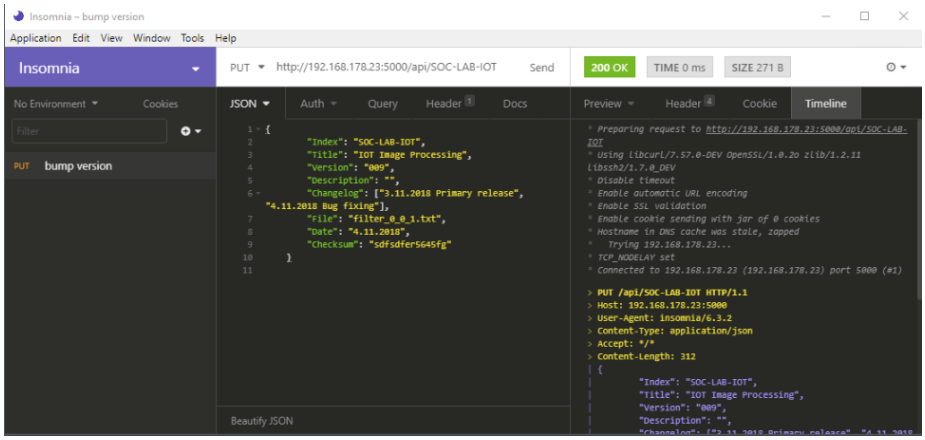

Figure 5: Insomnia

2.3.3.1 Get all information for repository <index>:

URL	URL PARAM	DATA PARAM	METHOD	SUCCESS RESPONSE	ERROR RESPONSE
/api/<index>	index=[string]	n/a	GET	200	404

2.3.3.2 Create repository <index>:

URL	URL PARAM	DATA PARAM	METHOD	SUCCESS RESPONSE	ERROR RESPONSE
/api/<index>	index=[string]	JSON	POST	201	400

JSON:

```
{
    "Index": "SOC-LAB-IOT",
    "Title": "IOT Image Processing",
    "Version": "001",
    "Description": "",
    "Changelog": ["3.11.2018 Primary release", "4.11.2018 Bug fixing"],
    "File": "filter_0_0_1.txt",
    "Date": "4.11.2018",
    "Checksum": ""
}
```

2.3.3.3 Update repository <index> or create it if not existing:

URL	URL PARAM	DATA PARAM	METHOD	SUCCESS RESPONSE	ERROR RESPONSE
/api/<index>	index=[string]	JSON	PUT	200	201

JSON:

```
{
    "Index": "SOC-LAB-IOT",
    "Title": "IOT Image Processing",
    "Version": "002",
    "Description": "",
    "Changelog": ["3.11.2018 Primary release",
    "4.11.2018 Bug fixing",
    "6.11.2018 added stuff"],
    "File": "filter_0_0_4.bin",
    "Date": "6.11.2018",
    "Checksum": "fdg851dfg654dfg6541dfg65514dfghdfg45534terg"
}
```

2.3.3.4 Delete repository <index>:

URL	URL PARAM	DATA PARAM	METHOD	SUCCESS RESPONSE	ERROR RESPONSE
/api/<index>	index=[string]	n/a	DELETE	200	404

2.3.3.5 Download file <filename>

URL	URL PARAM	DATA PARAM	METHOD	SUCCESS RESPONSE	ERROR RESPONSE
/api/download/ <path:filename>	filename=[string]	n/a	GET	200	404

2.4 Setup

To setup the hardware, one can either:

- Build the design himself as described in the previous sections
- Use the binaries provided in *<repo>/build*

An SD card of at least $4GB$ needs to be formatted as FAT32 and contain the following files:

- *BOOT.BIN*: Contains the FSBL, u-boot and the bistream for the PL
- *zImage*: The linux kernel
- *root.img*: The Android Operating System
- *devicetree.dtb*: The compiled devicetree
- *blake2b.ko*: The kernel module for the hashing device
- *image_filter.ko*: The kernel module for the image filters
- *ramdisk8M.image.gz*: The ramdisk
- *startup.sh*: The script that loads the kernel modules
- *startup_android.sh*: The script that starts Android. If this file is missing the boot process will only boot the linux kernel

The jumpers on the ZedBoard must be set up as follows:

JP1	open
JP2	short
JP3	short
JP4	(open / not populated)
JP5	(open / not populated)
JP6	short
JP7	GND
JP8	GND
JP9	3V3
JP10	3V3
JP11	GND
JP12	open
JP13	open
JP18	1V8

A USB hub can be connected at the USB-OTG connector. This can then host a USB Keyboard and the USB touchscreen connector A UART interface is available at the UART USB port. The SD card needs to be inserted into the SD card slot (on the backside of the ZedBoard, under the FMC connector). An Ethernet cable can be connected to the Ethernet port. This can be seen in Figure 6.

Once everything is set up, the power switch can be switched to the *ON* position and Android will boot. After boot, Android will present itself on the display and show the launcher as seen in Figure 7.

NOTE: The screen must be connected before boot, otherwise the Android VM will not show.

2.4.1 Dynamic Partial Reconfiguration in Android

The branch of the Linux kernel provided by Digilent [?] already included a device driver for partial reconfiguration.

Before the driver can be used, it must be set up during the boot routine. Listing 10 shows how to set up the device driver. In Line 3 the flag for partial reconfiguration is set.

From Android, DPR can be easily done by sending the partial bitstream to the device driver. Listing 11 shows how this is implemented.

After the reconfiguration is completed, the new logic part can be used.

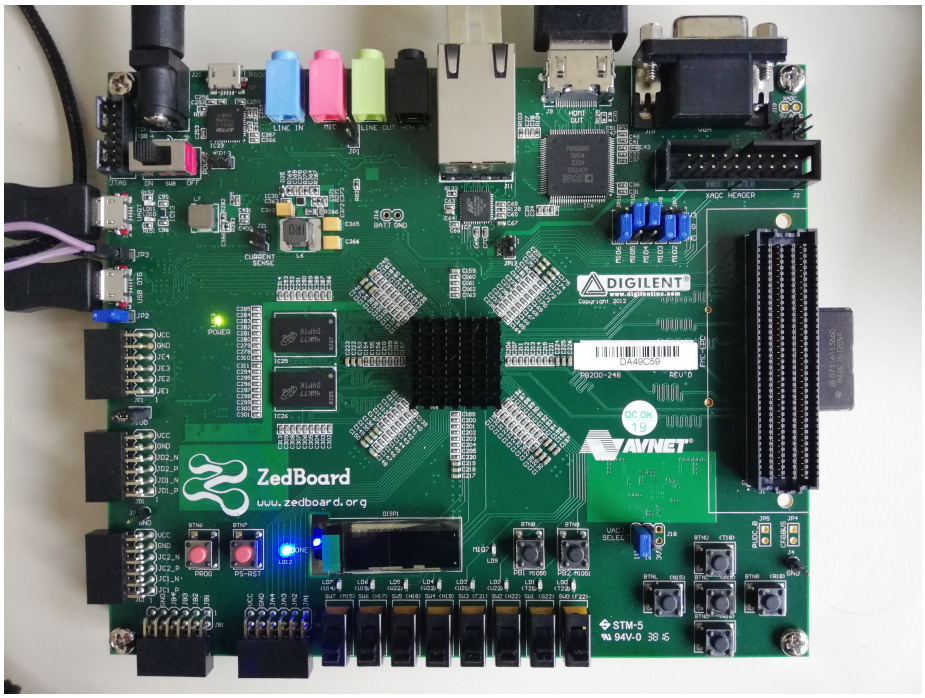

Figure 6: ZedBoard setup

Listing 10: Setup partial reconfiguration driver

```
1 echo "++ Adding partial reconfiguration device node"
2 mknod /dev/xdevcfg c 259 0
3 echo "1" > /sys/devices/axi.0/f8007000.devcfg/
      is_partial_bitstream
```

3 Known Issues

The presented implementation is far from perfect. Issues faced during development that could not be fixed during the course of the project are presented in this section.

3.1 Android Version

Android Gingerbread (2.3.7) was released on December 6th 2010, more than 8 years ago at the time of writing this document. 9 major versions were released since then. Unfortunately, the ZedBoard does not have the hardware requirements for more recent version (little RAM and no dedicated *Graphics Processing Unit (GPU)*). Although, the higher-range Zynq chips do have the

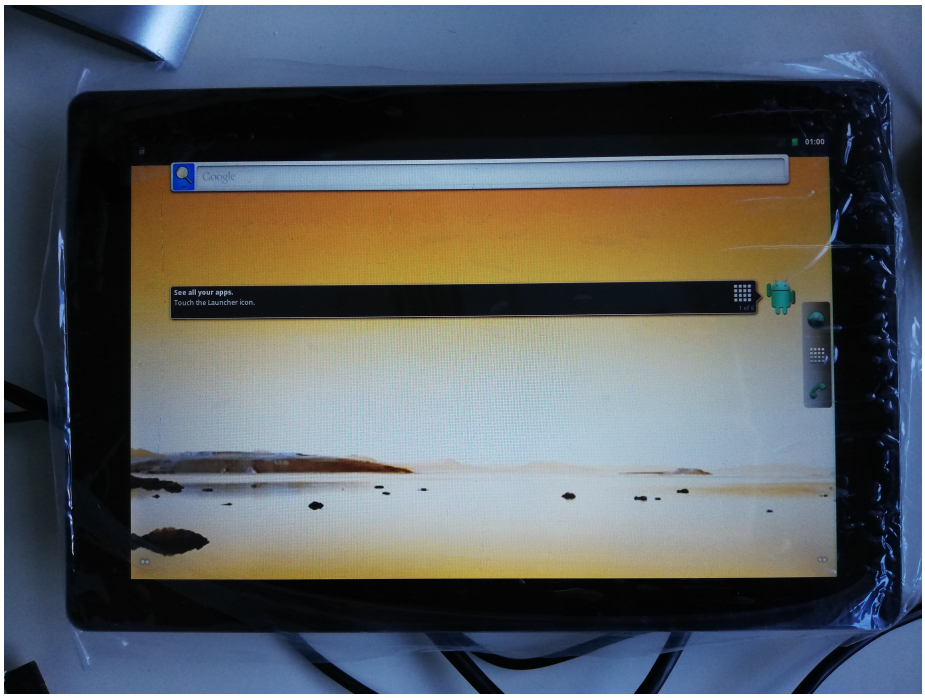

Figure 7: Android launcher on the ZedBoard display

Listing 11: Partial reconfiguration out of Android

```
1 cat new_bitstream.bin > /dev/xdevcfg
```

requirements to run the most up to date versions and Mentor Embedded even provides support for Android 6 (Marshmallow), 7 (Nougat) and 8 (Oreo). The same methodology as we presented could be employed on those higher range devices. The Mentor Embedded distribution was tested on the Zynq zcu102 developement board and was proven to be working. That distribution does not include a hardware design (the bitstream is distributed in binary format) and therefore not allow to alter the logic fabric on the FPGA. One could create a custom hardware design, but that task was deemed to be too time consuming considering the deadline for this project.

Noritsuna Imamura managed to get Android 5.0 running on the ZedBoard [?]. A linux swap partition was used to extend the RAM. Noritsuna states that his system is very slow. This is most probably due to the fact that it still uses software rendering and that the swap partition lies on the SD card and is therefore very slow. Nevertheless we tried to replicate his project but did not manage to do so. All the parts build without errors, but the system does not boot. Noritsuna has not configured the kernel to have a terminal console over the UART

interface, so we had no lead on where to start debugging.

3.2 Touchscreen Input

The touchscreen is not working in the current design. After configuring the kernel to support single- and multitouch devices and creating a *Input Device Configuration (IDC)*, the device is detected in Linux and input-events are sent to android. Due to unknown reasons, the events are never processed by the Android VM.

We observed the following:

- The device is recognized with all it's features by Android's built-in *getevent* utility
- Events are registered by Android's built-in *getevent* utility
- The translation from touch- to screen-coordinates works as designed
- Android's *InputReader.cpp* detects the device, creates an appropriate event handler and reacts to events
- When configured as multi-touch device, the events are not dispatched to the VM, because of the touchscreen never sending SYN_MT_REPORT messages. This could be amended by patching the *InputReader.cpp* to react to KEY_MT_SLOT messages instead. This was not tried because it seems like even if the events are dispatched, they are not picked up by the VM (see next point).
- When configured as single-touch device, the events are dispatched to the VM, but for unknown reasons they are never picked up. They are handed over from native code to Android's Dalvik Java *virtual machine (VM)* but never make it to the Java Input Queue in (*<repo>/android/frameworks/base/core/java/android/view*)
- We built *tslib*, a cross-platform library that provides access to touchscreens for Android. This library contains a calibration utility. That utility kills the Android VM and creates a minimal VM by itself. Even when started from Android, the utility recognizes all touches correctly. This suggests the issue lies within how the Android VM reacts to events and that the touchscreen driver is working fine
- Android Gingerbread has support for mice connected via USB-OTG. We observed that this is not working either. The behavior is much the same as with the touch-screen in single-touch configuration. Events are detected but not handled by the VM.

4 Future Improvements

In this section, ideas for further projects and how to improve the current design are presented.

4.1 Android Version

Since the higher end Zyqn development boards have official support for very recent Android versions, it would be great to migrate to project to one such platform, as discussed in Section 3.1. On a longer project it might make sense to create a custom hardware design to achieve this.

This would probably solve the issue discussed in Section 3.2 as well as it seems to be relevant to the ZedBoard. Also, Android natively supports external touch devices since version 6, so the hardware drivers are more sophisticated.

4.2 Image Processing Hardware

Image or video processing filters, which consume much more area on the FPGA on the one hand and have a much higher processing effort on the other hand, would be a very suitable application for a partial reconfigurable IOT device. At the Vivado IP Catalog, there are some cores, which would fit requirements for an efficient image and video processing purpose. These cores uses a AXI-Stream interface for the connection to the rest of the fabric. At this laboratory project these cores couldn't be used, because there was no license for generating a bitstream, which includes these cores, provided.

5 Conclusion

In general, we have shown in our work that DPR can be a useful method to exchange various applications based on hardware accelerators in IOT devices. Exchanging different filters for image or video processing applications is only one useful application where DPR can be an efficient method in terms of FPGA resources. Replacing different kinds of cryptographic algorithms or other functionalities are a useful application for an exchangeable DPR implementation as well. Finally there is a widely spread demand on powerful IOT devices, which can exchange some specific hardware blocks for different aims during runtime in a simple download process.