NANYANG TECHNOLOGICAL UNIVERSITY

ACCELERATED COMPUTING USING FPGA OVERLAYS WITH OS ABSTRACTION

by

RATHI CHETAN SANJAY
(G1502158F)

A Dissertation Submitted in partial fulfillment of
the requirements for the degree of
Master of Science in Embedded Systems

Supervised by

Assoc. Prof. Douglas L. Maskell

July 2016

Contents

List of Figures

List of Tables

Abbreviations

ACP	Accelerator Coherency Port
AMP	Asymmetric Multiprocessing
API	Application Programming Interface
BRAM	Block random access memory
BSP	Board Support Package
CDMA	Central Direct Memory Access
DMA	Direct Memory Access
FPGA	Field Programmable Gate Array
FSBL	First Stage Boot Loader
GP	General purpose
HP	High Performance
OCM	On-Chip Memory
OS	Operating system
PL	Programmable Logic
PS	Processing system

Abstract

Reconfigurable platforms for hardware acceleration have gained prominence as they deliver higher perfomance with lower power consumption. Such platfoms combine one or more general purpose processors with high speed computing fabrics like FPGA. The applications are accelerated by running control-intensive part on the processor and offloading compute intensive part to the programmable fabric. Due to lack of suitable abstractions, they have poor design productivity restricting their efficient use to hardware design experts. Overlay architectures provide an attractive solution for accelerated computing due to their improved design productivity through fast compilation, high-level design abstraction, and software-like programmability. Booting an OS on these platforms helps in co-ordinating multiple hardware tasks, efficient memory management and managing shared resources across the applications. In our work, we use one such overlay, the Vectorblox MXP Matrix processor, instantiated on the programmable fabric of Xilinx Zynq Zedboard for acceleration and analyzing the speedup obtained. It is programmed entirely in C/C++ by making use of vector-oriented data parallel programming model. We also setup Linux for using MXP to support processing data from files. Runtimes of benchmarks for applications using MXP APIs on Linux as well as in bare-metal mode were compared for getting the overheads due to Linux. Further we moved onto setup an asymmetric multiprocessing where bare metal application MXP application run on one CPU core and Linux is booted on another core. Apart from this, we also developed a system level driver which makes use of CDMA engine device driver for DMA transfers from DDR-DDR and DDR-BRAM. This helped in abstracting communication interface for memory subsystem using an OS.

Acknowledgment

I would like to express my deep and sincere gratitude to Assoc. Prof. Dr Douglas Leslie Maskell, for his constant and continuous support, encouragement and guidance. Furthermore, I would also like to acknowledge the crucial role of mentor Abhishek Kumar Jain, for his continuous support, effective suggestions and professional guidance in the entire phase of my dissertation.
I would also like to thank my friends and classmates for helping me with technical issues regarding my practical work. My heartfelt appreciation goes to my beloved parents, Mr. Sanjay Rathi and Mrs. Surekha Rathi, for their support and love throughout my studies at the University.

Chapter 1

Introduction

1.1 Motivation

Modern embedded systems often require extensive computing resources. In such systems, hardware acceleration has attained prominence due to the availability of embedded reconfigurable platforms which use high speed computing fabrics like Field Programmable Gate Array (FPGA). These platforms are scalable, consume low power and support isolated execution of tasks. For accelerating computations using programmable fabric, designing and debugging using RTL is required while waiting for slow place-and-route cycles as changes are made in the design. Non-availability of suitable abstractions prohibit commercial use of such platforms and restricts their effective use to hardware design experts. Overlay architectures have evolved as an attractive solution to this problem by providing software-like programmability and fast compilation. With the complexity of FPGA platforms growing exponentially the use of such overlay architectures might become mainstream practice [2]. Various levels of abstraction are possible from computation, programming, to communication interfaces and management. Furthermore using an Operating system (OS) in reconfigurable platforms helps in co-ordinating multiple hardware tasks, efficient memory management and managing shared resources across the applications. Also, integrating overlays with memory subsystem and processor is important to enable sharing and management of limited overlay resources. As communication interfaces and memory subsystem heavily affect the performance of such systems, these components need to be designed very carefully while providing abstraction for them using OS.

1.2 Contribution

The main contributions can be summarized as follows:

- Setting up Linux and accessing MXP overlay through it. Analyzing speedups obtained for MXP applications on Linux as well as in bare-metal mode and accelerating kernels.
- Setting up an Asymmetric Multiprocessing (AMP) System with Linux booted on one core and bare-metal MXP application running on another.
- Development of device driver for abstracting communication interface for DMA transfers from PS-PL

1.3 Organization

The remainder of the report is organized as follows:
Chapter 2 gives background information on acceleration computing, the platform used, FPGA overlays and OS abstraction. In chapter 3, we describe our experimentations with MXP vector processor. In chapter 4, we discuss the concept and methodology for setting up an AMP System. Chapter 5, describes the design and development of system level driver on top of AXI-CDMA device driver. We conclude in chapter 6 and discuss future work.

Chapter 2

Background

2.1 Accelerated Computing

For past several decades, there was reliance on Moores Law for getting better performance. Each year more transistors were added that were faster and consumed less power. But in recent years Moores Law has slowed down by a considerable amount. As frequency stopped scaling, it became much more difficult to squeeze out performance from a single sequential processor (CPU). So adding more CPU cores was seen as a solution. However with more CPU cores it became more challenging to gain performance out of these chips. This was due to the challenges in writing code that can assign computations across these different cores. Furthermore, some computations can't run in parallel at all. So the computing world ended up with multicore CPUs that could not accelerate all types of code. This made the designer's job even harder. Simply adding cores resulted into waste of transistors and also raised the cost to manufacture the processor without much benefit. Failure to improve performance of CPU, without affecting power budgets or using extremely complicated design methods resulted in the industry hitting a brick wall.

For gaining further performance, the processor design perspective had to change which led to development of heterogeneous computing systems. Heterogenous computing refers to systems that make use of more than one kind of processor. Performance gain or energy efficiency is achieved in these systems by adding dissimilar co-processors that integrate specialized processing capabilities for handling particular types of tasks. Accelerated computing is a computing model used for accelerating applications in the engineering and scientific domains wherein the computations are performed on specialized processors(a.k.a accelerators). It involves use of heterogeneous computing systems for enhancing performance for data parallel applications. The idea is to run code that is ideal for a particular kind of processor and is suitable for executing on it. For instance, serial code with lot of conditions and branches would be well suited to run on the CPU because thats most efficient for this type of code, whereas code that is massively parallel, and has less conditions would be well suited for execution on the accelerator.

2.2 Using FPGA Overlays for acceleration

FPGAs hold a special position when it comes to taking advantage of Moore's Law improvements in semiconductor technology [3]. The major FPGA vendors, Xilinx and Altera, have introduced reconfigurable platforms consisting of general purpose processors coupled with programmable logic. FPGAs have seen to be suitable for accelerating computations in a wide variety of applications. However developing an accelerator design using a hardware description language like Verilog is difficult, particularly requiring an expert in hardware design to perform the implementation, debugging and testing for developing real hardware. Due to difficulty of hardware design, long compilation times, and design productivity issues, FPGAs are restricted to niche applications which prevents their adoption for acceleration in general purpose computing. There is growing demand from software developers who are used to fast development cycles to make FPGAs more accessible to them by providing abstractions in the form of software Application Programming Interface (API).

Overlay architectures provide an attractive solution for accelerated computing due to their improved design productivity through fast compilation, high-level design abstraction, software-like programmability and run-time management [4],[5],[6],[7],[8],[9]. Other benefits include better design reuse, application portability across platforms and rapid reconfiguration that is much faster than partial reconfiguration on fine-grained FPGAs.

We use one such vector overlay known as MXP matrix processor for acceleration computing in our experimentations. It is a soft-core scalable vector processor developed by Vectorblox Computing Inc . It is provided as an IP core which when instantiated on the FPGA, enables us to accelerate data-parallel operations. Vectorblox provides rich C/C++ API support to overcome cumbersome hardware programming. The hardware design flow for using accelerator on FPGA goes through long design cycle(taking up hours to weeks) whereas using MXP, for a given configuration all we need to do is change the software code and the effects can be tested within minutes. MXPs parameterized design allows the user to specify the amount of parallelism needed, which can range from 1 to 128 or more parallel ALUs. It includes a parallel-access scratchpad memory to hold vector data and high-throughput Direct Memory Access (DMA) and scatter/gather engines. To provide maximum performance, the processor is expandable with custom vector instructions and DMA filters [10]. MXP seamlessly ties into existing Xilinx and Altera development procedures, simplifying system creation and deployment.

We have used Xilinx Zynq SoC [11] as a heterogeneous computing platform wherein the MXP soft-processor is instantiated on its programmable fabric. Next, we describe the platform in more detail.

2.3 Xilinx Zynq Zedboard

FPGA was completely the domain of hardware engineers earlier and software developers had stayed away from it. This reason resulted into Xilinx going beyond the FPGA and introducing the Extensible Programming Platform. These platforms are partitioned into Processing system (PS) consisting of one or more processors along with memory interfaces,bus and peripherals and the Programmable Logic (PL) where custom hardware can be instantiated. These two parts are connected via high throughput interconnect to maximize communication bandwidth. The principle is letting the main processor control an array of reconfigurable hardwares to perform compute-intensive tasks. FPGA fabric provides the power of reconfigurability alongwith application specific acceleration while letting the processor execute control intensive tasks. Xilinx Zynq Zedboard is a development and evaluation board which is based on Zynq-7000 All Programmable SoC architecure. It consists of Zynq Z7020-clg484 of speed grade -1(667 MHz) containing dual core ARM-Cortex A9 based processing system(PS) and programmable logic(PL) fabric in one package. The PS consists of a double-precision floating point unit, a hard DMA controller (PS-DMA),512 MB DDR RAM, commonly used peripherals and external memory interfaces. Block diagram for PS is shown in Figure 2.1. The components of PS are listed as below :

- Two ARM Cortex-A9 cores that are run-time configurable as a single processor, symmetric or asymmetric multiprocessor and are based on the ARMv7 ISA.
- NEON 128b SIMD coprocessor and VFPv3 per processor.
- 32KB instruction and L1 data caches per processor
- 512KB L2 cache that is shared between the processors
- Snoop Control Unit (SCU) and the ACP for cache coherent accesses.
- On-Chip Memory (OCM) with capacity of 256KB
- DDR controller comprising of AXI memory port interface, digital PHY and transaction scheduler.
- DMA controller with four channels for PS and PL

The PL is made up of Artix 7 FPGA fabric. It has 6 input LUTs and 36kb Block RAMs which can be configured as two 18 kb blocks. The processor in the system is first booted and PL is configured as a part of boot process or can be configured some time later in the future. PL can be either reconfigured completely or partially by making use of the partial reconfiguration(PR) feature. The PL configuration data is referred to as bitstream. PL is useful for real-time applications as it has predictable latency. Power can be managed by powering down the PL as it has a different power domain than the PS. The PL has a rich architecture capable of user configuration which are listed below [11]

- Configurable logic blocks (CLB) with 6-input lookup table (LUT)

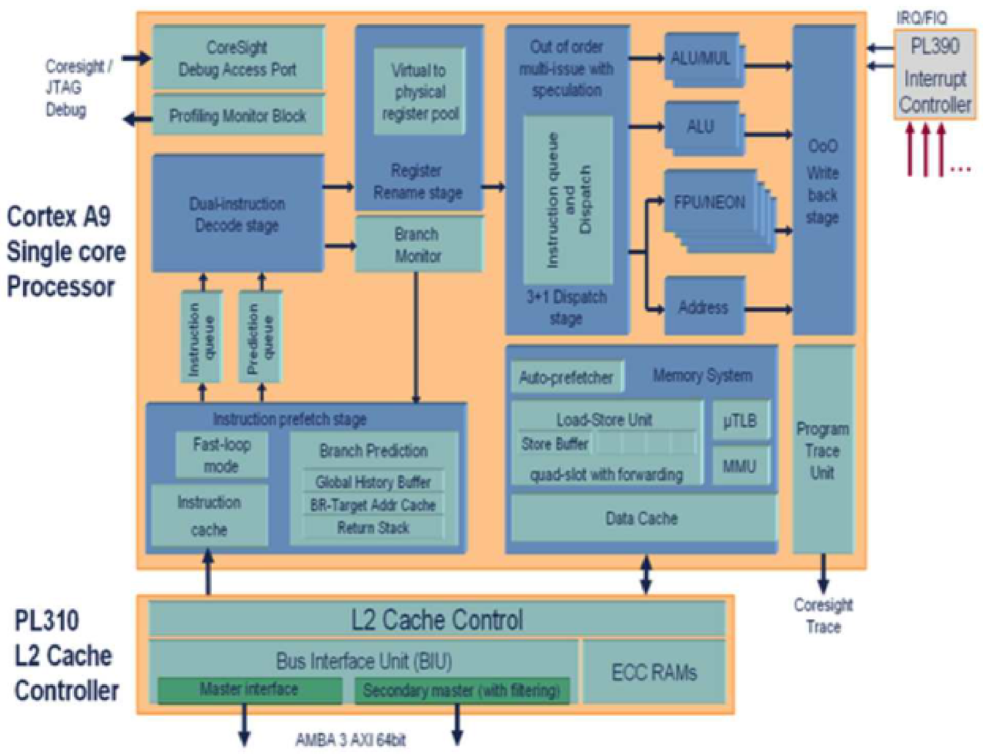


Figure 2.1: Processing System Block Diagram [1]

- 36KB block RAM with capability of dual port
- DSP48E1 Slice with optional pipelining, ALU and dedicated buses for cascading useful for Digital signal processing
- Low jitter clock distribution and low skew
- High performance I/Os that can be configured
- Dual Analog-to-Digital Converter (ADC) blocks with 12-bit and 1 MSPS rate

The ARM based reconfigurable system on Zedboard, makes use of multiple AXI interfaces for communication between the PS and the PL. Each interface provides for multiple AXI channels, which enables high throughput data transfer and eliminates performance bottlenecks for memory and I/O. There are three types of AXI interfaces to the fabric mentioned below:

- AXI_GP - Two 32-bit AXI master and two 32-bit AXI slave General purpose

(GP) ports.

- AXI_HP - Four 32-bit/64-bit configurable, AXI slave High Performance (HP) ports which are buffered alongwith direct access to DDR and OCM.
- AXI_ACP - One 64-bit AXI Accelerator Coherency Port (ACP) slave interface ensuring access to memory is coherent.

2.4 OS Abstraction

There has been lot of research on providing OS support for programmable fabric in order to provide a simple programming model to the user and for efficient run-time scheduling of software and hardware tasks [12, 13, 14, 15]. A technique for abstracting co-processors on the FPGA fabric in high performance reconfigurable computing (HPRC) systems was presented in [16]. ReconOS [17] which is based on an existing embedded OS (eCos) provides an execution environment by enhancing a programming model for multi-threading from software to reconfigurable hardware. RAMPSoCVM [18] provides hardware virtualization and runtime support for an SoC through APIs built on top of Embedded Linux for providing a standard interface for message passing.Various extensions of Linux have also been proposed to support FPGA hardware [19, 20].

Usage of OS in reconfigurable platforms helps in efficient memory management, co-ordinating multiple hardware tasks and managing shared resources across the applications [21]. Particularly, the presence of open-source OS like Linux provides better control over the scheduling of tasks, synchronization, abstracting communication interfaces, interrupt management etc. Due to overheads introduced, Linux applications will not perform as efficient compared to bare-metal applications, but as they are heavily abstracted from the underlying hardware it simplifies application development for software developers. Bare-metal applications require explicit handling of resources,communication and synchronization it poses a challenge for the developers to look into each of these issues. Some examples for OS support on FPGAs include, RIFFA [22][23], SIRC [24], and Xillybus [25].

Vectorblox provides some hints to setup Linux for using MXP [26] . Using these, alongwith the experience gained from booting Linux on Zedboard [27] combined with our knowledge in using Linux, we come up with the procedure for setting up Linux for using MXP. This procedure is described in detail in Section 3.3, so that in future this could be easily replicated and there would be more focus on using this system rather than dealing with setup issues. Furthermore, we also describe the procedure required for setting up an AMP system wherein Linux is booted on one CPU and other core runs bare-metal MXP application as described in chapter 4. We also created a system level driver which helps in providing communication abstraction for data transfer between the PS and the PL on Zedboard as described in chapter 5.

Chapter 3

MXP for Accelerated Computing

3.1 Vectorblox MXP

MXP has its local memory bank called as scratchpad and all vector operations are performed directly upon it which maximizes its performance because unlike traditional processors which have address and data registers, there are no load-stores of vector data. The architecture of MXP is composed of vector engine and DMA engine. The primary way to transfer data into or out of the scratchpad is by using DMA engine. Since MXP implements DMA to access the DDR memory directly, the cache-hierarchy is bypassed. Vector engine consists of multiple parallel vector lanes, the number of which can be configured from 1 to 256 and there is some 4 KB of scratchpad available for each vector lane.

MXP vector processor in instantiated onto the programmable logic of the Zynq Zedboard SoC as shown in Figure 3.1. The communication with ARM processor happens via general purpose ports whereas high performance ports are used for communication with the DDR memory. With 64 KB of scratchpad we can put 32 16-bit lanes/ 16 32-bit lanes / 64 8-bit lanes. The instruction port uses a dedicated AXI slave interface connecting to master GP port on PS. The scratchpad also connects its slave interface to PS through one of the master GP ports. The DMA engine is connected to the DDR memory controller though AXI slave HP ports. Maximum frequency achievable for the 16-lane MXP soft processor is 110 MHz.

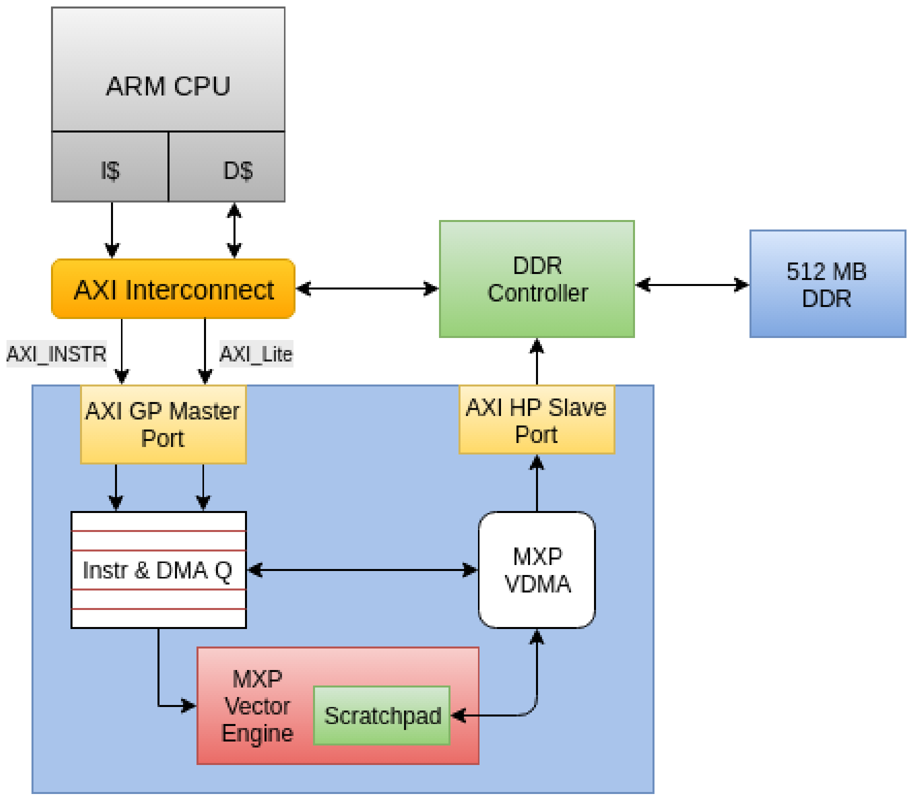


Figure 3.1: MXP Matrix processor on Zedboard

3.2 Programming Methodology

A basic program for MXP can be written by following the procedure below,

1. Allocate vectors on the local scratchpad.
2. DMA transfer from DDR to local scratchpad.
3. Set the vector length which indicates the number of vector elements on which the vector operations are to be performed.
4. Perform required vector computations and obtain result.
5. DMA transfer of result from local scratchpad to DDR.

A sample example to explain the methodology is shown in Figure 3.2.

```

int input_data[256] = { 1,2,3, 10, 11, 12, 13, ... ,256 };
int multi_factor = 7;

1. Allocating vectors in the scratchpad
    vbx_word_t* v_data;
    v_data = vbx_sp_malloc( 256*4 ); // 256 words allocated in scratchpad

2. Moving data from DDR to local scratchpad
    vbx_dcache_flush( input_data, 256*4 ); //Flush cache to ensure data is
                                             actually present in DDR

    vbx_dma_to_vector( v_data, input_data, 256*4 );

3. Setting of vector length to indicate number of elements
     vbx_set_vl( 256 ); // Number of elements

4. Performing scalar vector multiplication
     vbx( SVW, VMUL, v_data, multi_factor, v_data ); //output result written
                                                   back to v_data in scratchpad

5. Moving data from local scratchpad to DDR
     vbx_dma_to_host( data, vdata, 256*4 );
     vbx_sync();    // wait for all the vector and DMA engine operations
                                               to finish

                    Example: Multiplication of scalar with Vector
```

Figure 3.2: Sample program for MXP

The details and syntax for the API's can be found at [28]. The above bare-metal program works fine, however, what if we want to perform some processing on image,audio, video,etc. In that case we will need a filesystem through which we can extract bytes from the input file without having to manually enter values as in the example above. Also the results after processing needs to be written to a file in the appropriate file format depending on the input file type. Vectorblox does provide sources for Linux containing necessary device drivers alongwith prebuilt bitstreams and hardware design file generated through Vivado. We further provide detailed steps for setting up Linux to use MXP so that it is easy to replicate this thing in the future.

3.3 Setting up Linux for using MXP on Zedboard

We install Vivado along with SDK option so that cross-compiler required for building the Linux kernel sources is also installed without us needing to separately install some cross-compiler toolchain for ARM. We plan to use a persistent filesystem rather than ramdisk as ramdisk will lose changes made to filesystem when the board is powered off. So we setup Xillybus on Zedboard to have an sdcard ready with a root filesystem(which is persistent). Next we need to build First Stage Boot Loader (FSBL), device tree, U-boot, and the Linux kernel for booting up the board with the required MXP setup for which we need to clone the following sources(repositories) :

1. MXP repo : https://github.com/VectorBlox/mxp.git
2. Linux repo : https://github.com/VectorBlox/linux-xlnx.git
3. U-boot repo : https://github.com/Xilinx/u-boot-xlnx.git
4. Device-tree repo : https://github.com/Xilinx/device-tree-xlnx.git

Inside MXP repo, prebuilt bitstream(named system_wrapper.bit to be instantiated on the FPGA) and hardware design file(named system.hdf) have been provided for 8 and 16 vector lanes. We used the one with 16 vector lanes.

3.3.1 Building FSBL

Inside MXP repo, navigate to folder prebuilt_zedboard_arm_viv_v16 and type commands shown next. Once done rename executable.elf inside mxp_fsbl to fsbl.elf

```
hsi //This will switch to a TCL shell.
hsi% set hw_design [open_hw_design system.hdf]
hsi% generate_app -hw $hw_design -os standalone -proc ps7_cortexa9_0 -app
    zynq_fsbl -compile -sw fsbl -dir mxp_fsbl
hsi% quit
```

3.3.2 Building U-boot

1. Navigate to u-boot repo.Since we plan to use persistent filesystem rather than ramdisk below modification is required in file include/configs/zynq_common.h. Find sdboot entry and edit it to avoid loading ramdisk

```
// change this
"sdboot=echo Copying Linux from SD to RAM...;" \
       "mmcinfo;" \
       "fatload mmc 0 0x3000000 ${kernel_image};" \
       "fatload mmc 0 0x2A00000 ${devicetree_image};" \
       "fatload mmc 0 0x2000000 ${ramdisk_image};" \
       "bootm 0x3000000 0x2000000 0x2A00000\0" \
// to this
"sdboot=echo Copying Linux from SD to RAM...;" \
       "mmcinfo;" \
       "fatload mmc 0 0x3000000 ${kernel_image};" \
       "fatload mmc 0 0x2A00000 ${devicetree_image};" \
       "bootm 0x3000000 - 0x2A00000\0" \
```

2. Then to compile u-boot give commands as shown below :

```
export CROSS_COMPILE=arm-xilinx-linux-gnueabi-
export ARCH=arm
make zynq_zed_config
make
```

   Rename u-boot to u-boot.elf. While building Linux kernel we need mkimage utility, so add the tools/ folder to the $PATH variable.

3.3.3 Building Linux kernel

1. Navigate to Linux repo and give following commands

```
export CROSS_COMPILE=arm-xilinx-linux-gnueabi-
make ARCH=arm xilinx_zynq_defconfig
```

This will generate a .config file

2. For building the kernel alongwith support for MXP we need to find out the appropriate configuration parameters. To do this, we do some sort of reverse engineering as described further. If we look at example source codes provided in the bmark(benchmark) directory of MXP repo, we see the first function to be called in these sample examples is vbx_test_init(). For these source codes to run on Linux, this function calls VectorBlox_MXP_Initialize("mxp0","cma"). If we look at the definition of this function, it actually uses device files /dev/mxp0 and /dev/cma to do some memory mapping and other initializations. Since these device files are used, there must be corresponding device drivers for them in the Linux source. Sure enough we find files named mxp.c and cma.c in the drivers/char/ directory which are responsible for creating these device files. We then look at the Makefile in drivers/char/ directory and find entries for these files as shown below

```
obj-$(CONFIG_MXP) += mxp.o
obj-$(CONFIG_CM_ALLOCATOR) += cma.o
```

Now we have the configuration parameters and hence we edit the .config file generated in step 1 and set these parameters as :

```
CONFIG_MXP=m
CONFIG_CM_ALLOCATOR=y
```

3. Compile the kernel :

```
make ARCH=arm UIMAGE_LOADADDR=0x8000 uImage
```

This will generate compiled kernel uImage in arch/arm/boot/ directory

4. Compile the kernel modules :

```
make ARCH=arm modules
```

5. Insert the sdcard with Xillybus setup into the computer where this kernel is compiled and give:

```
make ARCH=arm modules_install INSTALL_MOD_PATH=/path/to/rootfs(
    ext4partition)/in/sdcard/
```

This will copy all the kernel modules built in step 4 into the path : INSTALL_MOD_PATH/lib/modules/{kernel_image_name}/

3.3.4 Building device tree

1. Inside Linux repo, compile device tree source(dts) to generate device tree blob(dtb)

   ```
   ./scripts/dtc/dtc -I dts -O dtb arch/arm/boot/dts/zynq-zed.dts -o mxp.
       dtb
   ./scripts/dtc/dtc -I dtb -O dts mxp.dtb -o mxp_linux.dts
   ```

   To avoid using ramdisk, we replace the contents of bootargs under the chosen node in mxp_linux.dts as shown

   ```
   //change this
   bootargs = "console=ttyPS0,115200 root=/dev/ram rw earlyprintk";
   //to this
   bootargs = "console=ttyPS0,115200 root=/dev/mmcblk0p2 rw earlyprintk
       rootfstype=ext4 rootwait devtmpfs.mount=0";
   ```

2. Copy folders from drivers/ directory inside MXP repo to the Device tree repo. Navigate to folder prebuilt_zedboard_arm_viv_v16 in MXP repo, and enter following commands

   ```
   hsi
   hsi% open_hw_design system.hdf
   hsi% set_repo_path /path/to/Device tree repo/
   hsi% create_sw_design mxp_device_tree -os device_tree -proc
       ps7_cortexa9_0
   hsi% generate_target -dir mxp_dts
   hsi% quit
   ```

   After this we see some dts/dtsi files created inside mxp_dts folder. We already have the device tree entries corresponding to various peripherals in mxp_linux.dts file created in step 1. So we are only interested in pl.dtsi files which will contain the device tree node for the MXP soft processor instantiated on programmable logic. Contents in pl.dtsi is as shown in Figure 3.3. Given the way in which the

```
amba_pl: amba_pl {
    #address-cells = <1>;
    #size-cells = <1>;
    compatible = "simple-bus";
    ranges ;
    vectorblox_mxp_arm_0: vectorblox_mxp@b0000000 {
        compatible = "xlnx,vectorblox-mxp-1.0";
        reg = <0xb0000000 0x10000 0x40000000 0x100000>;
        vblx, = <1>;
        vblx,archical = <0>;
        vblx,beats_per_burst = <16>;
        vblx,burstlength_bytes = <128>;
        vblx,c_instr_port_type = <2>;
        vblx,c_m_axi_addr_width = <32>;
        vblx,c_m_axi_data_width = <64>;
        vblx,c_m_axi_len_width = <4>;
        vblx,c_m_axi_supports_threads = <0>;
        vblx,c_s_axi_addr_width = <32>;
        vblx,c_s_axi_baseaddr = <0xB0000000>;
        vblx,c_s_axi_data_width = <32>;
        vblx,c_s_axi_highaddr = <0xB000FFFF>;
        vblx,c_s_axi_instr_addr_width = <32>;
        vblx,c_s_axi_instr_baseaddr = <0x40000000>;
        vblx,c_s_axi_instr_data_width = <32>;
        vblx,c_s_axi_instr_highaddr = <0x400FFFFF>;
        vblx,c_s_axi_instr_id_width = <6>;
        .........................................
        .........................................
        .........................................
        vblx,vector_lanes = <16>;
    };
};
```

Figure 3.3: DTS entry for MXP

device driver file mxp.c parses this device tree node we need to change contents of this file for the correct setup Specifically we need to change compatible string to

```
compatible = "vectorblox.com,vectorblox-mxp-1.0";
```

Also the register entry should contain instruction address first. So,node name and reg entry should be changed to

```
vectorblox_mxp@40000000{
reg = <0x40000000 0x100000 0xb0000000 0x10000>;
```

Copy the edited amba_pl node to the end of mxp_linux.dts

3. Finally compiling the device tree

```
./scripts/dtc/dtc -I dts -O dtb mxp_linux.dts -o devicetree.dtb
```

3.3.5 Packing into BOOT.BIN

Create a file with name, say bootimage.bif. Paste following contents into it

```
the_ROM_image:
{
    [bootloader] <path to fsbl.elf>
                <path to bitstream file>
                <path to u-boot.elf>
}
```

Save the file and generate boot.bin using the command below :

```
bootgen -image bootimage.bif -o i boot.bin -w on
```

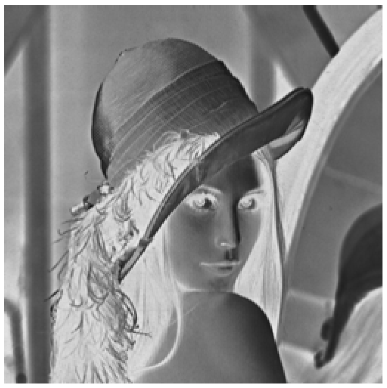
Copy files boot.bin, devicetree.dtb and uImage into the FAT partition of sdcard. Plug-in the sdcard and boot Zedboard. While the kernel is getting loaded we should some message like this

```
............
............
mxp_init
mxp_probe
............
............
```

meaning mxp driver is successfully loaded.

3.3.6 Sample example on Linux

Further to use MXP API's, on Linux we need to compile vbxapi library present in MXP repo.Once built this library needs to be linked while compiling the application we write on Linux. A sample example for performing the negation of an image is shown in Figure 3.4. The input and output images obtained after running the sample program are shown below



3.4 Experimentation and Results

The prebuilt bitstream for 16 vector lanes is used for the experimentations we describe further. Some configuration parameters available with this setup are as shown below

```
vector_lanes = 16
core_freq = 100.0e6
scratchpad_size = 65536
dma data width in bytes = 8
fxp_word_frac_bits = 16
fxp_half_frac_bits = 15
fxp_byte_frac_bits = 4
```

3.4.1 Accelerating PolyBench kernels

PolyBench [29] stands for the Polyhedral Benchmark suite. It consists of several benchmarks written with the purpose of having a uniform suite for the monitoring

```c
int main(int argc, char *argv[])
{
    pgm_t opgm;
    pgm_t ipgm;
    int img_size = 0;
    unsigned char *v_sub = NULL;
    unsigned char max_val = 255;
    //Initialization function to be called for using MXP in Linux
    VectorBlox_MXP_Initialize("mxp0","cma");

    //Reading image
    readPGM(&ipgm,"lena_256x256.pgm");
    img_size = (ipgm.width * ipgm.height);

    //Allocate buffer for output image
    opgm.width = ipgm.width;
    opgm.height = ipgm.height;
    opgm.buf = (unsigned char*)vbx_shared_malloc(img_size * sizeof(unsigned
        char));

    //Allocate vector on scratchpad
    v_sub = (unsigned char *)vbx_sp_malloc(img_size * sizeof(unsigned char ));

    //Transfer input bytes from memory to scratchpad
    vbx_dma_to_vector(v_sub, ipgm.buf, img_size);
    vbx_set_vl(img_size);

    //Scalar vector subtraction for taking negative of image
    vbx(SVBU, VSUB, v_sub, max_val, v_sub);

    //Writing result from scratchpad to memory
    vbx_dma_to_host(opgm.buf, v_sub, img_size);
    vbx_sync();

    //Writing output image
    writePGM(&opgm,"out_lena_negative.pgm");

    //Free allocated pointers
    vbx_sp_free();
    vbx_shared_free(ipgm.buf);
    vbx_shared_free(opgm.buf);
}
```

Figure 3.4: Sample MXP program on Linux

and execution of kernels. Typical features of Polybench include :

1. Availability of syntactic constructs so that there is no elimination of execution due to dead code in the kernel
2. For kernel instrumentation there is a single file which can be tuned during compile time.
3. Support for cache flushing operations which are performed before the kernel starts executing giving more accurate timings as cache hits and misses can affect execution times giving unpredictable results.
4. Supports setting of real-time scheduling to prevent interference from OS

We accelerate two kernels viz. atax and bicg present in this suite using MXP. Since MXP only supports data representation in fixed point format, we change the default data type which is double to integer for the computations to have proper comparison and idea about the speedup as ARM on Zedboard does support floating point operations.

3.4.1.1 ATAX

ATAX is abbreviation for A transpose A times X. It is a linear algebra kernel for matrix multiplication and vector transpose. We accelerate the kernel for small dataset with matrix size 500x500 and also for standard dataset size with matrix of 4000x4000. For small dataset depending upon the available space in scratchpad after allocating the required vectors we DMA maximum chunk of data to improve performance. The speedup obtained using MXP is as shown below

Table 3.1: Speedup for ATAX kernel

Dataset	Execution time on ARM alongwith NEON(in ms)	Execution time using MXP(in ms)	Speedup
Small	6.544	1.272	5.14
Standard	482.941	106.372	4.54

3.4.1.2 BiCG

BiCG stands for Biconjugate gradient method. In numerical algebra, it is used to solve system of linear equations of the form

$$Ax = b$$

In polybench, this is a sub kernel for the BICGSTAB(Biconjugate Gradient Stabilized method) which is an the iterative linear solver. We accelerate the kernel for small dataset with matrix size 500x500 and also for standard dataset size with matrix of 3200X3200. The speedup obtained using MXP is as shown below

Table 3.2: Speedup for BiCG kernel

Dataset	Execution time on ARM alongwith NEON(in ms)	Execution time using MXP(in ms)	Speedup
Small	5.88	0.867	6.78
Standard	217.278	72.014	3.02

3.4.2 Running benchmarks

We run some benchmark codes present in the MXP repo, these being provided by VectorBlox itself gives further idea about the kind of speedup that can be obtained using MXP soft-vector processor. Scalar time is measured for code running on ARM alongwith SIMD NEON unit and vector time represents time required when MXP is used for acceleration.

3.4.2.1 Results on Linux

The jumper settings on Zedboard are made so that it boots up Linux through the sdcard. The results obtained are shown in Table 3.3

Table 3.3: Speedup for benchmarks on Linux

Benchmark	Scalar time in seconds	Vector time in seconds	Speedup
vbw_mtx_fir_t	10.62e-3	1.751e-3	6.063
vbw_mtx_median_t	123.4e-3	4.885e-3	25.26
vbw_mtx_sobel	187.4e-3	24.94e-3	7.513
vbw_mtx_mm_t	276.5e-6	18.43e-6	15.00
vbw_vec_fir_t	847.9e-6	55.3e-6	25.33
imgblend	737.3e-6	313.3e-6	2.353

3.4.2.2 Bare-metal results

The jumper settings on Zedboard are made so that it boots in JTAG mode. Results obtained are shown in Table 3.4

Table 3.4: Speedup for bare-metal execution of benchmarks

Benchmark	Scalar time in seconds	Vector time in seconds	Speedup
vbw_mtx_fir_t	10.73e-3	889.0e-6	12.07
vbw_mtx_median_t	90.55e-3	4.826e-3	18.76
vbw_mtx_sobel	185.7e-3	24.72e-3	7.514
vbw_mtx_mm_t	76.00e-6	17.92e-6	4.242
vbw_vec_fir_t	836.0e-6	48.15e-6	17.36
imgblend	571.2e-6	298.0e-6	1.917

3.4.2.3 Overheads due to Linux on MXP

The overheads can be calculated by taking the difference between vector time taken on Linux versus bare-metal execution as shown in Table 3.5

Table 3.5: Overheads for benchmarks due to Linux

Benchmark	Vector time on Linux in seconds	Vector time for baremetal execution in seconds	Overhead in microsecs
vbw_mtx_fir_t	1.751e-3	889.0e-6	862
vbw_mtx_median_t	4.885e-3	4.826e-3	59
vbw_mtx_sobel	24.94e-3	24.72e-3	220
vbw_mtx_mm_t	18.43e-6	17.92e-6	0.5
vbw_vec_fir_t	55.30e-6	48.15e-6	7.15
imgblend	313.3e-6	298.0e-6	15.3

In the next chapter, we propose a possible solution to avoid the overheads incurred due to use of an OS, without losing the benefits it provides such as support for a filesystem.

Chapter 4

Asymmetric Multiprocessing on Zedboard

The processing system on Zynq Zedboard has two ARM Cortex-A9 cores that can be configured during run time as single processor, symmetric or asymmetric multiprocessor. Asymmetric Multiprocessing is a mechanism which provides us the opportunity to boot different operating system on each processor, boot operating system on one core and other can run bare metal application, or both cores can be running independent bare-metal applications. It allows us to assign different roles to each core so that we have separate cores,each performing different job within a cluster. Also, the processors may communicate with each other through shared resources if there is a need to do so.

In the current design, we boot Linux on one processor(CPU0) and other processor(CPU1) will be running bare-metal application. Xilinx provides an application note XAPP1078 [30] demonstrating a sample design for AMP on ZC702 board. Initially, we modified this sample design to get it working for Zedboard and implemented it to become conversant with the concepts and terminology involved for developing an AMP system. Further we moved on to develop a similar system alongwith the necessary modifications required for the setup, now with bare-metal application on CPU1 using MXP programming API's for acceleration. As seen further in this chapter we describe how one might write applications for CPU0 and CPU1 and their interaction.

The reading and writing of the image happens in Linux running on CPU0 whereas the necessary image processing using MXP is done in bare-metal application running on CPU1.This might provide a way to avoid overheads incurred due to use of an operating system while still being able to use its features. Also the executable file for the application to be run on CPU1 will actually be loaded by CPU0. This paves the way for dynamically changing the executable loaded on CPU1 during runtime depending on system requirement which we desribe as part of the future work.

4.1 Methodology

The processing system on Zynq SoC includes resources that are private to each CPU such as L1 cache, private timers, memory management unit(MMU), etc. Also there are resources shared by both CPUs such as L2 cache, RAM, On chip memory(OCM), Snoop control unit(SCU), Interrupt control Distributor(ICD). When running the system for AMP configuration we need to consider proper synchronization while accessing these shared resources and also prevent the processors from contending for them. The processors communicate through OCM as it provides low latency access as compared to DDR.

To avoid problems related to access of shared resources, following things are done,

1. Out of the 512 MB DDR memory available on Zedboard, Linux is assigned initial 384 MB and remaining 128 MB is assigned for running bare-metal application.

2. At any point in time, only one CPU will be writing or reading a location in the OCM to prevent contention.

3. Access to L2 cache from CPU1 is disabled because if CPU1 decides to flush L2, L2 cache being a shared resource, it could flush out the part of L2 that contains CPU0 code.

4.2 Setup on CPU0

Clone following sources :

1. Linux repo : https://github.com/Xilinx/linux-xlnx.git
2. U-boot repo : https://github.com/Xilinx/u-boot-xlnx.git
3. MXP repo : https://github.com/VectorBlox/mxp.git

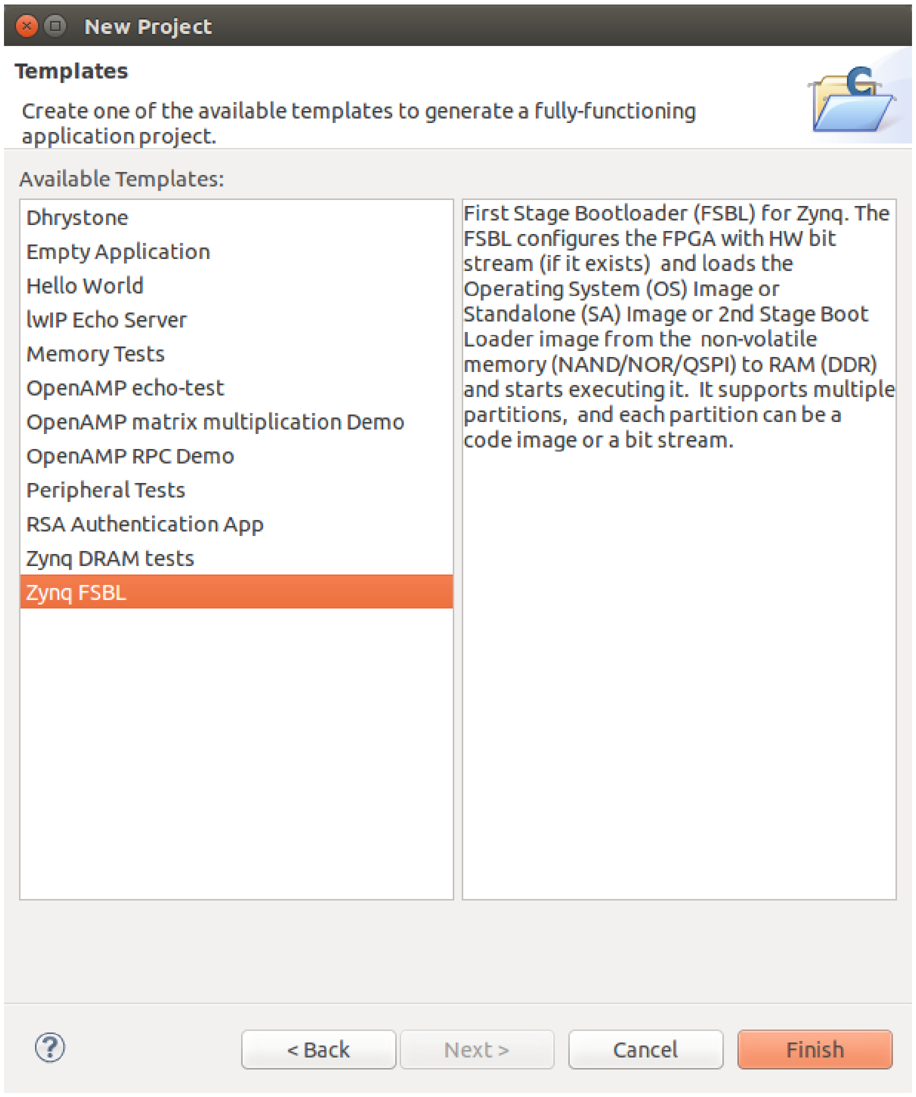
Similar to setup on MXP 3.3 we make use of the root filesystem available on SD card through Xillybus setup.

4.2.1 FSBL

When the board is booted the on chip ROM code will be loaded, this code will look for the FSBL executable and load into memory. FSBL will handle the following tasks,

1. Configuring the bitstream onto the FPGA.
2. Loading DDR controller.
3. Loading U-boot executable from SD card into DDR and further execute it. U-boot in turn acts as a second stage boot loader by loading and executing the Linux kernel onto the RAM.
4. Initialization of the phase locked loop.

First FSBL will look for bit file and if found will program the PL with it. Next it will look for all the executables that are available and load them into RAM. Once loaded it will start executing the first executable which was loaded. Using Xilinx SDK, FSBL can be generated by navigating the menu as File >Create >New >Application Project and selecting Zynq FSBL from template applications as shown in Figure 4.1



Create one of the available templates to generate a fully-functioning application project.

Available Templates:

- Dhrystone
- Empty Application
- Hello World
- lwIP Echo Server
- Memory Tests
- OpenAMP echo-test
- OpenAMP matrix multiplication Demo
- OpenAMP RPC Demo
- Peripheral Tests
- RSA Authentication App
- Zynq DRAM tests
- Zynq FSBL

First Stage Bootloader (FSBL) for Zynq. The FSBL configures the FPGA with HW bit stream (if it exists) and loads the Operating System (OS) Image or Standalone (SA) Image or 2nd Stage Boot Loader image from the non-volatile memory (NAND/NOR/QSPI) to RAM (DDR) and starts executing it. It supports multiple partitions, and each partition can be a code image or a bit stream.

Figure 4.1: Build FSBL through SDK

4.2.2 U-boot

Inside U-boot sources, changes required to prevent loading of ramdisk must be done similar to that described previously in Section 3.3.2. Further as the amount of RAM that is available to Linux running on CPU0 has to be limited to 384 MB, we need to edit the file include/configs/zynq_zed.h as described,

```
#define PHYS_SDRAM_1_SIZE (384 * 1024 * 1024)
```

The U-boot executable can then be obtained by following the steps described in Section 3.3.2.

4.2.3 Linux kernel

A simple way to implement an AMP system is to configure Linux as Symmetric Multiprocessing(SMP) but limit the number of CPUs seen by it to one. Doing so ensures that Linux takes care of correctly configuring the SCU and ICD for a multi-CPU environment. The linux sources can be compiled similar to the steps described in Section 3.3.3 with one caveat, unlike the setup on MXP, we don't have to enable any kernel configuration parameters here. Also symmetric multiprocessing is already enabled in the default configuration xilinx_zynq_defconfig.

4.2.4 Device-tree

Inside Linux sources, we make changes to bootargs entry in the device tree source(dts) file arch/arm/boot/dts/zynq-zed.dts to make use of persitent filesystem present in the ext4 partition of SD card (/dev/mmcblk0p2) as described in previous chapter. Further we add maxcpus=1 to bootargs so that Linux boots only on one core. So the bootargs entry should look something like this,

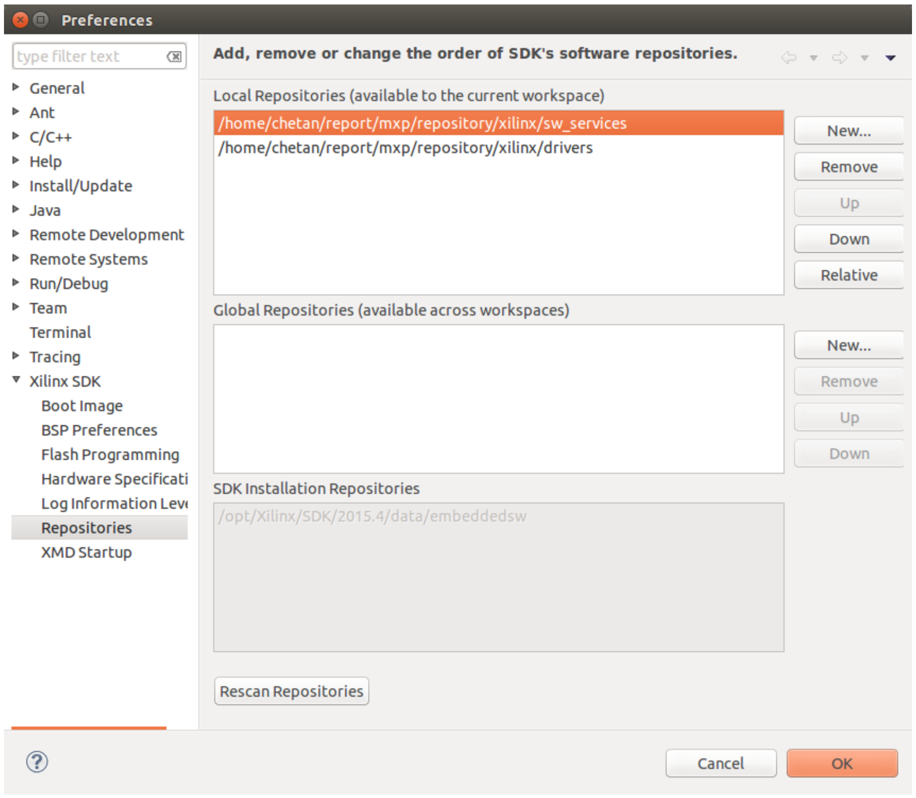
```
bootargs = "console=ttyPS0,115200n8 consoleblank=0 root=/dev/mmcblk0p2 rw
    rootwait earlyprintk maxcpus=1";
```

Also to limit the RAM available to Linux the memory node needs to be changed as

```
\\change this
ps7_ddr_0: memory@0 {
       device_type = "memory";
       reg = < 0x0 0x20000000 >;
   } ;
\\to this
ps7_ddr_0: memory@0 {
       device_type = "memory";
       reg = < 0x0 0x18000000 >;
   } ;
```

4.3 Setup on CPU1

1. For using the MXP APIs in our bare-metal application, we need to include the drivers/ folders present inside MXP repository and also if the use of vbxware library is required the sw_services/ folder should be added to the repository path in case these paths aren't already included. Using SDK, the repositories can be included by navigating the menu as Xilinx Tools >Repositories and adding required paths as shown in Figure 4.2. This will cause the drivers to be included as part of the Board Support Package (BSP) which we create next.



Add, remove or change the order of SDK's software repositories.

Local Repositories (available to the current workspace)

/home/chetan/report/mxp/repository/xilinx/sw_services
/home/chetan/report/mxp/repository/xilinx/drivers

Global Repositories (available across workspaces)

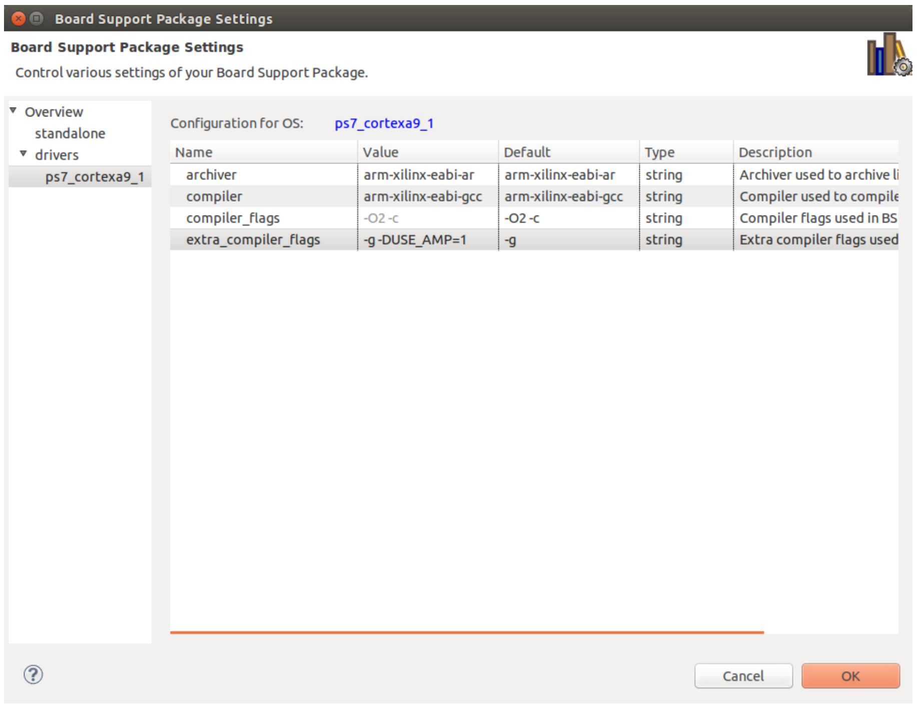
SDK Installation Repositories

/opt/Xilinx/SDK/2015.4/data/embeddedsw

Figure 4.2: Adding repository paths in SDK

2. We create a new standalone BSP for CPU1(ps7_cortexa9_1) with name amp_cpu1_bsp required for our bare-metal application as shown in Figure 4.3. To disable L2 cache access for CPU1, the assembly file boot.S contained in this standalone BSP should be modified. This change was already available with our version of

BSP in the form of preprocessor constant definition with name USE_AMP. So we modify the settings of this BSP to add the extra compiler flag -DUSE_AMP=1 to compile it for our desired setting.


Board Support Package Settings

Board Support Package Settings

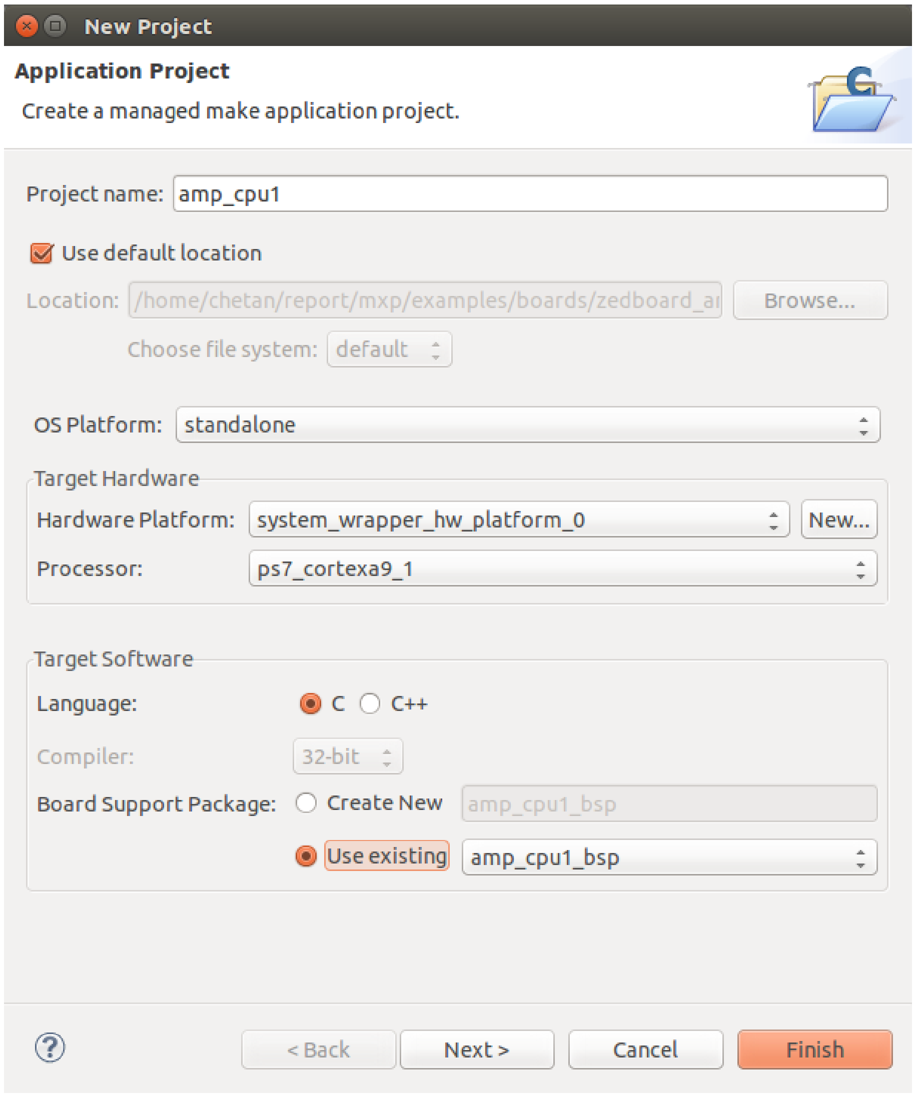
Control various settings of your Board Support Package.

Configuration for OS: ps7_cortexa9_1

Name	Value	Default	Type	Description
archiver	arm-xilinx-eabi-ar	arm-xilinx-eabi-ar	string	Archiver used to archive li
compiler	arm-xilinx-eabi-gcc	arm-xilinx-eabi-gcc	string	Compiler used to compile
compiler_flags	-O2 -c	-O2 -c	string	Compiler flags used in BS
extra_compiler_flags	-g -DUSE_AMP=1	-g	string	Extra compiler flags used

Figure 4.3: AMP setting for BSP

3. Next we create a new empty application project for CPU1(ps7_cortexa9_1) with name amp_cpu1 and make it use our existing BSP which we just created and compiled above as shown in Figure 4.4. After creating the project, we add a new source file for our bare-metal application code, the structure of which we will describe further in this chapter. The bare metal application should use the upper 128 MB of the DDR and also the FSBL should place its executable at 0x1800000 which is the starting address for the upper 128 MB.


New Project

Application Project

Create a managed make application project.

Project name: amp_cpu1

Use default location [✓]

Location: /home/chetan/report/mxp/examples/boards/zedboard_ar

Choose file system: default

OS Platform: standalone

Target Hardware

Hardware Platform: system_wrapper_hw_platform_0

Processor: ps7_cortexa9_1

Target Software

Language: [✓ C] C++

Compiler: 32-bit

Board Support Package: Create New amp_cpu1_bsp [✓ Use existing] amp_cpu1_bsp

Figure 4.4: Using existing BSP for application

To do this, we change the address range in the MEMORY section of the linker script file lscript.ld as shown,

```
\\change this
ps7_ddr_0_S_AXI_BASEADDR : ORIGIN = 0x100000, LENGTH = 0x1FF00000
\\to this
ps7_ddr_0_S_AXI_BASEADDR : ORIGIN = 0x18000000, LENGTH = 0x8000000
```

Also, depending on the number of bytes we might allocate in this baremetal application or the amount of stack the code might use during runtime, the heap and stack size should be changed in the linker script. If not done so, it might lead to undesirable results. Typically errors such as these can be difficult to track if there is proper error reporting available.

4.4 Booting the system

We create a file with name say boot.bif whose contents are described as follows

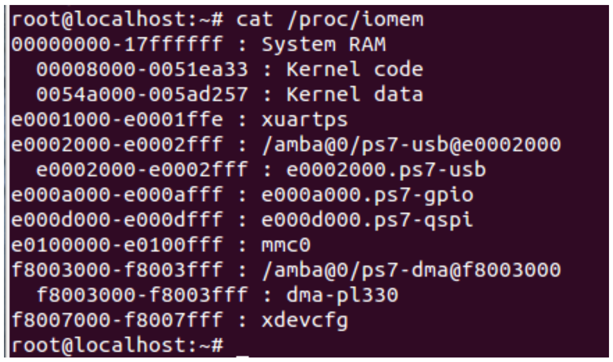
```
the_ROM_image:
{
    [bootloader] <path to zynq fsbl executable>
                 <path to bitstream file>
                 <path to u-boot executable>
                 <path to bare metal CPU1 executable>
}
```

Note that the bare metal application executable will also be part of binary file boot.bin. This implies CPU0 will be responsible for loading bare metal executable. Save the file and generate boot.bin using the command below

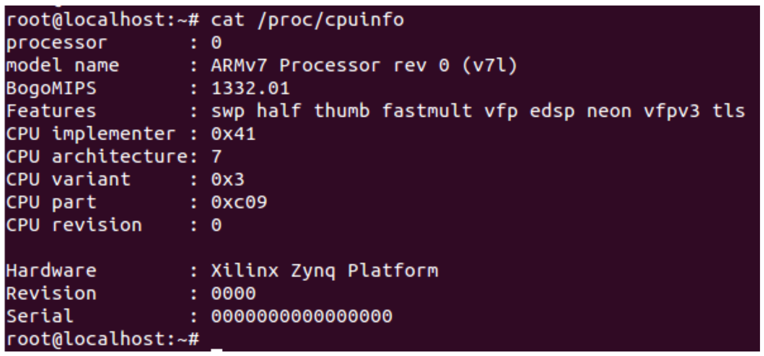
```
bootgen -image boot.bif -o i boot.bin -w on
```

Copy files boot.bin, devicetree.dtb and uImage into the FAT partition of sdcard. Setting jumpers on Zedboard to boot from SD card, the board is booted. After booting once the prompt hits the Linux shell, reading the contents of /proc/iomem and /proc/cpuinfo/ shows system RAM limited to 384 MB and only one CPU to

Linux as seen in the screenshots below,



```
root@localhost:~# cat /proc/iomem
00000000-17ffffff : System RAM
  00008000-0051ea33 : Kernel code
  0054a000-005ad257 : Kernel data
e0001000-e0001ffe : xuartps
e0002000-e0002fff : /amba@0/ps7-usb@e0002000
  e0002000-e0002fff : e0002000.ps7-usb
e000a000-e000afff : e000a000.ps7-gpio
e000d000-e000dfff : e000d000.ps7-qspi
e0100000-e0100fff : mmc0
f8003000-f8003fff : /amba@0/ps7-dma@f8003000
  f8003000-f8003fff : dma-pl330
f8007000-f8007fff : xdevcfg
root@localhost:~# _
```



```
root@localhost:~# cat /proc/cpuinfo
processor       : 0
model name      : ARMv7 Processor rev 0 (v7l)
BogoMIPS        : 1332.01
Features        : swp half thumb fastmult vfp edsp neon vfpv3 tls
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x3
CPU part        : 0xc09
CPU revision    : 0

Hardware        : Xilinx Zynq Platform
Revision        : 0000
Serial          : 0000000000000000
root@localhost:~# _
```

4.5 Structure of applications

After the Zedboard is turned on, the processing system powers up and starts executing the initial bootROM code. Once finished, CPU1 will be placed in a WFE (Wait for event) state and executes code at address 0xFFFFFE00. This code basically waits for an event and then checks if address 0xFFFFFFF0 has a non-zero value in a continuous loop. Once some non-zero address is written to 0xFFFFFFF0, CPU1 will jump to the written address and start executing. In our design FSBL will place the executable for CPU1 at address 0x18000000. The application for CPU0(running Linux) should write the value 0x18000000 to location 0xFFFFFFF0. This will cause CPU1 to wakeup, jump to specified address and start executing the application placed at that location by the FSBL. The OCM can be accessed by both the processors and can be used for sharing data between them. The bare-metal application on CPU1 should avoid creation of conflicts with regards to shared resources. The default standalone BSP of CPU1 will enable cache access for OCM whereas Linux will disable the same causing a shared resource conflict. The system level address map in Zynq TRM [11] shows that addresses ranging from 0xFFFC0000-0xFFFFFFFF belongs to OCM. Using above information, we describe how the applications are written for each processor so that the AMP configuration is successful and resulting interaction between the processors.

4.5.1 Application running on CPU0

1. Memory map the OCM address range through the mmap() system call using the file descriptor for /dev/mem. This provides us a way to access the OCM in Linux for writing data to it or reading data from it.

2. Write the data bytes from the input image to OCM by writing to successive addresses from the base address returned by mmap().

3. Write 0 to next immediate address in the OCM after filling in the input image data. This address acts as a flag to know whether CPU1 application is done with the image processing.

4. Write the address 0x18000000 to the location 0xFFFFFFF0 of the OCM. This will cause CPU1 to start executing the bare metal application, which performs the necessary image processing and writes the results back to OCM.

5. Assuming the output image size is same as that of input image, the bare metal application will write 1 to the immediate address after writing the output image. Meanwhile CPU0 will be polling for the value at this address to be true.

6. Once the value is found to be set, the output bytes are read from the OCM and the output image file is created using this data.

4.5.2 Application running on CPU1

1. To avoid shared resource conflict, configure the memory management unit(MMU) by appropriately setting the Translation Lookaside Buffer(TLB) attributes to disable cache access for OCM over its address range.

2. Allocate input and output buffers on the DRAM using malloc().

3. Copy data from the OCM to the input buffer. This is required because DMA transfer will occur from DRAM to MXP scratcpad.

4. DMA data from input buffer in DDR to scratchpad. Perform necessary vector computations for image processing and write results back to scratchpad. DMA output results from scratchpad to output buffer in DDR.

5. Copy results to OCM from the output buffer.

6. Indicate to CPU0 that processing is finished by writing 1 to the next immediate address after the output results are written.

Chapter 5

System level driver for AXI-CDMA

5.1 Introduction

FPGAs consist of one or more general purpose processors combined with the programmable fabric, where the fabric is used to accelerate the compute intensive tasks in the application. An important consideration in such a system is the integration of the accelerator with the processor and how efficiently can the software-hardware communication take place. Alongwith this, ability to abstract the communication while performing data transfers between the processor and the programmable logic using a system-level driver or so is also important. Communication abstraction provides a way to depict the memory-subsystem and the communication interfaces in a logical way by providing some sort of software API. With development of such system level driver, the developer only needs to understand the abstract functions and can use the system without any concern about underlying hardware. In the case of MXP, which we saw in earlier chapter, Vectorblox provides high level C/C++ APIs for using the vector processor for acceleration. This reduces development time. Also as per its programming methodology there were DMA transfers done from DDR to its local scratchpad and reverse once results are processed. Advantage of doing this, is the input data being closer to the processing elements of the accelerator results into lesser computation time as its available in the local memory for processing. Furthermore when the communication(DMA transfer) and computation times are similar, these operations if overlapped can in effect hide the communication latency.

Both Xilinx and Altera embed hard memory blocks(Block random access memory (BRAM)) iniside the fine grained programmable fabric which act as local memory space within PL region and is accessible by both, the PL region and processor. Xilinx provides three soft DMA IP cores [31] to be instantiated on the PL which can be used to enhance transactions between DDR-DDR and DDR-BRAM. These can provide high performance as they make use of the AXI_HP ports available with the Zynq architecture. They are listed as below :

- AXI-CDMA - Central Direct Memory Access which does transactions between memory mapped source and destination regions using AXI4 protocol.
- AXI-DMA - Direct Memory Access which does transactions memory and AXI4-Stream type target peripherals.
- AXI-VDMA - Video Direct Memory Accesss which does transactions between memory and AXI4-Stream type video target peripherals.

Our focus in this chapter will be developing a driver for making use of the Xilinx AXI-CDMA engine [32] device driver to do DMA transfers.

5.2 Hardware design

The Central Direct Memory Access (CDMA) engine is configured with 32-bit data width and burst length of 256 for the DMA transfers. The hardware design is created using Vivado, the block design of which is shown in Figure 5.1. The transfer path of the main signals is highlighted in red. The master port M_AXI on CDMA engine is connected to slave interface S_AXI of the BRAM controller and slave high performance port S_AXI_HP0 of the Zynq processing system. This S_AXI_HP0 port will be used to access the DDR controller and is used for the data transfers from DDR-DDR or DDR-BRAM or BRAM-DDR.

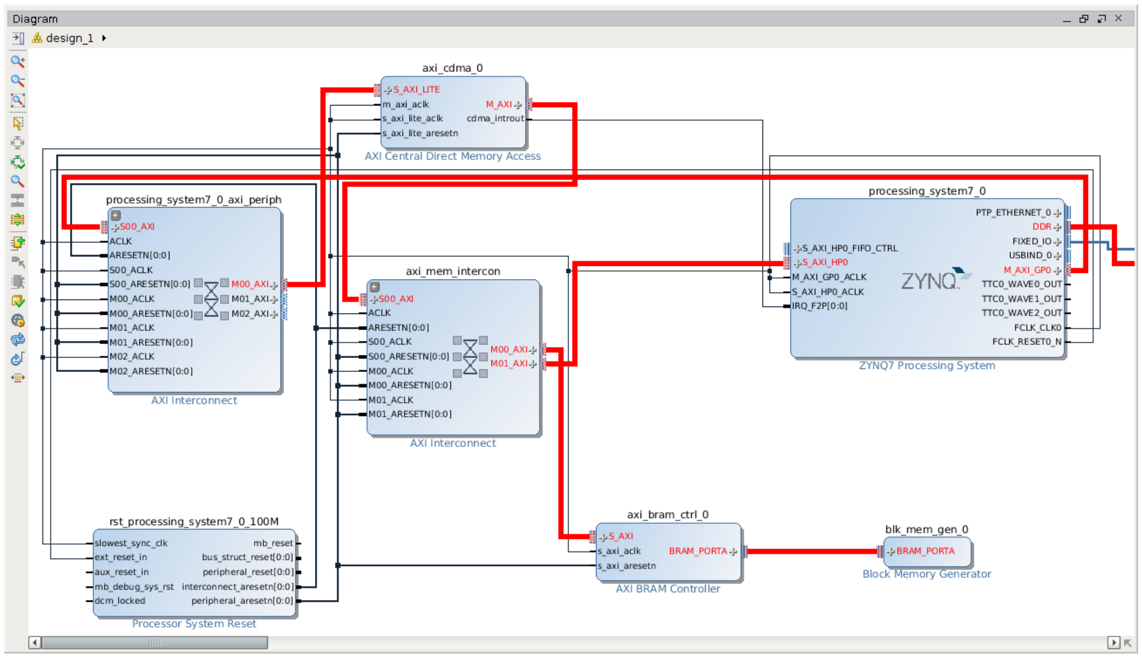


Figure 5.1: Vivado block design for PL-CDMA

The BRAM controller is required so that the AXI signals are converted to a form that can be used for accessing the BRAM. Also the master general purpose port M_AXI_GP0 on the PS is connected to slave interface S_AXI_LITE on the CDMA engine. This connection is used by the PS to program the CDMA engine registers with the appropriate parameters such as source address, destination address, number of bytes to be transferred, etc. The cdma_introut signal is interrupt output from the CDMA engine which is connected to IRQ_F2P on the Zynq processing system. This output is generated to indicate to the PS that the CDMA engine has finished a DMA transaction. After creating the design we can generate the hardware design file and the bitstream. The hardware design file can then be used for getting the device tree entry for the CDMA engine being instantiated through the bitstream.

5.3 Software design

For the CDMA engine instantiated on the PL, its device driver will read the device tree entry and set it up with the appropriate configuration parameters used for the hardware design. A sample device tree node to be added in the dts file for the same is shown in Figure 5.2

```
dma@7e200000 {
                #dma-cells = <0x1>;
                compatible = "xlnx,axi-cdma";
                interrupt-parent = <0x2>;
                interrupts = <0x0 0x1d 0x4>;
                reg = <0x7e200000 0x10000>;

                dma-channel@7e200000 {
                    compatible = "xlnx,axi-cdma-channel";
                    interrupts = <0x0 0x1d 0x4>;
                    xlnx,datawidth = <0x20>;
                    xlnx,device-id = <0x0>;
                    xlnx,max-burst-len = <0x100>;
                };
            };
```

Figure 5.2: DTS entry for CDMA

For a character driver, there can be many file operations that can be used for providing abstraction to the application by APIs(system calls). These file operations mapped to corresponding functions in our driver are as shown in Figure 5.3

```
  struct file_operations memory_fops = {
    .open = memory_open,
    .read = memory_read,
    .write = memory_write,
    .release = memory_release,
};
```

Figure 5.3: File operations structure for driver

In case of DDR-PL transactions memory_write performs transfer of data from DDR to BRAM, and memory_read will perform transfers from BRAM to DDR. In Linux, DMA is designed to be used by a higher level device driver in the kernel space.

There is a framework in Linux that allows access to DMA controller drivers(in our case AXI CDMA) in an abstract manner known as the DMA engine. Xilinx provides device drivers for the AXI CDMA, AXI DMA and AXI VDMA engines that plug into this DMA engine framework. We create such higher level device driver by making use of the DMA APIs provided by this framework. The software detailed design for the driver is shown in Figure 5.4

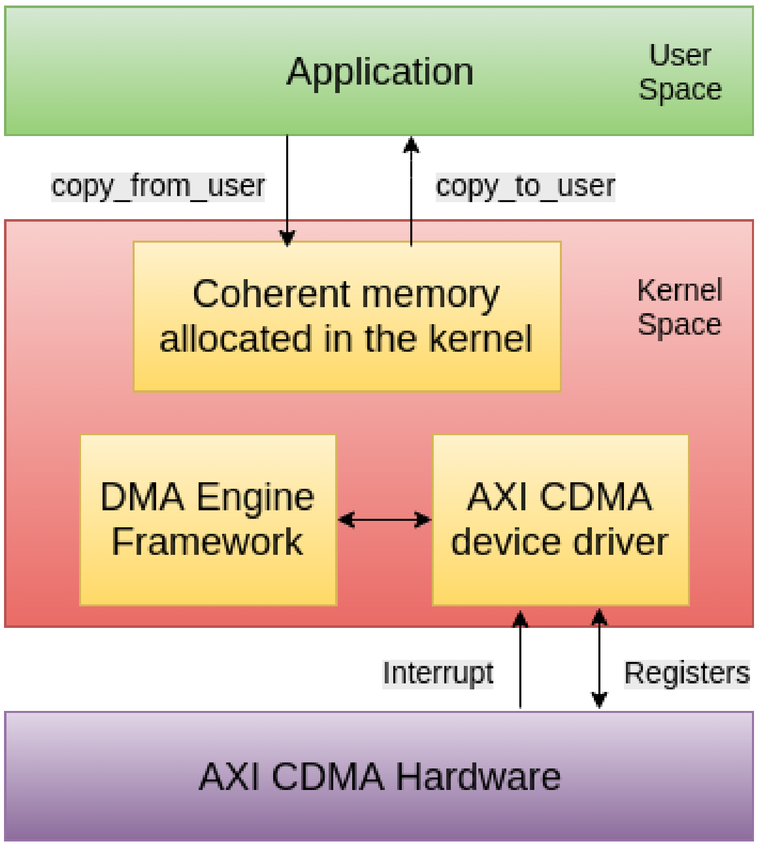


Figure 5.4: CDMA Software Detailed Design

5.4 Programming Flow

We perform a loopback in case of DDR-PL transfers i.e we transfer data from DDR-BRAM and again transfer the same data from BRAM-DDR to verify functionality of our driver. In this section we describe the procedure used in the different file operations functions of the driver.

5.4.1 Opening the device file

1. Request for a channel using dma_request_channel(). As there can be multiple PL-DMA drivers in use in the kernel depending upon what is instantiated on the fabric or even PS-DMA hard IP driver may be currently in use. To get channel specifically from the CDMA engine driver we need to provide the correct match value as shown :

```
direction = DMA_MEM_TO_MEM;

match = (direction & 0xFF) | XILINX_DMA_IP_CDMA | (device_id <<
    XILINX_DMA_DEVICE_ID_SHIFT);
```

2. Allocate source and destination buffers on the DDR using dma_alloc_coherent(). This function allocates cache coherent memory and returns the virtual and physical addresses of the allocated buffers.

5.4.2 Writing to the device file

1. Copy data from the user space into the allocated source buffer.

2. Call device_prep_dma_memcpy() providing source address as the physical address of the source buffer allocated in previous step. The destination address will be the starting address of the BRAM in case of DDR-PL transfers or physical address of the destination buffer allocated in previous step in case of DDR-DDR transfers. Also provide the number of bytes to be sent alongwith the channel to this function. This will return a channel descriptor.

3. Register a callback with the channel descriptor so that when DMA tranfer finishes an interrupt is received from the CDMA engine and the callback will get called.

4. Submit the channnel descriptor using dma_engine_submit(). This will return a cookie used for checking the transfer status.

5. Initiate transfer using dma_async_issue_pending(). Wait for completion of the DMA providing a timeout value to function wait_for_completion_timeout() so that it exits in case the interrupt is not received and timeout occurs. If the callback registered with the channel descriptor gets called, it means interrupt was received.

6. Get the status of the transfer done using dma_async_is_tx_complete() by providing it the cookie. The status should be returned as DMA_SUCCESS so as to verify that the operation was done successfully.

5.4.3 Reading from the device file

1. In case of DDR-PL transfers perform steps 2 to 6, similar to writing operation with the caveat that in step 2 the source address will be the starting address of BRAM and destination address is the physical address of the destination buffer allocated on DDR. For DDR-DDR no need to do these steps as the transfer was already done in the writing operation.

2. Copy data from the allocated destination buffer to user space.

5.4.4 Closing the device file

1. Deallocate the source and destination buffers of DDR using dma_free_coherent().

2. Terminate all the DMA operations that are making use of the channel by calling dma_engine_terminate_all().

3. Release the acquired channel using dma_release_channel().

5.5 Results

The profiling in user space was performed using clock_gettime() function while providing parameter as CLOCK_MONOTONIC_RAW which uses hardware-based time and is not subject to NTP(Network Time Protocol) adjustments [33]. Similarly, for profiling in kernel space,the kernel mode function for clock_gettime() i.e getrawmonotonic() was used. The timings are measured by averaging over 1024 transfers. DMA transfer latency indicates time measured only for the transfer function dma_async_issue_pending() which actually initiates the transfer and corresponding wait for completion. The kernel space latency indicates total time taken in the kernel space. The user space latency indicates time taken for the transfer as seen by the application which will include kernel space latency, system call overheads and call to profiling functions in the kernel space. Bandwidth is measured with respect to the DMA transfer latency. All the timings measured are in microseconds. In all cases, we observe bandwidth increases as number of samples tranferred becomes higher. Typically we observe that the overhead from user space with respect to kernel space is $\approx$ 2 microseconds.

5.5.1 DDR-DDR Communication

Table 5.1: Latency and bandwidth for DDR-DDR transfers

Number of Samples	DMA transfer latency	Kernel space latency	User space latency	Bandwidth in MB/s
32	13.338	16.092	18.45	2.29
64	13.374	16.488	18.702	4.56
128	16.128	18.396	20.718	7.57
256	20.628	23.112	25.722	11.84
512	19.944	23.184	26.082	24.48
1K	19.926	23.778	26.352	49.01
2K	22.626	28.134	30.906	86.32
4K	27.648	33.714	35.91	141.29
8K	38.322	46.008	48.60	203.86

5.5.2 DDR-PL Communication

We observe that transfers from DDR to BRAM are slower than that from BRAM to DDR indicating that writing to BRAM is typically slower than writing to DDR.

Table 5.2: Latency and bandwidth for DDR-BRAM transfers

Number of Samples	DMA transfer latency	Kernel space latency	User space latency	Bandwidth in MB/s
32	13.176	16.29	18.45	2.32
64	13.338	16.11	18.594	4.58
128	16.686	18.558	20.916	7.32
256	20.34	23.31	25.884	12.00
512	19.926	23.436	25.902	24.50
1K	20.772	21.978	25.992	47.01
2K	25.776	29.268	31.878	75.77
4K	33.192	38.142	40.914	117.69
8K	48.33	56.106	58.95	161.65

Table 5.3: Latency and bandwidth for BRAM-DDR transfers

Number of Samples	DMA transfer latency	Kernel space latency	User space latency	Bandwidth in MB/s
32	12.942	16.074	18.18	2.36
64	12.834	16.488	18.414	4.76
128	13.23	16.704	18.792	9.23
256	19.728	24.03	26.37	12.38
512	19.764	25.236	27.396	24.71
1K	20.034	27.09	29.358	48.75
2K	21.834	35.46	36.72	89.45
4K	27.504	47.502	49.914	142.02
8K	38.052	75.096	77.598	205.31

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This report focussed on the use of FPGA overlays as accelerators and providing OS support on them for abstracting the hardware details to the end user. We described in detail the necessary steps for configuring and booting Linux and accessing the MXP overlay through it. Polybench kernels, ATAX and BiCG were accelerated using MXP. For ATAX we were able to get $\approx 5\times$ speedup and for small dataset size on BiCG kernel $6.7\times$ speedup. For standard dataset size in BiCG we got a speedup of ≈ 3. Further standard benchmarks provided by MXP were run on Linux as well as in bare-metal mode. The overheads due to use of OS on the MXP overlay were calculated for these benchmarks.

In the quest for avoiding the overheads due to Linux on MXP, while still using the filesystem on an OS, we developed an AMP system, wherein Linux was booted on one CPU core and bare-metal application using MXP APIs was run on another. We described a way to structure the applications on each CPU for using the developed system and corresponding interaction between the cores.

We also developed a system-level driver on top of CDMA engine driver with the goal of abstracting the communication interface between the PS and PL. Using the driver we were able to do DMA transfers from DDR-DDR and DDR-PL by just writing or reading to the device file for the driver. The bandwidths for these transfers were measured. For 8K samples, we were able to achieve a bandwidth of 203 MB/s for writes to DDR. For writes to BRAM, we observed the bandwidth to be around 162 MB/s.

6.2 Future work

In this section we describe the future directions for the work described in Chapters 3, 4 and 5 .

6.2.1 For MXP

More kernels from polybench suite should be accelerated using MXP and the speedup obtained can be compared with other kinds of accelerator such as GPU. This will help to make an informed decision about the platform to use based on the performance requirements of the application. Also power measurement for these accelerators should be done to gain information about the energy consumption on each of these platforms.

6.2.2 For AMP

- The HP ports on Zynq have direct access to OCM alongwith DDR. So while using AMP for MXP, instead of copying data from OCM to DDR and back further exploration should be done to see if the DMA transfers can occur directly from OCM to scratchpad.
- The ability of changing the bare metal executable during run-time can be added to the existing setup. This will particularly require resetting the CPU1 and replacing with new executable before starting it again.
- In the current system, the two processors signal each other through polling for a particular memory location. This polling mechanism should be replaced with interrupt based signalling and checked for better performance.

6.2.3 For CDMA driver

- Instead of copying data between user space and kernel space buffers, the allocated coherent memory in the kernel space must be mapped to the user space by adding memory mapping call to the driver file operations. Furthermore, it is also possible to directly program the CDMA engine through user space without any requirement for kernel drivers. Bandwidths obtained with these approaches should be measured and checked if its better than current implementation.
- The hardware design and the driver should be modified further to add support for more channels thereby utilizing all the available HP ports for doing transfers in parallel which will further improve the bandwidth for the DMA transfers. This will require that the driver code be re-entrant to avoid conflicts and the number of channels should be scaled accordingly.
- In the current design, with the loopback, only the communication part for an accelerator is done. For accelerating computations, processing elements(PEs) should be attached to the BRAM so that once DMA transfer is done from the DDR to BRAM, processing can be done by the coprocessor on the data before transferring the results back to DDR.

Bibliography

[1] ARM Ltd. The ARM Cortex-A9 Processors. `http://www.arm.com/files/pdf/ARMCortexA-9Processors.pdf`.

[2] `http://olaf.eecs.berkeley.edu`.

[3] `http://www.eejournal.com/archives/articles/20130305-fpgawars/`.

[4] J. Coole and G. Stitt. Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 13–22, October 2010.

[5] A. K. Jain, K. D. Pham, J. Cui, S. A. Fahmy, and D. L. Maskell. Virtualized execution and management of hardware tasks on a hybrid ARM-FPGA platform. *Journal of Signal Processing Systems*, 77(1–2):61–76, Oct. 2014.

[6] Jesse Benson, Ryan Cofell, Chris Frericks, Chen-Han Ho, Venkatraman Govindaraju, Tony Nowatzki, and Karthikeyan Sankaralingam. Design, integration and implementation of the dyser hardware accelerator into opensparc. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, 2012.

[7] Neil W. Bergmann, Sunil K. Shukla, and Jrgen Becker. QUKU: a dual-layer reconfigurable architecture. *ACM Transactions on Embedded Computing Systems (TECS)*, 12:63:1–63:26, March 2013.

[8] D. Capalija and T.S. Abdelrahman. Towards synthesis-free JIT compilation to commodity FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 202–205, 2011.

[9] Davor Capalija and Tarek S. Abdelrahman. A high-performance overlay architecture for pipelined execution of data flow graphs. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, 2013.

[10] A. Severance and G. G. F. Lemieux. Embedded supercomputing in fpgas with the vectorblox mxp matrix processor. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2013 International Conference on*, pages 1–10, 2013.

[11] Xilinx Ltd. Zynq-7000 technical reference manual. `http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf`, 2013.

[12] Gordon Brebner. A virtual hardware operating system for the Xilinx XC6200. In *Field-Programmable Logic Smart Applications, New Paradigms and Compilers*, pages 327–336. 1996.

[13] C. Steiger, H. Walder, and M. Platzner. Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks. *IEEE Transactions on Computers*, 53(11):1393–1407, November 2004.

[14] M. Vuletic, L. Righetti, L. Pozzi, and P. Ienne. Operating system support for interface virtualisation of reconfigurable coprocessors. In *Proceedings of the Design, Automation and Test in Europe (DATE)*, pages 748–749, 2004.

[15] K. Rupnow. Operating system management of reconfigurable hardware computing systems. In *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, pages 477–478, 2009.

[16] Ivan Gonzalez and Sergio Lopez-Buedo. Virtualization of reconfigurable coprocessors in HPRC systems with multicore architecture. *Journal of Systems Architecture*, 58(6–7):247–256, June 2012.

[17] Enno Lübbers and Marco Platzner. ReconOS: multithreaded programming for reconfigurable computers. *ACM Transactions on Embedded Computing Systems (TECS)*, 9(1):8, October 2009.

[18] D. Gohringer, S. Werner, M. Hubner, and J. Becker. RAMPSoCVM: runtime support and hardware virtualization for a runtime adaptive MPSoC. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2011.

[19] H.K.-H. So, A. Tkachenko, and R. Brodersen. A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 259–264, 2006.

[20] K. Kosciuszkiewicz, F. Morgan, and K. Kepa. Run-time management of reconfigurable hardware tasks using embedded linux. In *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, pages 209–215, 2007.

[21] Aws Ismail. *Operating system abstractions of hardware accelerators on field-programmable gate arrays*. Thesis, August 2011.

[22] M. Jacobsen, Y. Freund, and R. Kastner. RIFFA: a reusable integration framework for FPGA accelerators. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 216–219, May 2012.

[23] M. Jacobsen and R. Kastner. RIFFA 2.0: A reusable integration framework for FPGA accelerators. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, September 2013.

[24] K. Eguro. SIRC: an extensible reconfigurable computing communication API. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 135–138, 2010.

[25] Xillybus Ltd. Xillybus: IP Core Product Brief. `http://xillybus.com/downloads/xillybus_product_brief.pdf`.

[26] http://vectorblox.github.io/mxp/mxp_quickstart_vivado.html.

[27] http://www.wiki.xilinx.com/Linux.

[28] http://vectorblox.github.io/mxp/mxp_reference.html.

[29] http://web.cse.ohio-state.edu/~pouchet/software/polybench/.

[30] http://www.xilinx.com/support/documentation/application_notes/xapp1078-amp-linux-bare-metal.pdf.

[31] http://www.wiki.xilinx.com/DMA+Drivers+-+Soft+IPs.

[32] http://www.xilinx.com/support/documentation/ip_documentation/axi_cdma/v3_03_a/pg034_axi_cdma.pdf.

[33] https://www.cs.rutgers.edu/~pxk/416/notes/c-tutorials/gettime.html.