TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Kareen Arutjunjan IAAB206357

A PARALLEL CI/CD FRAMEWORK FOR BUILDING AND TESTING MULTIPLE BOARDS WITH SYSTEM ON CHIP

Bachelor's Thesis

Supervisor: Petr Žejdl
PhD

Co-supervisor: Siim Vene
MsC

Tallinn 2024

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Kareen Arutjunjan IAAB206357

PARALLEELNE CI/CD RAAMISTIK ERINEVATE KIIPSÜSTEEM-PLAATIDE ARENDAMISEKS JA TESTIMISEKS

Bakalaureusetöö

Juhendaja: Petr Žejdl
PhD

Kaasjuhendaja: Siim Vene
MsC

Tallinn 2024

Author's Declaration of Originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Kareen Arutjunjan

14.05.2024

Acknowledgements

During my journey towards this thesis, I've been fortunate to encounter a remarkable group of individuals who provided invaluable support and guidance.

I couldn't have asked for better supervisors than Dr. Petr Žejdl and Dr. Marc Dobson, my exceptional leaders at CERN's CMS DAQ group. Their profound understanding of the CMS experiment and the Data Acquisition System laid the groundwork for grasping the project's true significance. Their knowledge and enthusiasm were truly contagious, and their support, which extended far beyond workdays, kept me motivated throughout the entire process. Their constructive criticism and enthusiastic engagement during our meetings significantly propelled my progress.

Big thanks to Philipp Brummer, Dominique Gigi, Dr. Dinyar Rabady, and the entire CMS DAQ team, for fostering a welcoming and supportive environment. Their kindness and willingness to help made my experience at CERN truly unforgettable.

I would like to thank my university supervisor Siim Vene for the trust, guidance, and freedom that he gave me during the course of my thesis.

Last but not least, to the incredible friends I made at CERN! The countless discussions we shared about science and various other topics helped me grow as a well-rounded individual. Their unwavering support throughout this journey was a constant source of strength, and I'm incredibly grateful to have them in my life.

Kareen Arutjunjan

14.05.2024

Abstract

CERN is working on the High-Luminosity LHC upgrade which is scheduled for installation in 2029 [1]. As part of this upgrade, the CMS experiment and its Data Acquisition (DAQ) system will undergo enhancements. Specifically, the CMS DAQ system will use System-on-Chip (SoC) from Xilinx on the new electronic boards. This SoC will run board control and monitoring software on a Linux Operating System (OS). This SoC will run control and monitoring software on a Linux Operating System (OS).

The SoC from Xilinx contains programmable logic, and processing system, which is built from various sources using a complex software stack. The problem is that any change in any of these sources implies a full rebuild of the whole system, and subsequently, thorough testing. In addition, this has to be done for every board type used by the developer, which further increases his workload. The problem is significant due to the involvement of multiple board types, developers, and diverse sources of firmware and software updates.

This thesis extends on previous work and experience done in CMS DAQ and proposes an automated and parallelized system for building and testing firmware and software specifically for the diverse SoC board types used within the CMS DAQ system. By leveraging a Continuous Integration and Continuous Delivery (CI/CD) pipeline, this approach aims to:

- **Reduce manual intervention:** Streamline the build and test process, minimizing manual effort and accelerating development.
- **Improve build reliability:** Integrate automated testing and fault detection mechanisms, ensuring a more robust process compared to manual methods.
- **Enhance developer experience:** Eliminate the need to recall and execute build steps, simplifying the development process.

This CI/CD based approach offers a significant improvement for the HL-LHC project by streamlining development and ensuring robust firmware and software builds for the CMS DAQ System's SoC boards.

The thesis is written in English and contains 57 pages of text, 6 chapters, 21 figures and 1 table.

Annotatsioon

CERN tegeleb kõrge valgustugevusega Suure Hadronite Põrguti (HL-LHC) uuendustööde kallal, mis on kavas ellu viia aastal 2029 [1]. Uuenduse käigus täiustatakse Kompaktsete Müüonite Solenoidi (CMS) eksperimenti ja selle andmekogumissüsteemi (DAQ). Täpsemalt kasutab CMS DAQ süsteem Xilinxi süsteemikiipi (SoC) uutel trükkplaatidel. See süsteemikiip kasutab GNU/Linuxi operatsioonisüsteemi juhtimis- ja seiretarkvara haldamiseks.

Xilinxi süsteemikiip sisaldab programmeeritavat loogikat ja töötlemissüsteemi, mis on loodud erinevatest allikatest, kasutades keerukat tarkvarapaketti. Probleem on selles, et mis tahes muutus nendes allikates eeldab kogu süsteemi täielikku ümberehitamist ja seejärel põhjalikku testimist. Lisaks tuleb seda teha iga arendaja poolt kasutatava trükkplaadi tüübi puhul, mis suurendab tema töökoormust veelgi. Probleem on märkimisväärne, kuna kaasatud on mitmeid trükkplaadi tüüpe, arendajaid ja erinevaid püsivara ja tarkvaravärskenduste allikaid.

Käesolev lõputöö laiendab varasemat CMS DAQ's tehtud tööd ja kogemust ning pakub välja automatiseeritud ja paralleelse raamistiku püsivara ja tarkvara ehitamiseks ja testimiseks spetsiaalselt CMS DAQ süsteemis kasutatavatele erinevatele trükkplaadi süsteemikiipide tüüpidele. Pidevlõimimise ja -valmiduse (CI/CD) töövoo abil püüab see lähenemisviis saavutada järgmisi eesmärke:

- **Käsitsi sekkumise vähendamine:** arenduse ja testimise protsesside automatiseerimine kiirendab arendustegevust ja vähendab käsitsi tehtavate sammude vajadust.
- **Töökindluse suurendamine:** automatiseeritud testimise ja vea tuvastamise mehhanismid tagavad töökindlamad protsessid võrreldes käsitsi meetodiga.
- **Arendajakogemuse parandamine:** kõrvaldades vajaduse käsitsi tehtavateks sammudeks ning lihtsustades arendusprotsessi.

See CI/CD-põhine raamistik pakub HL-LHC projektile märkimisväärset täiustust, lihtsustades arendustegevust ning tagades töökindla arenduse meetodi CMS DAQ süsteemikiipide trükkplaatide püsivara ja tarkvara koostamiseks.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 57 leheküljel, 6 peatükki, 21

joonist, 1 tabeli.

List of Abbreviations and Terms

ALICE	A Large Ion Collider Experiment
API	Application Programming Interface
APU	Application Processing Unit
ATCA	Advanced Telecommunications Computing Architecture
ATF	ARM Trusted Firmware
ATLAS	A Toroidal LHC Apparatus
AXI	Advanced eXtensible Interface
BSROOTFS	Board Specific root filesystem
CD	Continuous Delivery/Deployment
CERN	Conseil Européen pour la Recherche Nucléaire
CI	Continuous Integration
CI/CD	Continuous Integration & Continuous Delivery/Deployment
CLI	Command-Line Interface
CMS	Compact Muon Solenoid
DAQ	Data Acquisition
DHCP	Dynamic Host Configuration Protocol
DNF	Dandified YUM
ECAL	Electromagnetic Calorimeter
EEPROM	Electrically Erasable Programmable ROM
FPGA	Field-Programmable Gate Array
FSBL	First-Stage Bootloader
GPU	Graphics Processing Unit
HCAL	Hadron Calorimeter
HDL	Hardware Description Language
HL-LHC	High-Luminosity LHC
IPMC	Intelligent Platform Management Controller
JTAG	Joint Test Action Group
LHC	Large Hadron Collider
LHCb	LHC beauty
Linac4	Linear accelerator 4
MAC	Media Access Control
MPSoC	Multi-Processor System-on-Chip
NFS	Network File System

OS	Operating System
PCI	Peripheral Component Interconnect
PDU	Power Distribution Unit
PL	Programmable Logic
QEMU	Quick Emulator
RAM	Random Access Memory
RHEL	Red Hat Enterprise Linux
ROM	Read Only Memory
ROOTFS	Root Filesystem
RTM	Rear Transition Module
RU	Readout Unit
SATA	Serial Advanced Technology Attachment
SoC	System-on-Chip
SSBL	Second-Stage Bootloader
SSH	Secure Shell
TFTP	Trivial File Transfer Protocol
U-Boot	Universal Bootloader
USB	Universal Serial Bus
VM	Virtual Machine
XSA	Xilinx Shell Architecture
YAML	Human-readable data serialization language

Table of Contents

List of Figures

List of Tables

1. Introduction

This section provides a concise summary of the thesis by presenting an outline of the experiment and the historical background of the CMS Data Acquisition (DAQ) group. The research conducted in this thesis is an integral part of the CMS experiment's DAQ group.

1.1 Introduction to CERN

The European Organization for Nuclear Research, known as CERN, stands as a preeminent institution in the realm of particle physics. Established in 1954 [2], its primary objective is to explore the fundamental constituents of matter and understand the forces governing the universe's behavior at the smallest scales. Situated on the Franco-Swiss border near Geneva, CERN boasts a collaborative environment that brings together scientists, engineers, and researchers from around the globe [3]. At the heart of CERN's scientific endeavors is the Large Hadron Collider (LHC), the world's most potent particle accelerator [4]. The LHC enables groundbreaking experiments, including the discovery of the Higgs boson in 2012, a pivotal milestone in our comprehension of particle physics [5]. As we delve into the intricate fabric of the cosmos, CERN's contributions remain pivotal in advancing human understanding of the fundamental building blocks of our universe.

Figure 1. CERN Science Gateway [6]

1.2 The LHC at CERN

The Large Hadron Collider (LHC) at the European Organization for Nuclear Research (CERN) is a circular accelerator with a length of 27 kilometers. Commissioned in 2008, its purpose is to accelerate protons with nearly the speed of light and facilitate collisions. These collisions generate conditions similar to those in the early moments of the universe, allowing scientists to explore the fundamental particles and forces that govern our cosmos. The four main detectors – ATLAS, CMS, ALICE, and LHCb – capture and analyze the outcomes of these collisions, providing invaluable data for breakthrough discoveries. Notably, the LHC played a pivotal role in the confirmation of the Higgs boson's existence in 2012, a momentous achievement that underscored the collider's significance in advancing our understanding of particle physics. As the LHC continues to push the boundaries of scientific exploration, its contributions to unraveling the mysteries of the universe remain unparalleled. [4, 7]

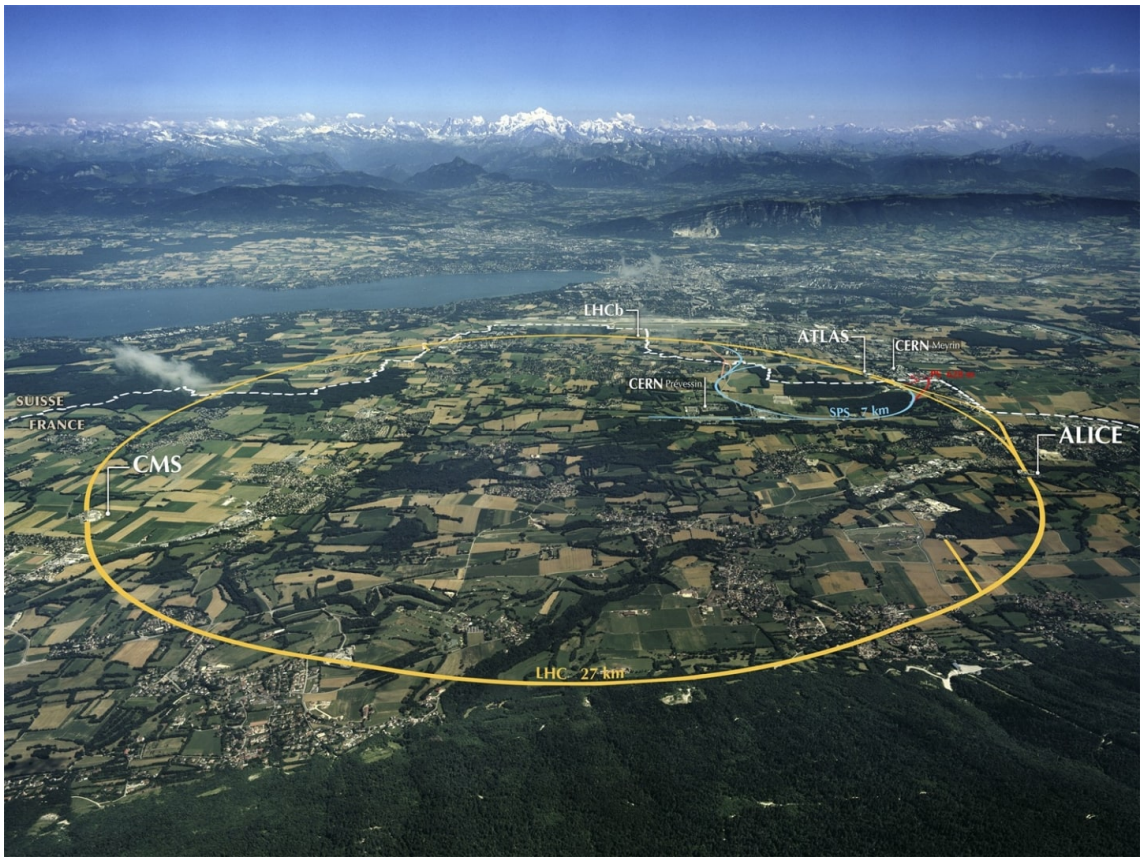

Figure 2. CERN, Large Hadron Collider [4]

1.2.1 The CMS sub-detectors

The CMS detector is composed of several sub-detectors: The silicon tracker, the electromagnetic calorimeter (ECAL), the hadron calorimeter (HCAL), and the muon chambers.

The silicon tracker, located at the innermost part of the detector, is responsible for reconstructing the paths of charged particles produced from collisions. This reconstruction allows for the measurement of particle momentum. The tracker is capable of reconstructing tracks from high-energy muons, electrons, hadrons, and tracks from the decay of short-lived particles [8].

The two calorimeters are designed to halt particles and measure the energy they release. The electromagnetic calorimeter (ECAL) specifically measures the energy of electrons and photons. It utilizes dense, highly transparent crystals that stop the particles. When electrons and photons pass through these crystals, they scintillate. The amount of light produced is directly proportional to the energy of the particle. Photo-detectors are attached to the crystals to measure the intensity of the light [9].

On the other hand, the hadron calorimeter (HCAL) measures the energy, positions, and arrival times of hadrons. It consists of alternating layers of absorbers (brass or steel) and scintillators. When a hadronic particle collides with an absorber layer, it is stopped and triggers an interaction that generates secondary particles. These secondary particles can then interact with subsequent absorber layers, leading to the creation of more particles and the formation of a particle shower. As the shower progresses, the particles pass through multiple scintillation layers, which are used to measure their energy, similar to the ECAL [10].

Muons and neutrinos are the sole particles that can pass through the calorimeters without being halted. Detecting neutrinos is particularly difficult due to their minimal interaction with matter. On the other hand, muons are monitored by the muon chambers situated outside the solenoid coil. To determine the trajectory of muons, a curve is fitted to the "hits" observed in the four muon stations (MS). Each station consists of multiple layers of gaseous ionization chambers that measure the particles' track and energy [10].

1.2.2 The CMS DAQ system

The DAQ system utilizes custom-designed electronics for both the front-end and back-end data acquisition. Front-end electronics reside close to the detectors and are responsible for signal amplification, shaping, and conversion into digital data. Back-end electronics aggregate, format, and transmit the data from the front-end to the DAQ system for further processing [11].

Data from the front-end electronics is transferred over high-bandwidth optical links to the DAQ and Timing Hub (DTH). The DAQ plays a crucial role in aggregating data from

various parts of the detector and distributing it to subsequent stages in the DAQ system [11].

A high-speed switching network interconnects different parts of the DAQ system and facilitates event building. Event building refers to the process of assembling complete event information from data fragments originating from various sub-detectors into a single data structure for further processing and storage [11].

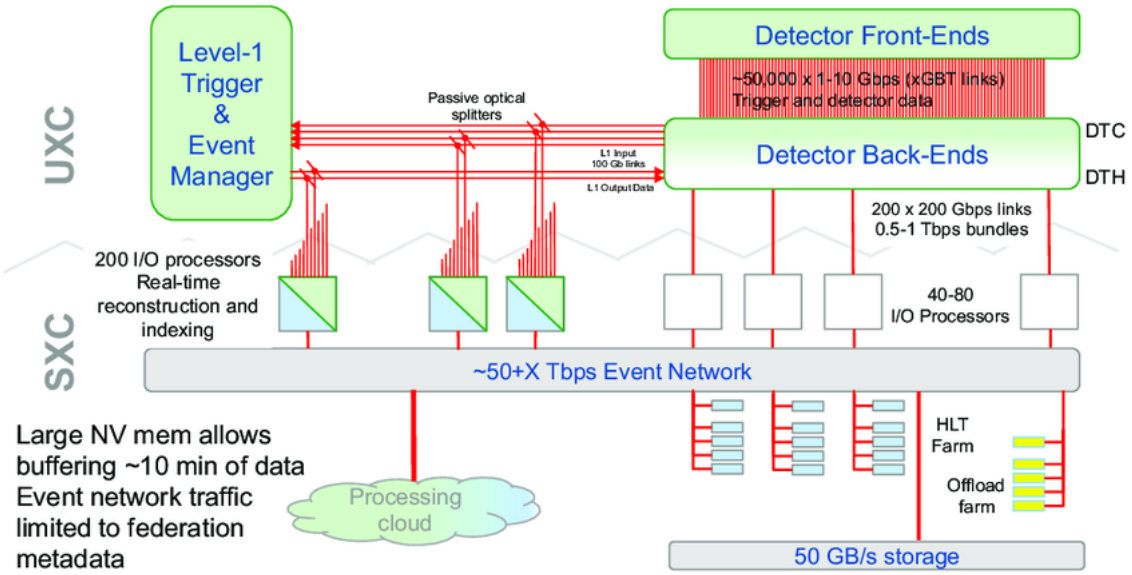

Figure 3. A diagram of the phase-2 CMS DAQ including the 40 MHz scouting system [11]

2. Project Description

2.1 Background

The current data acquisition system relies on a multitude of custom hardware components and Field-Programmable Gate Arrays (FPGAs). These elements are managed through bulky rack-mounted server PCs, which connect to the hardware via PCI bridges. While this architecture served effectively for the initial stages of LHC operation, it faces limitations in handling the exponential data growth anticipated with HL-LHC.

To address this challenge, a novel approach is being explored, integrating real-time processing capabilities directly onto the detector front-end electronics using System on Chip (SoC). The solution lies in the integration of embedded systems directly onto the hardware boards. These embedded systems will run control and monitoring tasks currently handled by the rack-mounted PCs. This eliminates the need for additional server racks and their associated cabling, resulting in a significantly more compact and efficient design.

2.1.1 The problem

The SoC from Xilinx contains programmable logic, and processing system, which is built from various sources using a complex software stack. The problem is that any change in any of these sources implies a full rebuild of the whole system, and subsequently, thorough testing. In addition, this has to be done for every board type used by the developer, which further increases their workload.

2.1.2 The solution

The aim of this thesis is to develop an automated and parallelized system designed to streamline the construction and testing of firmware and software across all System on Chip (SoC) board types utilized by developers. By minimizing manual interventions in the build and test processes, the objective is to accelerate the overall development cycle.

2.1.3 Requirements

Automation of Build Processes

Develop a system that automates the building, deployment, and testing of firmware images across various System on Chip (SoC) board types, reducing the need for manual intervention and accelerating the develop-ment cycle.

Enhanced Reliability

Integration of automated testing and fault detection mechanisms to ensure a more robust build process compared to manual methods, reducing the likelihood of errors and bugs being introduced.

Improved Developer Workflow

Eliminating the need for manual execution of build steps to simplify the development process, allowing developers to focus on higher-value tasks.

2.2 Objective

To address this objective, a parallelized automated build system will be implemented, employing a Continuous Integration & Continuous Deployment (CI/CD) pipeline. This system will utilize scripts to orchestrate various tools, manage communication with boards and the environment, retrieve results, and facilitate reporting at different stages of the development process.

This approach is deemed optimal for tackling the identified challenges, as CI/CD methodologies simplify the build process and eliminate the need to manually recall every step required to reproduce the final software. Furthermore, the integration of fault detection mechanisms and comprehensive tests enhances the reliability of building firmware images and operating systems for embedded boards, surpassing the reliability achievable through manual methods.

2.2.1 Deliverables

The tangible contribution of this thesis lies in the development of a practical, development-ready build system, which is automated and parallelised for building and testing the

firmware and software for System on Chip board types used in the CMS DAQ.

The final products of this thesis are:

1. A Parallel CI/CD Framework for Building Multiboard Systems on Chip
2. A Parallel CI/CD Framework for Testing Multiboard Systems on Chip
3. Container images for providing the Critical Infrastructure for the build system:
   (a) DNSMasq Server - for DNS (Domain Name System) and DHCP (Dynamic Host Configuration Protocol);
   (b) TFTP Server - for TFTP (Trivial File Transfer Protocol) service;
   (c) NFS Server - for NFS (Network File System) service;
   (d) Chrony Server - for NTP (Network Time Protocol) service;
4. Documentation on the whole build framework.

3. Research and Analysis

The CMS DAQ team is actively working on developing the SoC, which will align best for the upcoming High-Luminosity LHC upgrade. Today, there's a wide variety of embedded board types, which need to be supported at all times. Because of that in recent years, there's been a development of the common framework that would automize the process of building and testing the firmware images for embedded boards. The goal is to save time of the developers, so they can focus on developing the hardware, rather than working on the operational tasks.

3.1 Problems with the previous build framework

While the latest build framework, supearheaded by Vasileios Amoiridis [12], offers valuable functionality, it has limitations that need to be addressed:

- **Limited Build Capabilities:** The system is restricted to building firmware for a single board at a time, hindering efficiency for large-scale deployments.
- **Network Service Dependency:** The framework assumes pre-configured network services, introducing an additional complexity for the end-users.
- **IPMC support:** The framework uses Power Distribution Unit (PDU) for power cycling the boards, however the support for Intelligent Platform Management Interface (IPMC) is crucial for development purposes, as it mirrors the way the boards will be managed in the final deployment.
- **Linux Filesystem**:
  - **Redundant Filesystem Creation:** The pipeline re-creates the entire root filesystem for each firmware build, leading to unnecessary redundancy and prolonged build times. A pre-built and optimized root filesystem could be stored and reused, significantly improving efficiency.
  - **CentOS End-of-Life**: The framework relies on CentOS as the Operating System for embedded boards, which reaches its end-of-life in June 2024 [13].
- **Petalinux Integration:**
  - **Older Version:** The framework currently utilizes Petalinux 2021.2, falling behind the latest available version (2023.2). This limits access to potential improvements and bug fixes.
  - **Inefficient Project Usage:** The framework currently employs the entire Petalinux project for each board type, which makes incorporating changes complex,

as modifications within a single project wouldn't be readily transferable to others. Implementing a more modular structure would allow for code sharing and streamline the development process.

- **Lacking Support for DHCP-Client-Id:** The framework doesn't yet offer default support for dhcp-clientid. This functionality plays a vital role in assigning unique network addresses to devices via DHCP (Dynamic Host Configuration Protocol). Integrating dhcp-clientid by default would enhance the framework's versatility and streamline network setup for various embedded boards. [14]

3.2 Build framework

3.2.1 System-on-Chip (SoC)

A System-on-Chip (SoC) is a highly integrated circuit (IC) that combines multiple electronic components traditionally found on separate boards into a single package. These components typically include a central processing unit (CPU), memory interfaces, input/output (I/O) devices and controllers, and secondary storage interfaces. Additionally, modern SoCs often integrate specialized processing units like graphics processing units (GPUs), image signal processors (ISPs), and digital signal processors (DSPs) to enhance functionality for specific applications. [15]

The miniaturization and integration capabilities of modern semiconductor fabrication processes enable the creation of SoCs with remarkable processing power, memory capacity, and diverse functionalities on a single chip. This miniaturization offers several advantages, including:

- **Reduced size and weight:** SoCs are significantly smaller and lighter than traditional systems built with separate components. This is crucial for space-constrained applications like smartphones and wearable devices.
- **Lower power consumption:** By integrating components on a single chip, shorter signal paths and reduced leakage currents lead to lower overall power consumption, improving battery life in portable devices.
- **Enhanced performance:** SoCs can leverage on-chip communication for faster data exchange between components, potentially leading to improved system performance compared to traditional setups.
- **Reduced cost:** By eliminating the need for multiple discrete components and their associated packaging, SoCs can offer a more cost-effective solution for manufacturers.

3.3 SoC used in CMS DAQ

The CMS DAQ is currently studying the possibility of using the SoC from Zynq UltraScale+ family for the future upgrade. Zynq offers a powerful combination of processing resources that make it ideal for demanding applications like the CMS DAQ system:

- **ARM-based processing cores:** The Zynq UltraScale+ integrates multiple ARM processors. These include Cortex-A53 cores for tackling high-performance tasks and Cortex-R5 cores for real-time processing, a crucial aspect in data acquisition systems [16].
- **Programmable logic:** A key feature of these SoCs is the inclusion of a Field-Programmable Gate Array (FPGA) fabric. This programmable logic allows for custom hardware acceleration and flexibility in implementing functionalities that precisely meet the requirements of the CMS DAQ system [17].
- **On-chip peripherals:** Zynq UltraScale+ SoCs come equipped with various integrated peripherals. These peripherals facilitate interfacing with external devices and sensors commonly used in scientific data acquisition systems [16].

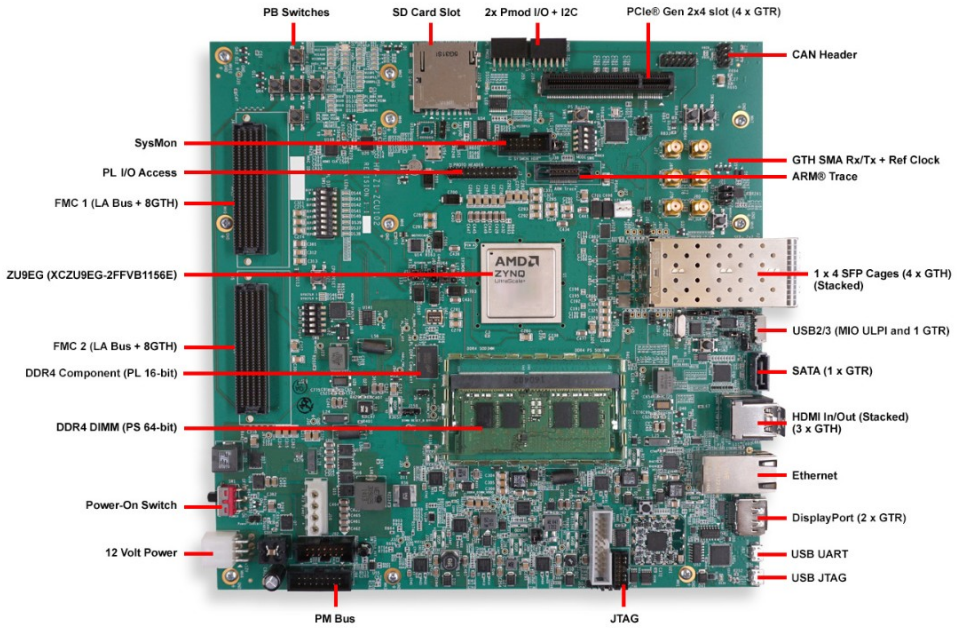

Figure 4. Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit [18]

3.4 SoC booting process

The boot process for a typical SoC involves several stages, each playing a specific role in initializing the system and preparing it for operation.

3.4.1 The First-Stage Bootloader (FSBL)

This lightweight program acts as a bridge between the low-level hardware initialization and the more complex operating system boot process. The FSBL takes over from the Boot ROM, further initializes critical hardware components like processing cores, memory controllers, and peripherals. Its primary responsibility is to locate and load the operating system image (e.g., Linux kernel) from the designated boot device (e.g., SD card) into the main system memory (DDR) [19].

3.4.2 The ARM Trusted Firmware (ATF)

The ATF operates before the traditional boot loader and provides a secure environment for platform initialization and early boot services. The ATF can perform tasks like secure boot verification and platform-specific hardware initialization, enhancing system security. With the ATF initialized by the FSBL, it becomes operational for U-Boot and Linux Kernel. U-Boot starts right after the FSBL [20].

3.4.3 The Second-Stage Bootloader

The second-stage bootloader, often represented by U-Boot, takes control once the FSBL completes its tasks. U-Boot extends the boot process by providing additional features such as interactive command-line interface (CLI), boot script execution, and support for various boot protocols (e.g., TFTP, NFS) [21].

3.4.4 The Linux Kernel Boot

Once the boot loader has loaded the Linux kernel image into memory, the Linux kernel takes center stage, initiating critical system services and loading essential device drivers. This stage prepares the environment for user applications to run on the SoC, marking the final step in bringing the system to a fully operational state [22].

3.5 SoC Network Boot

Network booting offers a compelling solution for deploying firmware and operating systems for a SoC. It eliminates the need for physical media like SD cards, simplifying the development and deployment workflow. This section explores the core aspects of network booting a SoC, including essential protocols, configuration, and security measures.

3.5.1 Network Identity (DHCP Client Identifier)

Traditionally the SoC relies on pre-configured MAC addresses stored in EEPROM (Electrically Erasable Programmable Read-Only Memory) for network identification during boot. This thesis explores an alternative approach that utilizes DHCP Client Identifier (dhcp_clientid) [23, 14, 24].

The use of DHCP Client Identifier aligns with industry standards and protocols, such as the Advanced Telecommunications Computing Architecture (ATCA), which dictates the use of dynamic network configurations to encode geographical information within the DHCP request. This allows network management tools to identify the physical location of a device based on its network identity, simplifying asset tracking and configuration management [25].

3.5.2 Obtaining the Boot Image

U-Boot leverages the Trivial File Transfer Protocol (TFTP) to download the boot image from a designated TFTP server [26]. To achieve this, U-Boot requires the server's IP address, which can be obtained through DHCP (Dynamic Host Configuration Protocol). U-Boot stores this retrieved IP address in an environment variable named "serverip." However, the default definition of "serverip" in Petalinux can lead to errors during TFTP operations if not explicitly set [14].

The default behavior of "serverip" prevents it from being overwritten by the DHCP request. To rectify this, U-Boot's "serverip" variable needs to be explicitly undefined before initiating a DHCP request, ensuring it receives the correct server IP address for successful TFTP retrieval of the boot image [21].

3.5.3 Network File System (NFS)

Network File System (NFS) allows the SoC to function without a local storage device for its root filesystem. This is achieved by instructing the kernel to mount the root filesystem via NFS using a specific boot argument ("root=/dev/nfs"). Additionally, the kernel necessitates the NFS server's IP address, the location (path) of the root directory on the server, and network configuration details. DHCP within the network provides all this information during boot through a separate boot argument ("ip=dhcp") [27].

3.6 Petalinux

PetaLinux is a toolkit crafted for developing embedded Linux systems, specifically for FPGA-based SoCs from Xilinx. It allows users to configure and customize essential components like bootloaders, kernels, device-trees, filesystems, and libraries within the Zynq MPSoC's firmware. Additionally, PetaLinux provides tools for project building and deployment. Based on the Yocto Project, an open-source initiative for creating embedded Linux distributions, PetaLinux offers a straightforward yet powerful platform for embedded system development [28].

3.6.1 Yocto Project Layers

The Yocto Project utilizes a layered approach for building embedded Linux systems. Layers are essentially directories containing metadata files and recipes. These layers can be stacked together to create a complete image for the target hardware [29].

In the context of Petalinux, there are two primary layer types to consider:

- **Petalinux Base Layers:** These layers form the foundation of a Petalinux project. They provide essential components like the Linux kernel, U-Boot bootloader, and various device driver recipes specific to Xilinx hardware. Petalinux offers a predefined set of base layers based on a chosen Zynq MPSoC and desired Linux kernel version [30].
- **Custom Layers:** The true power of Yocto layers lies in the ability to create custom layers. These layers allow to integrate additional software packages, hardware drivers not included in the base layers, or specific configurations tailored to the project's needs. One can create custom layers from scratch or leverage existing community-developed layers [30].

Benefits of Using Yocto Layers in Petalinux:

- **Modular Design:** Layers promote a modular design approach, making it easier to manage project complexity and maintain code reusability.
- **Customization:** Custom layers empower to incorporate specific software components or configurations not available in the base Petalinux layers.

Community Collaboration: The Yocto Project fosters a large and active community. One can benefit from pre-built recipes and layers contributed by the community, saving development time and effort.

3.6.2 BitBake and Recipes

Yocto Project recipes are the fundamental building blocks for creating software packages within an embedded system. Each recipe is a text file written in a specific language and processed by the BitBake tool. BitBake acts as a task executor, responsible for reading recipe files, determining dependencies between packages, and managing the entire build process [31].

A typical recipe specifies the following information:

- **Package Name and Version:** Identifies the software package the recipe builds [32].
- **Source Code Location:** Points to the source code for the package. This code can be downloaded from a remote repository or provided locally within a layer [32].
- **Dependencies:** Specifies any other packages required to build a particular package. BitBake ensures these dependencies are built and installed before attempting to build the current package [31].

Build Instructions: Defines the commands and steps necessary to build the software package, including compilation, linking, and installation.

3.6.3 Petalinux Project Structure

The Petalinux development environment offers a well-organized project structure to streamline the creation of embedded Linux systems for Xilinx devices. This structure separates project components into distinct directories, promoting clarity and manageability. Here's a breakdown of the key elements [33]:

- **project-spec:** This directory houses essential project specifications, including the Yocto Project layer configuration files. These files define the layers used in the project, including the Petalinux base layers and custom layers.
- **arch:** This directory contains architecture-specific components, typically containing pre-built libraries and toolchains optimized for the target Xilinx processor architecture (e.g., ARM) [33].
- **components:** This directory stores the source code for various components like Linux kernel, U-Boot bootloader, device drivers, and root filesystem recipes [33].
- **build:** This directory is generated during the build process and contains temporary build artifacts, object files, and the final output images, such as the boot image and root filesystem [33].
- **images:** This directory stores the final bootable images generated by the Petalinux project, including the boot image used for network booting and the root filesystem image containing the operating system [33].

3.7 Continuous Integration & Continuous Delivery/Deployment

This section explores the concept of Continuous Integration and Continuous Delivery/Deployment (CI/CD), a methodology that automates software development processes [34].

3.7.1 Continuous Integration (CI)

Continuous Integration (CI) refers to a software development practice that automates the integration of code changes from various developers into a single source code repository. Upon each integration, the software is automatically built and subjected to a battery of tests. The primary objective of CI is to facilitate the early detection of bugs and minimize the overall time required to validate a project. By automating the build process, CI ensures consistency and repeatability, guaranteeing that every build adheres to the same preconditions and follows identical steps [35].

3.7.2 Continuous Delivery/Deployment (CD)

Continuous Delivery (CD) builds upon Continuous Integration (CI) by extending automation to the deployment stage. After successful build and testing within the CI pipeline, CD automates the delivery of code changes to a testing environment. In some instances, CD might also encompass deployment to the production environment, in which case the process is referred to as Continuous Deployment. The terms "Continuous Delivery" and "Continuous Deployment" are sometimes used interchangeably as CI/CD. Ultimately, CD

aims to streamline the deployment process by automating testing procedures to verify the application's functionality before release [36].

3.7.3 GitLab CI/CD Pipelines

GitLab, a prominent software development and maintenance platform, offers built-in functionalities for CI/CD pipelines. CERN utilizes its own instance of GitLab (gitlab.cern.ch) for software development and management. CI/CD pipelines within GitLab can be conceptualized as a series of automated scripts executed in a predetermined order. These pipelines consist of two core components [37]:

- **Jobs:** These define specific actions to be performed within the pipeline, such as building the Firmware System Boot Loader (FSBL) [38].
- **Stages:** Stages dictate the execution sequence of jobs. For instance, building boot images must occur after image creation is complete.

Each CI/CD pipeline is linked to its corresponding GitLab repository, granting direct access to all repository files. Configuration of the pipeline is achieved through a dedicated file named ".gitlab-ci.yml." This file allows customization of the pipeline based on the application's specific requirements. Job configurations within the ".gitlab-ci.yml" file typically include details such as job scripts, artifacts, execution triggers, job tags, and stage assignments [39].

3.7.4 GitLab Runners

The execution of jobs within a stage is handled by GitLab Runners. These are open-source applications that collaborate with GitLab CI/CD to execute pipeline jobs. Runners can be deployed on various machines with diverse operating systems. GitLab Runners offer a selection of executors, catering to different job execution scenarios [40].

GitLab Runners also support the use of individual tags. Jobs configured with these tags can only be executed by corresponding Runners. This proves particularly beneficial when an application necessitates execution within a Docker container. A GitLab Runner equipped with a Docker executor will possess a specific tag that a job can reference within its configuration. Upon encountering such a job, the GitLab Runner will claim and execute it [40].

3.7.5 Gitlab Parallel Matrix

GitLab Parallel Matrix facilitate concurrent execution of jobs within a CI/CD pipeline, optimizing build and test execution time by enabling multiple tasks to run simultaneously [41].

In the context of the Zynq Buildsystem, leveraging GitLab Parallel Matrix Builds allows for the concurrent building of several boards without being bound by sequential execution. This parallelization strategy enhances efficiency by fully utilizing hardware resources and minimizing idle time during the build process [41].

For a deeper understanding of the implementation and benefits of parallel matrix build systems, refer to Section 4.5.1.

3.8 Summary

The CMS DAQ team is actively engaged in developing a System-on-Chip (SoC) tailored for the upcoming High-Luminosity LHC upgrade, requiring support for a diverse array of embedded board types. To streamline firmware image building and testing processes, a common framework is being developed. The adoption of GitLab Parallel Matrix Builds, as discussed in Section 3.7.5, represents a significant step forward, offering concurrent execution of tasks within CI/CD pipelines, thereby optimizing resource utilization and minimizing build times.

4. Implementation

This section details the development of the build framework for building, testing, and deploying firmware images and operating systems for the SoC. The framework will provide the necessary infrastructure for network booting, including: network services, storage services, and boot services. It will also handle power cycling of the embedded boards.

Note: Hardware design and programmable logic is provided by the hardware developer. The Operating System type and version is being set by the CMS DAQ Sysadmins.

4.1 Petalinux Pipeline

The Petalinux Pipeline is responsible for generating hardware images for the SoC. The pipeline leverages Petalinux framework, a comprehensive suite of tools for system configuration, kernel development, root filesystem creation, and final system image packaging. To further enhance the development experience, I have incorporated an internal tool developed at CERN called petalinux-template.

4.1.1 Infrastructure

The build process for Petalinux projects can be resource-intensive due to the size and complexity of the framework itself. Xilinx recommends a minimum of 8GB RAM, a 2 GHz CPU with eight cores, and 100GB of free hard drive space for their tools [42].

In our case, the anticipated parallel builds for multiple boards (around 5) significantly increased these requirements. To address this, I established a dedicated build server with the following specifications:

- 56-core Central Processing Unit (CPU)
- 92GB of Random Access Memory (RAM)
- 2TB Solid State Drive (SSD)

4.1.2 Petalinux Template

The Petalinux template provides a standardized and flexible infrastructure for Petalinux development, allowing users to fork the template and add their own patches, configurations,

and layers. It comes pre-equipped with various features, including scripts for interactive or automated building of boot files, a workflow supporting multiple boards with different hardware configurations, and functionality for adding custom layers and patches to the Petalinux project [43].

4.1.3 CI/CD Executor

To have the Petalinux project easily replicated and isolated for each board type, I am leveraging GitLab Runner configured with the Docker executor. The Docker image used in this setup originates from the CERN CCE project (https://gitlab.cern.ch/cce/docker_build). This project specializes in creating and maintaining Docker images specifically designed for Hardware Description Language (HDL) Electronic Design Automation (EDA) software, including support for Petalinux development.

The Petalinux image comes with pre-configured environment containing all the necessary tools and dependencies for building Petalinux projects.

Petalinux Pipeline: CI/CD Structure

The CI/CD structure for the Petalinux pipeline is quite simple. The pipeline consists of a single stage named "build" and one job named "Generate-Images".

Referring to the code snippet below:

CI: Extends

Nearly every stage extends the ".parallel" directive, which integrates the parallel matrix into the CI structure. This enables concurrent building across multiple boards, with variable configurations tailored to each variation.

CI: Include

The include keyword in GitLab CI allows to incorporate YAML configuration files from various locations into your main .gitlab-ci.yml file. This promotes modularity by breaking down complex pipelines, centralizes variable management through a single file accessible by all pipelines, and enables code reuse by defining common stages and scripts in separate files for inclusion across projects. This approach streamlines CI/CD configuration for better readability, maintainability, and consistency [41].

CI: build-petalinux.sh

The "build-petalinux.sh" script automates the entire Petalinux build process. It configures the project environment, utilizes Petalinux tools to construct board-specific system images, integrates project-specific customizations, and finally packages all necessary boot files for deployment on the target SoC. This script streamlines the build process, ensuring consistency and efficiency.

CI: artifacts

The artifacts keyword instructs the pipeline to upload specific files and directories generated during the build process to the cloud. These artifacts, which can encompass test reports, build outputs, or deployment packages, become readily available for download and further use after the job finishes.

In the provided code snippet (Figure X), upon successful build completion, the script compresses the built images into an archive named zynq-images.tar.gz. By designating this archive as an artifact (artifacts: paths: - 'zynq-images.tar.gz'), the CI/CD pipeline makes it downloadable for later use in the upcoming pipelines.

Figure 5. Petalinux Pipeline gitlab-ci.yml

```
stages:
  - "build"

include:
   - project: "hardware/zynq/zynq-buildsystem/orchestrator"
     ref: master
     file:
     - "variables.yml"
     - 'parallel-matrix.yml'

Generate-Images:
  extends: .parallel
  stage: build
  image: $PETALINUX_DOCKER_IMAGE
  script:
    - export BOARD_NAME=${BOARD_TYPE}
    - bash -e 'build-petalinux.sh'
    - tar -czf zynq-images.tar.gz -C petalinux-template/
  artifacts:
    paths:
      - 'zynq-images.tar.gz'
```

4.2 Linux Filesystem Pipeline

This section builds upon the work of V. Amoiridis [12], who established the foundation for the Linux Filesystem Pipeline with functionalities for root filesystem creation and a general CI/CD structure.

My contributions to the Pipeline focused on enhancing its capabilities. I upgraded the mkrootfs.py script to support Alma Linux 9, expanding its compatibility beyond CentOS 8. Additionally, I implemented a mechanism for storing the operating system within a centralized web server. This involved developing a portable web server encapsulated in a Docker image.

4.2.1 Building the Root Filesystem

This pipeline leverages CERN's centos-rootfs tool to build standardized and flexible root filesystems specifically tailored for ARM architectures [44]. Central to this tool are two key functionalities: first, establishing a development environment with QEMU support, essential for cross-platform development; and second, utilizing the mkrootfs.py script to build the root filesystem from RPM packages. This script interfaces seamlessly with the DNF package manager, facilitating the streamlined installation of dependencies and packages necessary for constructing a functional Linux ARM system.

4.2.2 Transitioning to Alma Linux

Since Cent OS Linux, a popular choice for embedded systems, is nearing its end-of-life on June 30th, 2024 [13]. CERN is transitioning to Alma Linux, a community-driven, Red Hat Enterprise Linux (RHEL) compatible distribution offering long-term stability and robust community support.

Since the centos-rootfs tool is built in a modular way, adding support for Alma Linux 9 was a relatively straightforward process. This involved updating some distribution-specific links and configuration variables within the script to reflect the new package repositories and naming conventions of Alma Linux.

4.2.3 Store root filesystem

Upon successful completion of the root filesystem build process, a bash script orchestrates the packaging of the generated files into a compressed tar archive. Subsequently, this

archive is transferred and stored on a remote web server, thereby establishing a centralized repository for storing root filesystem images.

This strategic approach is designed to optimize the development pipeline by mitigating the need for rebuilding the root filesystem from scratch with each iteration. Instead, by leveraging the stored images, developers can readily access and deploy pre-built root filesystems, significantly reducing turnaround time and resource utilization.

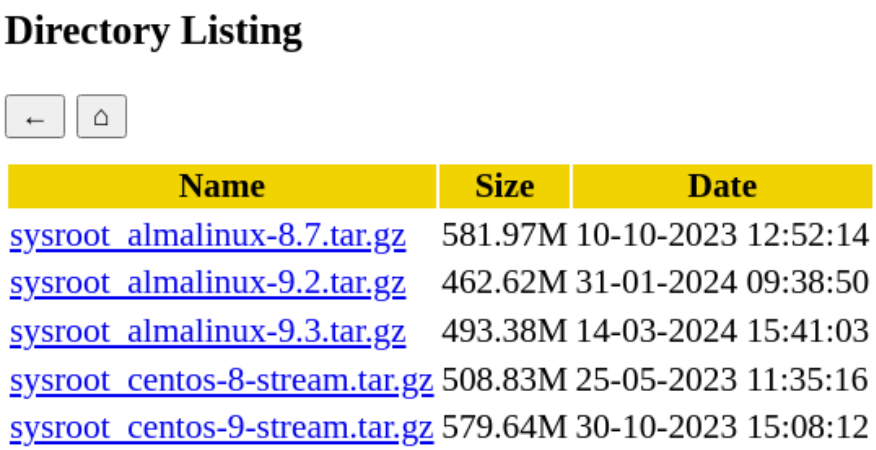

Figure 6. Directory Listing of Root File Systems

4.2.4 Portable web server

This project utilized Docker containers to streamline the deployment and management of the web server. Compared to traditional methods of directly installing software on the operating system, Docker offers several advantages. Firstly, containers encapsulate all essential components – the Nginx web server software, its configuration files, and any necessary dependencies – into a single, standardized unit.

The same image can be seamlessly deployed to any Docker host, regardless of the underlying operating system or hardware specifications. This eliminates the cumbersome task of manual configuration on each individual system, fostering increased efficiency and consistency across deployments. By leveraging Docker containers, this project achieved a streamlined and portable web server environment, enhancing developer productivity, reducing configuration complexities, and ensuring consistent behavior across deployments.

4.3 Boot and Test Pipeline

The Boot and Test Pipeline automates the deployment and testing of newly built system images on System-on-Chip (SoC) devices. Building upon the core concept and foundational code established by Vasileios Amoiridis [12], I focused on significantly improving the pipeline's functionality and user experience.

Key Improvements:

- **Portable Network Services:** A portable network services component was developed, eliminating the requirement for pre-configured network services. This simplifies deployment for users and reduces setup complexity.
- **Centralized Image Storage:** A mechanism for storing built system images on a centralized web server was implemented. This facilitates efficient image reuse and streamlines the development workflow.
- **IPMC Power Cycling Support:** Intelligent Platform Management Interface (IPMC) support was integrated to enable power cycling of the boards, aligning the development environment more closely with production settings and enhancing workflow realism.
- **Pipeline Optimizations:** Various stages of the pipeline were optimized, resulting in improved performance and efficiency in testing and image building processes.

These enhancements collectively contribute to a more robust and user-friendly experience, benefiting the overall SoC development process.

4.3.1 Infrastructure

This pipeline first imports the images built earlier in the Petalinux Pipeline for all the boards. Then constructs the infrastructure for network booting, deploys the images onto the target SoCs, and subsequently boots and executes comprehensive tests to verify system functionality.

Each embedded board is connected to the server via JTAG interface. This JTAG interface facilitates communication with the board's internal debugging circuitry, enabling real-time monitoring. The terminal output, consisting of console logs and debugging messages is streamed to the server for analysis and troubleshooting [45].

Furthermore, each board is connected with a controllable power source. This power source

can be managed through either an Intelligent Platform Management Controller (IPMC) or a Power Distribution Unit (PDU). An IPMC offers a more integrated approach, allowing for in-band power management directly through the connected network. Conversely, a PDU provides a more traditional, out-of-band method for controlling power utilizing dedicated network protocols or a physical interface [46, 47].

The development of this framework addressed the challenge of supporting both IPMC and PDU for power control, ensuring flexibility for different hardware configurations.

4.3.2 Build Images

The Build Images stage serves as the foundational step in the pipeline, responsible for provisioning board-specific images onto designated directories within the TFTP and NFS server root. This phase ensures that the target boards receive the necessary firmware and operating system components tailored to their configurations.

The zynq-images can be deployed to TFTP server right away, however the root filesystem needs some initial customization before it can be deployed to the NFS server. It's because the server only stores a generic OS, which doesn't yet have any board specific customizations like kernel modules.

The general workflow for the Build Images stage is:

1. Download zynq-images from Petalinux-Template Pipeline
2. Download root filesystem from webserver
3. Copy kernel modules from zynq-images into the root filesystem
4. Transfer board specific root filesystem to NFS server root
5. Transfer zynq-images to TFTP server root

4.3.3 Network Services

The Network Services stage leverages Docker containers orchestrated by Docker Compose to deploy all the network services for network boot operations.[1] The containerized approach to network services was specifically developed to provide a readily replicable and manageable network boot infrastructure for other teams, particularly those with less technical expertise in SoC development. By minimizing configuration overhead, promoting rapid development cycles, and fostering portability and scalability, the pipeline empowers

[1] Docker Compose is a tool for defining and running multi-container Docker applications.

these teams to efficiently deploy and manage their own network boot environments for embedded systems.

The containerized approach offers several key advantages:

- **Modularity**: Docker containers isolate individual services, simplifying development, deployment, and maintenance.
- **Scalability**: Services can be easily scaled up or down based on network traffic and resource requirements.
- **Portability**: Containerized applications are quite consistently across different environments, regardless of the underlying infrastructure.

The following services are configured and deployed during this stage:

- **NFS (Network File System)**: NFS acts as a distributed file system accessible over a network. This enables efficient sharing of the root filesystem across networked devices, eliminating the need for local storage on each client [27].
- **TFTP (Trivial File Transfer Protocol)**: Provides a lightweight and scalable solution for delivering bootloader and kernel images during network boot [26].
- **Dnsmasq**: Acts as a versatile DHCP and DNS server, providing network configuration and name resolution for client devices [48].
- **NTP (Network Time Protocol)**: Ensures consistent time synchronization across the network, crucial for logging and secure communication [49].

4.3.4 Boot

The Boot stage is a critical step in verifying that the developed software is valid, ensuring the successful loading of the operating system and firmware on the target board. This section details the multi-stage process and its flexibility in handling different board configurations. The Boot process consists of four distinct phases:

1. **Power Cycle and Boot Initiation**
   The initial phase triggers a system reset by performing a power cycle on the board. This initiates the boot sequence, loading the bootloader and preparing the hardware for further stages.
2. **Boot Monitoring**
   Following the power cycle, the pipeline enters a monitoring stage. Here, the system actively observes the terminal output from the board via a serial console, to con-

firm successful booting. This involves verifying specific log messages indicating successful hardware initialization and bootloader functionality.

3. **BOOT.BIN Update**
   The third stage focuses on updating the BOOT.BIN file, the low-level firmware responsible for early boot initialization. Since this cannot be updated remotely, an update on the board itself is necessary. This section highlights the framework's capability to handle two distinct board types:
   - **SD Card Boot:** For boards relying on SD cards for boot configuration, the framework locates and updates the BOOT.BIN on the card (specific tools or commands might be used depending on the implementation).
   - **QSPI Flash Memory:** Alternatively, boards utilizing QSPI flash memory store the BOOT.BIN internally. The framework adapts accordingly, updating the image within the flash storage with the use of a dedicated utility.
4. **Post-Update Verification:** To make sure, that the BOOT.BIN is valid, the process concludes with another power cycle. This final reboot verifies that the board successfully boots with the updated firmware, incorporating any hardware design changes or enhancements included in the new BOOT.BIN image.

It's important to note that encountering errors during any of the Boot process stages will trigger a pipeline halt. This is an indicator of a potential hardware design issue. By pinpointing the failing stage, developers can focus troubleshooting efforts on the corresponding hardware component or functionality. This facilitates a more efficient debugging process and identification of hardware-related problems.

4.3.5 Store Images

The final stage of this pipeline is initiated only upon successful completion of all preceding stages. This indicates that the board has successfully booted with the newly built operating system and firmware images, validating the integrity and functionality of the deployed software stack.

Upon successful boot, a bash script orchestrates the packaging of relevant files into a compressed tar archive, facilitating efficient storage and transfer. Subsequently, these archives are exported to a designated remote web server. The web server employs a structured directory hierarchy. Each target board possesses a dedicated directory within the server's file system. This dedicated directory serves as the repository for the board's specific boot image set, containing both the customized root filesystem and the corresponding zynq-images.

The web server employs a structured directory hierarchy. Each target board possesses a dedicated directory within the server's file system.

The centralized web-server offers a significant advantage. Anyone interested in deploying their own board can readily access and download production-ready image sets from the centralized repository. This eliminates the need for manual image creation and customization, significantly simplifying the process of setting up embedded boards.

Figure 7. ZCU-102 Image Directory Listing

4.4 Pipeline Orchestrator

The Pipeline Orchestrator acts as the central control unit, coordinating the execution of all previously described pipelines. The master pipeline serves the critical purpose of triggering downstream pipelines in a defined sequence. Similar to other framework components, the core concept of the pipeline orchestrator was proposed by V. Amoiridis at his thesis [12]. However, I have improved the original design by:

- Implementing centralized variables for improved maintainability and project-wide consistency.
- Enabling dynamic configuration to streamline pipeline customization.
- Developing a dedicated pipeline user interface for enhanced user experience.

4.4.1 CI/CD Structure

Unlike other pipelines, the orchestrator does not execute any scripts itself. Instead, it initiates the execution of downstream pipelines using the trigger keyword (please refer to the figure 8 for a code reference).

Figure 8. Trigger Downstream Pipeline

```
stages:
    - Build-Images

...

Trigger-Petalinux:
  stage: Build-Images
  trigger:
      project: "hardware/zynq/zynq-buildsystem/petalinux-template"
      branch: $PETALINUX_PROJECT_BRANCH
      strategy: depend

...
```

4.4.2 Centralized Variables

In order to make the codebase of the build system easy to manage, a decision was made to consolidate all variables into a centralized file called "variables.yml" and make it available for all the associated projects to read. Centralizing the variables allow for easy project-wide adjustments without modifying individual pipeline configurations. It's located in the orchestrator's repository, and other pipelines can read the file and incorporate these variables in the pipeline.

To enable other pipelines to read the centralized variables, each pipeline has the "include" keyword in it's "gitlab-ci.yml" file. This reads the remote file and incorporates all the listed variables to the CI environment. Please refer to the figure 9 for a code reference.

Figure 9. Import Variables from a Remote Repository

```
include:
  - project: "hardware/zynq/zynq-buildsystem/orchestrator"
    ref: master
    file: "variables.yml"
```

Pipeline Variables

It's important to understand that by default, only YAML-defined bridge variables are passed on to downstream pipelines. These bridge variables act as a predefined set of variables configured specifically for this purpose. However, the limitation arises when custom pipeline variables, are not automatically transmitted [41].

The line forward: "pipeline_variables: true" (please refer to the figure 10 for a code

reference) ensures that all variables defined within the triggered pipeline (in this case, petalinux-template) are accessible to the downstream pipeline initiated by the orchestrator.

Figure 10. Pass Variables to a Downstream Pipeline

```
Trigger-LinuxFilesystem:
  stage: Build-Images
  trigger:
      project: "hardware/zynq/zynq-buildsystem/linux-filesystem"
      branch: $LINUX_PROJECT_BRANCH
      strategy: depend
      forward:
        pipeline_variables: true
```

4.4.3 User Interface

While investigating the implementation of multi-board builds within our framework, I came across a valuable feature offered by GitLab: dropdown selection with choice options. Leveraging this feature simplifies the process of selecting the specific board(s) to build (see Figure 12). Users can choose a specific board type or leverage the "ALL" option to trigger parallel execution for all supported boards.

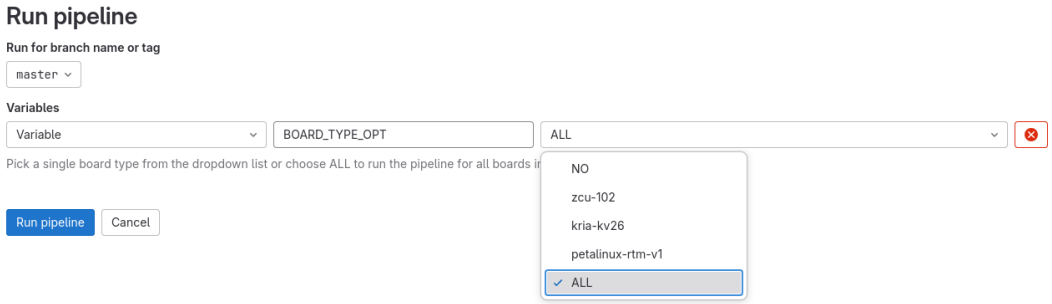

Figure 11. Gitlab UI

The code snippet in Figure 12 defines the configuration for the dropdown menu within a variables.yml file. The BOARD_TYPE_OPT variable specifies the options available in the menu, along with a clear description for user guidance. That variable is later passed to all downstream pipelines.

Figure 12. Code snippet: variables.yml

```
variables:

BOARD_TYPE_OPT:
  description: "Pick a single board type from the dropdown list \
  or choose ALL to run the pipeline for all boards in parallel"
  value: "NO"
  options:
    - "NO"
    - "zcu-102"
    - "kria-kv26"
    - "petalinux-rtm-v1"
    - "ALL"
```

4.5 Parallelisation Over Several Boards

The core strength of this build framework lies in its effective utilization of GitLab's Parallel Build functionality, as discussed in the subsection on Parallel Matrix (Section 3.7.5). Parallel execution proves particularly advantageous in two key areas: the PetaLinux Pipeline (Section 4.1) and the Boot and Test Pipeline (Section 4.3).

In the Petalinux Pipeline, the framework capitalizes on parallelization by creating dedicated nodes for each board type being built concurrently. This approach ensures that the time required to build multiple boards remains consistent with that of building a single board. By distributing the build process across parallel nodes, the framework maximizes resource utilization and minimizes overall build time, which would otherwise be prolonged in a sequential execution scenario.

Similarly, in the Boot and Test Pipeline, parallelization enables simultaneous testing of multiple boards. Traditionally, testing a single board can be time-consuming, but through parallel execution, the framework achieves comparable results across multiple boards in significantly less time. This expedited testing process is crucial for maintaining project timelines and facilitating rapid development iterations [41].

4.5.1 Performance Comparison: Sequential vs. Parallel Builds

This section evaluates the performance benefits of parallel execution within the build framework. By comparing the build and deployment times for single and multiple boards using both sequential and parallel approaches.

Sequential build

Building a single board typically takes approximately 25 minutes. However, as the number of boards increases, the build duration in a sequential execution grows proportionally. For example, building 5 boards sequentially would require an estimated total time of 5 boards $\times$ 25 minutes/board $=$ 125 minutes. Additionally, testing and deployment add another 4 minutes per board, bringing the total time for building and deploying 5 boards sequentially to 125 minutes $+$ (5 boards $\times$ 4 minutes/board) $=$ 145 minutes. See Figure 13 for visualization.

Parallel build

The core advantage of parallel execution lies in its ability to significantly reduce build times for multiple boards. With parallel processing, the build process for each board is executed concurrently, effectively keeping the total build time close to that of a single board. Building 5 boards in parallel would still take approximately 25 minutes, as the builds happen simultaneously. Testing and deployment also benefit from parallelism by reducing the overall time.

In this scenario, the estimated total time for building and deploying 5 boards in parallel would be around 25 minutes (build) $+ (5$ boards $\times\ 4$ minutes/board testing$) = 45$ minutes. This represents a significant improvement compared to the sequential approach, offering a time reduction of approximately 68.97% (145 minutes $-\ 45$ minutes) for building and deploying 5 boards. See Figure 13 for visualization.

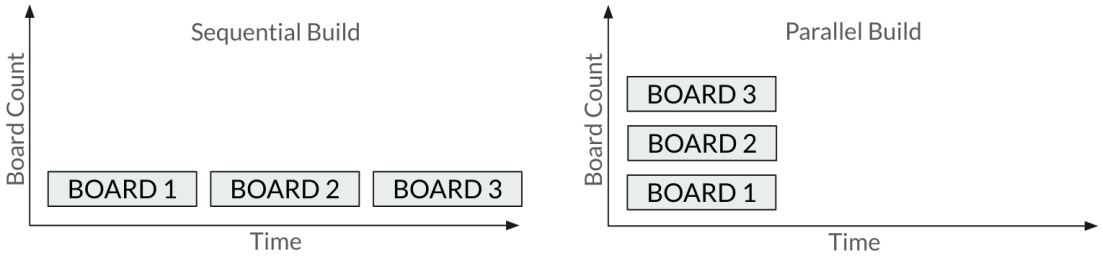

Figure 13. Parallel Matrix vs Sequential build

4.5.2 Variable configuration

Another great feature of Gitlab Parallel Build is the ability to define and pass variable build configuration for each board type. For example, in our laboratory setup, we have two board categories differentiated by their firmware storage method. Some boards utilize an SD card, while others store firmware in the onboard flash memory (QSPI). This necessitates different firmware update processes for each category.

Thanks to variable configuration in the Parallel Matrix, we can declare a variable named BOOT_VOLUME within the matrix. Later, during the image building stage of the Boot and Test Pipeline (see Section 4.3), the script checks the value of BOOT_VOLUME to determine the appropriate destination for the firmware image (e.g., QSPI flash or SD card).

The code snippet (Figure 14) showcases a basic implementation of the parallel matrix with variables for BOARD_TYPE and BOOT_VOLUME.

Figure 14. Code snippet: parallel-matrix.yml

```
.parallel:
  parallel:
    matrix:
        - BOARD_TYPE: kria-kv26
          BOOT_VOLUME: QSPI

        - BOARD_TYPE: petalinux-rtm-v1
          BOOT_VOLUME: mmcblk1p1
```

4.6 Summary

This chapter outlines the implementation of a build framework. The key components of the framework include the Petalinux Pipeline for hardware image generation, Linux Filesystem Pipeline for root filesystem creation, and Boot and Test Pipeline for automated deployment and testing.

Parallelization is a pivotal feature, enabling simultaneous testing of multiple boards and reducing overall build times. Additionally, the Pipeline Orchestrator coordinates pipeline execution and implements centralized variables for enhanced maintainability. Leveraging GitLab's Parallel Build functionality, the framework achieves efficient resource utilization and customized build processes for different board types, ensuring scalability and performance optimization throughout the development lifecycle.

5. Testing

To validate the functionality of the automated CI/CD framework, a comprehensive testing approach was employed. This involved The automated CI/CD framework is tested by deploying a working project within its infrastructure. Modifications are categorized by pipeline for clarity. The framework's functionality is comprehensively tested, including building and deploying firmware and software to the target boards, initiating the boot process, and actively monitoring for any boot-related issues.

Table 1. *Steps required to add a new board to the pipeline*

Nr	**Pipeline**	**Action**
1	Maestro Pipeline	Create a new entry in the following files with the new board(s) name: boot-config.yml, parallel-matrix.yml, variables.yml
2	Petalinux Pipeline	Copy the Petalinux project of the new board(s) into the Petalinux Pipeline.
2	Linux Filesystem Pipeline	-
3	Boot Pipeline	Ensure the new board(s) are connected to the lab server via JTAG and share the same network.

The Linux Filesystem Pipeline does not require any additional customization since its objective is to build a generic Linux root-file-system with the DAQ tools included.

5.1 Maestro Pipeline Modifications

This section outlines the steps required to incorporate a new board into the Maestro Pipeline.

1. The new board details have to be added to the variables.yml in order showcase the board as an option in the GitLab GUI.

The BOARD_TYPE_OPT variable controls which boards are processed by the pipeline. It works by the following order:

- **Default Value (NO):** Leaving this variable unset (default value of "NO") means the pipeline won't run at all.
- **Selecting a Board:** Choose a specific board type to run the pipeline for that board only.
- **Building All Boards:** Setting the variable to "ALL" instructs the pipeline to build all supported boards simultaneously.

Figure 15. Code snippet: variables.yml

```
variables:
    BOARD_TYPE_OPT:
      value: "NO"
      options:
        - "NO"
        ...
        - "zcu-102"
        ...
        - "ALL"
```

2. The new board details have to be added to the parallel-matrix.yml in order to enable parallelization.

- **BOARD_TYPE:** Specifies the type of board being used in the pipeline, set to "zcu-102."
- **XSA:** Refers to the Xilinx XSA file associated with the project, containing FPGA configuration information.
- **BOOT_VOLUME:** Identifies the boot volume or partition on the board's storage device.
- **KDUMP_VOLUME:** Specifies the volume or partition for kernel crash dump data.
- **TESTING_HOST:** Indicates the host name or IP address of the testing host machine.

Figure 16. Code snippet: parallel-matrix.yml

```
.parallel:
  parallel:
    matrix:
        - BOARD_TYPE: zcu-102
          XSA: project_1.xsa
          BOOT_VOLUME: mmcblk0p1
          KDUMP_VOLUME: mmcblk0p2
          TESTING_HOST: ATCA-LAB40-R02-01-01-ctrl
```

3. The new board details have to be added to the boot-config.yml for network configuration.

- **ATCA-...** The correct hostname for a machine in DAQ.
- **alias:** Alias for the host "ATCA-LAB40-R02-01-01-ctrl," set to "zcu102-lab40-r02-board01."
- **dhcp_client_id:** DHCP client identifier for the host, represented by a hexadecimal string.
- **tty_usb:** Identification for the USB serial port associated with the host, labeled as "ZCU00."
- **pdu_outlet:** Outlet number on the power distribution unit (PDU) controlling power to the host, specified as "1."
- **ip:** IP address assigned to the host, configured as "192.168.0.11."

Figure 17. Code snippet: boot-config.yml

```
hosts:
  ATCA-LAB40-R02-01-01-ctrl:
    alias: zcu102-lab40-r02-board01, zcu102
    dhcp_client_id: ff:00:00 ... 00:00:07:c0:01
    tty_usb: ZCU00
    pdu_outlet: 1
    ip: 192.168.0.11
```

5.2 Petalinux Pipeline Modifications

This section outlines the steps required to incorporate a new board into the Petalinux Pipeline:

- Choose a board name, eg. BOARD_TYPE=ZCU-102
- Create a directory: boards/$BOARD_TYPE
- Access the existing Petalinux project (e.g., <petalinux-project>) and copy the contents of the project-spec into the newly created folder (boards/$BOARD_-TYPE/project-spec).

5.3 Boot Pipeline Modifications

This section outlines the steps required to incorporate a new board into the Boot Pipeline:

- Connect the board to the server:
  - **(A):** Connect the board with the server using a JTAG connector.

- **(B):** Insert the board into the IPMC crate.

- Connect the board to the same network as the server.

5.4 Results

The automated CI/CD framework successfully executed all pipelines, signifying a smooth compilation process for all projects involved. This success is further visualized in Figure 18, showcasing the "Successfully Passed Stages" within the Maestro pipeline.

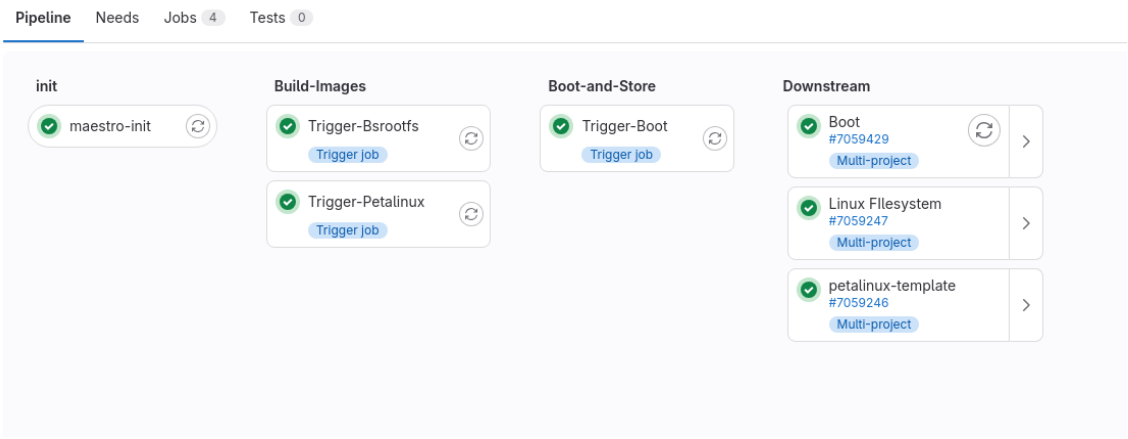

Figure 18. Successfully Passed Stages

As intended, the CI/CD framework demonstrated its parallel build capabilities by generating firmware images for multiple boards simultaneously. Figure 19 clearly illustrates this achievement.

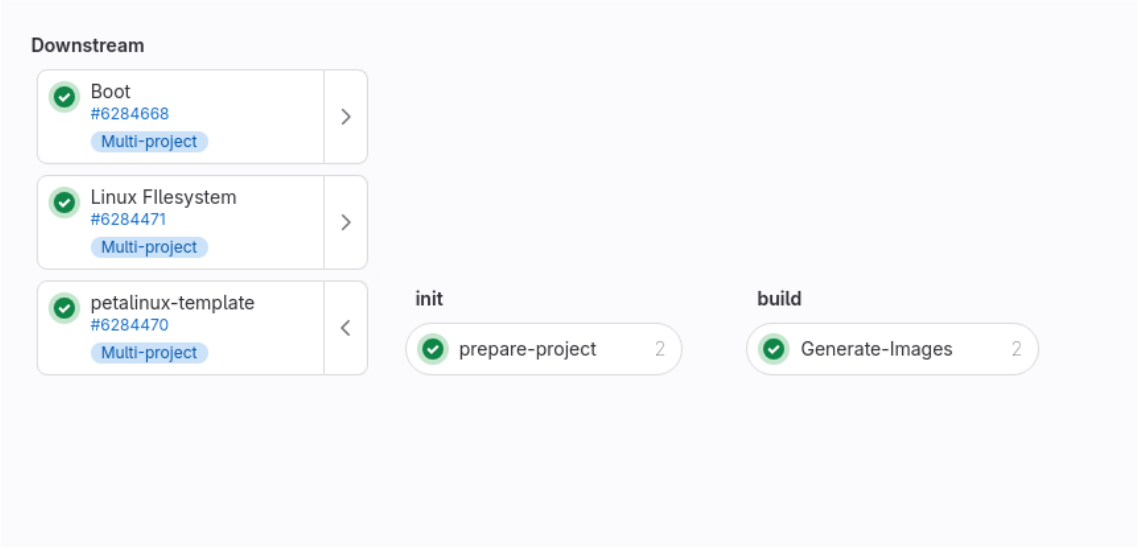

Figure 19. Successful Build of Firmware Images

Beyond building firmware, the framework successfully constructed an AlmaLinux 9.3 Filesystem, as evidenced in Figure 20.

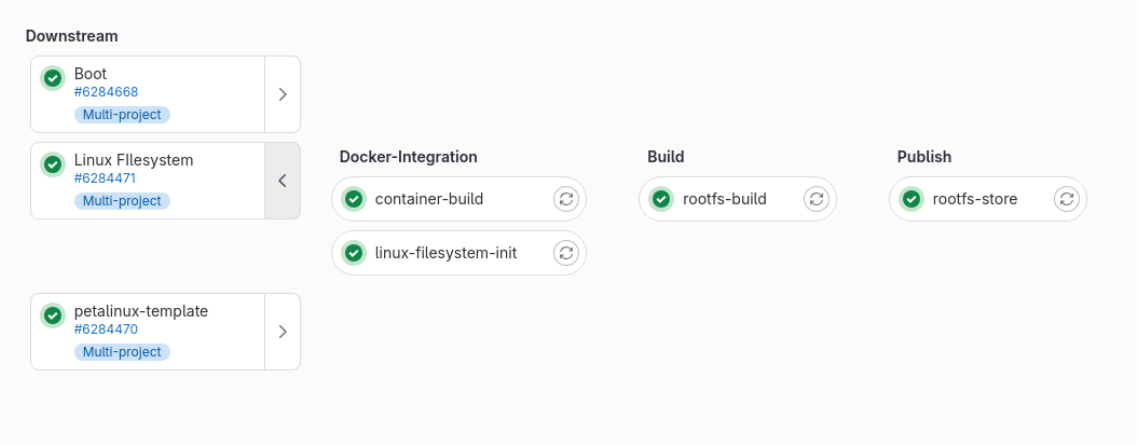

Figure 20. Successful Build of Linux

The framework has effectively demonstrated its ability to deploy images and boot boards in parallel, successfully passing all boot tests. These tests included verifying successful DHCP requests, the mounting of the root filesystem, loading the operating system without errors, and displaying the login prompt. Furthermore, the process also saw the successful update of the boot.bin file.

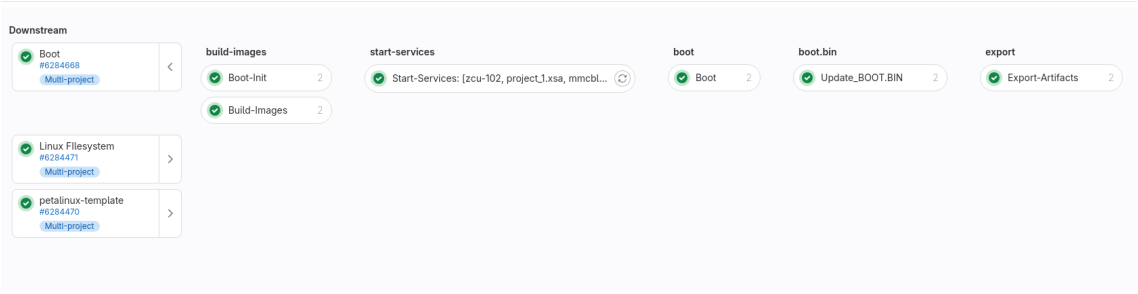

Figure 21. Successful Boot & Deployment

5.5 Summary

The automated CI/CD framework demonstrates its core functionality. It successfully builds, tests and deploys software and firmware images for a multitude of Systems on Chip, initiates the boot process, and actively monitors its progress. This accomplishment will streamline the development and deployment workflow for embedded systems within the CMS DAQ group.

6. Conclusion

This thesis successfully addressed the limitations of the manual development workflow for Zynq-based SoC boards in the CMS DAQ system. The prior approach lacked repeatability and relied on error-prone manual steps, hindering development efficiency.

As outlined in the objectives, this work aimed to achieve three key goals:

- **Automated Build and Testing:** Develop a system that automates building, deployment, and testing of firmware images across various SoC boards, reducing manual intervention and accelerating development cycles.
- **Enhanced Build Reliability:** Integrate automated testing and fault detection mechanisms for a more robust build process compared to manual methods, minimizing errors and bugs.
- **Improved Developer Workflow:** Eliminate the need for manual build steps, simplifying the development process and allowing developers to focus on higher-value tasks.

These objectives were achieved through the implementation of a parallel, automated build system leveraging a Continuous Integration/Continuous Deployment (CI/CD) pipeline. This CI/CD pipeline orchestrates various tools and scripts to effectively manage communication with boards and the development environment. Additionally, it retrieves results and facilitates comprehensive reporting at different stages of the build process.

The implemented CI/CD approach delivers on the proposed objectives. By eliminating manual intervention and ensuring build reproducibility, it simplifies and streamlines the build process. Furthermore, the integration of fault detection mechanisms and extensive testing enhances the reliability of built firmware images and operating systems for embedded boards, surpassing the limitations of manual methods.

The key contribution of this thesis is a production-ready, automated build system. This system is both parallelized and automated, empowering developers to efficiently build and test firmware and software across diverse SoC boards within the CMS DAQ system.

The deliverables outlined in the objectives section have all been successfully produced:

- **A Parallel CI/CD Framework for Building Multiboard Systems on Chip:** This framework automates the build process for various SoC boards.
- **A Parallel CI/CD Framework for Testing Multiboard Systems on Chip:** This framework integrates automated testing and fault detection mechanisms to ensure the reliability of built firmware and software.
- **Container Images for Providing the Critical Infrastructure for the Build System:** These pre-configured container images simplify deployment and ensure consistent functionality of essential services like DNS, DHCP, TFTP, NFS, and time synchronization.
- **Documentation on the Whole Build Framework:** Comprehensive documentation has been created to guide developers in using and maintaining the automated build system.

The research, development, and implementation of this project were shared with the wider scientific community through presentations to engineers and scientists at CERN. The feedback received during these discussions has been instrumental in shaping the framework's development.

Overall, this thesis successfully addressed the limitations of the existing manual build process and delivered a robust, automated CI/CD framework that streamlines the development workflow for Zynq-based SoC boards used in the CMS DAQ system.

6.1 Future work

6.1.1 Caching Functionality for the Petalinux Pipeline

The current implementation of the CI/CD framework performs a complete rebuild of the Petalinux project during each execution. While this approach ensures consistency, it can be time-consuming, especially for large projects. To address this limitation, future work could focus on implementing a caching mechanism within the Petalinux pipeline.

There are two primary caching strategies to consider:

1. **Gitlab Caching:** This approach would involve caching intermediate build artifacts generated by the Petalinux tools within the local build environment. These artifacts could include compiled object files, libraries, and other pre-processed components. On subsequent builds, the framework could check the cache for existing artifacts and only recompile those that have changed since the last successful build. This approach is relatively straightforward to implement but may require modifications to the Petalinux build scripts to manage the cache effectively [50].
2. **Shared State Cache:** This strategy utilizes a dedicated server to store cached build artifacts. The Petalinux pipeline could be extended to communicate with this server, checking for existing artifacts relevant to the current build and downloading them if available. This approach offers greater scalability as the cache can be shared across multiple build environments. However, it requires setting up and maintaining a dedicated server for cache storage, adding complexity to the overall infrastructure [51].

References

[1] High-Luminosity LHC. Wednesday 17th April, 2024. URL: https://home.cern/science/accelerators/high-luminosity-lhc.

[2] CERN. Wednesday 17th April, 2024. URL: https://home.cern/about.

[3] CERN collaborations. Wednesday 17th April, 2024. URL: https://home.cern/about/collaborations.

[4] The Large Hadron Collider (LHC). Wednesday 17th April, 2024. URL: https://home.cern/science/accelerators/large-hadron-collider.

[5] Higgs boson discovery. Wednesday 17th April, 2024. URL: https://www.sciencedirect.com/science/article/pii/S0370269312008581.

[6] CERN. *CERN Science Gateway*. Wednesday 17th April, 2024. URL: https://home.cern/news/press-release/knowledge-sharing/cern-unveils-its-science-gateway-project.

[7] Experiments at the LHC. Wednesday 17th April, 2024. URL: https://home.cern/science/experiments.

[8] Paolo Azzurri. *The CMS Silicon Strip Tracker*. Accessed: Tuesday 14th May, 2024. 2005. URL: https://arxiv.org/abs/physics/0512097.

[9] CMS Collaboration. *CMS TECHNICAL DESIGN REPORT FOR THE LEVEL-1 TRIGGER UPGRADE*. Wednesday 17th April, 2024. 2013. URL: https://cds.cern.ch/record/581342?ln=en.

[10] ENERGY OF HADRONS (HCAL). Wednesday 17th April, 2024. URL: https://cmsexperiment.web.cern.ch/detector/measuring-energy/energy-hadrons-hcal.

[11] *CMS DAQ Collaboration*. Accessed: Tuesday 14th May, 2024. 2017. URL: https://iopscience.iop.org/article/10.1088/1742-6596/898/3/032019.

[12] Vasileios Amoiridis. *An automated CI/CD framework for Zynq UltraScale+ MPSoC devices*. Wednesday 17th April, 2024. URL: https://cds.cern.ch/record/2823386?ln=en.

[13] RedHat. *What to know about CentOS Linux EOL*. Thursday 18th April, 2024. URL: https://www.redhat.com/en/topics/linux/centos-linux-eol.

[14] *Dynamic Host Configuration Protocol*. Wednesday 17th April, 2024. URL: https://en.wikipedia.org/wiki/Dynamic_Host_Configuration_Protocol.

[15] *System on a chip*. Wednesday 17th April, 2024. URL: https://en.wikipedia.org/wiki/System_on_a_chip.

[16] *Zynq™ UltraScale+™ MPSoC*. Wednesday 17th April, 2024. URL: https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html.

[17] *Field-programmable gate array*. Wednesday 17th April, 2024. URL: https://en.wikipedia.org/wiki/Field-programmable_gate_array.

[18] *Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit*. Wednesday 9th May, 2024. URL: https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html.

[19] *Bootloader*. Wednesday 17th April, 2024. URL: https://en.wikipedia.org/wiki/Bootloader.

[20] Javier Garcia. *Arm trusted firmware (atf)*. Wednesday 17th April, 2024. URL: https://ohwr.org/project/soc-course/wikis/ARM-Trusted-Firmware-(ATF).

[21] *The U-Boot Documentation*. Wednesday 17th April, 2024. URL: https://docs.u-boot.org/en/latest/.

[22] *Kernel Boot Process*. Wednesday 17th April, 2024. URL: https://0xax.gitbooks.io/linux-insides/content/Booting/.

[23] *EEPROM*. Wednesday 17th April, 2024. URL: https://en.wikipedia.org/wiki/EEPROM.

[24] *Access-Network-Identifier Option in DHCP*. Wednesday 17th April, 2024. URL: https://datatracker.ietf.org/doc/rfc7839/.

[25] Luigi Calligaris et al. *Novel developments on the OpenIPMC project*. Thursday 18th April, 2024. URL: https://arxiv.org/pdf/2310.19742.pdf.

[26] *The TFTP Protocol (Revision 2)*. Wednesday 17th April, 2024. URL: https://datatracker.ietf.org/doc/rfc1350/.

[27] *Network File System*. Wednesday 17th April, 2024. URL: https://en.wikipedia.org/wiki/Network_File_System.

[28] *PetaLinux Tools*. Wednesday 17th April, 2024. URL: https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html.

[29] *Yocto Project*. Wednesday 17th April, 2024. URL: https://docs.yoctoproject.org/.

[30] *Yocto Project Layers*. Wednesday 17th April, 2024. URL: https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/57836605/Creating+a+Custom+Yocto+Layer.

[31] *BitBake*. Wednesday 17th April, 2024. URL: https://docs.yoctoproject.org/bitbake/.

[32] *Yocto Project Recipe Reference Manual*. Wednesday 17th April, 2024. URL: https://docs.yoctoproject.org/dev-manual/new-recipe.html.

[33] AMD. *PetaLinux Tools Reference Guide*. Wednesday 17th April, 2024. URL: https://docs.amd.com/r/en-US/ug1144-petalinux-tools-reference-guide.

[34] Atlassian. *What is continuous integration?* Wednesday 17th April, 2024. URL: https://www.atlassian.com/continuous-delivery/continuous-integration.

[35] *Continuous Integration*. Wednesday 17th April, 2024. URL: https://en.wikipedia.org/wiki/Continuous_integration.

[36] *Continuous Delivery/Deployment*. Wednesday 17th April, 2024. URL: https://en.wikipedia.org/wiki/Continuous_delivery.

[37] GitLab. *CI/CD Pipelines*. Wednesday 17th April, 2024. URL: https://docs.gitlab.com/ee/ci/pipelines/.

[38] GitLab. *Jobs*. Wednesday 17th April, 2024. URL: https://docs.gitlab.com/ee/ci/jobs/.

[39] GitLab. *Repository*. Wednesday 17th April, 2024. URL: https://docs.gitlab.com/ee/user/project/repository/.

[40] GitLab. *GitLab Runners*. Wednesday 17th April, 2024. URL: https://docs.gitlab.com/runner/.

[41] GitLab. *Parallel jobs in pipelines*. Wednesday 17th April, 2024. URL: https://docs.gitlab.com/ee/ci/yaml/.

[42] AMD. *Installation Requirements*. Wednesday 17th April, 2024. URL: https://docs.amd.com/r/en-US/ug1144-petalinux-tools-reference-guide/Installation-Requirements.

[43] CERN. *Petalinux Template*. Wednesday 17th April, 2024. URL: https://gitlab.cern.ch/soc/petalinux-template.

[44] CERN. *CentOS ROOTFS*. Wednesday 17th April, 2024. URL: https://gitlab.cern.ch/soc/centos-rootfs/-/tree/master.

[45] *JTAG*. Wednesday 17th April, 2024. URL: https://en.wikipedia.org/wiki/QEMU.

[46] *Intelligent Platform Management Interface*. Wednesday 17th April, 2024. URL: https://en.wikipedia.org/wiki/Intelligent_Platform_Management_Interface.

[47] *Power distribution unit*. Wednesday 17th April, 2024. URL: https://en.wikipedia.org/wiki/Power_distribution_unit.

[48] *dnsmasq*. Wednesday 17th April, 2024. URL: https://en.wikipedia.org/wiki/Dnsmasq.

[49] *Network Time Protocol*. Wednesday 17th April, 2024. URL: https://en.wikipedia.org/wiki/Network_Time_Protocol.

[50] GitLab. *Caching in GitLab CI/CD*. Wednesday 17th April, 2024. URL: https://docs.gitlab.com/ee/ci/caching/.

[51] Yocto. *Enable sstate cache*. Wednesday 17th April, 2024. URL: https://wiki.yoctoproject.org/wiki/Enable_sstate_cache.

Appendix 1 – Non-Exclusive License for Reproduction and Publication of a Graduation Thesis[1]

I Kareen Arutjunjan

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis "A Parallel CI/CD Framework for Building and Testing Multiple Boards with System on Chip", supervised by Petr Žejdl and Siim Vene
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

14.05.2024

[1]The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.