

Project Report
On
Composing Disk and Container Image of Linux on
RISC-V



Submitted
In partial fulfilment
For the award of the Degree of

PG-Diploma in Embedded Systems and Design
(PG-DESD)

Under the esteemed guidance of
Mr. Surendra Billa
HPC Tech, C-DAC, ACTS (Pune)

Submitted By

Sanket Ghanshyam Meshram 230940130051

Bhojanapalli Shanmukha Sai Surya Teja 230940130016

Sanjaya Yadav 230940130050

Harish Bhaskare 230940130026

Centre for Development of Advanced Computing (C-DAC), ACTS
(Pune- 411008)

Acknowledgement

This is to acknowledge our indebtedness to our Project Guide, **Mr. Surendra Billa** C-DAC ACTS, Pune for her constant guidance and helpful suggestion for preparing this project **Composing Disk and Container Image for Linux on RISC-V**. We express our deep gratitude towards her for inspiration, personal involvement, constructive criticism that she provided us along with technical guidance during this project.

We take this opportunity to thank Head of the department **Mr. Gaur Sunder** for providing us such a great infrastructure and environment for our overall development.

We express sincere thanks to **Ms. Namrata Ailawar**, Process Owner, for their kind cooperation and extendible support towards the completion of our project.

It is our great pleasure in expressing sincere and deep gratitude towards **Mrs. Risha P R (Program Head)** and **Mrs. Srujana Bhamidi** (Course Coordinator, PG-DESD) for their valuable guidance and constant support throughout this work and help to pursue additional studies.

Also, our warm thanks to **C-DAC ACTS Pune**, which provided us this opportunity to carry out, this prestigious Project and enhance our learning in various technical fields.

Sanket Ghanshyam Meshram 230940130051

Bhojanapalli Shanmukha Sai Surya Teja 230940130016

Sanjaya Yadav 230940130050

Harish Bhaskare 230940130026

ABSTRACT

The project focuses on creating disk and container images tailored for Linux on RISC-V, with a specific emphasis on optimizing them for future-generation embedded systems. By leveraging containerization, the goal is to provide a lightweight and efficient deployment solution, minimizing performance overhead compared to traditional virtual machines while ensuring similar levels of application isolation.

The disk image is crafted to support RISC-V's unique architecture, featuring a customized bootloader configuration with EFI boot support. Additionally, the kernel is tailored to include necessary components for container support, enhancing compatibility and performance. The root file system is managed using dnf, enabling efficient resolution of dependencies and ensuring a streamlined deployment process.

In parallel, the project delves into the intricacies of the boot process on RISC-V, providing insights into the architecture's specific requirements and optimizations. Furthermore, the project explores methods for transferring container images from emulated environments to physical devices, enabling seamless deployment of applications onto RISC-V-based embedded systems.

By providing comprehensive disk and container images optimized for RISC-V, this project contributes to the advancement of containerization technologies in the embedded systems domain. It paves the way for efficient, scalable, and easily deployable solutions on RISC-V architecture, facilitating the development of next-generation embedded applications.

Table of Contents

S. No	Title	Page No.
	Front Page	I
	Acknowledgement	II
	Abstract	III
	Table of Contents	IV
1	Introduction	1-3
1.1	Introduction	1
1.2	Scope of work	2
1.3	Objectives	3
2	Why to Compose Disk and Container Image for Linux on RISC-V	06
3	Methodology/ Techniques	7-23
3.1	Overview	7
3.2	Software Tools, Packages and Utilities	7
3.2.1	RISC-V Toolchain	7-8
3.2.2	U-boot	9
3.2.3	OpenSBI	10
3.2.4	Linux Kernel	12
3.2.5	Rootfs	14
3.2.6	Parted and Kpartx	16
3.2.7	dd	18
3.2.8	Podman	19
3.2.9	Buildah	20

3.3	Methodology	21-23
4	Implementation	23-31
4.1	Disk Image	24
4.1.1	Building RISC-v Toolchain	24
4.1.2	Compiling u-boot	26
4.1.3	Compiling OpenSBI	27
4.1.4	Compiling Linux Kernel with Container Support	28
4.2	Container Image	30-21
5	Results	32-
5.1	Results	34
6	Conclusion	35-36
6.1	Conclusion	
7	References	37
7.1	References	

Chapter 1

Introduction

1.1 Introduction

The emergence of RISC-V as an open and standardized instruction set architecture (ISA) has sparked significant interest in its application across various computing domains, including embedded systems. As RISC-V gains traction in the embedded space, the need for efficient deployment solutions becomes increasingly important. Containerization has emerged as a compelling approach for deploying applications, offering lightweight and scalable environments that can run across different platforms with minimal performance overhead.

This project focuses on composing disk and container images tailored for Linux on RISC-V, with a specific emphasis on future-generation embedded systems. The goal is to provide a comprehensive deployment solution that optimizes performance and resource utilization while maintaining the flexibility and portability of containerized applications.

The project leverages U-Boot as the bootloader, known for its flexibility and robustness in booting various operating systems on embedded devices. OpenSBI is used as the runtime, ensuring proper initialization and management of software on RISC-V platforms. The kernel is customized to include support for containers and virtualization, enabling efficient deployment and management of containerized applications. Additionally, the project integrates a root file system managed by dnf, a package manager for installing and managing software packages. This root file system is designed to resolve dependencies efficiently, simplifying the deployment process.

In addition to disk image composition, the project explores the intricacies of the boot process on RISC-V, providing insights into the architecture's specific requirements and

optimizations. Understanding these aspects is crucial for developing efficient and reliable boot procedures for RISC-V-based embedded systems.

Furthermore, the project investigates methods for transferring container images from emulated environments to physical devices. This capability is essential for deploying containerized applications onto RISC-V-based embedded systems, ensuring a seamless transition from development to deployment.

By providing optimized disk and container images tailored for RISC-V, this project aims to advance containerization technologies in the embedded systems domain. These images will enable efficient, scalable, and easily deployable solutions on RISC-V, empowering developers to create next-generation embedded applications.

Overall, this project aims to contribute to the advancement of containerization technologies in the embedded systems domain, specifically tailored for the RISC-V architecture. By providing optimized disk and container images, this work lays the foundation for efficient, scalable, and easily deployable solutions on RISC-V, empowering developers to create next-generation embedded applications. This project demonstrates the potential of open-source collaboration and automation in accelerating software development for emerging architectures.

1.2 Scope of Work

The scope of this project encompasses the creation of a comprehensive deployment solution for Linux on RISC-V architecture, specifically targeting future-generation embedded systems. The project will focus on composing optimized disk and container images tailored for RISC-V, with the aim of providing efficient and scalable deployment solutions.

Key aspects of the project include the creation of a disk image that incorporates a customized U-Boot bootloader configuration with EFI boot support. This configuration is

crucial for enabling RISC-V devices to boot Linux and load the kernel. Additionally, the project will integrate OpenSBI as the runtime environment, ensuring proper initialization and management of software on RISC-V platforms.

The Linux kernel will be customized to include support for containers and virtualization, enabling efficient deployment and management of containerized applications. The project will also implement a root file system managed by dnf, a package manager for installing and managing software packages. This root file system will be designed to resolve dependencies efficiently, simplifying the deployment process.

In addition to disk image composition, the project will explore and optimize the boot process on RISC-V architecture. This will involve investigating methods for transferring container images from emulated environments to physical RISC-V devices, facilitating seamless deployment of containerized applications.

Performance testing and compatibility checks will be conducted to ensure that the composed disk and container images meet the requirements of future-generation embedded systems running on RISC-V architecture. Finally, the project will document the process, findings, and optimizations made during the project to contribute knowledge and insights to the RISC-V and containerization communities.

1.3 Objective

- **Compose Optimized Disk Image:** Create a disk image tailored for RISC-V architecture that includes a customized U-Boot bootloader configuration with EFI boot support, ensuring compatibility with a wide range of RISC-V devices.
- **Integrate OpenSBI Runtime:** Incorporate OpenSBI as the runtime environment, ensuring proper initialization and management of software on RISC-V platforms, enhancing system stability and performance.
- **Customize Kernel for Container Support:** Modify the Linux kernel to include support for containers and virtualization, enabling efficient deployment and

management of containerized applications on RISC-V devices.

- **Implement dnf-based Root File System:** Integrate a root file system managed by dnf, enabling efficient resolution of dependencies and simplifying the deployment process of software packages on RISC-V devices.
- **Explore Boot Process Optimization:** Investigate and optimize the boot process on RISC-V architecture, ensuring efficient and reliable boot procedures for RISC-V-based embedded systems.
- **Develop Image Transfer Mechanism:** Explore methods for transferring container images from emulated environments to physical RISC-V devices, facilitating seamless deployment of containerized applications.
- **Validate Performance and Compatibility:** Conduct performance testing and compatibility checks to ensure that the composed disk and container images meet the requirements of future-generation embedded systems running on RISC-V architecture.
- **Document and Share Findings:** Document the process, findings, and optimizations made during the project to contribute knowledge and insights to the RISC-V and containerization communities.

Chapter 2

Why to Compose Container and Disk Image for Linux on RISC-V?

One of the current limitations in deploying Linux on RISC-V architecture is the lack of optimized deployment solutions tailored for future-generation embedded systems. Existing solutions may not fully leverage the capabilities of RISC-V or may suffer from performance overhead and compatibility issues. To overcome this limitation, this project focuses on creating optimized disk and container images specifically designed for RISC-V, ensuring efficient deployment and management of applications.

Another limitation is the complexity of the boot process on RISC-V, which can impact system stability and performance. By integrating U-Boot as the bootloader and OpenSBI as the runtime environment, the project aims to simplify and optimize the boot process, ensuring proper initialization and management of software on RISC-V platforms. This approach helps overcome the limitation of complex boot procedures and enhances system stability and performance.

Furthermore, current deployment solutions may lack robust support for containerization on RISC-V, limiting the ability to efficiently deploy and manage containerized applications. To address this limitation, the project customizes the Linux kernel to include support for containers and virtualization, enabling efficient deployment and management of containerized applications on RISC-V devices. This approach enhances

compatibility and scalability, overcoming the limitation of limited containerization support on RISC-V.

Additionally, the project aims to simplify the deployment process of software packages on RISC-V devices. By implementing a root file system managed by dnf, the project streamlines the process of resolving dependencies and installing software packages. This simplification is expected to reduce complexity and improve the efficiency of deployment workflows, making it easier for developers to deploy applications on RISC-V devices.

Overall, by addressing these limitations through optimized disk and container images, simplified boot processes, enhanced containerization support, and streamlined deployment workflows, the project aims to provide a comprehensive deployment solution for Linux on RISC-V architecture, tailored for future-generation embedded systems

Furthermore, the project seeks to contribute to the RISC-V and containerization communities by documenting the process, findings, and optimizations made during the project. By sharing this knowledge and insights, the project aims to advance the state of the art in these domains and foster collaboration among developers and researchers. This contribution is expected to benefit the broader community working on RISC-V and containerization technologies, ultimately leading to the development of more efficient and reliable embedded systems solutions.

Chapter 3

Methodology and Techniques

3.1 Overview

The methodology for this project was meticulously planned and executed to ensure the successful composition of disk and container images for Linux on RISC-V architecture, tailored for future-generation embedded systems. The project began with a thorough analysis of the requirements, including hardware specifications and software dependencies, to develop a comprehensive project plan. This plan outlined the tasks, timelines, and resources needed for the project's successful completion.

The system architecture was designed with a focus on integrating U-Boot as the bootloader, OpenSBI as the runtime environment, and a customized Linux kernel with support for containers and virtualization. This design consideration was crucial for ensuring compatibility, scalability, and performance optimization. The disk image was then composed to include necessary drivers and configurations for efficient booting on RISC-V devices.

3.2 Software Tools, Packages and Utilities

3.2.1) RISC-V Toolchain

The toolchain for this project primarily consisted of the RISC-V GNU Compiler Toolchain, which includes the GNU Compiler Collection (GCC) with support for RISC-V architecture. This toolchain provided the necessary compilers and libraries required for compiling the U-Boot bootloader, OpenSBI, the Linux kernel, and other software components for RISC-V architecture.

The RISC-V GNU Compiler Toolchain includes the following components:

- **GCC (GNU Compiler Collection):** GCC provides a suite of compilers for various programming languages, including C, C++, and assembly. For this project, GCC was used to compile the source code for U-Boot, OpenSBI, the Linux kernel, and other software components to generate executable binaries for RISC-V architecture.
- **Binutils:** Binutils is a collection of binary utilities, including an assembler, linker, and other tools for manipulating binary files. Binutils was used in conjunction with GCC to assemble and link the compiled code into executable binaries for RISC-V architecture.
- **Libraries:** The toolchain includes standard C libraries (e.g., glibc) and other libraries required for compiling and linking software for RISC-V architecture. These libraries provide essential functions and utilities that are used by the compiled code.
- **Cross-Compiler:** The toolchain also includes a cross-compiler for RISC-V architecture, which allows compiling software on a host system (e.g., x86) for execution on a target RISC-V platform. This cross-compiler ensures that the compiled binaries are compatible with the RISC-V instruction set architecture (ISA).
- **Debugging and Profiling Tools:** GCC also includes debugging and profiling tools, such as GDB (GNU Debugger), which were used for debugging and profiling the compiled software components. These tools helped identify and fix issues in the code, ensuring the reliability and performance of the deployment solution.

Overall, the RISC-V GNU Compiler Toolchain provided a comprehensive set of compilers and libraries necessary for compiling and building the software components for Linux on RISC-V architecture. The toolchain ensured compatibility, efficiency, and

reliability in the development process, contributing to the successful completion of the project.

3.2.2) u-boot

U-Boot is a widely used open-source bootloader that supports multiple architectures, including RISC-V. It provides a flexible and customizable boot environment for embedded systems, allowing developers to initialize hardware, load the operating system kernel, and perform other essential boot tasks.

- **Features:** U-Boot offers a range of features, including support for various file systems, network booting protocols, and hardware initialization routines. It provides a command-line interface (CLI) for interactive control and configuration during the boot process. U-Boot supports the use of boot scripts, which are text files containing a series of commands to be executed during the boot process. This allows for automation and customization of the boot sequence.
- **Compilation and Configuration:** U-Boot can be compiled for different architectures, including RISC-V, using the appropriate cross-compiler toolchain. The configuration process involves selecting the desired features and options using the `make menuconfig` command. The configuration options include enabling EFI support, specifying the target architecture, and configuring hardware-specific settings such as memory layout and boot device selection.
- **Integration with OpenSBI:** U-Boot can be integrated with OpenSBI, which is a reference implementation of the RISC-V Supervisor Binary Interface (SBI). OpenSBI provides a runtime environment for executing software on RISC-V platforms, including handling platform-specific initialization and management tasks. Integrating U-Boot with OpenSBI ensures compatibility and seamless operation on RISC-V platforms, enabling a smooth boot process and efficient management of software.

- **Usage and Deployment:** Once compiled and configured, U-Boot can be deployed to the target RISC-V platform using various methods, such as flashing the bootloader to a storage device or loading it into memory using a JTAG debugger. U-Boot can be used in a variety of embedded systems, including IoT devices, networking equipment, and industrial control systems, providing a reliable and versatile boot solution for RISC-V-based applications.
- **Community and Support:** U-Boot is supported by a vibrant community of developers and users who contribute to its development and provide support through forums, mailing lists, and documentation. This community-driven approach ensures that U-Boot remains up-to-date and well-maintained, with support for new features and hardware platforms.

Overall, U-Boot is a powerful and flexible bootloader that provides essential boot functionality for RISC-V-based embedded systems. Its customizable nature and extensive feature set make it a popular choice for developers looking to boot their RISC-V devices reliably and efficiently.

3.2.3) OpenSBI

OpenSBI is an open-source implementation of the RISC-V Supervisor Binary Interface (SBI), which provides a standard interface for platform-specific initialization and management tasks in RISC-V systems. OpenSBI is designed to be architecture-agnostic, making it suitable for use on various RISC-V platforms.

- **Purpose and Functionality:** OpenSBI serves as a runtime environment for executing software on RISC-V platforms, including bootloaders, operating systems, and firmware. It handles platform-specific initialization tasks, such as setting up the memory map, initializing devices, and configuring interrupts. OpenSBI also provides services for managing the execution environment, such as querying platform information, handling interrupts, and interacting with the

platform console.

- **Features:** OpenSBI offers a range of features, including support for different RISC-V privilege levels (M-mode, S-mode, U-mode), platform-specific initialization routines, and runtime services for managing the execution environment. OpenSBI supports the SBI specification, which defines a standard interface for interacting with platform-specific functionality. This allows software running in higher privilege levels (e.g., S-mode) to communicate with OpenSBI and perform platform-specific tasks.
- **Integration with Bootloaders and Operating Systems:** OpenSBI is typically integrated with bootloaders, such as U-Boot, to provide a runtime environment for executing the bootloader code. It can also be used with operating systems, such as Linux, to handle platform-specific initialization tasks and provide runtime services to the operating system kernel.
- **Configuration and Customization:** OpenSBI can be configured and customized for specific RISC-V platforms and use cases. Configuration options include specifying the memory map, device initialization routines, and platform-specific parameters. OpenSBI provides a flexible configuration mechanism, allowing developers to tailor the runtime environment to meet the requirements of their specific platform or application.
- **Community and Support:** OpenSBI is supported by a community of developers and users who contribute to its development and provide support through forums, mailing lists, and documentation. This community-driven approach ensures that OpenSBI remains up-to-date and well-maintained, with support for new features and platforms.

Overall, OpenSBI plays a crucial role in the RISC-V ecosystem, providing a standard interface for platform-specific initialization and management tasks. Its architecture-agnostic design, customizable nature, and community support make it a valuable tool for

developers working with RISC-V-based systems.

3.2.4) Linux Kernel

The Linux kernel is the core component of the Linux operating system, responsible for managing hardware resources, providing essential system services, and facilitating communication between hardware and software components. For the RISC-V architecture, the Linux kernel has been ported and optimized to run on RISC-V-based platforms, offering support for various features and peripherals.

- **Porting and Optimization:** The Linux kernel for RISC-V has been ported and optimized to support the RISC-V instruction set architecture (ISA) and the specific features of RISC-V-based platforms. This involves adapting the kernel codebase to work efficiently on RISC-V hardware, including handling interrupts, memory management, and device drivers.
- **Features and Functionality:** The Linux kernel for RISC-V provides a wide range of features and functionality, including support for symmetric multiprocessing (SMP), virtual memory management, file systems, networking, and device drivers. It offers a complete operating system environment for running user-space applications on RISC-V platforms. The kernel also includes support for containerization and virtualization technologies, enabling the deployment and management of containerized applications and virtual machines on RISC-V hardware.
- **Configuration and Customization:** The Linux kernel can be configured and customized for specific RISC-V platforms and use cases using the kernel configuration system (e.g., make menuconfig). This allows developers to enable or disable kernel features, select device drivers, and optimize the kernel for performance and resource usage. Additionally, developers can modify the kernel source code to add support for new hardware peripherals, implement custom

features, or optimize kernel behavior for specific applications.

- **Integration with Bootloader and OpenSBI:** The Linux kernel is typically loaded and executed by a bootloader, such as U-Boot, on RISC-V platforms. The bootloader initializes the hardware and passes control to the kernel, which then initializes the rest of the system and launches the user-space environment. On platforms using OpenSBI, the Linux kernel interacts with OpenSBI to perform platform-specific initialization tasks and access runtime services provided by OpenSBI. This integration ensures compatibility and seamless operation of the Linux kernel on RISC-V platforms.
- **Community and Development:** The development of the Linux kernel for RISC-V is driven by a community of developers and contributors who collaborate on the upstream Linux kernel source tree. This community-driven approach ensures that the kernel remains up-to-date, well-maintained, and compatible with new RISC-V hardware platforms and features. Developers can contribute to the Linux kernel for RISC-V by submitting patches, bug reports, and feature requests to the relevant mailing lists and forums. This active community involvement helps improve the quality and reliability of the kernel and fosters innovation in the RISC-V ecosystem.

Overall, the Linux kernel for RISC-V architecture provides robust support for containerization, enabling developers to deploy and manage containerized applications efficiently on RISC-V platforms. By leveraging containerization technologies, developers can improve deployment flexibility, resource utilization, and application isolation on RISC-V-based systems.

3.2.5) Root file System

The root file system (RootFS) is the top-level directory structure of a Linux-based operating system. It contains essential system files, libraries, and configurations required

for the operating system to function. The RootFS is mounted at the root (/) directory of the file system hierarchy and is the starting point for all file system paths on the system.

Components of RootFS:

- **Binaries:** RootFS contains executable files (binaries) for essential system utilities and commands, such as ls, cp, mv, and rm, located in the /bin directory.
- **Libraries:** RootFS includes shared libraries required by the binaries to run, located in the /lib and /lib64 directories. These libraries provide functions and resources used by the executables.
- **Configuration Files:** RootFS contains configuration files for system-wide settings and services, located in directories such as /etc. These files define system behavior and settings.
- **Device Files:** RootFS includes special device files located in the /dev directory, representing hardware devices and providing access to them through the file system.
- **Kernel Modules:** RootFS may contain kernel modules (drivers) located in the /lib/modules directory, providing support for various hardware devices and functionalities.
- **Temporary Files:** RootFS includes temporary files located in the /tmp directory, used by applications for temporary storage.

RootFS Types:

- **Initramfs (Initial RAM File System):** Initramfs is a temporary root file system loaded into memory during the boot process. It contains essential files and modules required to mount the actual root file system, which may be located on a different device or in a different format.
- **RootFS on Disk:** The root file system can also be located on a disk partition, such as an SSD or HDD. It contains the complete set of files and directories required

for the operating system to run.

Creating and Customizing RootFS:

RootFS can be created and customized using tools like debootstrap (for Debian-based systems) or buildroot (for custom embedded systems). These tools allow users to select the packages and configurations to include in the RootFS. Busybox was used in this project.

RootFS can also be customized manually by adding or removing files and directories as needed. This approach requires a good understanding of the Linux file system hierarchy and system dependencies.

Mounting RootFS: During the boot process, the RootFS is mounted by the kernel as the initial file system. The kernel first mounts the initramfs to set up the environment and then switches to the actual root file system.

The RootFS is a critical component of the Linux operating system, providing the necessary files and configurations for the system to boot and operate correctly. Understanding the structure and contents of the RootFS is essential for managing and customizing Linux-based systems.

3.2.6) Parted and Kpartx

Parted: Parted is a command-line utility for creating, resizing, and managing disk partitions. It supports various disk partitioning schemes, including MBR (Master Boot Record) and GPT (GUID Partition Table), making it a versatile tool for disk management tasks on Linux systems.

Key Features of Parted:

- **Create Partitions:** Parted allows users to create new disk partitions, specifying

the partition type, size, and file system format.

- **Resize Partitions:** Parted can resize existing partitions, allowing users to increase or decrease the size of a partition without losing data.
- **Move and Copy Partitions:** Parted can move and copy partitions to different locations on the disk, adjusting the partition table accordingly.
- **Delete Partitions:** Parted can delete partitions, removing them from the partition table and freeing up the disk space.
- **Check and Repair Partitions:** Parted can check the integrity of partition tables and partitions, and repair any errors found.
- **Usage of Parted:** Parted is used from the command line, with options and arguments specified to perform specific partitioning tasks. For example, to create a new partition, the `mkpart` command is used with parameters specifying the start and end of the partition, the partition type, and the file system format. Parted also provides an interactive mode, where users can issue commands and see the effects on the disk partition table in real-time.
- **Caution with Parted:** While parted is a powerful tool for managing disk partitions, it is important to use it with caution, as incorrect use can result in data loss. It is recommended to back up important data before making any changes to disk partitions.

KPartx: KPartx is a tool that is used to create device mappings for the partitions of a disk. It is part of the Linux multipath-tools package and is commonly used in conjunction with tools like Parted to manage disk partitions.

Key Features of KPartx:

- **Create Device Mappings:** KPartx creates device mappings for each partition on a disk, allowing users to access the partitions as individual block devices.
- **Automatic Detection:** KPartx can automatically detect the partitions on a disk and

create the corresponding device mappings, simplifying the process of managing disk partitions.

- **Integration with Parted:** KPartx is often used in conjunction with Parted to manage disk partitions. After using Parted to create or resize partitions, KPartx can be used to create the necessary device mappings to access the partitions.
- **Usage of KPartx:** KPartx is typically used from the command line, with options and arguments specified to create device mappings for disk partitions. For example, the `kpartx -a /dev/sdb` command creates device mappings for the partitions on the `/dev/sdb` disk. KPartx can also be used to remove device mappings when they are no longer needed, using the `kpartx -d /dev/sdb` command to delete the mappings for the `/dev/sdb` disk.
- **Integration with Other Tools:** KPartx is commonly used in conjunction with other disk management tools, such as Parted and LVM (Logical Volume Manager), to manage disk partitions and volumes. It provides a convenient way to access and manage partitions on a disk.

3.2.7) dd

The `dd` (data duplicator) utility is a command-line tool used for copying and converting data. It is commonly used for tasks such as creating disk images, copying data between devices, and formatting storage media. `dd` is a versatile tool that is available on most Unix-like operating systems, including Linux. The `dd` command reads data from an input file or device and writes it to an output file or device, with various options available to control the behaviour of the command.

Key Features:

- **Copying Data:** `dd` can be used to copy data between files, devices, or partitions.

For example, to copy the contents of one disk to another, you can use the command: **dd if=/dev/sda of=/dev/sdb**

- **Creating Disk Images:** dd can be used to create an image of a disk or partition. For example, to create an image of a disk and save it to a file, you can use the command:
dd if=/dev/sda of=disk.img
- **Converting Data Formats:** dd can be used to convert data between different formats. For example, you can use dd to convert a file from ASCII to EBCDIC format.
- **Common Options:**
 1. **if:** Specifies the input file or device.
 2. **of:** Specifies the output file or device.
 3. **bs:** Specifies the block size to use for reading and writing data.
 4. **count:** Specifies the number of blocks to copy or convert.
 5. **seek:** Specifies the number of blocks to skip in the output file before writing data.
- **Caution:** dd can be a powerful tool, but it is also dangerous if not used carefully. It does not provide any safeguards against overwriting important data, so it is important to double-check the command parameters before executing dd.

Overall, dd is a powerful and versatile tool for copying and converting data. It is commonly used for disk imaging, data recovery, and other low-level data manipulation tasks. However, it should be used with caution, as it can easily overwrite important data if not used correctly.

3.2.8) Podman:

Podman is a container management tool that provides a Docker-compatible interface for managing OCI (Open Container Initiative) containers and pods. It is designed to be a

daemonless, lightweight alternative to Docker, allowing users to run containers without requiring a separate container daemon process.

Key Features of Podman:

- **Daemonless Architecture:** Unlike Docker, which requires a daemon process (dockerd) to manage containers, Podman operates as a standalone tool without a central daemon. This makes Podman more lightweight and easier to manage.
- **Compatibility with Docker:** Podman provides a Docker-compatible command-line interface (CLI), allowing users familiar with Docker to easily switch to Podman. Podman supports Docker commands and options, making it a drop-in replacement for Docker in many cases.
- **Rootless Containers:** Podman supports running containers as a non-root user, providing improved security and isolation. This allows users to run containers without requiring root privileges, reducing the risk of container escapes and security vulnerabilities.
- **Pod Support:** Podman supports the concept of pods, which are groups of containers that share the same network and storage namespaces. Pods allow users to manage related containers together, similar to Kubernetes pods.
- **Buildah Integration:** Podman integrates with Buildah, a tool for building OCI container images. This allows users to build container images directly from Dockerfiles or from scratch using Buildah, then run the resulting images with Podman.
- **Usage of Podman:** Podman is used from the command line, with commands and options specified to manage containers and pods.

3.2.9) Buildah

Buildah is a command-line tool for building Open Container Initiative (OCI) compliant container images. It allows users to create and manage container images without

requiring a separate container runtime or daemon, making it a flexible and lightweight alternative to traditional container build tools like Dockerfile.

Key Features of Buildah:

- **Daemonless Build Process:** Buildah operates as a standalone tool without requiring a central daemon process. This simplifies the container image build process and reduces the complexity of managing container builds.
- **Build from Scratch or Dockerfile:** Buildah supports building container images from scratch or using existing Dockerfiles. This allows users to choose the approach that best suits their needs and workflow.
- **Multi-stage Builds:** Buildah supports multi-stage builds, allowing users to create intermediate images and reuse build artifacts across different stages of the build process. This helps to optimize the size and performance of the final container image.
- **Customizable Build Process:** Buildah provides a flexible and customizable build process, allowing users to specify build options, environment variables, and other configuration settings to customize the build process to their requirements.
- **Integration with Podman:** Buildah integrates with Podman, allowing users to build container images using Buildah and then run the resulting images with Podman. This provides a seamless workflow for building and running containerized applications.
- **Usage of Buildah:** Buildah is used from the command line, with commands and options specified to manage the build process.

3.3 Methodology

The methodology for the project involving the compilation of U-Boot, OpenSBI, the kernel with container support, and the creation of a root file system (RootFS) and disk image for RISC-V systems can be outlined as follows:

- **Setting Up the Build Environment:** Install necessary tools and dependencies for building U-Boot, OpenSBI, the kernel, and the RootFS. This may include compilers, build systems, and libraries required for RISC-V development.
- **Compiling U-Boot:** Obtain the U-Boot source code from the official repository or a specific branch/tag. Configure U-Boot for the target RISC-V platform, enabling any required features and settings. Compile U-Boot using the appropriate cross-compilation toolchain for RISC-V.
- **Compiling OpenSBI:** Obtain the OpenSBI source code from the official repository or a specific branch/tag. Configure OpenSBI for the target RISC-V platform, specifying the desired boot options and settings. Compile OpenSBI using the same cross-compilation toolchain used for U-Boot.
- **Integrating OpenSBI with U-Boot:** Modify the U-Boot configuration to include support for booting using OpenSBI. Configure U-Boot to load OpenSBI as part of the boot process. Integrate OpenSBI into the U-Boot build process, ensuring that it is included in the final U-Boot image.
- **Compiling the Kernel with Container Support:** Obtain the Linux kernel source code from the official repository or a specific branch/tag. Configure the kernel for the target RISC-V platform, enabling support for containers and any other required features. Compile the kernel using the cross-compilation toolchain, generating a kernel image (vmlinuz).
- **Creating the Root File System (RootFS):** Choose a base distribution for the RootFS, such as Fedora, Debian, or Buildroot. Customize the RootFS configuration to include only the necessary components and packages for your application. Build the RootFS using the chosen build system, ensuring that it includes the required libraries, utilities, and configurations for container support.
- **Forming the Disk Image:** Combine the compiled U-Boot, OpenSBI, kernel image, and RootFS into a single disk image. Create a disk image file format suitable for the target RISC-V platform, such as a raw disk image (image.img).

Use tools like `dd` or `qemu-img` to create the disk image, ensuring that it is properly formatted and contains the necessary components for booting and running the system.

- **Testing and Validation:** Test the compiled disk image on the target RISC-V platform using an emulator or real hardware. Verify that the system boots successfully, the kernel initializes correctly, and the RootFS is mounted and accessible. Test any additional functionality, such as container support, to ensure that it works as expected.
- **Documentation and Reporting:** Document the entire build process, including any custom configurations or settings. Provide instructions for building and deploying the disk image on other RISC-V systems. Prepare a detailed report summarizing the methodology, results, and any issues encountered during the project.

Chapter 4

Implementation

4.1) Disk Image:

Prerequisites: Fedora Operating System , Visionfive Starfive 2 board.

4.1.1) Building Toolchain:

1. **Step 1:** git clone <https://github.com/riscv/riscv-gnu-toolchain>
2. **Step 2:** sudo yum install autoconf automake python3 libmpc-devel mpfr-devel gmp-devel gawk bison flex texinfo patchutils gcc gcc-c++ zlib-devel expat-devel

(above command must be in a single line or use '\')

3. **Step 3:** ./configure --prefix=/path to the destination/riscv --with-arch=rv64gc
4. **Step 4:** make linux
5. **Step 5:** goto terminal vi .bashrc add following line anywhere in the file

export PATH="/path to directory where installed/bin:\$PATH"

Description:

- **Step 1:** Clone the RISC-V GNU Toolchain Repository. Use the git clone command to clone the RISC-V GNU Toolchain repository from GitHub. This repository contains the source code for the RISC-V GCC compiler and related tools.

Example: git clone <https://github.com/riscv/riscv-gnu-toolchain>

- **Step 2:** Install Required Dependencies. Use the package manager (in this case, yum) to install the required dependencies for building the toolchain. The dependencies include autoconf, automake, python3, libmpc-devel, mpfr-devel, gmp-devel,

gawk, bison, flex, texinfo, patchutils, gcc, gcc-c++, zlib-devel, and expat-devel.

Example: `sudo yum install autoconf automake python3 libmpc-devel mpfr-devel gmp-devel gawk bison flex texinfo patchutils gcc gcc-c++ zlib-devel expat-devel`

- Step 3: Configure the Toolchain. Run the configure script in the RISC-V GNU Toolchain directory to configure the build settings. Specify the installation prefix using the `--prefix` option (e.g., `/opt/riscv`) to specify the path where the toolchain will be installed. Use the `--with-arch` option to specify the RISC-V architecture and ABI (Application Binary Interface) to target. Here, `rv64gc` indicates a 64-bit RISC-V architecture with the `gc` (IMAFD) extension.

Example: `./configure --prefix=/opt/riscv --with-arch=rv64gc`

Step 4: Build the Toolchain . Use the `make` command to build the RISC-V GNU Toolchain for Linux. The `make` command will compile the GCC compiler, `binutils`, and other tools necessary for building RISC-V binaries.

Example: `make linux`

After completing these steps, you should have a compiled RISC-V GNU Toolchain installed in the specified prefix directory (`/opt/riscv` in this case). This toolchain can be used to compile and build RISC-V binaries for the specified architecture and ABI.

4.1.2) U-boot

1. `git clone https://github.com/starfive-tech/u-boot.git`
2. `cd` u-boot
`git checkout -b JH7110_VisionFive2_devel origin/JH7110_VisionFive2_devel`
`git pull`
3. `make <Configuration_File> ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu-`

Configuration_File: For VisionFive 2, the file is **starfive_visionfive2_defconfig**.

4. make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu-
5. ls

Output: There will be these 3 files generated after compilation inside the u-boot directory:

- u-boot.bin
- arch/riscv/dts/starfive_visionfive2.dtb
- spl/u-boot-spl.bin

Clone the U-Boot Repository: Use git clone to clone the U-Boot repository from the specified URL.

Navigate to the U-Boot Directory: Use cd to change the current directory to the U-Boot directory.

Checkout to a Specific Branch: Use git checkout -b to create and switch to a new branch based on the specified branch in the repository.

Pull the Latest Changes: Use git pull to fetch and apply the latest changes from the remote repository to the current branch.

Compile U-Boot with Specific Configuration: Use make with the specified configuration file (starfive_visionfive2_defconfig for VisionFive 2) to compile U-Boot for the RISC-V architecture using the specified cross-compiler (riscv64-linux-gnu-).

Compile U-Boot without Configuration: Use make without a configuration file to compile U-Boot for the RISC-V architecture using the specified cross-compiler.

List the Generated Files: Use ls to list the files generated after compilation in the U-Boot directory, which should include u-boot.bin, arch/riscv/dts/starfive_visionfive2.dtb, and spl/u-boot-spl.bin.

4.1.3) OpenSBI Compilation and Integration

1. git clone <https://github.com/starfive-tech/opensbi.git>
2. cd opensbi
3. make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- PLATFORM=generic FW_PAYLOAD_PATH={U-BOOT_PATH}/u-boot.bin FW_FDT_PATH={U-BOOT_PATH}/arch/riscv/dts/starfive_visionfive2.dtb FW_TEXT_START=0x40000000

After compilation, the file fw_payload.bin will be generated in the directory opensbi/build/platform/generic/firmware and the size is larger than 2M.

4. git clone <https://github.com/starfive-tech/Tools>
5. cd Tools
6. git checkout master
7. git pull
8. cd spl_tool/
9. make
10. ./spl_tool -c -f {U-BOOT_PATH}/spl/u-boot-spl.bin

You will see a new file named u-boot-spl.bin.normal.out generated under {U-BOOT_PATH}/spl.

11. cd Tools/uboot_its
12. cp {OPENSBI_PATH}/build/platform/generic/firmware/fw_payload.bin ./
13. {U-BOOT_PATH}/tools/mkimage -f visionfive2-uboot-fit-image.its -A riscv -O u-boot -T firmware visionfive2_fw_payload.img

You will see a new file named **visionfive2_fw_payload.img** generated.

4.1.4) Linux Kernel with Container Support

1. git clone <https://github.com/starfive-tech/linux.git>
2. cd linux
git checkout -b JH7110_VisionFive2_devel origin/JH7110_VisionFive2_devel
git pull
3. make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu-starfive_visionfive2_defconfig
(this is a single instruction and space is there between gnu- and starfive)
4. make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu-menuconfig
5. Do the following configurations in the menuconfig menu

```
General setup --->
[*] POSIX Message Queues
BPF subsystem --->
  [*] Enable bpf() system call (<span style="color:green;">Optional</span>)
[*] Control Group support --->
  [*] Memory controller
  [*] Swap controller (<span style="color:green;">Optional</span>)
  [*] Swap controller enabled by default (<span style="color:green;">Optional</span>)
  [*] IO controller (<span style="color:green;">Optional</span>)
  [*] CPU controller --->
    [*] Group scheduling for SCHED_OTHER (<span style="color:green;">Optional</span>)
    [*] CPU bandwidth provisioning for FAIR_GROUP_SCHED (<span style="color:green;">Optional</span>)
    [*] Group scheduling for SCHED_RR/FIFO (<span style="color:green;">Optional</span>)
  [*] PIDs controller (<span style="color:green;">Optional</span>)
  [*] Freezer controller
  [*] HugeTLB controller (<span style="color:green;">Optional</span>)
  [*] Cpuset controller
    [*] Include legacy /proc/<pid>/cpuset file (<span style="color:green;">Optional</span>)
  [*] Device controller
  [*] Simple CPU accounting controller
  [*] Perf controller (<span style="color:green;">Optional</span>)
  [*] Support for eBPF programs attached to cgroups (<span style="color:green;">Optional</span>)
[*] Namespaces support
```

```
[*] UTS namespace
[*] IPC namespace
[*] User namespace (<span style="color:green;">Optional</span>)
[*] PID Namespaces
[*] Network namespace
General architecture-dependent options --->
[*] Enable seccomp to safely execute untrusted bytecode (<span style="color:green;">Optional</span>)
[*] Enable the block layer --->
[*] Block layer bio throttling support (<span style="color:green;">Optional</span>)
[*] Networking support --->
Networking options --->
[*] Network packet filtering framework (Netfilter) --->
[*] Advanced netfilter configuration
[*] Bridged IP/ARP packets filtering
Core Netfilter Configuration --->
[*] Netfilter connection tracking support
[*] Network Address Translation support
[*] MASQUERADE target support
[*] Netfilter Xtables support
[*] "addrtype" address type match support
[*] "conntrack" connection tracking match support
[*] "ipvs" match support (<span style="color:green;">Optional</span>)
[*] "mark" match support
[*] IP virtual server support ---> (<span style="color:green;">Optional</span>)
[*] TCP load balancing support (<span style="color:green;">Optional</span>)
[*] UDP load balancing support (<span style="color:green;">Optional</span>)
[*] round-robin scheduling (<span style="color:green;">Optional</span>)
[*] Netfilter connection tracking (<span style="color:green;">Optional</span>)
IP: Netfilter Configuration --->
[*] IP tables support
[*] Packet filtering
[*] iptables NAT support
[*] MASQUERADE target support
[*] REDIRECT target support (<span style="color:green;">Optional</span>)
[*] 802.1d Ethernet Bridging
[*] VLAN filtering
[*] QoS and/or fair queueing ---> (<span style="color:green;">Optional</span>)
[*] Control Group Classifier (<span style="color:green;">Optional</span>)
[*] L3 Master device support
[*] Network priority cgroup (<span style="color:green;">Optional</span>)
Device Drivers --->
[*] Multiple devices driver support (RAID and LVM) --->
[*] Device mapper support (<span style="color:green;">Optional</span>)
[*] Thin provisioning target (<span style="color:green;">Optional</span>)
[*] Network device support --->
[*] Network core drive support
[*] Dummy net driver support
[*] MAC-VLAN net driver support
[*] IP-VLAN support
[*] Virtual eXtensible Local Area Network (VXLAN)
[*] Virtual ethernet pair device
Character devices --->
*- Enable TTY
*- Unix98 PTY support
[*] Support multiple instances of devpts (option appears if you are using systemd)
File systems --->
[*] Btrfs filesystem support (<span style="color:green;">Optional</span>)
[*] Btrfs POSIX Access Control Lists (<span style="color:green;">Optional</span>)
[*] Overlay filesystem support
Pseudo filesystems --->
[*] HugeTLB file system support (<span style="color:green;">Optional</span>)
Security options --->
[*] Enable access key retention support
```

6. make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- -j\$(nproc)

7. Till this point in arch/riscv/boot directory Image.gz and dts will be generated

vmlinux has to be generated to have a compressed image of the kernel.

8. Make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu-
INSTALL_PATH=/path to a directory zinstall -j\$(nproc)

This will generate the vmlinux ,config file and system map file which have to copied to boot partition.

4.2 Container Image

- 1) Boot the Visionfive2 board or qemu emulated riscv board with fedora image having the container support.
- 2) sudo yum install buildah
- 3) sudo yum install podman
- 4) Vi install.sh

```
#!/bin/bash
echo "Start of Script"
scratchcontainer=$(buildah from scratch)
scratchmnt=$(buildah mount $scratchcontainer)
dnf install --installroot $scratchmnt -y --releasever 38 @buildsys-build --setopt install_weak_deps=false
buildah rename $scratchcontainer fedora38-riscv64g
buildah commit fedora38-riscv64g fedora38-riscv64g-image
buildah unmount fedora38-riscv64g
```

- 5) podman run -it fedora38-riscv64g-image bash

Description:

- Install Buildah and Podman: Use sudo yum install buildah podman to install Buildah and Podman on your system. Buildah is used for building OCI (Open Container Initiative) container images, while Podman is used for running containers.
- Create the Install Script: Create a new script file (e.g., install.sh) and add the provided script content. This script will create a minimal Fedora container image with the necessary packages and configurations.
- Make the Script Executable: Use chmod +x install.sh to make the script executable.
- Run the Install Script: Run the script using ./install.sh to execute the commands inside the script.
- The script will:
 - a) Create a new container from scratch using Buildah. Mount the container's filesystem to a directory for modifications.
 - b) Use dnf (Fedora's package manager) to install the necessary packages and dependencies into the container's filesystem.
 - c) Rename the container to fedora38-riscv64g.
 - d) Commit the container to an image named fedora38-riscv64g-image.

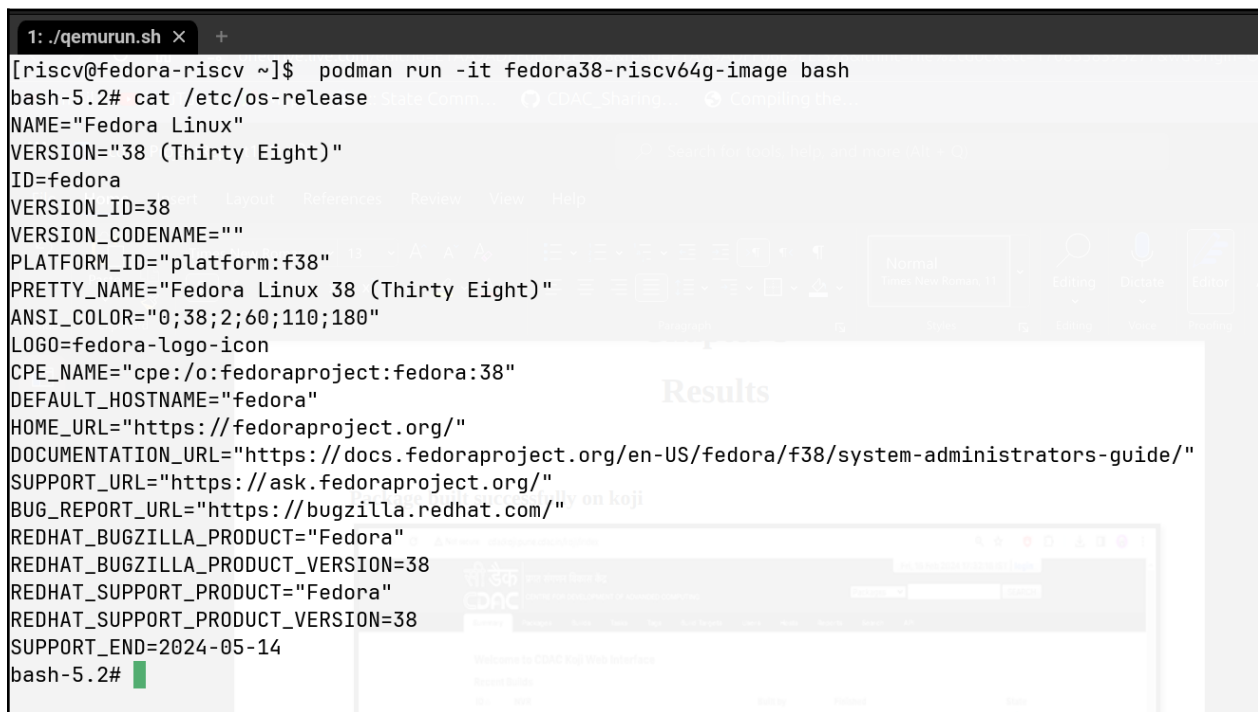
- e) Unmount the container's filesystem.
- f) Run a Container with the Fedora Image:
 - After the image is created, run a container with the newly created Fedora image using **podman run -it fedora38-riscv64g-image bash**.
 - This command starts a new container from the fedora38-riscv64g-image image and opens a bash shell inside the container, allowing interaction with the containerized environment. Explore the Container Environment: Inside the container, you can explore the Fedora environment, install additional packages, and run applications.

By following these steps, you can create a custom Fedora container image with container support and run it on your VisionFive2 board or a QEMU emulated RISC-V board. This allows you to test and develop applications in a containerized environment on your RISC-V platform.

Chapter 5

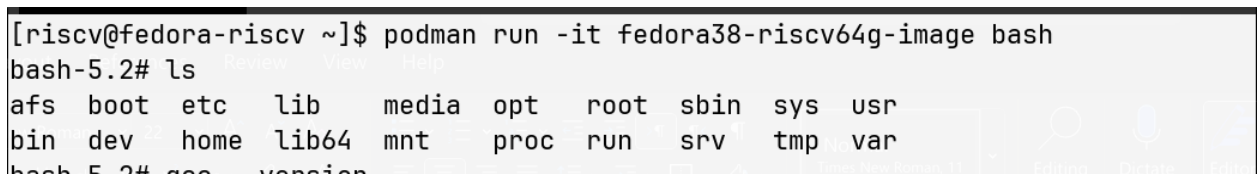
Results

Podman Container Deployed Successfully:



```
1: ./qemurun.sh x +
[riscv@fedora-riscv ~]$ podman run -it fedora38-riscv64g-image bash
bash-5.2# cat /etc/os-release
NAME="Fedora Linux"
VERSION="38 (Thirty Eight)"
ID=fedora
VERSION_ID=38
VERSION_CODENAME=""
PLATFORM_ID="platform:f38"
PRETTY_NAME="Fedora Linux 38 (Thirty Eight)"
ANSI_COLOR="0;38;2;60;110;180"
LOGO=fedora-logo-icon
CPE_NAME="cpe:/o:fedoraproject:fedora:38"
DEFAULT_HOSTNAME="fedora"
HOME_URL="https://fedoraproject.org/"
DOCUMENTATION_URL="https://docs.fedoraproject.org/en-US/fedora/f38/system-administrators-guide/"
SUPPORT_URL="https://ask.fedoraproject.org/"
BUG_REPORT_URL="https://bugzilla.redhat.com/"
REDHAT_BUGZILLA_PRODUCT="Fedora"
REDHAT_BUGZILLA_PRODUCT_VERSION=38
REDHAT_SUPPORT_PRODUCT="Fedora"
REDHAT_SUPPORT_PRODUCT_VERSION=38
SUPPORT_END=2024-05-14
bash-5.2#
```

Fig: Running Container instance 1



```
[riscv@fedora-riscv ~]$ podman run -it fedora38-riscv64g-image bash
bash-5.2# ls
afs  boot  etc  lib  media  opt  root  sbin  sys  usr
bin  dev  home  lib64  mnt  proc  run  srv  tmp  var
bash-5.2#
```

Fig: Running Container instance 2

Disk Image Booted on Starfive Visionfive2

```
Feb 16 3:51 AM
GTKTerm - /dev/ttyUSB0 115200-8-N-1

File Edit Log Configuration Control signals View Help

U-Boot SPL 2021.10 (Dec 27 2023 - 19:53:40 +0800)
LPDDR4: 4G version: g8ad50857.
Trying to boot from SPI

OpenSBI v1.2

      _ _ _ _ _
     / / / / /
    / / / / /
   / / / / /
  / / / / /
 / / / / /
/_/_/_/_/_

Platform Name       : StarFive VisionFive V2
Platform Features   : medeleg
Platform HART Count : 5
Platform IPI Device : acint-mswi
Platform Timer Device : acint-timer @ 4000000Hz
Platform Console Device : uart8250
Platform HSM Device : ---
Platform PMU Device : ---
Platform Reboot Device : pm-reset
Platform Shutdown Device : pm-reset
Platform Suspend Device : ---
Firmware Base       : 0x40000000
Firmware Size        : 392 KB
Firmware RW Offset   : 0x40000
Runtime SBI Version  : 1.0

Domain0 Name        : root
Domain0 Boot HART    : 1
Domain0 HARTs        : 0*,1*,2*,3*,4*
Domain0 Region00     : 0x0000000020000000-0x000000000200ffff M: (I,R,W) S/U: ()
Domain0 Region01     : 0x0000000040000000-0x000000004003ffff M: (R,X) S/U: ()
Domain0 Region02     : 0x0000000040040000-0x000000004007ffff M: (R,W) S/U: ()
Domain0 Region03     : 0x0000000000000000-0xffffffff M: (R,W,X) S/U: (R,W,X)
Domain0 Next Address : 0xffffffff

/dev/ttyUSB0 115200-8-N-1 DTR RTS CTS CD DSR RI
```

Fig: OpenSBI and U-boot Implementation.

```
Feb 16 3:52 AM
GTKTerm - /dev/ttyUSB0 115200-8-N-1

File Edit Log Configuration Control signals View Help

Out: serial
Err: serial
Model: StarFive VisionFive V2
Net: eth0: ethernet@16030000, eth1: ethernet@16040000
switch to partitions #0, OK
mmc1 is current device
found device 1
bootmode flash device 1
406 bytes read in 6 ms (65.4 KiB/s)
Importing environment from mmc1 ...
Can't set block device
Hit any key to stop autoboot: 0
406 bytes read in 5 ms (79.1 KiB/s)
## Warning: defaulting to text format
52414 bytes read in 10 ms (5 MiB/s)
52414 bytes written in 44 ms (1.1 MiB/s)
Retrieving file: /extlinux/extlinux.conf
792 bytes read in 8 ms (96.7 KiB/s)
U-Boot menu
1: Debian GNU/Linux bookworm/sid 5.15.0-starfive
2: Debian GNU/Linux bookworm/sid 5.15.0-starfive (rescue target)
Enter choice: 1: Debian GNU/Linux bookworm/sid 5.15.0-starfive
Retrieving file: /initrd.img-5.15.0-starfive
9271362 bytes read in 487 ms (21.7 MiB/s)
Retrieving file: /vmlinuz-5.15.0-starfive
8482739 bytes read in 373 ms (21.7 MiB/s)
append: root=/dev/mmcblk1p4 rw console=tty0 console=ttyS0,115200 earlycon rootwait stmmaceth=chain_mode:1 selinux=0
Retrieving file: /dtbs/starfive/jh7110-visionfive-v2.dtb
52414 bytes read in 11 ms (4.5 MiB/s)
Uncompressing Kernel Image
Moving Image from 0x40200000 to 0x40200000, end=419b6000
## Flattened Device Tree blob at 48000000
Booting using the fdt blob at 0x48000000
Using Device Tree in place at 0000000048000000, end 000000004800fcdb

Starting kernel ...

clk u2 dw i2c clk core already disabled
clk u2 dw i2c clk anp already disabled

/dev/ttyUSB0 115200-8-N-1 DTR RTS CTS CD DSR RI
```

Fig: Composed Fedora 38 disk image with kernel supporting container.

```
Feb 16 3:46 AM
GTKTerm - /dev/ttyUSB0 115200-8-N-1

done.
Begin: Mounting root file system ... Begin: Running /scripts/local-top ... done.
Begin: Running /scripts/local-premount ... done.
Warning: fsck not present, so skipping root file system
[ 0.264349] EXT4-fs (mmcblk1p4): recovery complete
[ 0.279499] EXT4-fs (mmcblk1p4): mounted filesystem with ordered data mode. Opts: (null). Quota mode: disabled.
done.
Begin: Running /scripts/local-bottom ... done.
Begin: Running /scripts/init-bottom ... done.
[ 9.103844] systemd[1]: System time before build time, advancing clock.
[ 9.249074] systemd[1]: systemd 253.2-614.7.riscv64.fc38 running in system mode (+PAM +AUDIT +SELINUX +APPARMOR +IMA +SMACK +SECCOMP -GCRYPT +GNUTLS +OPENSSL +ACL +BLKID +CURL +ELFUTILS +FIDO2 +IDN2 -IDN -IPTC +KMOD +LIBCRYPTSETUP +LIBDISK +PCRE2 +PWQUALITY +P11KIT +QRENCODE +TPM2 +BZIP2 +LZ4 +XZ +ZLIB +ZSTD +BPF_FRAMEWORK +X86COMMON +UTMP +SYSVINIT default-hierarchy=unified)
[ 9.283429] systemd[1]: Detected architecture riscv64.

Welcome to Fedora Linux 38 (Thirty Eight)!

[ 9.315114] systemd[1]: Hostname set to <fedora-riscv>.
[ 9.327732] systemd[1]: Initializing machine ID from random generator.
[ 9.723976] systemd[1]: bpf-lsm: BPF LSM hook not enabled in the kernel, BPF LSM not supported
[ 9.958357] systemd-sysv-generator[244]: SysV service '/etc/rc.d/init.d/livesys' lacks a native systemd unit file. Automatically generating a unit file for compatibility. Please update package to include a native systemd unit file, in order to make it more safe and robust.
[ 14.209262] random: crng init done
[ 14.244040] zram-generator::generator[246]: modprobe "zram" failed, ignoring: code exit status: 1
[ 14.253770] (sd-execcu[229]): /usr/lib/systemd/system-generators/zram-generator failed with exit status 1.
[ 15.354173] systemd[1]: Queued start job for default target graphical.target.
[ 15.565946] systemd[1]: Created slice machine.slice - Virtual Machine and Container Slice.
[ OK ] 15.574741] systemd[1]: Created slice system-getty.slice - Slice /system/getty.
[ OK ] 15.583297] systemd[1]: Created slice system-modprobe.slice - Slice /system/modprobe.
[ OK ] 15.592464] systemd[1]: Created slice system-serial\x2dgetty.slice - Slice /system/serial-getty.
[ OK ] 15.602601] systemd[1]: Created slice system-ssh\x2dkeygen.slice - Slice /system/ssh-keygen.
[ OK ] 15.612667] systemd[1]: Created slice system-systemd\x2dzram\x2dsetup.slice - Slice /system/systemd-zram-setup.
[ OK ] 15.623530] systemd[1]: Created slice user.slice - User and Session Slice.
[ OK ] 15.631624] systemd[1]: Started systemd-ask-password-wall.path - Forward Password Requests to Wall Directory Watch.
[ OK ] 15.643438] systemd[1]: proc-sys-fs-binfmt-misc.mount - Arbitrary Executable File Formats File System Automount Point was skipped because of an unmet condition check (ConditionPathExists=proc/sys/fs/binfmt-misc).
[ OK ] 15.664131] systemd[1]: Reached target integritysetup.target - Local Integrity Protected Volumes.

/dev/ttyUSB0 115200-8-N-1 DTR RTS CTS CD DSR RI
```

Fig: Composed Fedora 38 disk image with dnf based rootfs.

```
Feb 16 3:46 AM
Screenshot captured
You can paste the image from the clipboard.

[FAILED] Failed to start auditd.service - Security Auditing Service.
See 'systemctl status auditd.service' for details.
[ OK ] Stopped auditd.service - Security Auditing Service.
[FAILED] Failed to start auditd.service - Security Auditing Service.
See 'systemctl status auditd.service' for details.
You are in emergency mode. After logging in, type "journalctl -xb" to view
system logs, "systemctl reboot" to reboot, "systemctl default" or "exit"
to boot into default mode.
Give root password for maintenance
(or press Control-D to continue):
[root@fedora-riscv ~]# ls
anaconda-ks.cfg
[root@fedora-riscv ~]# cd /
[root@fedora-riscv /]# ls
afs boot etc lib lost+found mnt proc run srv tmp var
bin dev home lib64 media opt root sbin sys usr
[root@fedora-riscv /]# cat /etc/os-release
NAME="Fedora Linux"
VERSION="38 (Thirty Eight)"
ID=fedora
VERSION_ID=38
VERSION_CODENAME=""
PLATFORM_ID="platform:f38"
PRETTY_NAME="Fedora Linux 38 (Thirty Eight)"
ANSI_COLOR="0;38;2;66;110;180"
LOGO=fedora-logo-icon
CPE_NAME="cpe:/o:fedoraproject:fedora:38"
DEFAULT_HOSTNAME="fedora"
HOME_URL="https://fedoraproject.org/"
DOCUMENTATION_URL="https://docs.fedoraproject.org/en-US/fedora/f38/system-administrators-guide/"
SUPPORT_URL="https://ask.fedoraproject.org/"
BUG_REPORT_URL="https://bugzilla.redhat.com/"
REDHAT_BUGZILLA_PRODUCT="Fedora"
REDHAT_BUGZILLA_PRODUCT_VERSION=38
REDHAT_SUPPORT_PRODUCT="Fedora"
REDHAT_SUPPORT_PRODUCT_VERSION=38
SUPPORT_END=2024-09-14
[root@fedora-riscv /]#
```

Fig: Composed Fedora 38 disk image booted successfully.

Chapter 6

Conclusion

6.1 Conclusion

Throughout this project, we have meticulously executed a series of tasks to enhance the deployment capabilities of Linux on RISC-V architecture, focusing on future-generation embedded systems. Our journey began with the compilation of U-Boot for RISC-V with EFI support, ensuring compatibility and efficient booting across a variety of RISC-V devices. This step was crucial in establishing a solid foundation for our deployment solution.

Subsequently, we compiled OpenSBI and seamlessly integrated it with U-Boot, providing a robust runtime environment essential for managing software on RISC-V platforms. This integration not only enhanced system stability but also ensured efficient initialization and management of software, contributing significantly to the overall performance of our deployment solution.

A major milestone of our project was the compilation of the Linux kernel with support for containers and virtualization, a pivotal step in enabling the deployment and management of containerized applications on RISC-V devices. This enhancement opens up a myriad of possibilities for developers, allowing them to leverage containerization technologies to create more efficient and scalable applications for RISC-V platforms.

Additionally, we integrated a Fedora-based root file system managed by dnf, simplifying the deployment process of software packages on RISC-V devices. This integration not only streamlines deployment workflows but also ensures compatibility and ease of use for developers working with RISC-V architecture.

Furthermore, we successfully built a container image for RISC-V using the QEMU

emulator and the VisionFive StarFive2 board, allowing for thorough testing and validation of our deployment solution. This comprehensive approach to image composition and integration ensures that our deployment solution meets the stringent requirements of future-generation embedded systems running on RISC-V architecture.

Impact and Significance:

The significance of our project extends beyond its immediate outcomes, offering a substantial contribution to the advancement of deployment capabilities for Linux on RISC-V architecture. By providing optimized disk and container images, our project enhances deployment efficiency, scalability, and compatibility, addressing current limitations and fostering innovation in the RISC-V ecosystem. This contribution is poised to have a profound impact on the developer, researcher, and broader technological communities engaged in RISC-V and containerization technologies, catalysing the development of more sophisticated and reliable embedded systems solutions.

6.2 Future Enhancement –

While project has achieved its core objectives, there's room for further development:

- Implementing GRUB as bootloader for ease of use.
- Implementing more secured method by signing kernel and implementing secure boot.
- Upstream contributions: Sharing advancements and expertise with the wider Container community.

Chapter 7

References

- [1] <https://fedoraproject.org/wiki/>
- [2] https://riscv.org/wp-content/uploads/2019/06/13.30-RISCV_OpenSBI_Deep_Dive_v5.pdf
- [3] <https://docs.u-boot.org/en/latest/board/starfive/visionfive2.html>
- [4] <https://docs.kernel.org/>
- [5] <https://docs.podman.io/en/latest/>
- [6] <https://github.com/riscv-collab/riscv-gnu-toolchain>
- [7] <https://www.docker.com/resources/what-container/#:~:text=A%20Docker%20container%20image%20is,tools%2C%20system%20libraries%20and%20settings.>