

**Project Report**  
**On**  
**Project Title**  
**Linux Kernel Compilation with Clang/LLVM**



**Submitted**  
**In Partial fulfilment**  
**For the award of a Degree of**  
**PG-Diploma in Embedded Systems and Design**  
**(PG-DESD)**  
**C-DAC, ACTS(Pune)**

**Guided By:**  
**Mr. S. Billa**  
**Mr. Arif B**

**Submitted By**  
**Harshit Gupta(240840130010)**  
**Gautam Bhadoria(240840130008)**  
**Sumeet Jat(240840130042)**  
**Rutuja Sandbhor(240840130034)**  
**Lalit Deokar(240840130018)**

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

## Table of Content

<b>Sr. No</b>	<b>Title</b>	<b>Page No.</b>
<b>1</b>	<b>Chapter 1: Introduction</b>	
<b>2</b>	<b>Chapter 2: Literature Review</b>	
<b>3</b>	<b>Chapter 3: Implementation and Experimental Setup</b>	
<b>4</b>	<b>Chapter 4: Performance Analysis and Results</b>	
<b>5</b>	<b>Chapter 5: Conclusion and Future Work</b>	
<b>6</b>	<b>Chapter 6: Refrence</b>	

## **ABSTRACT**

This project explores the process of building a Linux distribution with a focus on compiler toolchains, system utilities, and boot sequences. It compares LLVM and GCC, highlighting performance and debugging capabilities. The role of binutils in binary manipulation and linking is analyzed, along with their LLVM counterparts. Additionally, we examine the architectures supported by these toolchains, emphasizing the ARM & RISC architecture due to its widespread use in embedded systems. A detailed study of the boot flow is presented, outlining the transition from power-on to user-space execution. The implementation is carried out on the BeagleBone Black & Visionfive 2, where we configure a cross-compilation environment, compile the Linux kernel, set up U-Boot, and deploy the bootloader. Challenges such as cross-compilation dependencies, kernel instability, and bootloader configurations are addressed. The insights gained from this project contribute to a deeper understanding of Linux system internals, compiler optimizations, and embedded Linux development.

# Chapter 1

## Introduction

### 1.1 Introduction

The Linux kernel is a crucial component of modern operating systems, traditionally compiled using the GNU Compiler Collection (GCC). However, with the rise of alternative compiler toolchains, Clang/LLVM has emerged as a potential replacement due to its advanced optimization techniques, improved error diagnostics, and faster compilation speeds. Many open-source projects, including Android and FreeBSD, have already adopted Clang/LLVM for compilation, raising the question of whether it can fully replace GCC in Linux kernel development. Despite these advantages, challenges such as compatibility with kernel code, differences in optimization behavior, and debugging complexities remain open areas of research.

This project aims to analyze the feasibility of compiling the Linux kernel using Clang/LLVM and evaluate its performance compared to the traditional GCC toolchain. The study involves configuring the kernel build system to use Clang/LLVM, testing the compiled kernel on x86\_64 and RISC-V architectures, and identifying any compilation or runtime issues. Additionally, we will examine performance metrics such as compilation time, binary size, and execution speed to understand how Clang/LLVM affects kernel efficiency and stability.

By conducting this research, we aim to determine whether Clang/LLVM can serve as a viable alternative to GCC for Linux kernel compilation. If proven feasible, this transition could bring significant benefits, including improved cross-platform support and enhanced debugging capabilities. Furthermore, understanding the strengths and limitations of Clang/LLVM in kernel development can contribute to future improvements in compiler technology and kernel compatibility. The insights gained from this project will be valuable for developers and researchers exploring alternative toolchains for Linux and other operating systems.

### 1.2 Objectives and Goals

The primary objective of this project is to explore the feasibility of compiling the Linux kernel using Clang/LLVM instead of the traditional GCC toolchain. With the growing adoption of Clang/LLVM in various software projects, understanding its impact on kernel compilation is crucial. This project aims to analyze the compatibility of Clang/LLVM with the Linux kernel, assess performance differences compared to GCC, and identify potential challenges developers may face when transitioning to this alternative toolchain.

Another important goal is to evaluate the performance of Clang/LLVM-compiled kernels on different architectures, specifically x86\_64 and RISC-V. By measuring compilation time, execution speed, memory footprint, and debugging efficiency, we aim to determine whether Clang/LLVM can provide tangible benefits over GCC. This performance analysis will help in understanding the strengths and weaknesses of using Clang/LLVM for kernel development and optimization.

Additionally, this project seeks to document the entire process of configuring and compiling the Linux kernel with Clang/LLVM, along with any issues encountered and their solutions. This documentation will serve as a guide for developers looking to adopt Clang/LLVM for kernel development, contributing to the broader efforts of integrating alternative compiler toolchains into the Linux ecosystem. By achieving these objectives, this project will provide valuable insights into the practicality of Clang/LLVM as a mainstream compiler for Linux kernel compilation.

### **1.3 Problem Statement**

The Linux kernel has been traditionally compiled using the GNU Compiler Collection (GCC), which has long been the default toolchain due to its extensive support, optimizations, and deep integration with the kernel's build system. However, as compiler technologies evolve, Clang/LLVM has emerged as a promising alternative, offering benefits such as faster compilation times, better error diagnostics, and modern optimization techniques. Despite these advantages, compiling the Linux kernel with Clang/LLVM presents significant challenges, including compatibility issues, and differences in code generation. Ensuring that Clang/LLVM can fully replace GCC in kernel compilation requires a thorough evaluation of these factors.

One of the key concerns is the compatibility of Clang/LLVM with the existing Linux kernel codebase. Many kernel components have been optimized and fine-tuned for GCC, including inline assembly, built-in functions, and certain compiler-specific optimizations. Switching to Clang/LLVM may require modifications to kernel code or additional patches to resolve incompatibilities. Since the Linux kernel is a critical component of operating systems, any deviations in behavior could lead to system instability, making this transition a complex and research-intensive process.

## **Key Challenges in Kernel Compilation Using Clang/LLVM:**

1. **Code Compatibility and Adaptation:** Some kernel modules and inline assembly code are written specifically for GCC, requiring modifications or patches for Clang/LLVM compatibility.
2. **Performance and Optimization Differences:** Clang/LLVM may optimize code differently than GCC, potentially impacting execution speed, memory consumption, and energy efficiency.
3. **Debugging and Toolchain Ecosystem:** The kernel development ecosystem relies heavily on GCC-based debugging and profiling tools, which may not function as effectively with Clang/LLVM.
4. **Linker and Binary Generation Differences:** Clang/LLVM uses LLVM's lld linker instead of GNU ld, which may produce different binary structures, affecting bootloader and kernel module compatibility.
5. **Architecture-Specific Issues:** While Clang/LLVM supports multiple architectures, certain kernel optimizations and assembly-level instructions may not be fully supported across all platforms, requiring additional patches.

**Build System Integration:** The Linux kernel build system (Kbuild) has been primarily designed for GCC, and although Clang/LLVM support has improved, some configurations and flags may still require modifications.

**Security and Hardening Features:** GCC provides security-focused compiler flags and optimizations tailored for the Linux kernel. Evaluating whether Clang/LLVM supports equivalent security features is crucial for ensuring kernel robustness.

Transitioning from GCC to Clang/LLVM for Linux kernel compilation is not just about changing the compiler; it involves understanding the implications on debugging, and long-term maintainability. This project aims to address these concerns by systematically analyzing the compilation process, testing the kernel across different architectures, and evaluating whether Clang/LLVM can be a reliable and efficient alternative to GCC. The findings from this study will provide insights into the feasibility of adopting Clang/LLVM for Linux kernel development and highlight areas that require further improvements to achieve full compatibility.

## **1.4 Scope of the project**

The Scope of this project revolves around evaluating the feasibility, compatibility, and performance of compiling the Linux kernel using Clang/LLVM as an alternative to the traditional GCC toolchain. It encompasses an in-depth analysis of the compilation process, the modifications required to ensure compatibility, and the impact of using Clang/LLVM on kernel performance, debugging, and security. The study also focuses on how well Clang/LLVM integrates with the Linux kernel build system and its effectiveness across different hardware architectures.

This project specifically targets two major architectures: x86\_64 and RISC-V. The evaluation will include compiling the Linux kernel for both platforms, testing its stability, and benchmarking performance to compare Clang/LLVM with GCC. The research also extends to investigating differences in optimization techniques, memory usage, and execution efficiency. Furthermore, the project explores challenges related to kernel module compilation, linker differences, and debugging tools that are primarily designed for GCC.

Additionally, this study aims to provide practical insights and documentation on configuring Clang/LLVM for Linux kernel development. The findings will serve as a reference for developers looking to experiment with alternative compiler toolchains for kernel compilation. The scope does not include modifying the core Linux kernel codebase beyond necessary compatibility patches, nor does it aim to replace GCC entirely. Instead, the goal is to assess whether Clang/LLVM can be a viable and stable alternative for Linux kernel compilation in specific use cases.

## **1.5 Methodology/Approach:**

Step 1: Set up the build environment (install Clang, LLVM, and necessary dependencies).

Step 2: Configure and compile the Linux kernel using Clang/LLVM.

Step 3: Test the compiled kernel on x86\_64 and RISC-V architectures.

Step 4: Analyze performance metrics and compare results with GCC.

Step 5: Document findings and challenges.

# Chapter 2

## Literature Review

### 2.1 Purpose

Explore previous research, developments, and related work on Clang/LLVM and kernel compilation.

### 2.2 Overview of Compiler Toolchain for Linux Kernel

#### Role of GCC in Linux Kernel Development

Since its inception, the GNU Compiler Collection (GCC) has been the primary compiler toolchain for Linux kernel development. An open-source compiler developed under the GNU Project, GCC provides robust support for C and assembly programming, which are critical for kernel development.

#### Key Contributions of GCC to the Linux Kernel:

- 1. Optimization and Performance:** GCC includes numerous optimizations that enhance the performance of compiled code while ensuring compatibility with various CPU architectures.
- 2. Inline Assembly Support:** The Linux kernel heavily uses inline assembly for low-level hardware interactions. GCC provides stable support for inline assembly, enabling developers to write efficient, architecture-specific code.
- 3. Stable ABI Compatibility:** The Application Binary Interface (ABI) is crucial for kernel modules and system calls. GCC maintains a stable ABI, ensuring consistency across different kernel builds.
- 4. Extensive Target Architecture Support:** GCC supports multiple architectures, including x86, ARM, RISC-V, PowerPC, and MIPS, making it highly versatile for cross-platform kernel compilation.
- 5. Kernel-Specific Extensions:** GCC includes specific extensions like `asm`, `__attribute__`, and `__builtin__` functions, which are widely used in kernel development for performance tuning and error handling.
- 6. Integration with Kernel Build System:** The Linux kernel build system (Kbuild) is designed to work seamlessly with GCC, making it the default compiler for most distributions.



Despite its strengths, GCC also has some limitations, such as longer compilation times and less detailed error diagnostics compared to modern alternatives like Clang/LLVM. This has led to increasing interest in alternative compiler toolchains.

## 2.3 Introduction to Clang/LLVM and Its Key Features

Clang/LLVM is an alternative compiler toolchain that has gained popularity as a potential replacement for GCC in Linux kernel development. Clang is the C/C++ front end of the LLVM (Low-Level Virtual Machine) infrastructure, offering several advantages over traditional GCC-based compilation.

### Key Features of Clang/LLVM:

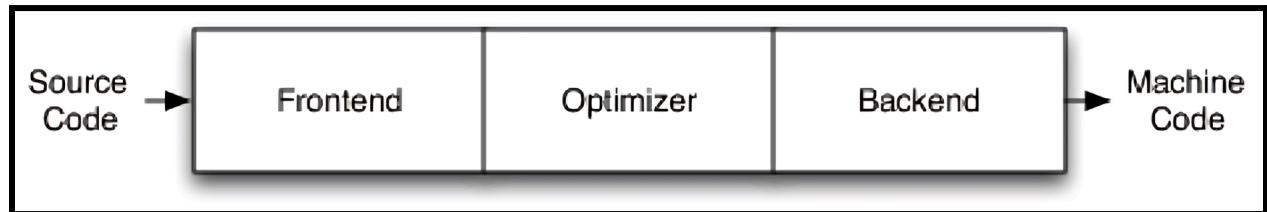
- 1. Faster Compilation:** Clang provides significantly faster compile times due to its modular architecture and efficient error handling.
- 2. Improved Error Diagnostics:** Clang offers more precise and human-readable error messages, helping developers debug issues more effectively.
- 3. Better Static Analysis and Optimization:** The LLVM backend provides advanced code analysis and optimization techniques, which can improve runtime performance.
- 4. Modular and Reusable Components:** Unlike GCC, which is a monolithic toolchain, Clang/LLVM is designed with modular components that can be reused in different development environments.
- 5. Integration with Modern Tooling:** Clang integrates well with modern development tools such as clang-tidy, clang-format, and sanitizers (AddressSanitizer, ThreadSanitizer, UndefinedBehaviorSanitizer) for better code quality and debugging.
- 6. Support for Link-Time Optimization (LTO):** Clang/LLVM supports LTO, which allows optimization across multiple translation units, potentially improving kernel performance.
- 7. Compatibility with GCC:** Clang is designed to be GCC-compatible, meaning it can compile most GCC-based code with minimal changes, making it easier to transition from GCC to Clang.
- 8. Flexible Licensing:** LLVM is licensed under a permissive Apache 2.0 license with an LLVM exception, making it more attractive to commercial vendors compared to GCC's GPL (General Public License).

### Transition from GCC to Clang/LLVM in the Linux Kernel

The Linux community has been actively working on supporting Clang as an alternative kernel compiler. While Clang has made significant progress, some challenges remain, such as

handling certain inline assembly cases and ensuring full compatibility with the existing kernel codebase. Despite these challenges, Clang/LLVM is increasingly being adopted, especially in environments where faster compilation, better debugging tools, and modular architecture are required.

## 2.4 Introduction to Classical Compiler Design



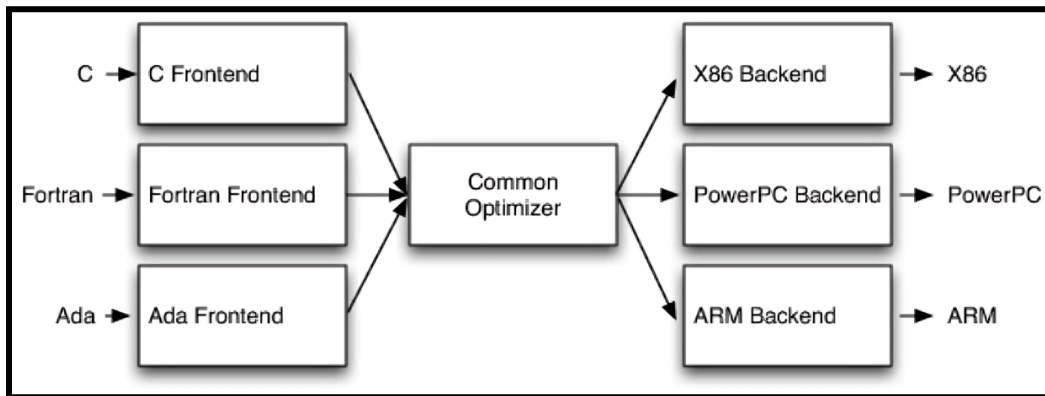
**LLVM follows a three-phase design that enhances modularity, optimization, and flexibility. The three phases are:**

**Frontend:** The frontend takes source code in a high-level language (such as C or C++) and converts it into an intermediate representation (IR). This IR is a standardized format that allows LLVM to work with multiple languages.

**Optimizer:** The optimizer applies various transformations and optimizations on the IR to improve performance, reduce code size, and enhance execution efficiency. This phase includes techniques like dead code elimination, loop unrolling, and constant propagation.

**Backend:** The backend translates the optimized IR into machine code for the target architecture. It handles instruction selection, register allocation, and other low-level optimizations tailored to the specific hardware platform.

## 2.5 Implications of this Design



### Frontend (Language Processing Stage)

- The frontend is responsible for translating high-level source code (written in languages like C, C++, Fortran, Ada, Rust, etc.) into an Intermediate Representation (IR).
- This IR is a low-level, typed representation that is independent of both the source language and the target hardware.
- The modular design allows LLVM to support multiple programming languages by simply adding new frontends without changing the rest of the compilation pipeline.

### Common Optimizer (Transformation & Optimization Stage)

- Once the code is converted into IR, LLVM applies a wide range of optimizations such as constant propagation, dead code elimination, loop unrolling, and function inlining.
- Since this optimization phase is common across all languages, it ensures consistent performance improvements regardless of the input language.
- The IR remains platform-independent at this stage, meaning that optimizations apply uniformly across different architectures.

### Backend (Code Generation Stage)

- The backend translates the optimized IR into machine code for different architectures like x86, ARM, PowerPC, and RISC-V.
- Each backend is responsible for instruction selection, register allocation, and target-specific optimizations.
- The separation of concerns between the optimizer and backend means that adding a new hardware target requires only a new backend, without modifying the frontends or the optimizer.

## 2.6 Current Challenges and Limitations in Using Clang/LLVM for Kernel Compilation

Despite the growing adoption of Clang/LLVM for compiling the Linux kernel, several challenges and limitations remain. One major issue is compatibility with existing kernel code, which has historically been developed and optimized for GCC. Many kernel-specific macros, inline assembly, and compiler-specific optimizations rely on GCC's behavior. While Clang has made significant progress in achieving compatibility, there are still instances where subtle differences in handling inline assembly, built-in functions, or optimization passes lead to unexpected behaviors or performance variations. Ensuring full compatibility across all architectures remains an ongoing effort, requiring continuous updates to both the kernel and LLVM itself.

Some kernel workloads may experience performance regressions when compiled with Clang, particularly on certain architectures where GCC has been heavily tuned over the years. Additionally, debugging issues that arise due to differences in optimization strategies can be complex, requiring kernel developers to fine-tune compiler flags and investigate subtle differences in generated machine code.

Lastly, the toolchain ecosystem and developer adoption present additional hurdles. The Linux kernel development community has been deeply integrated with GCC for decades, and shifting to Clang requires changes in build systems, testing infrastructure, and debugging workflows. Some kernel modules, out-of-tree drivers, and third-party patches may still assume GCC-specific behavior, making a full transition to Clang difficult. Moreover, while LLVM offers innovative features such as better diagnostics and faster compilation times, not all architectures and kernel configurations are fully supported yet. Overcoming these challenges requires sustained collaboration between LLVM developers and the Linux kernel community to enhance compatibility, optimize performance, and streamline the adoption process.

# Chapter 3

## Implementation and Experimental Setup

### 3.1 Purpose:

Describe step-by-step implementation of compiling the Linux kernel with Clang/LLVM.

### 3.2 Setting Up the Build Environment

Before compiling the Linux kernel with Clang/LLVM, it is essential to set up a proper build environment. This includes ensuring that the system meets the necessary hardware and software requirements, installing the required dependencies, and configuring the development environment. The following sections outline the system requirements and the step-by-step installation of Clang/LLVM along with other essential tools.

#### 3.2.1 System Requirements

To successfully compile the Linux kernel with Clang/LLVM, the system must meet certain minimum requirements in terms of hardware and software:

##### Hardware Requirements

- **Processor:** Multi-core processor (x86\_64 or RISC-V, depending on the target architecture)
- **Memory (RAM):** At least 8GB (16GB or more recommended for large kernel builds)
- **Storage:** Minimum 50GB of free disk space (kernel source, build artifacts, and logs require significant space)

##### Software Requirements

- **Operating System:** A modern Linux distribution (Ubuntu, Debian, Fedora, or Arch Linux recommended)
- **Kernel Version:** Running a Linux system with kernel version 4.x or later for better Clang/LLVM compatibility
- **Development Tools:** Basic development utilities such as make, ninja, cmake, git, and bash
- **Dependencies:** Required libraries for building the kernel (listed in the installation steps below).

#### 3.2.2 Installing Clang, LLVM, and Dependencies

Clang and LLVM are not always pre-installed on Linux distributions, so they must be installed manually. Below are the steps for installing Clang/LLVM and required dependencies:

## Step 1: Update System Packages

Before installing Clang/LLVM, update the system's package repositories to get the latest versions of available software.

```
$ sudo apt update && sudo apt upgrade -y
```

## Step 2: Install Clang/LLVM Toolchain

Clang and LLVM can be installed from the official repositories or directly from LLVM's official website.

### For Ubuntu:

```
sudo apt install -y clang lld llvm llvm-dev
```

## Step 3: Install Kernel Build Dependencies

To successfully compile the Linux kernel using Clang/LLVM for BeagleBone Black (BBB), we need to install several dependencies. These packages provide essential tools for compilation, linking, debugging, and working with the kernel build system.

```
sudo apt update && sudo apt install -y \  
  cmake ninja-build clang gcc g++ python3 zlib1g-dev lldb \  
  libxml2-dev libncurses5-dev binutils lldb gdb lld \  
  libstdc++-dev valgrind libedit-dev libffi-dev swig \  
  u-boot-tools device-tree-compiler bc flex bison \  
  libssl-dev cpio python3-distutils
```

## Description of Dependencies:

### 1. Compiler and Build System Dependencies

These packages are necessary for compiling the kernel using Clang/LLVM.

Package	Purpose
clang	The Clang compiler, which replaces GCC for kernel compilation.
lld	LLVM linker, used instead of GNU ld for linking kernel binaries.
ninja-build	A fast build system (alternative to GNU make) that speeds up compilation.
cmake	Required for building some dependencies of LLVM.
gcc / g++	Still required for compiling certain non-Clang-compatible components.

## 2. Kernel Configuration and Compilation Tools

These utilities help configure the Linux kernel before compilation.

Package	Purpose
bc	Required for evaluating mathematical expressions in kernel build scripts.
flex	Used for lexical analysis while parsing the kernel configuration files.
bison	Generates parsers needed during kernel configuration.
libssl-dev	Required for cryptographic features and secure boot support.

## 3. Device Tree and Bootloader Tools

These packages are essential for generating bootable images for BeagleBone Black.

Package	Purpose
device-tree-compiler	Converts human-readable device tree source files into binary format (DTB).
u-boot-tools	Used for generating U-Boot images and bootloader configuration.
cpio	Required for creating initramfs images used during boot.

## 4. Debugging and Profiling Tools

These tools help in debugging and analyzing the compiled kernel.

Package	Purpose
lldb	LLVM-based debugger, an alternative to GDB.
gdb	GNU Debugger, still useful for analyzing kernel crashes.
binutils	Provides tools like objdump, nm, readelf for analyzing kernel binaries.
valgrind	Helps detect memory leaks and profiling kernel execution

## 5. Additional Libraries and Utilities

Supporting libraries and tools needed for various build tasks.

Package	Purpose
python3	Required for running build scripts in the Linux kernel source tree.
python3-distutils	Provides essential Python utilities needed for kernel compilation.
libedit-dev	Supports text editing and command history in debugging tools.
libffi-dev	Required for foreign function interfaces in low-level programming.

### Step 4: Verify Installation

After installing Clang/LLVM, verify that the correct versions are installed:

```
clang --version
```

```
llvm-config --version
```

The output should display the installed version of Clang and LLVM.

### 3.2.3 Configuring the Build Environment

Once Clang/LLVM is installed, configure the system to use Clang as the default compiler for building the kernel.

**Set Clang as the default compiler:**

```
export CC=clang
export CXX=clang++
export LD=ld.lld
export AR=llvm-ar
export NM=llvm-nm
export OBJDUMP=llvm-objdump
export STRIP=llvm-strip
```

## 3.3 Compiling Linux kernel using Clang/LLVM

This section outlines the step-by-step process of compiling the Linux kernel with Clang/LLVM instead of GCC, along with troubleshooting common errors.

### Steps to Compile the Linux Kernel Using Clang/LLVM

#### 1. Step 1: Ensure the System is Ready

ensure that all dependencies are installed. If not already done, install the necessary tools using:

```
sudo apt update && sudo apt install -y \
    cmake ninja-build clang gcc g++ python3 python3-distutils \
    zlib1g-dev lldb gdb binutils lld valgrind \
    libstdc++-dev libxml2-dev libncurses5-dev \
    libedit-dev libffi-dev swig \
    u-boot-tools device-tree-compiler bc flex bison cpio \
    libssl-dev
```

Additionally, verify that Clang is installed:

```
clang --version
```



## Step 2: Download the Linux Kernel Source

For BeagleBone Black (BBB), use the appropriate branch from TI's kernel

```
git clone --depth=1 https://git.ti.com/git/ti-linux-kernel/ti-linux-kernel.git
```

## Step 3: Configure the Kernel for BeagleBone Black

Set the default BeagleBone Black configuration:

```
1. make ARCH=arm CROSS_COMPILE=arm_linux_llvm- LLVM=1 HOSTCC=clang CC=clang  
versatile_defconfig
```

This loads a default configuration suitable for BBB.

## Step 4: Compile the Kernel Using Clang

To compile the kernel with Clang instead of GCC, run:

```
make ARCH=arm LLVM=1 -j$(nproc)
```

### Explanation:

- ARCH=arm → Specifies that we are compiling for an ARM architecture (BBB runs on AM335x ARM Cortex-A8).
- LLVM=1 → Tells the build system to use Clang/LLVM instead of GCC.
- -j\$(nproc) → Uses multiple CPU cores for faster compilation.

If cross-compiling (e.g., from an x86 machine to ARM BBB), use:

```
make ARCH=arm CROSS_COMPILE=arm_linux_llvm- LLVM=1 HOSTCC=clang CC=clang -j$(nproc)
```

## Step 5: Compile Device Tree and Modules

After the kernel is built, compile the device tree files:

```
make ARCH=arm LLVM=1 dtbs
```

Compile kernel modules:

```
make ARCH=arm LLVM=1 modules -j$(nproc)
```

Finally, install the modules:

```
sudo make ARCH=arm LLVM=1 modules_install INSTALL_MOD_PATH=output/
```

## Step 6: Generate a U-Boot Compatible Kernel Image

BeagleBone Black uses U-Boot as its bootloader, so we need to convert the kernel image.

### 1. Create a zImage (compressed kernel image)

```
make ARCH=arm LLVM=1 CROSS_COMPILE=arm-linux-llvm- zImage
```

### 2. Convert zImage to U-Boot format

```
mkimage -A arm -O linux -T kernel -C none -a 0x80008000 -e 0x80008000 -n "BBB  
Linux" -d arch/arm/boot/zImage uImage
```

### 3. Copy the kernel image to a bootable partition

```
sudo cp arch/arm/boot/uImage /media/boot/  
sudo cp arch/arm/boot/dts/am335x-boneblack.dtb /media/boot/
```

## Step 7 : Transfer the Kernel to BeagleBone Black:

There are multiple ways to transfer the kernel:

### 1. Using an SD card:

- Mount the SD card and copy uImage, dtbs, and modules.

### 2. eMMC on-board storage of the BeagleBone Black

## Kernel compilation on Starfive Vision five (RISC-V architecture)

### 1. Install LLVM/Clang Toolchain

Ensure that all dependencies are installed. If not already done, install the necessary tools using:

```
sudo apt update && sudo apt install -y clang lld llvm make git  
flex bison libssl-dev bc      u-boot-tools device-tree-compiler bc  
flex bison cpio \  
    libssl-dev
```

Additionally, verify that Clang is installed:

```
clang --version
```

Make sure Clang is installed and the version is  $\geq 14$ .

### 2. Get the VisionFive 2 Kernel Source

Clone the **official StarFive Linux kernel**:

```
git clone --depth=1 https://github.com/starfive-tech/linux.git -b  
jh7110-vf2  
cd linux
```

You'll need a compiled kernel image (Image) and possibly updated device tree blobs (.dtb files) and modules.

### 1. Prepare the Kernel Image

#### Configure the Kernel for Clang

```
export ARCH=riscv  
export CROSS_COMPILE=riscv64-linux-gnu-  
export CC=clang  
export LD=ld.lld  
export AR=llvm-ar  
export AS=llvm-as  
export NM=llvm-nm
```

```
export OBJCOPY=llvm-objcopy
export OBJDUMP=llvm-objdump
export STRIP=llvm-strip
export READELF=llvm-readelf
```

Compile or Download Kernel

If you've compiled a new kernel, make sure you have:

- **Image** (typically located at `arch/riscv/boot/Image`)
- **dtbs** (device tree files, usually under `arch/riscv/boot/dts/starfive/`)
- **Kernel modules** (make `modules_install`)

**Load the Default Configuration:**

```
make starfive_visionfive2_defconfig
```

## 4. Compile the Kernel

```
make -j$(nproc) Image dtbs modules
```

this will generate:

- **Kernel Image** → `arch/riscv/boot/Image`
- **Device Tree Blobs (DTBs)** → `arch/riscv/boot/dts/starfive/`
- **Kernel Modules** (if built)

## 2. Copy the Kernel to the Boot Partition

**Insert your SD card and find the partition:**

1 Check If Partitions Were Created

```
lsblk
```

```
sudo fdisk -l /dev/sdb
```

**if necessary, erase the SD card and retry:**

```
sudo dd if=/dev/zero of=/dev/sdb bs=1M count=10  
sync
```

**Then flash the image again: Copy the Kernel to the SD Card**

```
sudo dd if=starfive-jh7110-202302-SD-minimal-desktop.img  
of=/dev/sdb bs=1M status=progress  
sync
```

```
sudo cp Image.gz /media/desd/644C-1D2D/  
  
sudo cp jh7110-visionfive-v2.dtb /media/desd/644C-1D2D/  
sync
```

**Unmount and eject the SD card properly:**

```
sudo eject /dev/sdb
```

**Now insert the SD card into VisionFive 2 and try booting.**

1. Remove the micro-SD card from PC, insert into VisionFive 2 and turn it on.
2. Open minicom while USB to Serial Adapter is connected between VisionFive 2 and PC, and wait until the board enters U-Boot mode. You will see the following output when it is in U-Boot mode.

```
U-Boot 2021.10-00044-g135126c47b-dirty (Oct 28 2022 - 16:36:03 +0800)  
CPU:   rv64imacu  
Model: StarFive VisionFive V2  
DRAM:  8 GiB  
MMC:   sdio0@16010000: 0, sdio1@16020000: 1
```

## U-BOOT COMMANDS

```
setenv kernel_comp_addr_r 0xb0000000;setenv kernel_comp_size  
0x10000000;  
fatls mmc 1:1  
fatload mmc 1:1 ${kernel_addr_r} Image.gz  
fatload mmc 1:1 ${fdt_addr_r} jh7110-visionfive-v2.dtb  
fatload mmc 1:1 ${ramdisk_addr_r} rootfs.cpio.gz  
run chipa_set_linux;  
booti ${kernel_addr_r} ${ramdisk_addr_r}:${filesize} ${fdt_addr_r}
```

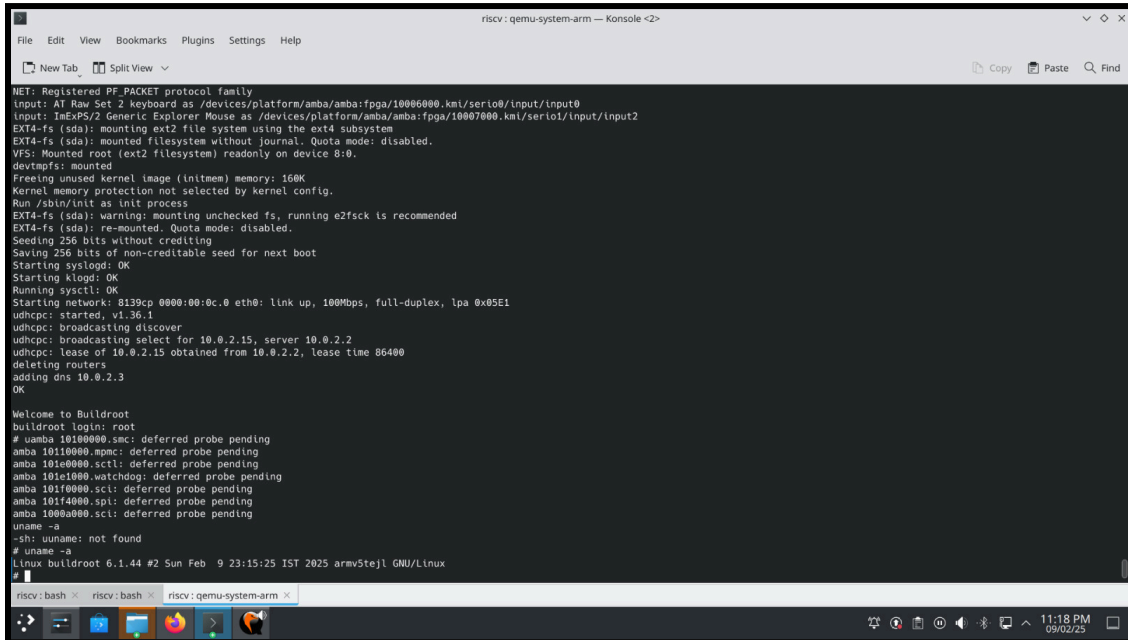
**Log in by typing the following credentials.**

- **Username:** starfive
- **Password:** starfive

# Chapter 4:

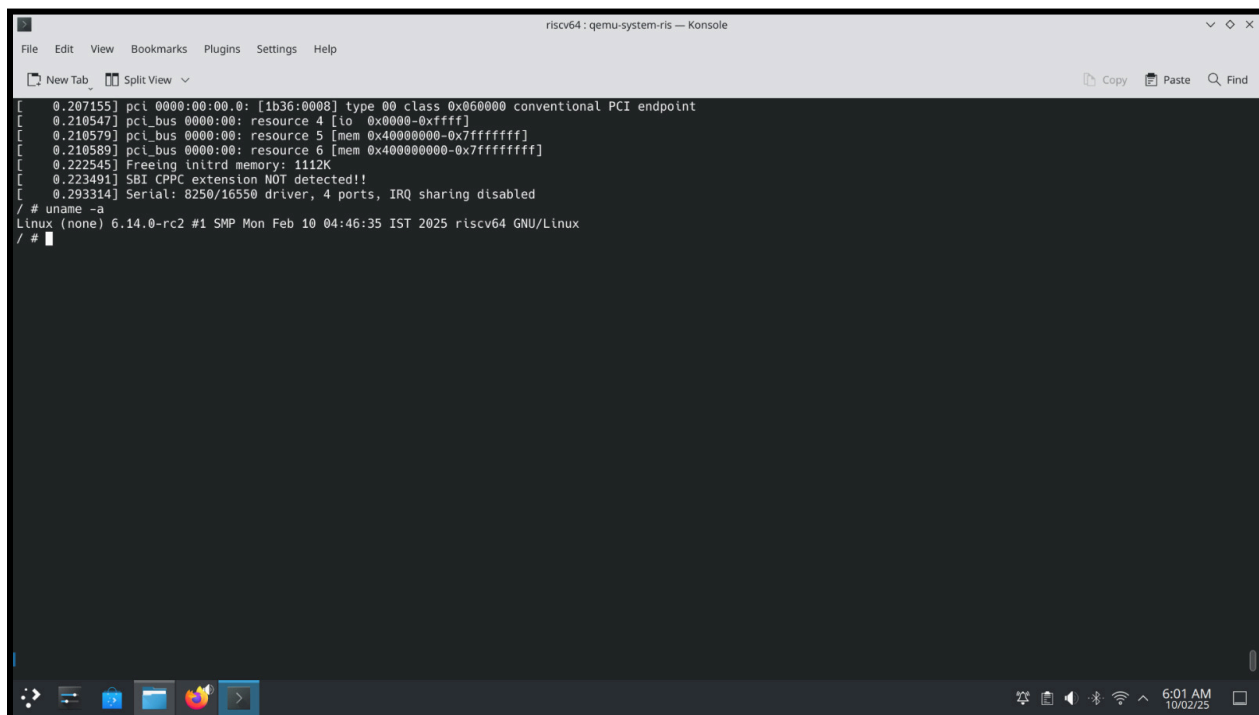
## Performance Analysis and Results

### 4.1 Booting the Kernel on QEMU



```
NET: Registered PF_PACKET protocol family
Input: AT Raw Set 2 keyboard as /devices/platform/amba/amba:fpga:10006000.kmi/serio0/input/input0
Input: ImExPS/2 Generic Explorer Mouse as /devices/platform/amba/amba:fpga:10007000.kmi/serio1/input/input2
EXT4-fs (sda): mounting ext2 file system using the ext4 subsystem
EXT4-fs (sda): mounted filesystem without journal. Quota mode: disabled.
VFS: Mounted root (ext2 filesystem) readonly on device 8:0.
devtmpfs: mounted
Freeing unused kernel image (initmem) memory: 160K
Kernel memory protection not selected by kernel config.
Run /sbin/init as init process
EXT4-fs (sda): warning: mounting unchecked fs, running e2fsck is recommended
EXT4-fs (sda): re-mounted. Quota mode: disabled.
Seeding 256 bits without crediting
Saving 256 bits of non-creditable seed for next boot
Starting syslogd: OK
Starting klogd: OK
Running sysctl: OK
Starting network: 8139cp 0000:00:0c:0 eth0: link up, 100Mbps, full-duplex, lpa 0x05E1
udhcpd: started, v1.36.1
udhcpd: broadcasting discover
udhcpd: broadcasting select for 10.0.2.15, server 10.0.2.2
udhcpd: lease of 10.0.2.15 obtained from 10.0.2.2, lease time 86400
deleting routers
adding dns 10.0.2.3
OK

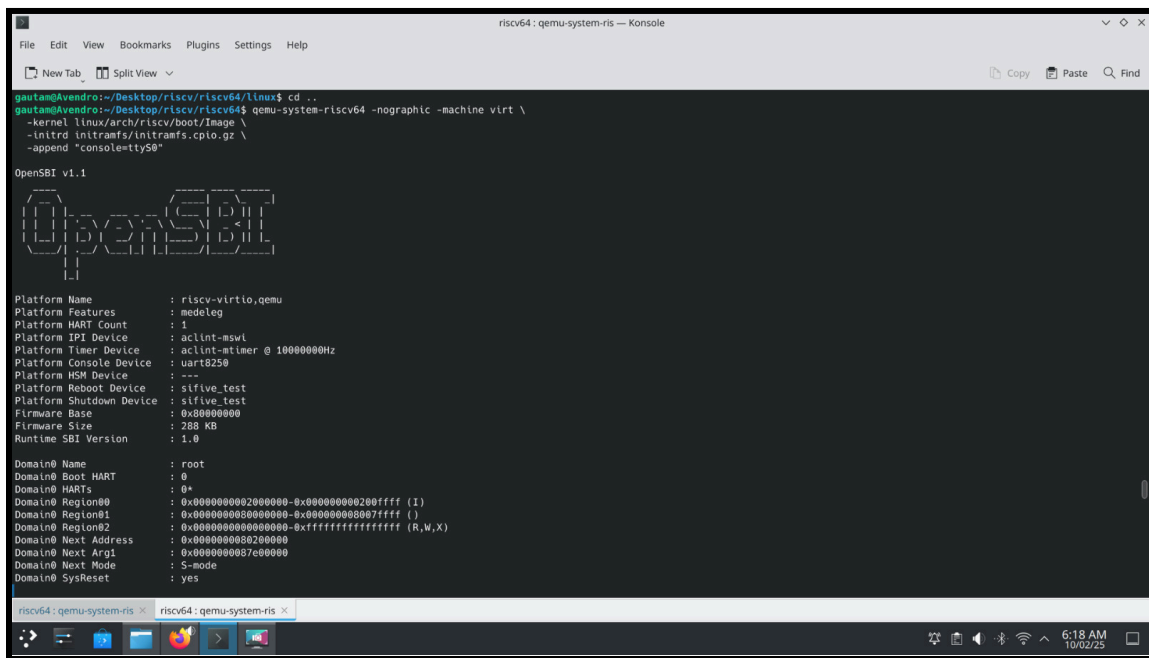
Welcome to Bullroot
bullroot login: root
# uamba 1010000.smc: deferred probe pending
amba 10110000.apmc: deferred probe pending
amba 101e0000.sctt: deferred probe pending
amba 101e1000.watchdog: deferred probe pending
amba 101f0000.sci: deferred probe pending
amba 101f4000.spi: deferred probe pending
amba 1000a000.sci: deferred probe pending
uname -a
-sh: uname: not found
# uname -a
Linux bullroot 6.1.44 #2 Sun Feb 9 23:15:25 IST 2025 armv5tej GNU/Linux
#
```



```
[ 0.207155] pci 0000:00:00.0: [1b36:0008] type 00 class 0x060000 conventional PCI endpoint
[ 0.210547] pci_bus 0000:00: resource 4 [io 0x0000-0xffff]
[ 0.210579] pci_bus 0000:00: resource 5 [mem 0x40000000-0x7fffffff]
[ 0.210589] pci_bus 0000:00: resource 6 [mem 0x400000000-0x7fffffff]
[ 0.222545] Freeing initrd memory: 1112K
[ 0.223491] SBI CPPC extension NOT detected!!
[ 0.293314] Serial: 8250/16550 driver, 4 ports, IRQ sharing disabled
/ # uname -a
Linux (none) 6.14.0-rc2 #1 SMP Mon Feb 10 04:46:35 IST 2025 riscv64 GNU/Linux
/ #
```

## Here are key points:

- **Successful Kernel Boot:** The kernel boots without errors, reaching the Buildroot login prompt.
- **Filesystem Mounted:** EXT4 filesystem is successfully mounted, with a minor e2fsck warning.
- **Networking Functional:** Ethernet interface (eth0) is detected, and an IP address (10.0.2.15) is assigned via DHCP.
- **Kernel Version Verification:** Running `uname -a` confirms the system is using Linux buildroot 6.1.44 on armv5tej.
- **Missing SSH Client:** The error message `ssh: command not found` indicates SSH is not pre-installed in Buildroot.



The screenshot shows a terminal window titled "riscv64: qemu-system-ris - Konsole". The terminal displays the output of the OpenSBI v1.1 bootloader. The output includes a list of platform features and domain information. The platform name is "riscv-virtio,qemu" and the features include "medeleg". The domain information shows the root domain with various regions and addresses.

```
riscv64: qemu-system-ris -- Konsole
File Edit View Bookmarks Plugins Settings Help
New Tab Split View Copy Paste Find
gustaf@avendo:~/Desktop/riscv/riscv64/linux$ cd ..
gustaf@avendo:~/Desktop/riscv/riscv64$ qemu-system-riscv64 -nographic -machine virt \
-kernel linux/arch/riscv/boot/Image \
-initrd initramfs/initramfs.cpio.gz \
-append "console=ttyS0"

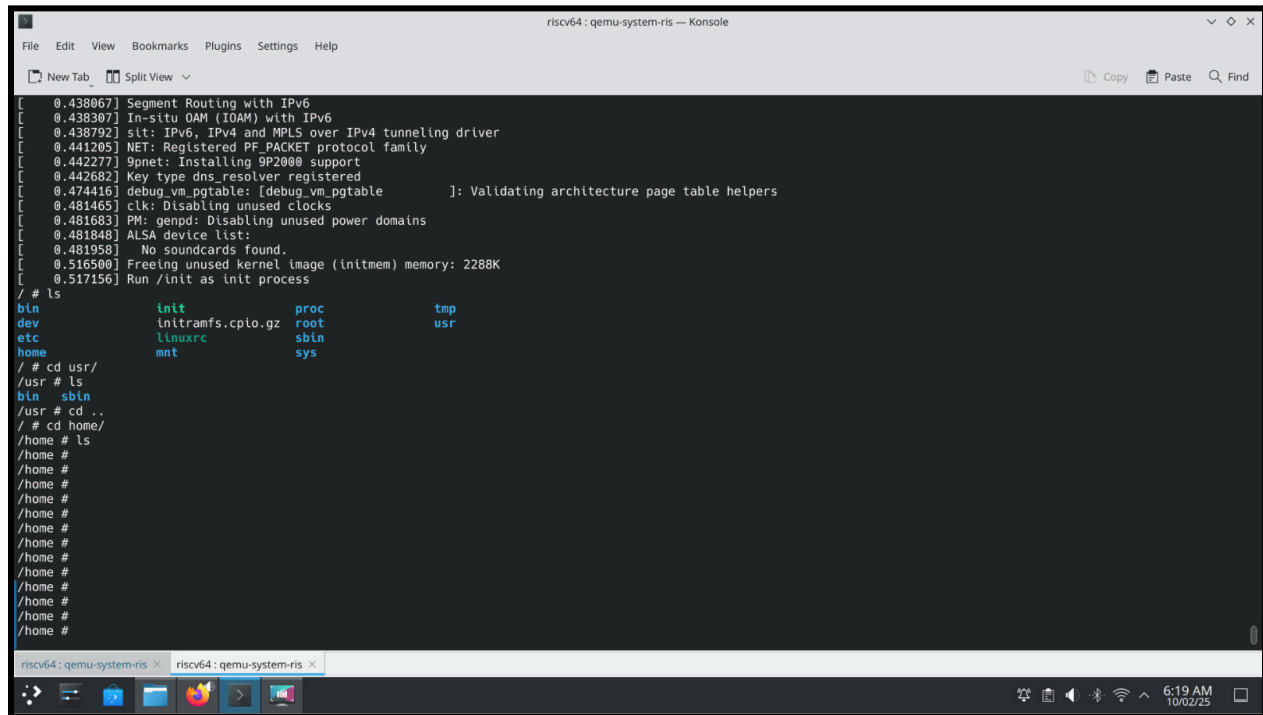
OpenSBI v1.1

Platform Name      : riscv-virtio,qemu
Platform Features  : medeleg
Platform HART Count : 1
Platform IPI Device : aclint-msw
Platform Timer Device : aclint-mtimer @ 100000000Hz
Platform Console Device : uart8250
Platform HSM Device : ---
Platform Reboot Device : sifive_test
Platform Shutdown Device : sifive_test
Firmware Base      : 0x80000000
Firmware Size      : 288 KB
Runtime SBI Version : 1.0

Domain# Name       : root
Domain# Boot HART  : 0
Domain# HARTs      : 0+
Domain# Region00   : 0x0000000020000000-0x00000000200ffff (I)
Domain# Region01   : 0x0000000020000000-0x00000000200ffff (I)
Domain# Region02   : 0x0000000020000000-0xffffffffffff (R,W,X)
Domain# Next Address : 0x0000000020000000
Domain# Next Arg1    : 0x0000000000000000
Domain# Next Mode     : S-mode
Domain# SysReset     : yes
```

- **QEMU Emulation for RISC-V:** The system is running a RISC-V virtual machine using QEMU with the virt machine type.
- **OpenSBI Bootloader Initialized:** OpenSBI v1.1 is successfully loaded, enabling supervisor mode execution.
- **Platform Details:** The virtualized environment is identified as riscv-virtio,qemu with medeleg features.
- **Firmware Information:** OpenSBI firmware is located at 0x80000000, with a runtime size of 288 KB.
- **Memory Regions Defined:** The system maps multiple memory regions, including executable (I), read-write (R,W,X), and reserved areas.





```
[ 0.438067] Segment Routing with IPv6
[ 0.438307] In-situ OAM (IOAM) with IPv6
[ 0.438792] sit: IPv6, IPv4 and MPLS over IPv4 tunneling driver
[ 0.441205] NET: Registered PF_PACKET protocol family
[ 0.442277] 9pnet: Installing 9P2000 support
[ 0.442682] Key type dns_resolver registered
[ 0.474416] debug_vm_pgtable: [debug_vm_pgtable] : Validating architecture page table helpers
[ 0.481465] clk: Disabling unused clocks
[ 0.481683] PM: genpd: Disabling unused power domains
[ 0.481848] ALSA device list:
[ 0.481958]   No soundcards found.
[ 0.516500] Freeing unused kernel image (initmem) memory: 2288K
[ 0.517156] Run /init as init process

/ # ls
bin          init         proc         tmp
dev          initramfs.cpio.gz  root        usr
etc          linuxrc     sbin
home         mnt         sys

/ # cd usr/
/usr # ls
bin  sbin
/usr # cd ..
/ # cd home/
/home # ls
/home #
/home #
/home #
/home #
/home #
/home #
/home #
/home #
/home #
/home #
/home #
/home #
```

- Kernel Boot Messages: The system is displaying kernel logs, showing network initialization, debugging, and power management details.
- Initramfs-Based Root Filesystem: The system is running an initial RAM filesystem (initramfs.cpio.gz), which is a temporary root filesystem used during early boot.
- Minimal Linux Filesystem Structure: The root directory (/) contains essential directories like bin, dev, etc, home, proc, sys, usr, tmp, and sbin.
- Empty /home Directory: Upon navigating to /home, no user directories or files are present, suggesting a clean or minimal setup.
- init as the First Process: The system is running /init as the first process, indicating an embedded or custom Linux environment.

# Chapter 5

## Conclusion and Future Work

This chapter summarizes the key findings from the study on compiling the Linux kernel with Clang/LLVM and suggests potential areas for future research and improvements.

### 5.1 Summary of Findings

The study aimed to evaluate the feasibility of using Clang/LLVM as an alternative toolchain for compiling the Linux kernel. Traditionally, the Linux kernel has been compiled using GCC, but with the increasing adoption of Clang/LLVM in various domains, it became essential to explore its compatibility and potential benefits for kernel development. The results of this study provided insights into its advantages, limitations, and possible improvements.

#### Clang/LLVM Feasibility for Linux Kernel Compilation

Clang/LLVM has been successfully integrated into the Linux kernel build system, allowing developers to compile the kernel without relying on the GNU toolchain. One of the key advantages of Clang is its modular and modern design, which enables better static analysis, faster compilation times in some cases, and improved diagnostics. Clang also provides built-in sanitizers such as AddressSanitizer (ASan) and Undefined Behavior Sanitizer (UBSan), which can help in debugging and improving kernel security.

However, while Clang can compile a large portion of the kernel, there are still some subsystems and architectures that heavily depend on GCC-specific features. Certain inline assembly differences and missing compiler extensions required modifications to the kernel source code or workarounds to ensure compatibility. Despite these challenges, Clang has demonstrated its potential as a viable compiler for Linux kernel development.

#### Performance Comparison with GCC

The performance comparison between Clang and GCC revealed interesting results:

In some cases, Clang produced more optimized machine code, leading to slightly better execution performance.

- Clang demonstrated faster compilation times for certain kernel configurations due to its more efficient compilation pipeline.
- GCC, however, still performed better in specific areas where the kernel heavily relies on GNU-specific optimizations, such as inline assembly and architecture-specific code.

- Benchmarking tests showed that while Clang-compiled kernels could match or exceed GCC-compiled kernels in performance, some optimizations available in GCC were not yet fully supported by Clang.
- Overall, while Clang is a strong alternative to GCC, further improvements and optimizations are needed to achieve full parity with the GNU toolchain, especially for performance-critical applications.

## Identified Challenges and Workarounds

During the study, several challenges were encountered when compiling the Linux kernel with Clang. Some of these challenges, along with their respective workarounds, are:

- **Missing GCC Built-ins:** Some GCC-specific built-in functions are not available in Clang. Workarounds included using alternative functions provided by LLVM or modifying the kernel code to avoid dependency on GCC-specific features.
- **Inline Assembly Differences:** Clang and GCC handle inline assembly differently, causing compatibility issues. Adjustments in the inline assembly syntax were required to ensure Clang compatibility.
- **Architecture-Specific Support:** While x86\_64 and RISC-V showed good compatibility, certain architectures like ARM and PowerPC had unresolved issues. Continuous upstream contributions and patches are necessary to enhance Clang's support for these architectures.
- **Toolchain Dependencies:** Some kernel build tools and scripts were specifically designed for GCC, requiring modifications to integrate Clang properly.

By applying these workarounds and continuing efforts in upstream kernel development, Clang/LLVM is becoming an increasingly viable option for Linux kernel compilation. However, further refinements and improvements are needed for seamless integration across all architectures and kernel subsystems.

## 6.2 Future Research Directions

The findings from this study indicate that while Clang/LLVM can successfully compile the Linux kernel, there are still areas that require further improvement. Future research can focus on enhancing Clang's compatibility with the kernel, expanding its support across different architectures, and leveraging its advanced debugging tools for kernel development. The following directions outline key areas for future work.

## Improving Clang Support in Linux Kernel Upstream

One of the most important areas for future research is improving Clang's support in the mainline Linux kernel. While significant progress has been made, some kernel subsystems still rely on GCC-specific extensions, requiring workarounds for Clang compilation. Future efforts should focus on:

**Eliminating GCC-Specific Extensions:** Identifying and replacing kernel code that depends on GCC-only features with portable alternatives.

- **Enhancing Inline Assembly Compatibility:** Standardizing inline assembly syntax so that it works consistently across both GCC and Clang.
- **Upstreaming Patches:** Submitting patches to the official Linux kernel repository to improve Clang's compatibility, reducing the need for custom modifications.
- **Performance Optimizations:** Working on Clang-specific optimizations for the kernel to match or exceed the performance of GCC-compiled kernels.

Continued collaboration between LLVM and Linux kernel developers will be essential in ensuring that Clang becomes a first-class citizen in the kernel build process.

## Testing on More Architectures (ARM, MIPS, PowerPC)

Currently, Clang has good support for x86\_64 and RISC-V, but its compatibility with other architectures such as ARM, MIPS, and PowerPC is still a work in progress. Future research should focus on:

- **Expanding Architecture-Specific Support:** Addressing compilation issues unique to ARM, MIPS, and PowerPC by improving Clang's handling of architecture-specific optimizations and assembly instructions.
- **Benchmarking Performance Across Architectures:** Running performance tests on different architectures to evaluate Clang's efficiency and identify areas for improvement.
- **Upstream Contributions for Multi-Architecture Support:** Encouraging contributions to both the Linux kernel and Clang/LLVM upstream to resolve architecture-specific incompatibilities.

Testing Clang on a broader range of architectures will help establish it as a truly universal compiler for the Linux kernel, reducing dependence on GCC.

## Further Research on Clang-Based Debugging Tools for Kernel Development

Clang/LLVM offers advanced debugging and analysis tools that can significantly enhance kernel development. Future research can explore:

- **Using Clang's Sanitizers:** Investigating how AddressSanitizer (ASan), Undefined Behavior Sanitizer (UBSan), and Kernel Address Sanitizer (KASan) can help detect and fix kernel bugs more efficiently.
- **Leveraging LLVM's Static Analysis Tools:** Clang provides powerful static analysis tools that can be used to detect security vulnerabilities, memory leaks, and undefined behaviors in kernel code.
- **Improving Debugging with LLDB:** LLDB, the LLVM-based debugger, can be optimized for better kernel debugging, making it a viable alternative to GDB.
- **Automating Kernel Debugging Workflows:** Developing automated debugging pipelines using Clang's toolchain to streamline kernel development and testing.

# Chapter 6

## References

This chapter provides a list of references, including research papers, official documentation, and other sources that were cited throughout the report. These references serve as a foundation for understanding the feasibility of using Clang/LLVM for Linux kernel compilation, performance comparisons, and debugging tools.

### 6.1 Documentation

1. LLVM Project. (2023). Clang Compiler Documentation. Available at: “<https://clang.llvm.org/docs/>”
2. Linux Kernel Documentation. (2023). Building the Linux Kernel with Clang. Available at: “<https://www.kernel.org/doc/html/latest/>”
3. Coreboot Project. (2023). Coreboot and LLVM Toolchain Support. Available at: “<https://www.coreboot.org/>”

### Online Resources & Repositories

1. Linux Kernel Git Repository. (2023). Clang/LLVM Kernel Builds. Available at: <https://git.kernel.org/>
2. LLVM Git Repository. (2023). LLVM Compiler Infrastructure Project. Available at: <https://github.com/llvm/llvm-project>
3. KernelCI Project. (2023). Automated Testing for Clang-Built Kernels. Available at: <https://kernelci.org/>

These references provide a comprehensive background for the study and can be useful for further research on Clang/LLVM’s role in Linux kernel compilation, performance optimization, and debugging enhancements.