University of São Paulo
Institute of Mathematics and Statistics
Bachelor of Computer Science

Contributing to a Continuous Integration infrastructure to the Linux Kernel

Automating patch collection and integration with a CI infrastructure

Marcelo Mendes Spessoto Junior

Final Essay

mac 499 — Capstone Project

Supervisor: Paulo Meirelles
Co-supervisor: Rodrigo Siqueira

São Paulo
2024

Acknowledgements

First of all, I would like to thank those who helped me getting into FLOSS communities: Paulo Meirelles for granting me many opportunities to work on projects regarding free software, and Rodrigo Siqueira and David Tadokoro for the guidance on contributing with the kworkflow project. I would also like to show gratitude to the FLUSP extension group for giving me new perspectives over software development.

I would also want to thank the Instituto of Matemática e Estatística da Universidade de São Paulo (IME-USP) and all its professors, students, and employees for contributing to create a friendly environment to improve as a student, professional and person.

Finally, I would like to show gratitude to my family for supporting me on this journey.

Resumo

Marcelo Mendes Spessoto Junior. **Contributing to a Continuous Integration infrastructure to the Linux Kernel: *Automating patch collection and integration with a CI infrastructure***. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2024.

A implementação de uma infraestrutura de Integração Contínua (IC) vem se tornando uma prática fundamental no contexto moderno de desenvolvimento de software. A capacidade de automatizar o processo de testes de código afeta profundamente o fluxo de produção ao permitir consistência na validação de novos trechos de código em um projeto. Esse conceito se difundiu no contexto da comunidade do kernel Linux, um projeto de alto grau de complexidade e com certas peculiaridades em seu fluxo de desenvolvimento, como o uso de listas de e-mail. Para a implementação de IC nesse projeto, estratégias únicas devem ser adotadas, visando a automação da coleta de patches das listas de e-mail, bem como a compilação e implantação de imagens do kernel. Este trabalho visa contribuir para a implementação de uma infraestrutura de IC hospedada no IME, por meio do desenvolvimento de um software de coleta de patches projetado para ser integrado a uma infraestrutura de IC. O objetivo é planejar um laboratório de testes de imagens do kernel e produzir uma prova de conceito de um programa de rastreamento de novos patches para ser integrado ao laboratório.

Palavras-chave: Integração Contínua. Linux. Testes de software. Listas de e-mail.

Abstract

Marcelo Mendes Spessoto Junior. **Contribuindo para a infrastrutura de Integração Contínua do Kernel Linux: *Automatizando a coleta de patches e integração com uma infrastrutura de IC***. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2024.

Implementing a Continuous Integration (CI) infrastructure is becoming a core practice in the modern context of software development. The ability to automate the code testing process significantly impacts the production workflow by ensuring consistency in validating new code segments in a project. This concept has spread within the Linux kernel community, a highly complex project with certain peculiarities in its development workflow, such as mailing lists in modern software development. Unique strategies should be adopted to implement CI for this project, aiming at automating the collection of new patches from mailing lists and compiling and deploying kernel images. This work aims to contribute to implementing a CI infrastructure hosted at IME by developing a patch collection software designed to be integrated into a CI infrastructure. The objective is to plan a testing lab for kernel images and to produce a proof-of-concept patch-tracking software to be integrated into the lab.

Keywords: Continuous Integration. Linux. Software testing. Mailing lists.

List of Abbreviations

API	Application Programming Interface
CI	Continuous Integration
CLI	Command Line Interface
DB	Database
DRM	Direct Rendering Management
DUT	Device Under Testing
FLOSS	Free/Libre and Open Source Software
GPU	Graphics Processing Unit
GUI	Graphical User Interface
IIO	Industrial Input/Output
JCaC	Jenkins Configuration as Code
IME	Instituto de Matemática e Estatística
OS	Operating System
PDU	Power Distribution Unit
SSH	Secure Shell
USP	Universidade de São Paulo
VM	Virtual Machine
VCS	Version Control System
XML	Extensible Markup Language

List of Figures

1.1 A cyclic diagram representing how kernel releases are developed. 9

3.1 An overview of an example CI-Tron infrastructure, with the gateway and two DUTs. 22
3.2 The kernel lore page for the `amd-gfx` mailing list, with the most recent messages. 23

4.1 The file hierarchy defining the Docker Compose environment that runs the pipeline. Rounded blue items represent files, while squared items represent directories. 31
4.2 The main interface of vivian. 35

B.1 Discovering a virtual DUT from the vivian interface 49
B.2 Accessing DUT configuration from the vivian interface 50
B.3 Accessing the logs of a DUT from the vivian interface 50

List of Programs

1.1 The introductory segment of MAINTAINERS file, describing how each entry is represented . . . 6
1.2 Some entries from the MAINTAINERS file covering the IIO subsystem . 7
4.1 The XML representation of an entry from the kernel lore . . . 27
4.2 The struct abstraction where the lore entries are stored . . . 27
4.3 The Dockerfile for the Jenkins service . . . 29
4.4 The Dockerfile for the Jenkins service . . . 31
4.5 The Dockerfile for the lore-fetcher service . . . 32
4.6 The final docker-compose.yml file . . . 32
4.7 An example of a MarsDB file . . . 36
4.8 A python script to render a custom URL into a jinja2 file . . . 38
A.1 Code from the configuration parser. . . . 45
A.2 A basic unit test for the configuration abstraction. . . . 46
A.3 Code from a prototypal mailing feature using SMTP to send automated texts. The next step would be sending mails with data from the respective patch entry. . . . 47
C.1 A first iteration of the pipeline `.gitlab-ci.yml`. It contains a stage for syncing the Gitlab repository with the original kernel tree (mirror) and a stage for building the kernel and saving the image as an artifact (build). . 51
C.2 A second iteration of the Gitlab CI pipeline focusing on parsing the artifact generated from the build stage. . . . 52

Contents

Introduction 1

1 The Linux Kernel 3
- 1.1 The Linux Kernel . . . 3
- 1.2 Free/Libre and Open Source Software . . . 3
- 1.3 The Linux Kernel and its development workflow . . . 4
  - 1.3.1 Kernel repository structure and build system . . . 4
  - 1.3.2 Kernel subsystems . . . 6
  - 1.3.3 The kernel development workflow . . . 8

2 Continuous Integration 11
- 2.1 Relevant CI Practices . . . 12
  - 2.1.1 Automated Builds . . . 12
  - 2.1.2 Automated Testing . . . 13
  - 2.1.3 Code Coverage . . . 14
- 2.2 Common CI Services and Tools . . . 14
  - 2.2.1 Popular CI Services . . . 14
  - 2.2.2 Tools for Kernel CI . . . 15

3 Developing a CI System for the AMD Display Subsystem 17
- 3.1 Introduction to the AMD Display . . . 17
  - 3.1.1 The DRM Subsystem . . . 17
  - 3.1.2 AMD Display . . . 18
- 3.2 A Bare-Metal CI Infrastructure with CI-Tron . . . 18
  - 3.2.1 The Basic Principles for a Good Bare-Metal CI . . . 18
  - 3.2.2 Preparing a device under testing . . . 19
  - 3.2.3 Deploying the testing environment to a DUT . . . 20
  - 3.2.4 The gateway . . . 21
- 3.3 Integrating a CI system with a mailing list . . . 22

3.3.1 Retrieving new e-mails . . . 23

4 Results **25**
4.1 Tracking new patches . . . 25
4.1.1 The lore-fetcher application . . . 26
4.2 Implementing a CI Pipeline for Kernel Compilation and Exploring LAVA 28
4.2.1 Configuring Jenkins . . . 28
4.2.2 Configuring LAVA for Basic Virtual Deployments . . . 33
4.3 Using CI-Tron from a Virtual Environment . . . 34
4.3.1 The Vivian Virtual Environment . . . 34
4.3.2 Configuring GitLab CI to Expose CI-Tron Runners . . . 35
4.3.3 Writing a Pipeline in GitLab CI . . . 37

5 Conclusion **39**
5.1 Final Remarks on the `lore-fetcher` Development . . . 39
5.1.1 Patch Collection . . . 39
5.1.2 Integration with CI Services . . . 40
5.1.3 Returning Results . . . 40
5.2 Final Remarks on the CI Infrastructure . . . 40
5.3 Future Plans for the CI . . . 41
5.3.1 Why Not KernelCI? . . . 41

6 Personal appreciation **43**

Appendixes

A Code snippets from `lore-fetcher` **45**

B The vivian interface **49**

C Scripts from the Gitlab CI pipeline **51**

References **53**

Introduction

The Linux kernel is one of the most relevant programs in modern computing. Having more than three decades of development, it is widespread in a major amount of mobile devices, network servers, and high-performance computers. It acts as the core component of a computer's operating system (OS), providing an interface between the physical device (hardware) and the applications that will run on it (software). This role is crucial to enable the usability of any device, but implementing such a program is extremely difficult, as many complex tasks must be covered by it. This implies a challenging development process, which affects how the code will be written, tested, and maintained.

Despite these challenges, Linux, which was initially a small project by Linus Torvalds, experienced extreme growth and popularity. An important factor in its gaining traction as a feasible kernel option was its Free/Libre and Open Source Software (FLOSS) development model. FLOSS software differs from traditional projects because it follows a philosophy that promotes the freedom to use and modify the given program. This enabled the appearance of a community of Linux users and developers, allowing people to freely write code for it with an organic hierarchy of contributors and maintainers. The way the community was organized also implied the creation of multiple subsystems, i.e., smaller sub-communities that are highly specialized in a specific functionality of the kernel.

During the last thirty years, programming practices have evolved, and the Linux community has made efforts to incorporate these changes for kernel development. Recently, many contributors and maintainers have started to focus on adopting methodologies that are common to Continuous Integration (CI). CI is the practice of enabling developers to constantly integrate their code into the shared code repository. It also relates to using techniques to form a development environment where these integrations flow smoothly, such as automating the testing of new code to check if it is ready to be merged.

The Linux kernel, however, has a lot of peculiarities that make it more difficult to apply such CI practices. The way the community is organized and its members' communication with themselves is almost the same from 1991 when the project was published, and this affects how new code additions will be available to the community and how to automate the process of collecting these new patches.

Also, it is important to note that a kernel differs from most software projects because of its nature of interacting with the physical device. This will change the dynamics of how to test the code, as hardware becomes a relevant criterion to validate the code.

This work aims to study the adoption of CI practices in the Linux kernel community, identify the current caveats, and propose and apply a solution to make the process more

efficient. The first chapter will cover in more detail the Linux kernel and its development workflow (which includes the way the code is organized, some frameworks that are used to build the program, and how the community is organized to approve and merge patches).

The next step will be to properly define and describe modern CI and common practices that are employed to maximize the automation of code integration. This includes an explanation of software build and test and how these development stages can be automated for the sake of an efficient application of CI. Finally, there will be a brief overview of the most popular CI tools and services.

With a good background in both the Linux kernel, its community, and Continuous integration, it will be possible to analyze in Chapter 3, from the perspective of the AMD Display subsystem, how CI is currently present in the kernel development and how it is limited by the technical and social aspects. This will lead to the presentation of CI-Tron as a viable tool to deal with the technical complexities of the kernel and the introduction of the Kernel Lore as a possibility to integrate the old mailing list used by the kernel community with CI services.

In Chapter 4, there will be a description of the results of this work, detailing the progress of studying and developing solutions to improve CI for the kernel and impressions with CI-Tron. Chapter 5 contains the conclusion and expectations on how to proceed with the project, based on the results.

Chapter 1

The Linux Kernel

This chapter will cover the basic concepts regarding the Linux Kernel and its development ecosystem. Implementing a CI system for a given program is heavily based on its technical peculiarities, principles, and organization of its user and developer community.

1.1 The Linux Kernel

The operating system (OS) is the most relevant software component of a computer device. It represents the most important abstraction of the software layer, enabling users to easily access their programs and interact with the computer without having to face hardware directly.

It is usually divided into two components: the userspace and the kernel. The first represents the space where higher-level programs are run and are closer to the final user of the system; the latter is the core component of the entire OS, orchestrating processes, managing memory, abstracting the interaction with peripherals, and dealing with I/O operations.

A kernel is, therefore, a very complex software. It is expected to contain many lines of code and specific device drivers to interact with every desired device to be plugged into the system. This also implies the intricate relationship between this program and the hardware that will run it, severely affecting the strategies to test the code properly.

The Linux Kernel was first released in 1991 by Linus Torvalds. The UNIX operating system strongly inspires its traits. Unlike UNIX, Linux is free software. This was a key point for its popularity and worldwide adoption, enabling volunteers to contribute to the project, while the free software philosophy also granted a stronger appeal for personal, commercial, and political usage.

1.2 Free/Libre and Open Source Software

Before diving deep into how the Linux Kernel is developed and maintained, it is necessary to understand that it is a Free/Libre and Open Source Software (FLOSS), and its

development cycle and success are intensely shaped by this fact.

Being a product of the Free Software movement's advocacy, a FLOSS is a program that addresses the following four essential freedoms (GNU, n.d.):

- The freedom to run the program as you wish, for any purpose (Freedom 0);
- The freedom to study how the program works and change it so it does your computing as you wish (Freedom 1);
- The freedom to redistribute copies so you can help others (Freedom 2);
- The freedom to distribute copies of your modified version to others (Freedom 3). Doing this gives the whole community a chance to benefit from your changes.

As a symbol of the Free Software movement, the Linux Kernel is licensed under the GPLv2 (GNU General Public License version 2), ensuring the four essential freedoms while enforcing software derived from it to use the same license.

Because of Freedom 1 and Freedom 3, Linux and other free software must have their source code available, i.e., be open-source software. Every free software is open-source, but the opposite is not always accurate since software can have its code publicly available but not promote other freedoms. Comprehending that the target software is open-source is important to understanding the dynamics of how it is developed and maintained, and therefore, it helps plan the implementation of its CI system.

Because of these characteristics, Linux is a strongly community-oriented software project. It allows its users to modify their own kernel and encourages any user to voluntarily contribute to the source code for the main tree kernel. The free software principles promote a decentralized approach to code, affecting how it is used and developed since code is not restricted to a small development team but anyone in the world who is interested in participating in the process.

1.3 The Linux Kernel and its development workflow

Considering the kernel's historical, technical, and philosophical contexts, it is possible to grasp its peculiarities and plan a proper CI system. The rest of this chapter focuses on the development workflow of the kernel as it is in the present, ignoring the existing efforts of the Linux community to implement proper continuous integration infrastructure.

1.3.1 Kernel repository structure and build system

The Linux Kernel project was expected to have around 27.8 million lines of code in 2020. This implies a fundamental necessity to modularize its code properly. This work will not focus on every aspect of how the kernel tree is organized. Instead, it will highlight the kernel's robust system so that it can adequately compile its coherently defined components.

Compiling all of the mentioned amount of code is not viable. It would take an unfeasible quantity of time to properly compile the code, much memory for the linking process, and a considerable portion of storage to keep the resulting image. However, it is important

to notice that most of the code contained in the kernel tree can easily be discarded when working on a specific portion of the software. The repository contains the implementation of all device drivers supported by Linux, but, for a developer, only the devices present in the testing environment (and the device on which a driver is being implemented, if that is the case) are desired. Also, there are kernel features that are meant for specific development needs, which may not be useful for a specific debugging case, automated testing, and personal use of the compiled kernel.

The Linux community has dealt with the problem with a robust compiling framework: the kernel build system. Almost every directory in the kernel tree contains a Kconfig file, which defines compilation symbols and associates these symbols with the files to be compiled. For a Kconfig, it is possible to add a new symbol and define its properties, which may include:

- if it can be a boolean symbol (compiled(y) or not compiled(n) statuses) or a tristate symbol (compiled(y), compiled as module(m), or not compiled(n) statuses). In this context, a module is a portion of compiled code that is not directly compiled into the kernel but separately and can be dynamically loaded at runtime, allowing more compact kernel images.
- if it depends on other symbols to be enabled, i.e., if the symbol can be enabled only if the other symbols it depends on are enabled, either with (y) or (m).
- a help message explaining which feature will be activated if the symbol is enabled.

From any directory that contains a Kconfig file, there is a Makefile that orchestrates the compilation of the files according to the chosen symbols. The root directory has a central Makefile where the developer can build their desired compilation configurations. It offers different build targets that provide different ways to customize the compilation configuration. Commands such as `make tinyconfig`, `make localmodconfig`, `make defconfig` generate a configuration that, respectively, is minimal, contains only kernel features that are loaded on the host device, or contains the basic configurations that most devices need. There are also `make menuconfig` and `make nconfig`, which provide graphical user interfaces (GUI) where it is possible to navigate through each symbol and set them.

With a very customizable piece of software, new problems arise regarding how it will be properly tested. Different configurations produce different kernel images that may not present the same results during testing. Evaluating results for every possible configuration is also not feasible, but validating minimal kernel images with only the desired driver may not detect every issue. The Syzbot project detects kernel crashes by fuzzing, i.e., providing random configuration options and finding new issues with conflicting symbols.

Besides defining portions of code to be compiled, the root Makefile from the Linux repository also enables the selection of a desired compiler. The compiler difference may generate new compile errors/warnings. Therefore, it is also important for a CI system to try compiling with both options.

Finally, testing the desired feature as a module or built directly into the kernel may also be desired. Covering the code is not feasible for such a vast and customizable project. In this context, defining the proper compilation configurations must be strategic and

cover the optimal amount of code with the minimum possible quantity of compilations to avoid wasting computing resources.

1.3.2 Kernel subsystems

The kernel complexity affects how it is compiled and how its development community is organized. As it has been mentioned before, the kernel is a vast project with completely different but coherently divided interconnected component abstractions. Diving deep into a specific portion of code while ignoring the rest is possible. It is important to define explicitly how the code is divided, as this makes the addition of new contributors feasible, as newcomers may specialize in their area of interest while not being obligated to have a deep understanding of the project itself and even operating systems.

However, the best aspect of this is that it makes it possible to delegate the maintenance of different portions of code to the kernel subcommunities that work specifically on them. It would be impossible for a single team of maintainers to handle thousands of contributions for countless different areas of the project.

The kernel is separated into subsystems. Each subsystem has its own community of contributors. It is also common for some subsystems to be so complex that they are further divided into smaller subsystems, which may benefit from a common API. This is the case of the Direct Rendering Management (DRM) subsystem for kernel graphical capacities, which are fragmented into different development groups that usually work over the DRM code to develop their specific drivers for Graphics Processing Units (GPU) devices.

This partition resulted in the division of the kernel into different development trees. There is the main kernel tree, which contains the code of Linux releases, but developers usually have their own tree on which they work, and the incoming patches are sent in batches to the main tree after a release cycle. Alongside a custom kernel tree, a team of maintainers manages that tree, a specific mailing list for sending patches, and, sometimes, an Internet Relay Chat for further communication.

To promote an easier integration of newcomers to a specific kernel tree, there is, at the root of a kernel tree, a `MAINTAINERS` file with a list of subsystems, each possibly detailed with the maintainers of that kernel community, the tree to work on, the path to the directory with the source code that is developed, and the mailing list. The way each entry is represented is shown by program 1.1.

```
1    List of maintainers
2    ===================
3
4    Descriptions of section entries and preferred order
5    ---------------------------------------------------
6
7      M: *Mail* patches to: FullName <address@domain>
8      R: Designated *Reviewer*: FullName <address@domain>
9        These reviewers should be CCed on patches.
10     L: *Mailing list* that is relevant to this area
11     S: *Status*, one of the following:
12       Supported: Someone is actually paid to look after this.
13       Maintained: Someone actually looks after it.
```

```
14        Odd Fixes: It has a maintainer but they don't have time to do
15         much other than throw the odd patch in. See below..
16        Orphan: No current maintainer [but maybe you could take the
17         role as you write your new code].
18        Obsolete: Old code. Something tagged obsolete generally means
19         it has been replaced by a better system and you
20         should be using that.
21      W: *Web-page* with status/info
22      Q: *Patchwork* web based patch tracking system site
23      B: URI for where to file *bugs*. A web-page with detailed bug
24        filing info, a direct bug tracker link, or a mailto: URI.
25      C: URI for *chat* protocol, server and channel where developers
26        usually hang out, for example irc://server/channel.
27      P: *Subsystem Profile* document for more details submitting
28        patches to the given subsystem. This is either an in-tree file,
29        or a URI. See Documentation/maintainer/maintainer-entry-profile.rst
30        for details.
31      T: *SCM* tree type and location.
32        Type is one of: git, hg, quilt, stgit, topgit
33      F: *Files* and directories wildcard patterns.
34        A trailing slash includes all files and subdirectory files.
35        F: drivers/net/ all files in and below drivers/net
36        F: drivers/net/* all files in drivers/net, but not below
37        F: */net/*  all files in "any top level directory"/net
38        One pattern per line. Multiple F: lines acceptable.
39      X: *Excluded* files and directories that are NOT maintained, same
40        rules as F:. Files exclusions are tested before file matches.
41        Can be useful for excluding a specific subdirectory, for instance:
42        F: net/
43        X: net/ipv6/
44        matches all files in and below net excluding net/ipv6/
45      N: Files and directories *Regex* patterns.
46        N: [^a-z]tegra all files whose path contains tegra
47                        (not including files like integrator)
48        One pattern per line. Multiple N: lines acceptable.
49        scripts/get_maintainer.pl has different behavior for files that
50        match F: pattern and matches of N: patterns. By default,
51        get_maintainer will not look at git log history when an F: pattern
52        match occurs. When an N: match occurs, git log history is used
53        to also notify the people that have git commit signatures.
54      K: *Content regex* (perl extended) pattern match in a patch or file.
55        For instance:
56        K: of_get_profile
57          matches patches or files that contain "of_get_profile"
58        K: \b(printk|pr_(info|err))\b
59          matches patches or files that contain one or more of the words
60          printk, pr_info or pr_err
61        One regex pattern per line. Multiple K: lines acceptable.
```

Program 1.1: *The introductory segment of MAINTAINERS file, describing how each entry is represented*

Each entry of the MAINTAINERS list follows a similar format to the example of the two entries depicted by program 1.2.

```
1   IIO SUBSYSTEM AND DRIVERS
2   M: Jonathan Cameron <jic23@kernel.org>
```

```
3    R: Lars-Peter Clausen <lars@metafoo.de>
4    L: linux-iio@vger.kernel.org
5    S: Maintained
6    T: git git://git.kernel.org/pub/scm/linux/kernel/git/jic23/iio.git
7    F: Documentation/ABI/testing/configfs-iio*
8    F: Documentation/ABI/testing/sysfs-bus-iio*
9    F: Documentation/devicetree/bindings/iio/
10   F: Documentation/iio/
11   F: drivers/iio/
12   F: drivers/staging/iio/
13   F: include/dt-bindings/iio/
14   F: include/linux/iio/
15   F: tools/iio/
16
17   IIO UNIT CONVERTER
18   M: Peter Rosin <peda@axentia.se>
19   L: linux-iio@vger.kernel.org
20   S: Maintained
21   F: Documentation/devicetree/bindings/iio/afe/current-sense-amplifier.yaml
22   F: Documentation/devicetree/bindings/iio/afe/current-sense-shunt.yaml
23   F: Documentation/devicetree/bindings/iio/afe/voltage-divider.yaml
24   F: drivers/iio/afe/iio-rescale.c
```

Program 1.2: *Some entries from the MAINTAINERS file covering the IIO subsystem*

The IIO UNIT CONVERTER is a sub-unit of the IIO (Industrial Input/Output) subsystem and shares some of its information, such as the mailing list and tree (the latter implicitly – a contributor may send a patch from any tree as long as it is updated and synced with the newest code). This shows that the kernel contains a strong but also organic and not rigid hierarchy. This implies that the development process is somehow decentralized, and different kernel communities may develop their own tools and methodologies for development while also sticking to the basic workflow. This is the case for many communities that may, for example, use GitLab to track issues, e.g., Nouveau and AMD. It is not mandatory, but many developers adopt this idea to increase productivity and organization.

This freedom to adapt parts of the development workflow may spread to different aspects of coding. This is where the subsystem hierarchy influences the implementation of CI systems: maintainers are responsible for deciding if they will adopt CI for the development workflow and implement and maintain any infrastructure used.

1.3.3 The kernel development workflow

It has been mentioned before that patches are sent through mailing lists instead of using Version Control System hosts (e.g., GitHub and GitLab). This is a basic rule followed by any maintainer. Contributors must commit their patches using the git software, a Version Control System that handles new code additions as discretized contributions (commits) and allows easy management of each code version. The commits can be sent to a development mailing list with the commands `git format-patch` and `git send-email`, which, respectively, produce a file with a conventional format for sending to mailing lists and send the formatted message from the file to the mailing list.

Some software projects, such as `kworkflow`, provide simplified commands and procedures to send patches. If the patch is correctly formatted, a maintainer can easily review it and apply it to their local tree with the `b4` software and test it. The only problem, however, is that the maintainer can already feel the lack of automation at this step. A manual patch application process is usually done to test the source code. Much of this work tries to fix the lack of automation between receiving a patch from a mailing list and piping it directly to a CI system.

The Linux release cycle

Kernel releases come out every 10 to 11 weeks and are time-based instead of feature-based. The release cycle begins after Linus Torvalds releases a new kernel. This starts a merge window of 2 weeks, where subsystem maintainers send to Linus the patches they have applied to their trees and will be applied to the main tree. The maintainers will be busy sending their patches to Linus and may not be responsive to new contributions during this period. After this period, the first release candidate is released, known as rc1. Then, the rc development is set to focus on bug fixes, and new release candidates (rc2, rc3, etc.) are released weekly until all major bugs and regressions are solved. When this happens, a 3-week "quiet period" begins, containing the week before the release and the 2-week merge window that begins the new cycle. Figure 1.1 shows an example of how a new kernel version would be released.

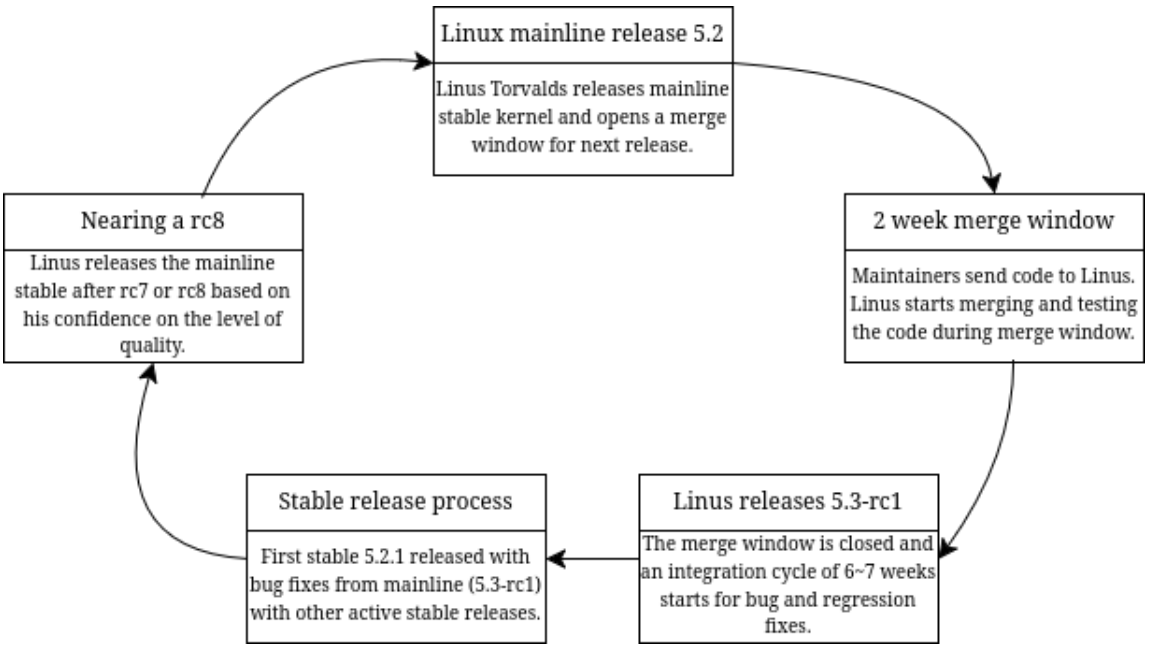

Figure 1.1: *A cyclic diagram representing how kernel releases are developed.*

It is considered stable after the kernel from the main tree is released. Some stable releases become long-term and will get more extended support periods to fix critical bugs, even after new stable releases. This sums up how the kernel is developed at a macro level. However, for a CI contribution, this work will focus on developing it at a subsystem level, i.e., testing and integrating patches, as they are sent to the subsystem mailing list for a maintainer's review and not during a "quiet period" or release candidate development.

Chapter 2

Continuous Integration

In the development process of modern software products, it is common for multiple developers to work over the same codebase simultaneously and have to integrate their work into a common development branch. This approach can lead to different product quality problems, as developers are unaware of others' changes to code. This strategy typically results in conflicting changes that cause new build errors/warnings and the introduction of new bugs. Therefore, naive methodologies for code integration cause extra overhead, as extra time and money are spent exclusively on integration features, especially if the interval between code integrations is long.

In 1991, the term Continuous Integration (CI) was coined to define a solution for managing code integration (Brooch, 1991), where code from different development branches is staged in a common integration branch, and the correctness of code can be asserted. This includes practices such as:

- Addition of new code to a mainstream branch;
- Automated build processes;
- Application of automated tests for every new integration build;
- Triggering new builds for every code integration.

However, it is important to note that despite not being initially considered a CI practice, modern continuous integration also requires a development principle where developers must commit their code constantly, as much as possible. Code additions also become an important communication method between developers. This is especially true for commercial software projects with a dedicated group of contributors who have good internal communication skills and are available to modify the source code constantly.

This fact implies that regular open-source projects do not technically implement true CI (Fowler, 2006). In the context of volunteer and usually anonymous developers, maintainers can't effectively track every small integration to the codebase. Therefore, it is usually enforced that contributors open new merge requests to the mainline branch only when the entire feature is completed, as opposed to the CI model of pushing changes at every buildable commit.

The incomplete adherence of open-source projects to CI practices does not invalidate the consistent efforts of the community to incorporate most of its techniques, such as automation of builds and tests, which are still valuable to reduce the overhead of this development workflow. This chapter will cover most CI practices compatible with open-source projects and common CI tools.

2.1 Relevant CI Practices

A CI job is essentially the entire pipeline cycle of integrating new code, building, and testing it. Implementing a promising and automated pipeline for executing jobs is the most relevant aspect of working with Continuous Integration. The following subsections describe the major steps usually run during a CI job.

2.1.1 Automated Builds

The most important step for testing new code is applying the new changes and asserting that the program can be built from the new source code version. This usually means running a minimal operating system environment and, with only the project source, being able to run the installation scripts, get dependencies, compile and produce a binary (if it is programmed with a compilable language), and do everything else needed to run the product properly.

Automating the process of preparing software for execution can be done with scripting. Having a minimal environment for the build, however, is not trivial. Preparing a specific device with a pure environment is not viable as it requires manual work and management of previous biased artifacts produced by previous builds. Virtual environments are the most appropriate solution for this problem.

Virtualization

Virtualization technologies have been popular in the IT industry for decades. Virtual Machines (VMs) are software abstractions that represent an entire OS and associated hardware (peripherals, storage). This environment is denominated the guest system, while the OS running on the real device hardware is called the host system.

By running multiple guest systems on a single host system, deploying multiple isolated server services on a single device is possible instead of needing multiple dedicated machines. This makes the deployment of services more cost-effective and space-efficient. The usage of VMs also increases overall security, as the programs running on the guest system cannot use the resources of the host OS.

The benefit of VMs is that they can be easily used for software testing. Virtualization often appears at the beginning of the CI pipeline, where a minimal and ephemeral guest environment is invoked to be exclusively used for the specific build. This solves the problem of easily reproducing unbiased testing environments.

The provision of testing environments for CI improved further with the introduction of tools like LXC and Docker. These virtualization solutions provide containers, which behave

mostly the same as VMs but share the host system kernel instead of having their own. This enables the environments to be more lightweight. Containers demand fewer resources from the host device to be deployed, making it more viable to start a new build for a CI pipeline.

2.1.2 Automated Testing

After asserting that it is possible to run the program from the contents of its source code, the following important process for the CI pipeline is to check if the new code additions didn't decrease the quality of the software product. Testing the code is essential for ensuring a stable level of quality.

The quality of software is usually measured with verification and validation procedures, where the terms are usually defined as follows:

- Verification: The act of checking if the implementation of the software behaves as expected (i.e., if it is "being developed the right way").
- Validation: The act of checking if the software produced meets the expected requirements (i.e., if "the right thing is being developed").

Ensuring the quality of software is often associated with preparing a correct test suite that reassures that the product is being correctly validated and verified. There are plenty of different approaches to testing software, and developers must plan the best set of tests that covers most of the code in quantity and quality in an efficient way.

The development of test suites for the Linux Kernel is not the focus of this work but rather the implementation of an infrastructure and pipeline to run existing test suites. Therefore, only fundamental or required software testing concepts will be detailed.

Testing Scope

Tests can focus on verifying and validating different portions of code. Some tests may evaluate a single module, while others may check different modules simultaneously and how they interact with each other. There are three categories of tests regarding their level:

- Unit tests: Tests that evaluate exclusively a single component.
- Integration tests: Tests where the interaction between different components is evaluated.
- System tests: Tests where the entire cycle of software execution is tested, also known as end-to-end testing (E2E).

Artifacts

During a CI pipeline build or test stage, artifacts are usually produced. An artifact is any file produced during the build or test stage that may contain interesting information about the executed job. Examples of typical artifacts are compiled binaries of the software product.

2.1.3 Code Coverage

It is very common for some projects to have jobs or job stages to calculate the program's code coverage. Code coverage is the practice of checking the percentage of the source code that is executed during the test suite.

Code coverage is an important statistic for a software project and is useful for tracking the state of the test suite for a given program. There are plenty of tools to evaluate code coverage that store the results in artifacts statically.

Popular CI services provide features for parsing these artifacts and displaying a dashboard to visualize the resulting code coverage better.

2.2 Common CI Services and Tools

There are plenty of software solutions for implementing CI pipelines. A distinction will be made for defining the programs described in this section:

- "CI service" refers to programs that provide a complete framework for implementing a CI pipeline and system.
- "CI tool" refers to programs that improve overall workflow for executing CI jobs but do not represent a proper CI solution.

This section will cover popular CI tools and services explored during this work, starting from services that have general purposes and are also used for regular userspace projects, and then funneling to the tools explicitly used in the kernel community.

2.2.1 Popular CI Services

GitHub Actions and GitLab CI

Most modern software projects are hosted on VCS Hosts, such as GitHub and GitLab. These VCS Hosts are the most popular and offer CI services exclusively for defining jobs and pipelines. Since the source code of the projects is also hosted there, the integration between the code repository and CI pipeline is seamless.

GitHub Actions and GitLab CI have different policies but work very similarly. Code maintainers define the general steps of the CI pipeline(s) with the configuration syntax. Each service provides extensive documentation on how to write these files and the available keywords for defining the job. Finally, these files are stored in the root of the code repository.

From the repository page, the code maintainer can also configure, via the service web interface, more generic configurations, such as which computer resource nodes are available for executing jobs (VMs hosted by the service providers or self-hosted devices and environments), also known as runners.

The CI services from VCS Hosts are usually connected to the respective code repository they host, but jobs do not necessarily have to be triggered exclusively by modifying the

existing source code. Configuring scheduled jobs via the service web interface or triggering jobs remotely with REST APIs is also possible. This approach allows developers to trigger jobs periodically or according to an external event.

These CI services stand out because they gather all relevant information in one place, integrating version control and CI capabilities. Their widespread adoption is mainly due to the seamless integration with the repositories they host, their rich feature sets, and user-friendly interfaces. GitHub Actions and GitLab CI enable developers to customize pipelines to fit the specific needs of their projects, whether it is running automated tests, deploying applications, or generating reports.

These tools also provide built-in security mechanisms, such as secrets management and permissions control, ensuring that sensitive information (e.g., API keys) remains secure. Additionally, they support extensive plugin ecosystems and community-contributed actions or templates, allowing for rapid CI pipeline implementation without reinventing the wheel.

Jenkins

Jenkins is one of the most popular and mature open-source CI tools available. It is widely adopted in various industries and stands out due to its flexibility and extensibility. Unlike GitHub Actions or GitLab CI, Jenkins is self-hosted and does not tie directly to any specific VCS platform, making it a platform-agnostic solution.

Jenkins relies on a vast array of plugins that allow developers to extend their capabilities to support different build tools, testing frameworks, and deployment pipelines. Configurations can be managed through its web interface, scripts, or declarative pipeline syntax in code repositories. One of Jenkins' key advantages is the level of control it provides to the user over the entire CI/CD process. However, this flexibility often comes at the cost of complexity in setup and maintenance. Ensuring that the system scales well and remains secure typically requires a significant investment in resources and expertise.

Travis CI

Travis CI is a cloud-based CI service that gained early adoption in the open-source community due to its generous free-tier offering for public repositories. It simplifies the process of defining CI pipelines using a configuration file placed in the root of a repository, similar to GitHub Actions and GitLab CI.

While Travis CI popularity has waned in recent years due to increasing competition and limitations in its free-tier policies, it remains a viable option for teams that value simplicity and a lightweight solution. Its primary limitation lies in its restricted features compared to more modern services and a lack of integration flexibility in tools like Jenkins or GitLab CI.

2.2.2 Tools for Kernel CI

For kernel development, where the scale and complexity of the projects demand specialized solutions, several tools (described below) have emerged to address unique challenges.

KernelCI

KernelCI is a collaborative project explicitly aimed at testing and validating Linux kernel changes across various hardware and configurations. It serves as a community-driven CI framework that integrates contributions from various organizations and developers to provide extensive test coverage for the Linux kernel.

KernelCI runs automated tests on kernels built from different branches, ensuring that changes do not introduce regressions across supported platforms. It uses a distributed architecture, where the testing infrastructure comprises various labs and devices the community contributes. The results are aggregated and made publicly accessible through its dashboard.

LAVA

LAVA (Linaro Automated Validation Architecture) is another tool often used in kernel development to automate hardware testing. It is designed to manage and schedule test jobs across multiple physical and virtual devices. LAVA excels in scenarios where real hardware is required for validation, such as testing device drivers or firmware. By combining LAVA with other CI tools like KernelCI, kernel maintainers can achieve comprehensive test coverage, including functional validation and performance benchmarking.

These tools and services provide a wide range of options for integrating Continuous Integration practices into software development workflows, whether for general-purpose software or specialized projects like kernel development. By adopting the appropriate solutions, developers can streamline their workflows, minimize errors, and improve the overall quality and reliability of their software products.

Chapter 3

Developing a CI System for the AMD Display Subsystem

Building on kernel development, software testing, and Continuous Integration concepts, this chapter focuses on applying this knowledge in the context of GPU kernel driver development, specifically for the AMD display subsystem. The main objective of this work is to study and develop a CI infrastructure to be used primarily for kernel graphics.

3.1 Introduction to the AMD Display

The desired CI infrastructure being developed here is meant to use only free software and to be as generic as possible to satisfy the needs of most kernel development workflows. However, the primary target is the AMD GPU driver development community, beginning with the AMD display subsystem.

3.1.1 The DRM Subsystem

The Direct Rendering Management (DRM) subsystem is one of the most relevant kernel subsystems. It is responsible for interfacing with most GPUs by exposing an API that user-space programs can access to request more complex graphical operations from the GPU hardware.

The DRM is the kernel component that has exclusive access to the GPU. It initializes and maintains the hardware resources. Whenever a user-space program wants to use the GPU, it sends requests to the DRM, which allocates the required resources. This ensures that the DRM correctly manages how graphical resources are distributed and used by programs, avoiding conflicts and ensuring that resources are not used simultaneously by different processes.

The DRM community continues to grow as time passes. More kernel responsibilities have been assigned to this subsystem over time. The DRM scope has expanded to include handling the framebuffer, screen mode settings, and GPU switching, which became a new responsibility as the simultaneous use of integrated and discrete GPUs gained popularity.

3.1.2 AMD Display

The entire codebase for the AMD drivers is extensive and fragmented into minor subsystems. This work aims to provide a CI infrastructure for the AMD Display subsystem. The AMD Display is AMD's display engine. This section will not describe its implementation in detail but instead focus on the development community and relevant information for planning a CI system.

The AMD Display is divided mainly into two separate components:

- **The Display Core (DC):** Contains code for OS-agnostic components. It represents hardware-level code that does not depend directly on the operating system.
- **The Display Manager (DM):** Contains code that depends on the operating system (in this case, systems running Linux). It includes hooks to interact with the base amdgpu drivers and the DRM itself.

There is a significant effort to validate the code for the DC subsystem. Exhaustive tests are run on many AMD GPUs. The test suite used for checking the validity of the code is the igt-gpu-tools, the most comprehensive collection of tests for kernel GPU drivers. The internal AMD CI farm, i.e., network of DUTs, also checks for compilation issues across various architectures.

One of the most significant issues with AMD internal CI is that it is not open. Maintainers must apply patches manually and pipe them into their CI infrastructures. There is no dynamic, real-time public overview of results.

An open and pre-merge automated CI system would enhance the existing development workflow and enable a transparent process for contributors.

3.2 A Bare-Metal CI Infrastructure with CI-Tron

Designing a proper CI system for kernel code presents many challenges. Unlike regular user-space software, kernel code cannot be run on a simple virtualized environment on any device, as the underlying hardware plays a crucial role. Therefore, allocating the required hardware and resetting it are fundamental concerns. A promising solution for a bare-metal CI system, CI-Tron, is being developed and has been chosen to address these challenges. CI-Tron serves as a bare-metal gateway, orchestrating the testing process across the devices in a CI farm.

Following Martin Roukala's blog post series for implementing a CI system, which will be summarized in the following sections, is key to understanding how it works under the hood.

3.2.1 The Basic Principles for a Good Bare-Metal CI

Two major principles were adopted when planning the generic CI system that later became CI-Tron (Roukala, 2021):

- **Stability:** If a test is re-executed, it must return the same results.

- **Reproducibility:** It must be possible to run the same tests on a device with the same hardware and obtain the same results.

These principles imply the need for a portable tool that follows modern container environment principles. Deploying and preserving isolation should be easy, ensuring the system is not affected by non-deterministic external factors.

In a bare-metal context, additional concerns must be addressed to achieve these objectives:

- The system should power cycle the machine between tests to reset the hardware.
- Devices must be diskless, as a disk could store biases from previous testing or usage.
- Pre-compute as much as possible outside of the test machine to minimize the test environment's impact on evaluation.

3.2.2 Preparing a device under testing

Some aspects must be considered to prepare a DUT for CI correctly according to CI-Tron standards.

Power cycle of a DUT

It must be possible to boot and shut down the device remotely to run new kernel images for testing. As pointed out, CI-Tron also requires that a device be turned on and off to ensure a hardware reset. This can be achieved with some hardware and configuration.

First, the device must be booted remotely. This is easily done with the Wake-on-LAN feature supported by most modern motherboards. However, there is no simple solution for powering off a machine. It is necessary to cut its energy supply and wait a significant amount of time.

This is trickier to set up, as the device must first be configured appropriately for power cycling. The DUT must be powered exclusively from the power cable. This means portable devices such as laptops must have their internal batteries removed. Then, it must be connected to a Power Distribution Unit (PDU). Having the DUTs and the gateway connected to the PDU will allow the latter to manage the power supply for the former devices.

Netbooting the DUT

Another fundamental aspect of preparing a DUT, besides its power cycle, is deploying the proper kernel to be evaluated. Unlike other CI solutions, CI-Tron uses the iPXE bootloader, which can download a remote kernel and use remote kernel images in the netbooting process.

The Preboot Execution Environment (PXE) is a specification for a client-server environment for booting software from the network instead of locally. The iPXE bootloader implements such a specification and is a tiny bootloader that can be installed on a disk or USB stick.

Serial connection

Finally, a DUT must have a serial connection to emulate the screen and keyboard via a serial console. This requires that the DUT's motherboard supports a serial port.

Using Secure Shell (SSH) connections to interact with the Operating System from the DUT is easier to set up but does not cover the entire process. Interacting with and troubleshooting a device via SSH is impossible if it fails to start the SSH server.

3.2.3 Deploying the testing environment to a DUT

To properly test a kernel, deploying the entire Operating System containing the kernel under testing is desirable since the testing scripts and resources will be located at the user-space level. Martin broke the Operating System into three deployable units (Roukala, 2022a):

- The kernel: the software component that will be evaluated.
- The rootfs (user-space): all user-space-level dependencies to test the kernel.
- The initramfs: as the name suggests, it is the initial content to fill the RAM. It contains drivers and firmware to access the user-space image and initial scripts to be run during the early boot sequence. It is the only optional component to be deployed, as it is possible to compile drivers and firmware built directly into the kernel.

Deploying the kernel and initramfs is as simple as netbooting with iPXE, which was previously decided upon and configured while setting the DUT. When booting, the DUT will request an IP address from the DHCP server and download the kernel and initramfs. This is an excellent method as it can be performed on diskless devices, which is beneficial for reproducibility.

Deploying the rootfs is more complicated. A regular solution would be to change root with `chroot`, install a distro of choice, install the dependencies, and compress the rootfs. It is tricky as it is difficult to maintain – the rootfs underlying system tends to get many updates, requiring the repetition of the process from the start with no cached operations.

However, a more interesting, modern, and elegant solution is possible with containers. Instead of deploying compressed rootfs to the DUT, it is also possible to package the rootfs in a container. The container is a lightweight solution that is easy to maintain and reproduce, as containers only require their images to be rebuilt to update the system. Most build stages are cached, so only new operations are effectively computed.

Using boot2container for the CI system

The only problem with packaging the rootfs in a container is optimally entering the container after the booting process. A simplified and likely suboptimal process to boot a DUT directly to a container is as follows:

1. Download a pre-configured initramfs with the kernel from the netboot process. This initramfs will be responsible for initializing the device and executing the next steps.
2. The DUT connects to the network and downloads the container.

3. The DUT initializes the container.

The main concern regarding this model is that steps 2 and 3 will likely be executed by a container runtime, like Docker or Podman, which are larger. Inserting it into the initramfs would significantly increase its size. This is not viable if the DUT is netbooted, as every iteration of the process requires a new download of the initramfs.

The `boot2container` project is a compact and optimal initramfs designed explicitly for easily netbooting a device directly to a containerized environment. It consists of an exhaustive attempt to find alternatives to interact with containers without using container runtimes directly by breaking down their main procedures and using more compact software alternatives for each step of running a container.

3.2.4 The gateway

Alongside the CI farms, it is also necessary to implement the proper CI gateway, which will be responsible for assigning jobs for DUTs and deploying the kernel images for testing. CI-Tron is meant to be the software that runs on the CI gateway.

First, the CI gateway is the only device in a CI farm connected to the Internet and the private network where the DUTs are connected (Roukala, 2022b). The Internet is required as the gateway will likely expose the runners to users, either directly or via GitLab/GitHub, and the private network is necessary to communicate with DUTs, which will not be connected to a public network. The relationship between the gateway and the DUTs can be seen on the figure 3.1.

Since the CI gateway is publicly exposed to the Internet, it is vulnerable to attacks and must be constantly updated. Like other server services, CI-Tron can be installed locally on the device and directly updated, but this is inadequate for software that must always be available online to respond to user requests. Updates must be atomic to ensure the software is not partially updated while active, and it must be possible to roll back from a broken update easily. Because of this, it is advisable to run CI-Tron inside a container while booting the gateway directly into it with `boot2container`. As mentioned, updating a container requires only pulling the newest image and running a new container from it. If the update negatively affects the system, replacing the newest container with an older one that works is possible. This also enables an easier way to test a new version, as deploying containers is easier to maintain than installing all the dependencies locally on a device.

Live-patching the container

Sometimes, however, it is impossible to update a program by shutting its container down and deploying a new one (Roukala, 2022c). It may be busy performing an important task. In this case, it is possible to provision newer versions of the running programs and dependencies with the Ansible software. This tool automates the management of devices by remotely running a defined script on each device, which may include upgrading existing software.

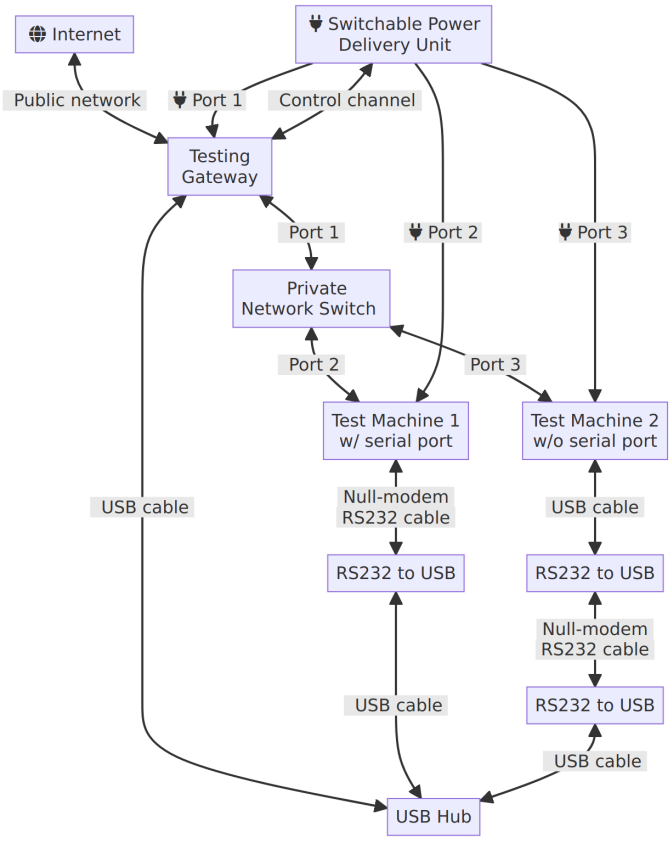

Figure 3.1: *An overview of an example CI-Tron infrastructure, with the gateway and two DUTs.*

3.3 Integrating a CI system with a mailing list

Another problem arises with the actual kernel development workflow: the usage of mailing lists. Most open-source projects use a VCS host for development. These services have built-in CI services (e.g., GitLab CI and GitHub Actions) that are automatically triggered with new merge/merge requests. This does not apply to mailing lists; therefore, a workaround must be used to enable the automation of CI pipelines from patches sent through e-mail. There are two possible solutions: migrating to a new workflow model or working with the mailing lists, each having benefits and drawbacks.

Many developers in the community openly advocate for the end of mailing lists for kernel development. The problem, however, is that the current system is deeply ingrained into the development workflow, and many contributors do not reject it entirely. New possibilities are constantly being discussed, but skepticism is also faced while proposing radical changes. As a result, the future of the current workflow is uncertain but will likely remain the same in the immediate future.

Projects like *kworkflow* and *patch-hub* offer a new perspective on the current contribution model. Instead of changing it entirely and starting with something else, these projects propose providing new tools to work alongside the current workflow and increase its efficiency. This work follows this path, aiming to provide further integration between the existing mailing lists and bare-metal CI infrastructures.

3.3.1 Retrieving new e-mails

There are many options to track new e-mails from mailing lists. An immediate solution is to implement a simple tracking bot. It could be subscribed to a specific development mailing list and register the new e-mails it receives. However, the process of subscribing to a mailing list to receive e-mails is unnecessary for tracking patches, as there are online services that already register the messages from kernel lists. The only problem is piping the messages such services collect to a desired CI pipeline.

One of the most well-known patch tracking projects is `Patchwork`. There are already many efforts to connect it to other services. There are initiatives to create Patchwork webhooks to trigger other programs, scripts and code to pipe Patchwork patches elsewhere. This work, however, uses another tool for properly monitoring new patches: the kernel lore.

The kernel lore

The kernel lore is a public web archive for kernel mailing lists, and its interface can be seen on figure 3.2. It is simple, contains a not-well-documented but robust API, and efficiently registers new messages in real time. Other recent projects rely on the kernel lore, e.g., `patch-hub`.

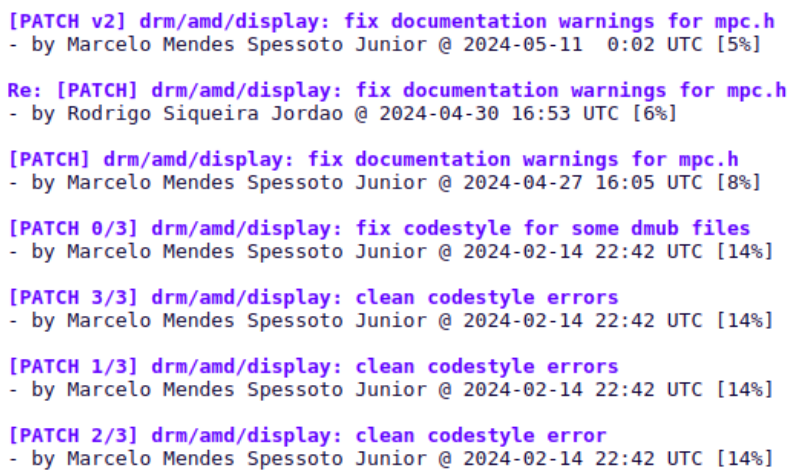

Figure 3.2: *The kernel lore page for the `amd-gfx` mailing list, with the most recent messages.*

One of the deliverables of this work is the `lore-fetcher` software, a program that constantly queries for new patches in the mailing list and aims to pipe them to another program in the CI pipeline.

Chapter 4

Results

This chapter describes the overall progress of the development of the CI system, presenting the different components of the CI pipeline and their execution. Some choices persisted during the entire process, while others were experimental and involved experimenting with multiple approaches. Each section describes a different stage of the prototypal CI system, analyzing the tools and technologies used, the approaches adopted, and the benefits and drawbacks of each implementation strategy tested.

4.1 Tracking new patches

As mentioned in the previous chapter, adopting mailing lists for patch submission poses a problem when implementing automation for CI systems. With a regular VCS host, it is easy to configure an automated CI system that tests each new merge request or commit in a default or customized testing environment. Mailing lists have a generic purpose and are not meant to provide complex and feature-rich development environments. This implies that improving development workflows is often achieved by removing the usual overhead of using mailing lists for a given context rather than directly improving them. Kernel development tools and mailing lists are not directly connected but provide interfaces to facilitate their use in a specific technical context. Commands such as `git format-patch` and `git send-email`, previously mentioned, adopt this approach. Patchwork and Kernel Lore are public interfaces for accessing, from the web, public discussions and patch submissions within the kernel community.

The kworkflow project, also known as kw, simplifies the development process even further by providing easier usage of the existing facilities offered by the community. For example, it gives abstractions over the kernel build system and configuration management mentioned in Chapter 1. Additionally, it has improved most workflows involving mailing lists. Its patch-hub feature uses the public API from Kernel Lore to provide a similar service through a command-line interface (CLI), making it more convenient to track community development. Despite recent efforts to introduce webhooks to Patchwork, this work is heavily inspired by the patch-hub implementation to track patches due to its more straightforward approach. Using similar queries to get the most recent patches, a small

program called lore-fetcher was written in the Go programming language to run as a daemon and verify if a new patch was found. If it detects a new contribution, it uses the API of other CI services (e.g., Jenkins) to trigger new automated pipelines.

4.1.1 The lore-fetcher application

Requesting new patches from the kernel lore

The Kernel Lore has an API that was extensively studied while developing the Kworkflow patch-hub. The lore-fetcher application uses almost the same query as the Kworkflow patch-hub:

- q=rt..+AND+NOT+s:Re allows composition of the rt.. and NOT+s:Re filters. The latter filter removes any message that is a reply to another, ensuring that only patches (which are not replies to other messages) are collected. The former filter is necessary to allow filters with a NOT prefix. Collecting only patches is particularly useful for the context of lore-fetcher, as the application is meant to track patches for CI pipelines. Therefore, any other type of message sent to mailing lists is unnecessary.
- x=A ensures the response from the API is an Atom Feed in Extensible Markup Language (XML) format. This is necessary for properly preprocessing the response. Although kworkflow also uses this filter, it is beneficial for lore-fetcher, as the Go programming language allows efficient parsing of the XML format.

Therefore, the lore-fetcher application queries the newest messages with the following query:

https://lore.kernel.org/<mailing-list>/?q=rt:..+AND+NOT+s:Re&x=A

The <mailing-list> expression is replaced by the mailing list specified by the user before the execution of the program. This is set during configuration and will be discussed in greater detail later. The program is expected to run as a daemon, i.e., to keep running in the background. The application constantly makes new queries to the kernel API and processes the results, triggering the next CI steps if a new patch is found. The requests are made using Go's built-in HTTP package.

Processing API responses

As previously mentioned, Kernel Lore returns the application response in XML format. The Go programming language, known for its wide array of built-in modules, provides an easy solution for parsing XML. The `encoding/xml` package provides methods for performing Marshalling and Unmarshalling of data. Marshalling transforms the memory representation of an abstraction, such as an object, into an ideal format for storing or transmitting it, while Unmarshalling does the opposite. In the context of lore-fetcher, the XML package is used to unmarshall data represented in XML format into data structures during the program execution. The following methods were used:

- `func NewDecoder(r io.Reader) *Decoder`: Creates a new XML parser by reading the XML content of the `r` parameter.

- `func (d *Decoder) Decode(v any) error`: Unmarshalls the data stream of the decoder `d` into the memory address `v`.

The main idea is to declare a new decoder with the `NewDecoder` method that parses initial XML data, which is the response from Kernel Lore. Then, the `Decode` method stores the Decoder's stream into a proper internal struct. Before Unmarshalling, it is necessary to declare a struct type where the data can be stored. This is done by declaring a struct type in Go and tagging each field with `xml:"<xml-tag>"` to ensure proper mapping between XML attributes and struct fields.

The XML structure of Kernel Lore responses is complex, but it is possible to notice that each message is wrapped in `<entry></entry>` tags containing tags such as `<author>` for the author's name and email, `<title>` for the message title, and `<link>` for accessing the patch page on the web. An example is shown by program 4.1.

```
1    <entry>
2      <author>
3        <name>Srinivasan Shanmugam</name>
4        <email>srinivasan.shanmugam@amd.com</email>
5      </author>
6      <title type="html">[PATCH 4/5] drm/amdgpu: Add missing &amp;#39;inst&amp;#39;
           parameter to VCN v2.5 clock gating functions</title>
7      <updated>2024-11-11T02:46:33Z</updated>
8      <link href="http://lore.kernel.org/amd-gfx/20241111024612.1881727-4-
           srinivasan.shanmugam@amd.com/"/>
9      ...
10   </entry>
```

Program 4.1: *The XML representation of an entry from the kernel lore*

The struct from program 4.2 was defined to store the contents of each entry:

```
1    type Feed struct {
2      Entries []struct {
3        Name string `xml:"author>name"`
4        Email string `xml:"author>email"`
5        Title string `xml:"title"`
6        Link struct {
7          Href string `xml:"href,attr"`
8        } `xml:"link"`
9      } `xml:"entry"`
10   }
```

Program 4.2: *The struct abstraction where the lore entries are stored*

During development, it became necessary to enable support for decoding XML in the ASCII-US encoding used by Kernel Lore responses, as the `xml` module only decodes UTF-8 by default.

Other explored features

Two additional features were explored while developing the `lore-fetcher` application. The first is a configuration parser capable of parsing configurations from a TOML file.

This is especially relevant for packaging the application in a containerized environment. The implementation was straightforward using the Go `viper` library. The second feature allows the submission of automated email messages via SMTP to share CI results with contributors.

4.2 Implementing a CI Pipeline for Kernel Compilation and Exploring LAVA

The initial attempt at implementing a pipeline for CI involved tracking new patches, sending them to a Jenkins pipeline to build the kernel image, and integrating Jenkins with a LAVA instance to deploy the image and collect results.

4.2.1 Configuring Jenkins

Integrating lore-fetcher with a CI pipeline required triggering a Jenkins build using the identified patch. This step necessitated the configuration of a Jenkins server and its agents and the definition of the build job.

Configuring the Jenkins Server and Agents

The Jenkins architecture is designed to support distributed build environments, where job execution is scheduled across various computing nodes, ranging from physical devices to virtualized environments such as containers or VMs. These nodes are called agents in Jenkins, while the central server is the controller.

Configuring the controller is typically straightforward, as the installer wizard guides the user through the process. It installs essential plugins to cover the most common requirements and gets the server operational and ready to set up agents and jobs. Setting up Jenkins agents is more complex. Jenkins provides plugins for almost every type of agent, each with its own documentation and setup procedures. One important distinction in configuring agents is choosing between setting up nodes or clouds.

Nodes are individual agent units that must be manually configured. Examples include SSH agents, which can be static containers, VMs, or physical devices equipped with an SSH server and Java to allow the controller to trigger them for building and running jobs. Clouds dynamically provision new agents as needed by the controller. These may include Docker daemons (communicated via API to spin up new containers), Kubernetes clusters, or cloud providers such as AWS and Azure. Configuring an agent involves consulting the relevant plugin documentation and setting up both the node/cloud and the controller to interact correctly. For experimentation purposes, a more straightforward approach to agent configuration was employed. An alternative solution is using the Docker Pipeline plugin for Jenkins, which allows the definition of container-based agents directly within the job pipeline code, eliminating the need for additional agent configuration. The following subsection explains the steps for defining such a text-based pipeline.

Jenkins Pipelines

A Jenkins Pipeline is a job definition written in Groovy that automates the execution of a job. Its syntax typically includes specifications for the agent, parameters, and the job stages. Each stage consists of steps, including executing shell commands, printing output, or leveraging advanced features provided by plugins. For this work, a Pipeline was created to define a Docker agent within the Pipeline code, install the necessary dependencies for building and testing the kernel, configure Git to apply the fetched patch, set the kernel's tinyconfig build configuration, compile the kernel, and clean up the environment after the build process.

The program 4.3 shows the final result.

```
1    pipeline {
2            agent {
3               docker {
4                  image 'ubuntu:latest'
5               }
6            }
7            parameters {
8               string(name: 'PATCH')
9            }
10           stages {
11              stage('Install dependencies'){
12                 steps {
13                    sh 'apt update -y'
14                    sh 'apt install -y b4 git bc binutils bison dwarves flex gcc
                           git gnupg2 gzip libelf-dev libncurses5-dev libssl-dev
                          make openssl pahole perl-base rsync tar xz-utils'
15                 }
16              }
17              stage('Setup git configs'){
18                 steps{
19                    sh 'git config --global user.email "jenkins@ci.com"'
20                    sh 'git config --global user.name "jenkins"'
21                 }
22              }
23              stage('Test tinyconfig Compilation'){
24                 steps {
25                    sh 'rm -rf linux'
26                    sh 'git clone --depth 1 "https://gitlab.freedesktop.org/
                           agd5f/linux.git" linux'
27                    dir("linux"){
28                       sh "b4 shazam ${params.PATCH}"
29                       sh 'make tinyconfig'
30                       sh 'make'
31                    }
32                 }
33              }
34              stage('Cleanup'){
35                 steps {
36                    sh 'rm -rf linux'
37                 }
38              }
39           }
```

```
40          }
```

Program 4.3: *The Dockerfile for the Jenkins service*

The cleanup steps performed before and after building the patch were quick solutions to problems encountered during job execution and required further analysis to identify more robust and efficient solutions.

The PATCH parameter, defined at the beginning of the pipeline and used to apply the patch with the `b4 shazam` command, contains the href link to the patch. As a parameter, its value is specific to each execution and must be manually provided by the Jenkins administrator whenever the job is triggered via the dashboard. Such jobs, where input parameters are required, are referred to as parameterized jobs.

Even when parameterized, Jenkins jobs can be triggered remotely by making POST requests to the job's URL, including the parameter values and an authentication token as query parameters. While additional authentication steps are typically required, these can be bypassed using the Build Authorization Token Root Plugin, which was adopted for pipeline experimentation. Modifying the lore-fetcher to send these POST requests whenever new patches are received makes it possible to trigger Jenkins jobs in real-time for custom patches.

Writing the Configuration as Code

Jenkins supports defining its entire setup as Code, using popular plugins that allow the environment configuration to be described in files. These plugins can parse the configuration and automatically recreate the setup. This approach, known as "Configuration as Code," is a cornerstone of DevOps practices. Written configurations are easier to maintain, share, reproduce, and modify while reducing the likelihood of human error during repetitive environment setups.

Two essential plugins were utilized to achieve this:

- Jenkins Configuration as Code (JCaC): This plugin enables defining the complete Jenkins server configuration, including agents, credentials, and other settings, in a YAML file. JCaC parses this configuration and applies it at server startup.
- Job DSL: This plugin allows jobs to be defined entirely as Code. JCaC can invoke the Job DSL plugin during the server's boot process to create jobs based on the provided configuration.

Packaging Applications in a Docker Environment

To validate the deployment of the patch tracker application alongside a Jenkins server configured to listen for new job triggers, virtualization was employed to ensure the pipeline's reproducibility. Docker Compose, a tool for orchestrating multiple Docker containers, was used to define and manage the environment. This required creating a `docker-compose.yml` file to describe the services to be containerized. By writing Dockerfiles for both the lore-fetcher and Jenkins services, specifying the steps necessary to configure these services, and combining them in a Docker Compose file, a fully operational

pipeline for patch compilation was implemented. The resulting file structure for the Docker Compose environment is shown below on figure 4.1.

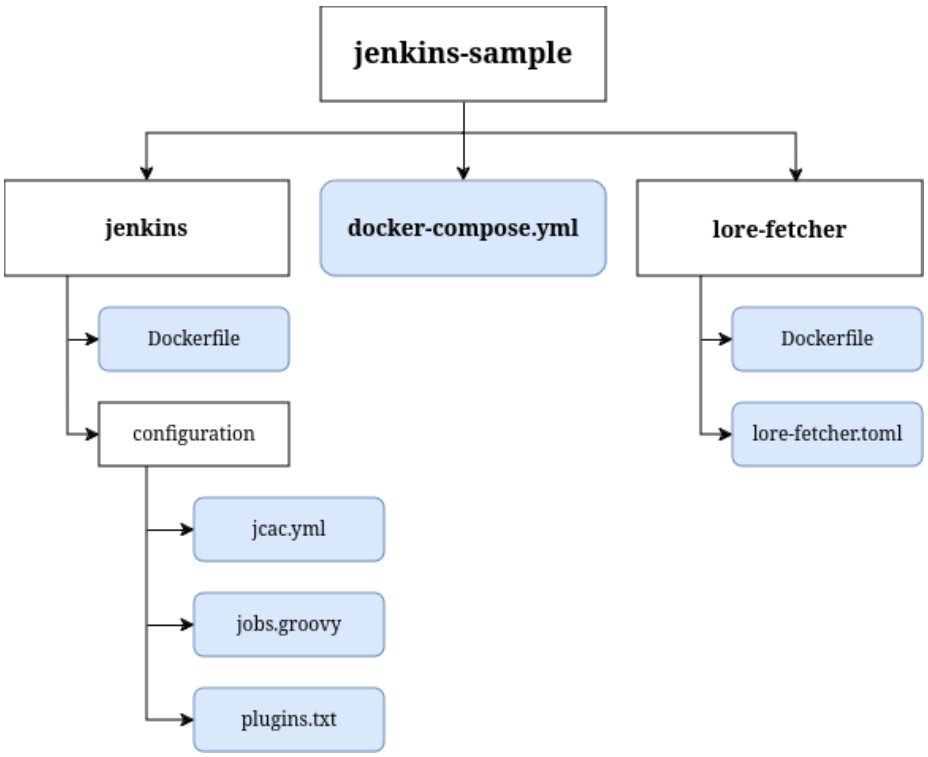

Figure 4.1: *The file hierarchy defining the Docker Compose environment that runs the pipeline. Rounded blue items represent files, while squared items represent directories.*

Considering the `jenkins-sample` directory as the root of the hierarchy, the CI services had dedicated directories with their respective configuration files. For the Jenkins service, a `configuration` directory contained the following:

- `jcac.yml`: Defines configurations for the Jenkins Configuration as a Code plugin, responsible for parsing general settings for the server master node.
- `jobs.groovy`: Contains job definitions for the Job DSL plugin. A hook in the `jcac.yml` file ensures `jobs.groovy` is parsed into Jenkins jobs when the server boots for the first time.
- `plugins.txt`: Lists the plugins required for the dockerized Jenkins instance, such as pipeline-related plugins and configuration parsing plugins. This file is referenced in the Dockerfile to initialize the service with the necessary plugins.

The Dockerfile for the Jenkins service was made as shown by program 4.4.

```
1   FROM jenkins/jenkins
2
3   USER root
4   COPY /configuration/ /usr/local/configuration/
5
6   # Install docker so it can create Docker container from pipeline
7   RUN apt-get update -y && \
```

```
 8       apt-get install -y ca-certificates curl && \
 9       install -m 0755 -d /etc/apt/keyrings && \
10       curl -fsSL https://download.docker.com/linux/debian/gpg -o /etc/apt/
             keyrings/docker.asc && \
11       chmod a+r /etc/apt/keyrings/docker.asc && \
12       echo \
13       "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker
             .asc] https://download.docker.com/linux/debian \
14       $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
15       tee /etc/apt/sources.list.d/docker.list > /dev/null && \
16       apt-get -y update && \
17       apt-get install -y docker-ce docker-ce-cli containerd.io docker-buildx-
             plugin docker-compose-plugin
18
19    RUN jenkins-plugin-cli -f /usr/local/configuration/plugins.txt
```

Program 4.4: *The Dockerfile for the Jenkins service*

Using the base Jenkins container image from the official Docker registry, the Dockerfile copies configuration files from the host system and mounts them into the container's `/usr/local/configuration` path. It installs the necessary Docker dependencies to enable the instantiation of virtual container agents and uses a Jenkins CLI command to install the plugins listed in the `plugins.txt` file. The configuration defined in the `jcac.yml` file is then parsed by the JCaC plugin upon its installation.

The configuration for the `lore-fetcher` service is more straightforward. It involves cloning the `lore-fetcher` repository, building the program, and copying configuration files from the host system. The Dockerfile for this service is as represented by program 4.5.

```
1    FROM golang
2
3    WORKDIR /root/
4    RUN git clone https://github.com/MarceloSpessoto/lore-fetcher.git
5
6    WORKDIR /root/lore-fetcher
7    RUN ./install.sh
8
9    COPY lore-fetcher.toml /etc/lore-fetcher/lore-fetcher.toml
10
11   ENTRYPOINT [ "lore-fetcher", "--fetch" ]
```

Program 4.5: *The Dockerfile for the lore-fetcher service*

The final `docker-compose.yml` file, defining the environment where the containers will be executed is shown by the program 4.6

```
1    services:
2      jenkins:
3        build: jenkins
4        ports:
5          - "8080:8080"
6        volumes:
7          - /var/run/docker.sock:/var/run/docker.sock
8        environment:
```

```
 9         - JAVA_OPTS=-Djenkins.install.runSetupWizard=false
10         - CASC_JENKINS_CONFIG=/usr/local/configuration/jcac.yml
11     lore-fetcher:
12       build: lore-fetcher
```

Program 4.6: *The final docker-compose.yml file*

It defines two services, `jenkins` and `lore-fetcher`. The former has its port **8080** exposed at the host port **8080**, the host docker socket mounted at it (to allow agent containers to be instantiated from it), and the following environment variables:

- `JAVA_OPTS=-Djenkins.install.runSetupWizard=false`: Configure Jenkins to disable the interactive setup wizard at the first start. Crucial for containerized services that are expected to be automated and have no manual intervention;
- `CASC_JENKINS_CONFIG`: When the JCaC plugin is installed, it searches for a configuration file to parse automatically. It searches the file in the path defined in this variable.

4.2.2 Configuring LAVA for Basic Virtual Deployments

After successfully compiling kernel patches, the next step of the prototyped CI pipeline involves deploying the resulting images onto a DUT (Device Under Test) and executing an appropriate test suite. The initial attempt to implement this stage involved exploring LAVA as the primary tool.

As previously mentioned, the LAVA architecture consists of a master node and worker nodes, but implementing this setup in a real-world scenario is more complex than initially expected. Workers must be connected to a set of DUTs requiring prior configuration. A summary of the configuration process is provided below.

Preparing a virtual environment to simulate nodes and devices to gain familiarity with the tool before acquiring real hardware proved more challenging than anticipated. LAVA requires at least one master node and one worker node. Additionally, optional services, such as databases, can run alongside the nodes. Writing a single Dockerfile to create a container forces the user to address the entire process of installing the exact dependencies and configuring the initial setup as code, which involves numerous steps and intricate details. Fortunately, an official sample `docker-compose.yml` file exists specifically to provide easier initial access to a working LAVA instance. However, it is still necessary to configure VM DUTs to ensure proper usage within LAVA. The required steps can be summarized as follows:

- From the master node shell, an admin user must be configured using the LAVA CLI. This user is necessary for configuring LAVA workers, registering DUTs, and running jobs if some steps are performed via the web interface.
- LAVA requires every registered DUT to be associated with a **device type**. The user must define new device types and include specifications that differentiate them from others, allowing LAVA to interact appropriately. While this topic is complex, LAVA

provides built-in device types for commonly used configurations, including VM devices. The desired device types must also be enabled for the LAVA instance.

- After enabling a device type, a new device must be registered and assigned to the corresponding device type.
- A device dictionary must be linked to the device. This dictionary defines the device's computational capabilities in a file using the jinja2 format for configuration templates. The official LAVA documentation provides an expected device dictionary template for VM devices.

After configuring the VM device, tests were conducted to verify the proper functioning of the CI system by triggering manual jobs executed by the DUT. Triggering manual jobs is straightforward using the web interface. The syntax for defining a job follows the YAML configuration language, which is extensively used in this project (e.g., for the Jenkins Configuration as Code file). Job definitions are well-documented in the official LAVA documentation but can be summarized into three main stages:

- Defining how the DUT will be booted;
- Specifying how the image will be deployed onto the DUT;
- Outlining the test suite to be executed on the DUT.

These steps encompass most of the exploration conducted with LAVA. The main conclusion is that, despite being the most mature project for deploying kernel images, LAVA is complex, and its documentation is still insufficient. Moreover, the proposed solution is generic, addressing CI challenges for most kernel communities but offering limited specific benefits for the DRM (Direct Rendering Manager) community. The integration between LAVA and Jenkins services was not implemented, as the decision was to explore CI-Tron instead. However, it is known that services like KernelCI integrate these tools effectively. This does not mean LAVA is entirely discarded for this work; instead, the CI-Tron approach became the primary focus during the study and development of the kernel CI system.

4.3 Using CI-Tron from a Virtual Environment

The previous section presented the first approach adopted to implement a CI system for the kernel. It began with the initial steps of the pipelines, which included tracking patches, triggering new jobs with a CI service (Jenkins), and starting a LAVA job from there.

This section will describe the second approach taken during the development of this work. While the beginning of the pipeline (tracking patches with an automated tool) remains the same, the chosen CI service and the underlying infrastructure for deployment are new.

4.3.1 The Vivian Virtual Environment

The CI-Tron project is highly hardware-dependent. Its repository contains code for the orchestration of the CI farm (i.e., the gateway), and most of the remaining configuration

is expected to be done physically, following the steps described in Chapter 3 to achieve the CI farm structure previously mentioned.

This does not imply that the usage and testing of CI-Tron are unfeasible without hardware. A secondary project is included in the official repository called `vivian`. It is a VM configured to run a complete virtual environment for CI-Tron, including the management of a virtual PDU and DUTs. The `vivian` virtual environment was crucial for exploring new possibilities of CI and gaining a deeper understanding of CI-Tron without taking risks with physical devices.

Invoking this environment from the source code is simple, as there is a `Makefile` target that contains all the logic for generating and instantiating this environment automatically, with no overhead. By running `make vivian`, the environment is set up and a visual interface, that can be seen on figure 4.2, is displayed to represent a perspective from the gateway device over the virtual farm.

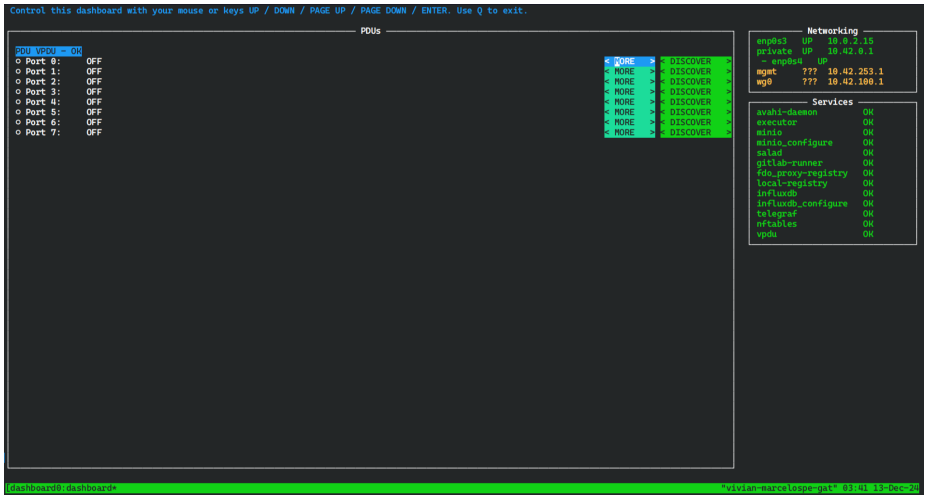

Figure 4.2: *The main interface of vivian.*

The `Makefile` also contains a target to invoke new virtual DUTs without the visual interface. This operation and more specific ones can be executed by interacting with the REST API exposed by the `executorctl` program that runs on the orchestrator (either virtual or physical).

4.3.2 Configuring GitLab CI to Expose CI-Tron Runners

The CI-Tron project enables a practical way of integrating the farm with GitLab CI: it offers the possibility of exposing each DUT from the instance to the service, abstracting all the logic of handling the farm and allowing developers to focus on writing the jobs for the GitLab CI service.

This requires manipulating the MarsDB database, a Database (DB) file that registers the actual configuration state of the CI instance. This file is located on the gateway device (physical or virtual) and can be edited manually during the execution of the CI-Tron service, where every change is automatically applied to the CI farm.

To access this file from a `vivian` setup, it is possible to enable the virtual environment with the `make vivian` target and then access the shell of the virtual gateway, either by quitting the dashboard or by using `make vivian-connect` on another shell of the host device to establish an SSH connection. The file is at the guest's `/config/mars_db.yaml`.

In a virtual CI instance, with a single virtual DUT registered, the program 4.7 shows an example of a MarsDB file that was automatically generated.

```
pdus:
  VPDU:
    mac_address: null
    driver: vpdu
    config:
      hostname: localhost:9191
    reserved_port_ids: []
duts:
  '52:54:00:11:22:00':
    mac_address: '52:54:00:11:22:00'
    base_name: virtio
    ip_address: 10.42.0.2
    tags:
    - cpu:cores:2
    - mem:size:1GiB
    - virtio:family:VIRTIO
    - virtio:pciid:0x1af4:0x1050:0x0
    - virtio:codename:VIRTIO
    - cpu:arch:x86_64
    - firmware:non-efi
    manual_tags: []
    local_tty_device: ttyS1
    gitlab:
      freedesktop:
        token: <invalid default>
        exposed: true
        runner_id: -1
    pdu: VPDU
    pdu_port_id: '0'
    pdu_off_delay: 0.1
    ready_for_service: false
    is_retired: false
    first_seen: 2024-12-14 01:39:52.940089
    comment: null
gitlab:
  freedesktop:
    url: https://gitlab.freedesktop.org
    registration_token: null
    runner_type: instance_type
    group_id: null
    project_id: null
    access_token: null
    expose_runners: false
    maximum_timeout: 21600
    gateway_runner:
      token: <invalid default>
      exposed: false
```

```
48          runner_id: -1
```

Program 4.7: *An example of a MarsDB file*

With the YAML syntax, the basic components of the CI instance are declared and defined:

- The `pdus` field contains the declaration and definition of every PDU detected and managed by the gateway;
- The `duts` field contains the list of DUTs in the farm. Each entry is declared with the MAC address of the respective DUT and includes definitions for its PDU, hardware, and GitLab instance(s) where the device is exposed;
- The `gitlab` field contains the definitions for the GitLab instances where the devices will be exposed. To effectively expose them, the user must manually fill in these fields so the gateway can parse this information to authenticate with the given instances and expose all devices via the GitLab CI API. The essential fields are:
  - `url`: The URL of the GitLab instance. By default, it uses the freedesktop instance, but others can be used (e.g., www.gitlab.com for the official instance);
  - `registration_token`: An access token generated within GitLab containing the `create_runner` scope alongside Maintainer role permissions;
  - `runner_type`: The type of runner to register. It can be instance, group, or project type, each defining the scope of projects the runner has access to;
  - An `id` field corresponding to the `runner_type`;
  - `access_token`: Another GitLab token with the `read_api` scope.

After properly configuring the GitLab instance credentials in the MarsDB file, the runners from the CI farm can be seen on the GitLab runners page and used in GitLab CI pipelines.

4.3.3 Writing a Pipeline in GitLab CI

Once the CI infrastructure (physical or virtual) is in place, it is necessary to write a CI pipeline that describes the steps to be executed by a job. Using free software projects that utilize CI-Tron for CI (dxvk and Mesa) as a reference, a prototypical CI pipeline was written to trigger jobs for compiling and testing kernel images.

With GitLab CI, the CI pipelines are defined in a `gitlab-ci.yml` file. It is also common for projects to contain a `.gitlab-ci` directory, which includes helper scripts and files to be used by the `gitlab-ci.yml`. Given this, the experimentation with CI pipelines involved creating a repository on GitLab (the freedesktop instance) to contain the kernel tree from the AMD community.

The pipeline consisted of a simple compilation stage (similar to what was done for Jenkins) and a deployment stage, where the compiled image is deployed onto the device. The strategy for deploying the image was based on the fact that a CI-Tron farm parses

the kernel image to be loaded from a configuration file containing the URL to the image. This means that compiled images must be publicly available somewhere to be fetched by the DUT.

The plan for the pipeline involved compiling the image and saving the result as an artifact. Then, a CI stage was applied where a generic copy of a configuration file, present in the repository, had its URL field modified to contain the URL of the corresponding CI job artifact before being parsed by the DUT. Fetching the artifact URL at runtime is possible because GitLab CI offers environment variables containing information like the Job ID, which is essential for constructing the URL for the artifacts of a specific job of a specific project.

The configuration files are in the jinja2 format. The jinja2 file template has excellent syntax for rendering custom data. By writing a Python script using the jinja2 library, as shown by program 4.8, it is possible to parse and modify files following the jinja2 syntax easily. Custom variable names can be inserted into the jinja2 file using the {{ }} delimiter, and the jinja2 Python library offers custom variables to replace the placeholders with the custom values (in this case, the URL).

```
1    from jinja2 import Environment, FileSystemLoader
2    from os import environ
3
4    def get_artifact_url():
5       project_id = environ.get("CI_PROJECT_ID")
6       job_id = environ.get("CI_JOB_ID")
7       job_token = environ.get("CI_JOB_TOKEN")
8       artifact_url = f"https://gitlab.freedesktop.org/api/v4/projects/{
            project_id}/jobs/{job_id}/artifacts/arch/x86/boot/bzImage?job_token={
            job_token}"
9       return artifact_url
10
11   env = Environment(loader = FileSystemLoader('.gitlab-ci'))
12   template = env.get_template('IGT.yml.j2')
13   output = template.render(kernel_url = get_artifact_url())
14   with open(".gitlab-ci/output.yml", "w") as out:
15      print(output, file = out)
```

Program 4.8: *A python script to render a custom URL into a jinja2 file*

The code above represents a script that modifies the URL containing the kernel image artifact. With methods from the jinja2 library, it is possible to render the `kernel_url` variable into a URL containing the entire path to fetch an artifact from the GitLab Application Programming Interface (API). At the same time, specific information, like Job ID, Project ID, and Job authentication token, are collected from environment variables from Gitlab CI runtime and formatted into the final URL.

Chapter 5

Conclusion

This work covered a highly contextualized and niche topic, requiring extra effort to study and understand the problem. Fortunately, the time invested contributed to a deeper comprehension of Continuous Integration (CI) and how its concepts are applied in the current state of the kernel development community. This chapter summarizes the various topics addressed during the project while presenting their final remarks and possible future plans.

5.1 Final Remarks on the `lore-fetcher` Development

The first development cycle of this work and its core proposal consisted of implementing a patch-tracking application to automate the integration between the kernel mailing lists and the CI infrastructure. The application was also designed to retrieve results from the CI pipeline and return them to the community.

The final prototype achieved three important milestones:

- The application can automatically collect patches from the Kernel Lore archives.
- The application can "pass" collected patches to a CI service (Jenkins), i.e., trigger new jobs with a custom parameter referencing the new patch collected.
- The application can retrieve a string-based response by e-mail.

5.1.1 Patch Collection

The idea of automated tracking of new patches is not entirely new in the kernel community, but a standard solution still doesn't exist. Maintainers usually develop their own tools tailored to their specific use cases, leaving the field open to experimentation. This motivated the development of a new tracking system based on the Kernel Lore archives. Many strategies for querying the newest entries from Lore were previously documented and implemented by other free software projects, such as `patch-hub`, which provides a

robust system for retrieving the latest messages from kernel mailing lists. Consequently, techniques for acquiring patches for a CI infrastructure could be adapted from already established programs working with Lore.

The final program proved effective in achieving the desired results. Using Kernel Lore as the underlying data source is a consistent strategy for tracking the newest patches, fulfilling all the needs of the proposed CI system. This approach will likely remain the default for future iterations of `lore-fetcher`.

5.1.2 Integration with CI Services

By the end of the project, the `lore-fetcher` application was also capable of triggering new jobs on a Jenkins server. This was made possible through the Jenkins API, which allows seamless interaction with the service.

Although Jenkins was later replaced by GitLab CI, mainly due to the seamless integration between CI-Tron and GitLab CI, the concept of interacting with CI services via APIs remains highly relevant. GitLab CI provides an API that fulfills the exact requirements of Jenkins API.

5.1.3 Returning Results

Finally, a prototype system for returning results via e-mail was developed. Despite its apparent validity, it was decided during the project that more modern approaches could be used to share CI results with the community. A publicly hosted dashboard emerged as the most promising solution, providing a unified location for accessing all results. This does not mean sending response e-mails to patch authors is an invalid result return mechanism. However, this approach should be used as a complementary mechanism alongside the dashboard rather than as the main solution.

5.2 Final Remarks on the CI Infrastructure

After developing a small program for tracking patches, the next step was studying how to organize a CI infrastructure. This involved experimenting with CI services such as Jenkins and GitLab CI and tools like LAVA and CI-Tron.

Although this was not expected to be the project's core focus, the patch collection automation was more straightforward than initially anticipated. Consequently, the need to prioritize the CI infrastructure itself became more evident. Fortunately, virtual environments were used to evaluate both LAVA and CI-Tron, significantly enhancing the studying effectiveness by avoiding the bureaucracy of acquiring physical hardware.

Experimenting with virtual infrastructures played a crucial role in consolidating a stable understanding of the challenges related to CI in the kernel community. Ultimately, CI-Tron was chosen over LAVA for several reasons:

- CI-Tron offers a more straightforward approach to CI while maintaining robustness in managing DUTs (Devices Under Test) and preserving essential testing principles

such as test reproducibility.

- CI-Tron has strong ties with the Linux graphics development communities. For example, its maintainer, Martin Roukala, has a history of contributing to Nouveau (free software drivers for NVIDIA GPUs). Moreover, CI-Tron is widely adopted by MESA (a free software implementation of the OpenGL API). This alignment is particularly beneficial for the context of CI in the AMD Display community.

However, the main drawback of using CI-Tron is its relatively recent development and lack of maturity compared to LAVA. This means the software is more likely to contain bugs, have limited documentation, and lack certain features. Thus, contributing to CI-Tron will be a key focus moving forward. The plan to develop a CI infrastructure for the AMD Display community now includes actively working on new patches for CI-Tron and integrating with its community.

5.3 Future Plans for the CI

This project has been instrumental in understanding the challenges of implementing CI practices in the kernel community. Experimenting with various tools (e.g., LAVA, CI-Tron), proposing new ones (`lore-fetcher`), and leveraging generic-purpose CI services (GitLab CI and Jenkins) to build kernel pipelines were crucial steps toward comprehending the differences between CI for kernel development and CI for userspace programs.

The conclusions drawn from this work are expected to contribute to a long-term project to create a CI infrastructure for the AMD Display community. The vision extends beyond a single CI farm to encompass a federated CI system, where multiple CI farms act as nodes within a larger hierarchy. This system would serve the AMD Display community while also being made available as free software for other communities.

Implementing such a solution will require managing multiple farms and scheduling jobs across them. Based on this study, a promising approach involves exposing individual farms through generic CI services (e.g., GitLab CI) and using a master node to orchestrate them via APIs. The `lore-fetcher` application would either expand its scope to manage federated CI systems or integrate its code into a broader solution. The immediate next step is to implement a single CI farm at IME-USP. In the coming months, CI-Tron will be further explored to create a complete CI pipeline – from collecting patches with `lore-fetcher` to testing them with the igt-gpu-tools test suite and presenting the results on a dashboard.

5.3.1 Why Not KernelCI?

KernelCI is an established solution for kernel-related CI, offering CI farms without the burden of managing them. It even includes DRM-CI, an interface tailored for the DRM community.

However, KernelCI presents two main limitations for this project:

1. It does not cater to those who want to self-host their CI infrastructure. Many development communities prefer to manage their own infrastructure, relying exclusively on their allocated resources and maintaining complete control over them. While

KernelCI provides a shared solution for the broader kernel community, this project empowers individual communities to manage their own CI infrastructure.

2. Integrating CI-Tron farms with KernelCI is not as seamless as with LAVA.

Self-hosting is particularly relevant for the AMD community, which can afford the resources for its CI infrastructure while benefiting from academic research and greater control over its system.

Chapter 6

Personal appreciation

It was during my sophomore year at university that I started using Linux for the first time. I had heard about it, but have never considered actually using a Linux Distro as an OS. I didn't even know what a "Linux Distro" was, and names like Ubuntu, Debian, and Arch were unknown to me. After some encouragement from professors to avoid using Windows as an OS for development, I decided to try dual-booting Ubuntu and Windows, and, since then, I've become a Linux user and fan, and have distro-hopped more times than I'm able to count.

During my third year of my bachelor's degree in computer science, I decided to join the rebirth of the FLUSP extension group. Despite having more experience with the open-source background at that point, contributing to free and open-source projects was still mysterious to me. When starting the course, I already had vague notions about what was necessary to develop games, web and mobile applications, scientific programs, etc. But the concept of FLOSS software was something completely new to me that I had just observed from far away for the last two and a half years.

Driven by curiosity, I had a great time and have learned a lot. I've also got to know Professor Paulo Meirelles who presented to me many opportunities. This led me to become an administrator of Rede GNU/Linux, a computer network where I've grasped a lot of practical knowledge on how to manage Linux systems.

I've also had the great opportunity to dive deep into free and open-source communities. I've joined the kworkflow community while also making my first contributions to the Linux Kernel. It wouldn't take long before joining the Google Summer of Code program for kworkflow. These opportunities got me to meet people who would be important to me on my journey to contribute to the Kernel, such as Rodrigo Siqueira and David Tadokoro who guided me when first contributing to kworkflow until now.

This project of developing a Continuous Integration infrastructure for the AMD Display community helped me to learn a lot about DevOps practices, important development tools, and the kernel itself. It was also the biggest project I've worked on so far, which required from me a lot of independence and self-organization. Hopefully, I could still rely on the helpful suggestions of Paulo, Rodrigo, David. During the last semester, I also worked with

Rafael Passos, who helped me figure out which steps we should take next.

Finally, the plan of working with CI led me to get exposed to the CI-Tron project, after a recommendation of Rodrigo. I would soon get integrated with its free software community and also get closer contact with its maintainer Martin Roukala. The CI-Tron community is small, but also dedicated and extremely friendly, and it became an honor to join the community.

All these experiences enabled me to meet closely a wide array of free software communities and it has been very demanding and challenging to keep track of them all. Hopefully, all the efforts have been worth it. I've been grasping a lot of knowledge and have found a new perspective on computers and software development that I didn't expect to know about when first started at university.

Appendix A

Code snippets from `lore-fetcher`

```
1   package configurator
2
3   import (
4     "fmt"
5     "strconv"
6
7     "lore-fetcher/internal/fetcher"
8
9     "github.com/spf13/viper"
10  )
11
12  type Configurator struct {
13    configuration map[string]string
14  }
15
16  func NewConfigurator() *Configurator {
17    var configurator Configurator
18    configurator.configuration = make(map[string]string)
19    return &configurator
20  }
21
22  func (configurator *Configurator) ParseConfiguration(){
23    viper.SetConfigType("toml")
24    viper.SetConfigName("lore-fetcher")
25    viper.AddConfigPath("/etc/lore-fetcher/")
26    if err := viper.ReadInConfig(); err != nil {
27      panic(fmt.Errorf("error reading config file: %w", err))
28    }
29
30    configurator.configuration["mailing-list"] = viper.GetString("fetcher.
          mailing-list")
31    configurator.configuration["fetch-interval"] = strconv.Itoa(viper.GetInt("
          fetcher.fetch-interval"))
32    configurator.configuration["jenkins-server"] = viper.GetString("fetcher.
          jenkins-server")
33    configurator.configuration["jenkins-token"] = viper.GetString("fetcher.
          jenkins-token")
```

```
34      configurator.configuration["jenkins-pipeline"] = viper.GetString("fetcher.
            jenkins-pipeline")
35
36      configurator.configuration["from-mail"] = viper.GetString("mailer.from-mail
            ")
37      configurator.configuration["to-mail"] = viper.GetString("mailer.to-mail")
38      configurator.configuration["password"] = viper.GetString("mailer.password")
39    }
40
41    func (configurator *Configurator) IsConfigurated(key string) bool{
42      return configurator.configuration[key] != ""
43    }
44
45    func (configurator *Configurator) GetConfiguration(key string) string{
46      return configurator.configuration[key]
47    }
48
49    func (configurator *Configurator) SetConfiguration(key string, value string)
          {
50      configurator.configuration[key] = value
51    }
52
53    func (configurator *Configurator) ConfigureFetch(fetcher *fetcher.Fetcher){
54      fetcher.MailingList = configurator.configuration["mailing-list"]
55      fetcher.FetchInterval, _ = strconv.Atoi(configurator.configuration["fetch-
            interval"])
56      fetcher.JenkinsServer = configurator.configuration["jenkins-server"]
57      fetcher.JenkinsPipeline = configurator.configuration["jenkins-pipeline"]
58      fetcher.JenkinsToken = configurator.configuration["jenkins-token"]
59    }
```

Program A.1: *Code from the configuration parser.*

```
1     package configurator
2
3     import (
4       "testing"
5
6       "lore-fetcher/internal/evaluator"
7       "lore-fetcher/internal/fetcher"
8       "lore-fetcher/internal/mailer"
9     )
10
11    func TestConfigurator(t *testing.T){
12      fetcher := fetcher.NewFetcher()
13      evaluator := evaluator.Evaluator{}
14      mailer := mailer.Mailer{}
15      configurator := Configurator{}
16      configurator.ParseConfiguration(fetcher, &evaluator, &mailer, "../../
            testdata/")
17      if(mailer.ToMail != "mock_receiver"){
18        t.Errorf("got %s, expected %s", mailer.ToMail, "mock_receiver")
19      }
20    }
```

Program A.2: *A basic unit test for the configuration abstraction.*

```go
package mailer

import (
  "fmt"
  "net/smtp"
  "time"
  "lore-fetcher/internal/types"
)

type Mailer struct {
  FromMail string
  ToMail string
  Password string
}

func (mailer Mailer) SendResults(resultBuffer chan types.Patch){
  for {
    patch := <- resultBuffer
    auth := smtp.PlainAuth("", mailer.FromMail, mailer.Password , "smtp.gmail.com")
    to := []string{mailer.ToMail}
    msg := []byte("Subject: " + "Tests for patch '" + patch.Title + "' completed.\r\n" +
      "\r\n" +
      patch.ResultString +
      "\r\n")
    err := smtp.SendMail("smtp.gmail.com:587", auth, mailer.FromMail, to, msg)

    if err != nil {
      fmt.Println(err)
    }
    fmt.Println("[", time.Now(), "]: Sent result to", mailer.FromMail)
  }
}
```

Program A.3: *Code from a prototypal mailing feature using SMTP to send automated texts. The next step would be sending mails with data from the respective patch entry.*

Appendix B

The vivian interface

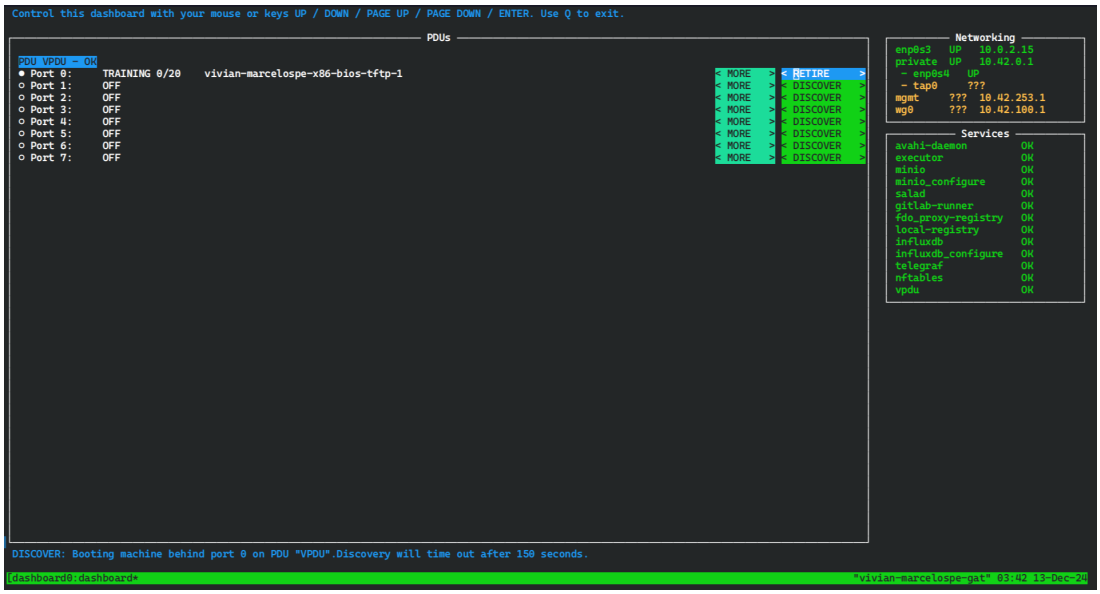

Figure B.1: *Discovering a virtual DUT from the vivian interface*

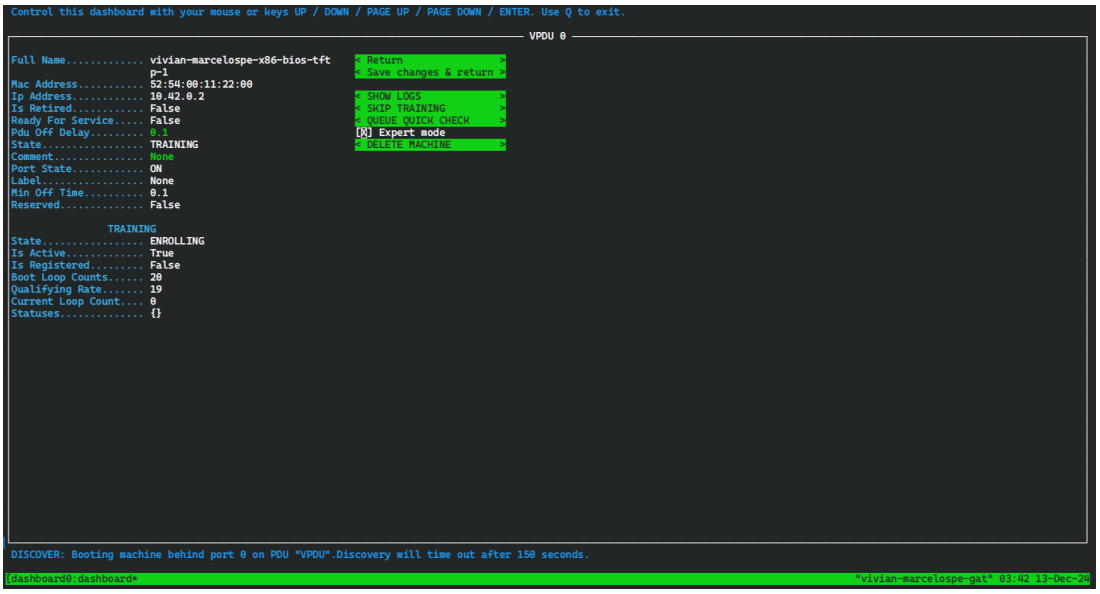

Figure B.2: *Accessing DUT configuration from the vivian interface*

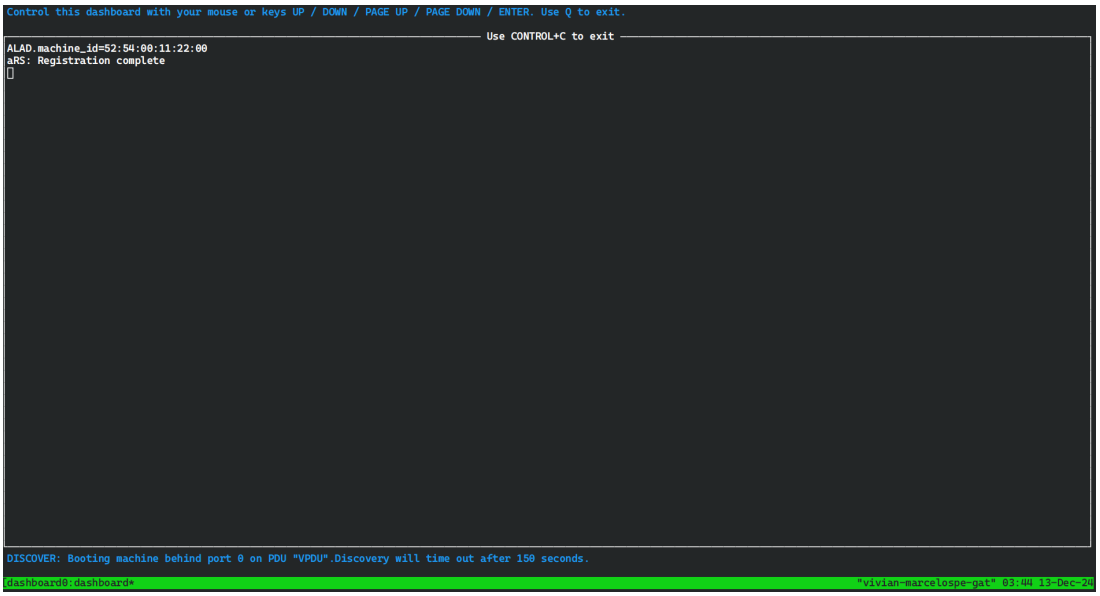

Figure B.3: *Accessing the logs of a DUT from the vivian interface*

Appendix C

Scripts from the Gitlab CI pipeline

```
stages:
  - mirror
  - build

mirror:
  image: bitnami/git:latest
  stage: mirror
  rules:
    - if: $CI_PIPELINE_SOURCE == "schedule"
      when: on_success
    - when: never
  before_script:
    - git config --global user.name "${GITLAB_USER_NAME}"
    - git config --global user.email "${GITLAB_USER_EMAIL}"
  script:
    - git remote add upstream https://gitlab.freedesktop.org/agd5f/linux.git
        || /bin/true
    - git fetch upstream
    - git reset --hard upstream/amd-staging-drm-next
    - git push "https://${GITLAB_USER_LOGIN}:${CI_MIRROR_TOKEN}@${
        CI_REPOSITORY_URL#*@}" "HEAD:master" --force

build:
  stage: build
  image: ubuntu
  artifacts:
    paths:
      - arch/x86/boot/bzImage
  variables:
    PATCH: 'https://lore.kernel.org/amd-gfx/20241018133308.889-1-Yunxiang.
        Li@amd.com/T/#t'
  script:
    - apt update
    - apt install -y b4 git bc binutils bison dwarves flex gcc git gnupg2
        gzip libelf-dev libncurses5-dev libssl-dev make openssl pahole perl-
        base rsync tar xz-utils
    - git config --global user.name "${GITLAB_USER_NAME}"
    - git config --global user.email "${GITLAB_USER_EMAIL}"
```

```
34        - b4 shazam "${PATCH}"
35        - make tinyconfig
36        - make
```

Program C.1: *A first iteration of the pipeline `.gitlab-ci.yml`. It contains a stage for syncing the Gitlab repository with the original kernel tree (mirror) and a stage for building the kernel and saving the image as an artifact (build).*

```
1     - mirror
2     - build
3     - test
4
5     rror:
6     image: bitnami/git:latest
7       - git reset --hard upstream/amd-staging-drm-next
8       - git push "https://${GITLAB_USER_LOGIN}:${CI_MIRROR_TOKEN}@${
            CI_REPOSITORY_URL#*@}" "HEAD:master" --force
9
10    st:
11    artifacts:
12      paths:
13        - .gitlab-ci/output.yml
14    stage: test
15    image: python
16    script:
17      - pip install jinja2
18      - python3 .gitlab-ci/generate_job.py
```

Program C.2: *A second iteration of the Gitlab CI pipeline focusing on parsing the artifact generated from the build stage.*

References

[Brooch 1991] Grady Brooch. *Object Oriented Design with Applications*. Benjamin Cummings, 1991 (cit. on p. 11).

[Fowler 2006] Martin Fowler. *Continuous Integration*. 2006. URL: https://martinfowler.com/articles/continuousIntegration.html (cit. on p. 11).

[GNU n.d.] GNU. *What is Free Software?* URL: https://www.gnu.org/philosophy/free-sw.en.html (cit. on p. 4).

[Roukala 2021] Martin Roukala. *Setting Up a CI System Part 1: Preparing Your Test Machines*. 2021. URL: https://mupuf.org/blog/2021/02/08/setting-up-a-ci-system-preparing-your-test-machine/ (cit. on p. 18).

[Roukala 2022a] Martin Roukala. *Setting Up a CI System Part 2: Generating and Deploying Your Test Environment*. 2022. URL: https://mupuf.org/blog/2021/02/10/setting-up-a-ci-system-part-2-generating-and-deploying-your-test-environment/ (cit. on p. 20).

[Roukala 2022b] Martin Roukala. *Setting Up a CI System Part 3: Provisioning Your CI Gateway*. 2022. URL: https://mupuf.org/blog/2022/01/10/setting-up-a-ci-system-part-3-provisioning-your-ci-gateway/ (cit. on p. 21).

[Roukala 2022c] Martin Roukala. *Setting Up a CI System Part 4: Live Patching Your CI Gateway*. 2022. URL: https://mupuf.org/blog/2022/04/15/setting-up-a-ci-system-part-4-live-patching-your-ci-gateway/ (cit. on p. 21).