KernelCI Modular Pipeline

Rationale **2**

Configuration **4**
- System configuration 4
- User settings 4

Step 1: Building kernels **5**
- Overview 5
- Reference implementation 5
- Alternative implementations 5

Step 2: Running tests **6**
- Overview 6
- Reference implementation 6
- Alternative implementations 6

Step 3: Processing incoming results **7**
- Overview 7
- Reference implementation 7
- Alternative implementations 7

Step 4: Visualising test results **8**
- Overview 8
- Reference implementation 8
- Alternative implementations 8

Authors:

- Guillaume Tucker (Collabora)

Date: 29th November 2019

Version: 2.0

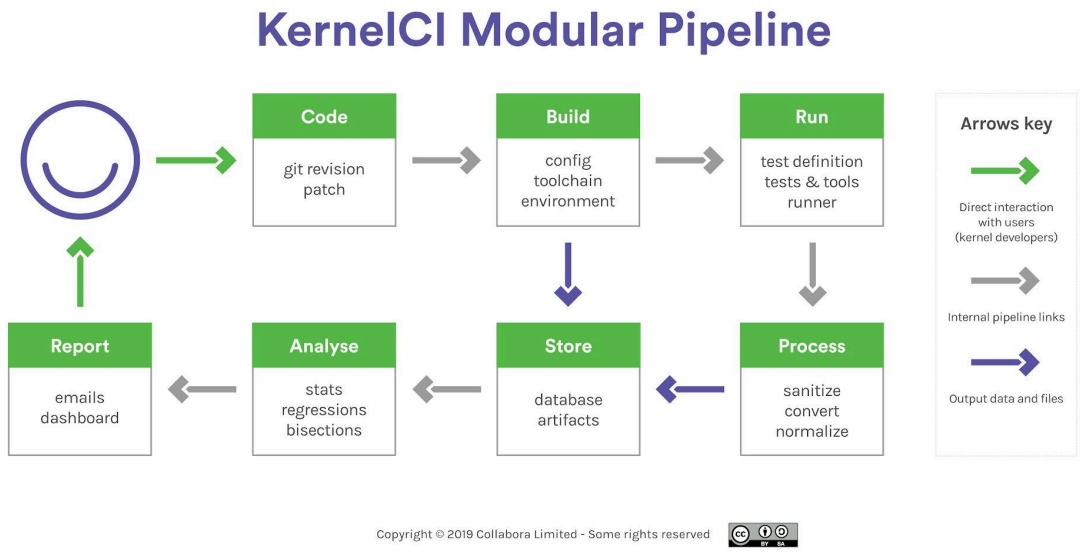
Rationale

KernelCI is a project to continuously test the upstream Linux kernel. Results are posted to mailing lists and on kernelci.org.

This document defines the main design aspects of a modular KernelCI pipeline design. The aim is to have a reference architecture used by default and based on the current production system, but also allowing each component to be replaced with any arbitrary alternative.

For example, the central pipeline automation can be based on Jenkins as per the current kernelci.org production instance but alternative options such as the Azure Pipeline may be used as well. Similar scenarios apply to running jobs in LAVA labs by default while also enabling other non-LAVA labs to contribute test results, or storing the results in Mongo DB by default while enabling alternative database engines such as BigQuery.

This essentially breaks down the system into the following KernelCI pipeline steps, each one being associated with some types of components:



1. Building kernels
    - Automated build system
    - Storage server for the build artifacts
    - Results database for the build meta-data
2. Running tests on a range of platforms
    - Test lab
3. Processing the incoming test results
    - Results parser & analyser
    - Storage server for the logs and any test results data assets
    - Results database for the test results data
4. Visualising the stored test results
    - Dashboard

The key aspects to take into account in order to make this possible are to:

- Provide primitive functions in a generic programming language such as Python
- Make it possible to run each KernelCI pipeline step manually in a shell
- Provide documented APIs and callbacks for each step of the pipeline
- Test each step individually in a shell (should be automated)
- Test the whole reference implementation end-to-end on a staging instance
- Make it possible to run any autonomous instance with arbitrary components

The main motivations for this level of flexibility are to:

- Enable a wider community by providing entry points for existing build and test services
- Enable independent instances to use the KernelCI code in other environments than kernelci.org
- Make it easy for developers to recreate the KernelCI pipeline locally or run some parts of it to reproduce some results and debug issues

Configuration

There are two main categories of things to configure in KernelCI: the system itself (trees to build, tests to run, components to use...) and user settings (API tokens, developer options…). Both are outlined in the following sections respectively:

System configuration

For kernelci.org mainly but also for other instances using the same KernelCI tools, the system configuration is made up of several YAML files to cover the various elements of the pipeline (git branches, tests, labs, rootfs definitions, email report recipients…). Each configuration entry can have filters and references to other entries to express the pipeline behaviour.

For example, some test suites may only be run when some specific kernel options are enabled at build time. Some labs may only allow certain branches to be tested. Email reports may be sent with different format variants to different recipients. Results from different branches may be saved into separate databases.

The schema of these YAML files should be documented and versioned in order to make it possible to keep them in a standalone repository and have control over their compatibility with the code using them.

User settings

While the system configuration is typically public and describes what end users should expect from it, some other settings need to be kept private such as secret API tokens and local preferences. These are mostly some information needed by the KernelCI primitive functions to run but which doesn't change between each iteration. While it's not strictly needed for an automated system which can directly produce all the required command line arguments, it's a lot more convenient when using the tools manually on the command line. For a developer, providing an API token on the command line is not very safe and also tedious. User settings could also include specific paths to store results or debug features to enable.

The settings file format uses `[section]` headers followed by "`name: value`" entries as supported by the Python `configparser` module. The KernelCI primitive functions and command line tools should be used to parse the settings file and make use of them in the various steps of the pipeline.

The supported section names and their values should be documented and versioned, to go hand-in-hand with the YAML system configuration.

Step 1: Building kernels

Overview

The starting point of the KernelCI pipeline is to detect when some new code has become available and build it for a number of variants (CPU architecture, kernel config, compiler…). The list of monitored Git repositories (“trees”) and branches should be defined in the YAML build configurations file, with attributes to specify the variants to build. Something similar can be done for monitoring new patches sent to mailing lists, although this isn’t something KernelCI currently support. The KernelCI primitive functions and command line tools should provide a way to achieve this manually in a shell and make it easy to automate anywhere.

This step produces build artifacts (kernel binaries, build logs) and meta-data which all need to be pushed to a central server. Meta-data should be sent to the results database as described in Step 3, and should include a reference to the location of the published build artifacts. At least one storage and one results database components should be defined in the pipeline configuration with any API URLs, access tokens and the types of protocols to use to communicate with them.

Reference implementation

The reference implementation should be done with Jenkins Pipeline jobs using the primitive functions and command line tools. It should use the current kernelci-backend storage API.

Alternative implementations

Implementing an automated solution other than a Jenkins Pipeline should in principle only involve the aspects specific to that alternative automated build system, such as job scheduling, resource management and credentials. Other protocols than the current kernelci-backend API may be supported by the KernelCI primitive functions to enable sending the build artifacts to another storage server and meta-data to another database. Multiple storage server and results database components may be listed in the configuration file, in which case a copy of the build artifacts and meta-data will be pushed to each of them.

Step 2: Running tests

Overview

Tests are to verify that the various kernel binaries that have been built behave as expected when run on test platforms. For example: booting to a login prompt, checking for kernel error messages, verifying that all the hardware is initialised and running functional tests to cover particular areas of the kernel such as `kselftest`, `igt` or `v4l2-compliance`. Tests are typically run on development boards but higher level kernel tests such as `xfstools` may also be run in a virtual environment.

The list of tests to run should be described in the YAML test configurations file as combinations of test plans and platforms. The KernelCI primitive functions and command line tools should provide a way to achieve this manually in a shell.

At least one test lab component should be defined in the settings file. This is where the test definitions will be submitted to in order to be run. This step will also be using the storage and results database used in Step 1 to retrieve kernel binaries and associated meta-data.

Reference implementation

The reference implementation should be done with Jenkins Pipeline jobs using the primitive functions and command line tools. It should use LAVA to schedule tests, and the kernelci-backend results API with its associated storage server as used in Step 1.

Alternative implementations

Support for other job schedulers may be added to the KernelCI primitive functions. Multiple labs may be listed in the settings file, with the API URL, any access tokens and the type of protocol. Each lab should be given a name which can be referred to in the test configurations YAML file to filter tests on a per-lab basis. Support for each type of lab may depend on particular types of storage and database components used in Step 1 to retrieve kernel builds.

Step 3: Processing incoming results

Overview

Test results will all be sent to a central service where they will be processed and stored in a database. Other actions may be performed at this stage such as sending email reports, detecting regressions and triggering bisections. This applies to both runtime tests and build results with warnings and errors produced by the compilers.

At least one results database component should be defined in the settings file with an API URL, any required access tokens and the type of protocol to use to communicate with it.

Reference implementation

The reference implementation should be using the current kernelci-backend. The primitive functions and command line tools should be used to interact with it to implement this step and also perform regular operations (dealing with tokens, retrieving statistics…). The reference results database API should be documented and tested in order to enable alternative clients to access its data and use it in production.

Alternative implementations

The primitive functions may implement other protocols to communicate with alternative results databases. Other database APIs may also be compatible with the reference one to receive the same requests and data format but use it differently. If more than one database component is listed in the YAML database configurations file then all the results will be sent to each of them.

Step 4: Visualising test results

Overview

Once the results have been stored in the database, they can be queried by arbitrary tools to visualise and search them. There is no real specification as how this may be done, and several solutions may be used to address different use-cases.

Reference implementation

The reference implementation should use the existing kernelci-frontend in conjunction with the existing kernelci-backend to provide the actual data. The kernelci-backend API should be documented in order to enable alternative ways to visualise the data.

Alternative implementations

Alternative implementations may use the reference kernelci-backend using a read-only API token, or an alternative results database and storage server as defined in previous steps of the KernelCI pipeline. There may be several backends and visualisation solutions coexisting in order to address different use-cases and ramp up new solutions while keeping legacy ones online.