wa Documentation

Release 3.1.3

Arm Limited

Jul 19, 2019

Contents

Workload Automation (WA) is a framework for executing workloads and collecting measurements on Android and Linux devices. WA includes automation for nearly 40 workloads and supports some common instrumentation (ftrace, hwmon) along with a number of output formats.

WA is designed primarily as a developer tool/framework to facilitate data driven development by providing a method of collecting measurements from a device in a repeatable way.

WA is highly extensible. Most of the concrete functionality is implemented via *plug-ins*, and it is easy to *write new plug-ins* to support new device types, workloads, instruments or output processing.

Note: To see the documentation of individual plugins please see the *Plugin Reference*.

Contents

CHAPTER 1

What's New

1.1 What's New in Workload Automation

1.1.1 Version 3.1.2

Fixes/Improvements

Framework:

- Implement an explicit check for Devlib versions to ensure that versions are kept in sync with each other.
- Added a `View` parameter to ApkWorkloads for use with certain instruments for example `fps`.
- Added `"supported_versions"` attribute to workloads to allow specifying a list of supported version for a particular workload.
- Change default behaviour to run any available version of a workload if a specific version is not specified.

Output Processors:

- `Postgres`: Fix handling of `screen_resoultion` during processing.

Other

- Added additional information to documentation
- Added fix for Devlib's `KernelConfig` refactor
- Added a `"label"` property to `Metrics`

1.1.2 Version 3.1.1

Fixes/Improvements

Other

- Improve formatting when displaying metrics
- Update revent binaries to include latest fixes
- Update DockerImage to use new released version of WA and Devlib
- Fix broken package on PyPi

1.1.3 Version 3.1.0

New Features:

Commands

- `create database`: Added *create subcommand* command in order to initialize a PostgresSQL database to allow for storing WA output with the Postgres Output Processor.

Output Processors:

- `Postgres`: Added output processor which can be used to populate a Postgres database with the output generated from a WA run.
- `logcat-regex`: Add new output processor to extract arbitrary "key" "value" pairs from logcat.

Configuration:

- *Configuration Includes*: Add support for including other YAML files inside agendas and config files using `"include#:"` entries.
- *Section groups*: This allows for a `group` entry to be specified for each section and will automatically cross product the relevant sections with sections from other groups adding the relevant classifiers.

Framework:

- Added support for using the *OutputAPI* with a Postgres Database backend. Used to retrieve and *process* run data uploaded by the `Postgres` output processor.

Workloads:

- `gfxbench-corporate`: Execute a set of on and offscreen graphical benchmarks from GFXBench including Car Chase and Manhattan.
- `glbench`: Measures the graphics performance of Android devices by testing the underlying OpenGL (ES) implementation.

Fixes/Improvements

Framework:

- Remove quotes from `sudo_cmd` parameter default value due to changes in devlib.
- Various Python 3 related fixes.
- Ensure plugin names are converted to identifiers internally to act more consistently when dealing with names containing –'s etc.
- Now correctly updates RunInfo with project and run name information.
- Add versioning support for POD structures with the ability to automatically update data structures / formats to new versions.

Commands:

- Fix revent target initialization.
- Fix revent argument validation.

Workloads:

- `Speedometer`: Close open tabs upon workload completion.
- `jankbench`: Ensure that the logcat monitor thread is terminated correctly to prevent left over adb processes.
- UiAutomator workloads are now able to dismiss android warning that a workload has not been designed for the latest version of android.

Other:

- Report additional metadata about target, including: system_id, page_size_kb.
- Uses cache directory to reduce target calls, e.g. will now use cached version of TargetInfo if local copy is found.
- Update recommended *installation* commands when installing from github due to pip not following dependency links correctly.
- Fix incorrect parameter names in runtime parameter documentation.

1.1.4 Version 3.0.0

WA3 is a more or less from-scratch re-write of WA2. We have attempted to maintain configuration-level compatibility wherever possible (so WA2 agendas *should* mostly work with WA3), however some breaks are likely and minor tweaks may be needed.

It terms of the API, WA3 is completely different, and WA2 extensions **will not work** with WA3 – they would need to be ported into WA3 plugins.

For more information on migrating from WA2 to WA3 please see the *Migration Guide*.

Not all of WA2 extensions have been ported for the initial 3.0.0 release. We have ported the ones we believe to be most widely used and useful. The porting work will continue, and more of WA2's extensions will be in the future releases. However, we do not intend to port absolutely everything, as some things we believe to be no longer useful.

Note: If there a particular WA2 extension you would like to see in WA3 that is not yet there, please let us know via the GitHub issues. (And, of course, we always welcome pull requests, if you have the time to do the port yourselves :-)).

New Features

- Python 3 support. WA now runs on both Python 2 and Python 3.

  > **Warning:** Python 2 support should now be considered deprecated. Python 2 will still be fully supported up to the next major release (v3.1). After that, Python 2 will be supported for existing functionality, however there will be no guarantee that newly added functionality would be compatible with Python 2. Support for Python 2 will be dropped completely after release v3.2.

- There is a new Output API which can be used to aid in post processing a run's output. For more information please see *Output*.
- All "augmentations" can now be enabled on a per workload basis (in WA2 this was available for instruments, but not result processors).
- More portable runtime parameter specification. Runtime parameters now support generic aliases, so instead of specifying `a73_frequency: 1805000` in your agenda, and then having to modify this for another target, it is now possible to specify `big_frequency: max`.
- `-c` option can now be used multiple times to specify several config files for a single run, allowing for a more fine-grained configuration management.
- It is now possible to disable all previously configured augmentations from an agenda using `~~`.
- Offline output processing with `wa process` command. It is now possible to run processors on previously collected WA results, without the need for a target connection.
- A lot more metadata is collected as part of the run, including much more detailed information about the target, and MD5 hashes of all resources used during the run.
- Better `show` command. `wa show` command now utilizes `pandoc` and `man` to produce easier-to-browse documentation format, and has been enhanced to include documentation on general settings, runtime parameters, and plugin aliases.
- Better logging. The default `stdout` output is now more informative. The verbose output is much more detailed. Nested indentation is used for different phases of execution to make log output easier to parse visually.
- Full `ChromeOS` target support. Including support for the Android container apps.
- Implemented on top of devlib. WA3 plugins can make use of devlib's enhanced target API (much richer and more robust than WA2's Device API).
- All-new documentation. The docs have been revamped to be more useful and complete.

Changes

- Configuration files `config.py` are now specified in YAML format in `config.yaml`. WA3 has support for automatic conversion of the default config file and will be performed upon first invocation of WA3.
- The "config" and "global" sections in an agenda are now interchangeable so can all be specified in a "config" section.
- "Results Processors" are now known as "Output Processors" and can now be ran offline.
- "Instrumentation" is now known as "Instruments" for more consistent naming.
- Both "Output Processor" and "Instrument" configuration have been merged into "Augmentations" (support for the old naming schemes have been retained for backwards compatibility)

1.2 Migration Guide

Contents

1.2.1 Users

Configuration

Default configuration file change

Instead of the standard `config.py` file located at `$WA_USER_HOME/config.py` WA now uses a `confg.yaml` file (at the same location) which is written in the YAML format instead of python. Additionally upon first invocation WA3 will automatically try and detect whether a WA2 config file is present and convert it to use the new WA3 format. During this process any known parameter name changes should be detected and updated accordingly.

Plugin Changes

Please note that not all plugins that were available for WA2 are currently available for WA3 so you may need to remove plugins that are no longer present from your config files. One plugin of note is the `standard` results processor, this has been removed and it's functionality built into the core framework.

Agendas

WA3 is designed to keep configuration as backwards compatible as possible so most agendas should work out of the box, however the main changes in the style of WA3 agendas are:

Global Section

The `global` and `config` sections have been merged so now all configuration that was specified under the "global" keyword can now also be specified under "config". Although "global" is still a valid keyword you will need to ensure that there are not duplicated entries in each section.

Instrumentation and Results Processors merged

The `instrumentation` and `results_processors` sections from WA2 have now been merged into a single `augmentations` section to simplify the configuration process. Although for backwards compatibility, support for the old sections has be retained.

Per workload enabling of augmentations

All augmentations can now been enabled and disabled on a per workload basis.

Setting Runtime Parameters

Runtime Parameters are now the preferred way of configuring, cpufreq, hotplug and cpuidle rather setting the corresponding sysfile values as this will perform additional validation and ensure the nodes are set in the correct order to avoid any conflicts.

Parameter Changes

Any parameter names changes listed below will also have their old names specified as aliases and should continue to work as normal, however going forward the new parameter names should be preferred:

- The workload parameter `clean_up` has be renamed to `cleanup_assets` to better reflect its purpose.
- The workload parameter `check_apk` has been renamed to `prefer_host_package` to be more explicit in it's functionality to indicated whether a package on the target or the host should have priority when searching for a suitable package.
- The execution order `by_spec` is now called `by_workload` for clarity of purpose. For more information please see *Configuration*.
- The `by_spec` reboot policy has been removed as this is no longer relevant and the `each_iteration` reboot policy has been renamed to `each_job`, please see *Configuration* for more information.

Individual workload parameters have been attempted to be standardized for the more common operations e.g.:

- `iterations` is now `loops` to indicate the how many 'tight loops' of the workload should be performed, e.g. without the setup/teardown method calls.
- `num_threads` is now consistently `threads` across workloads.
- `run_timeout` is now consistently `timeout` across workloads.
- `taskset_mask` and `cpus` have been changed to consistently be referred to as `cpus` and its types is now a `cpu_mask` type allowing configuration to be supplied either directly as a mask, as a list of a list of cpu indexes or as a sysfs-style string.

Output

Output Directory

The *output directory*'s structure has changed layout and now includes additional subdirectories. There is now a `__meta` directory that contains copies of the agenda and config files supplied to WA for that particular run so that all the relevant config is self contained. Additionally if one or more jobs fail during a run then corresponding output directory will be moved into a `__failed` subdirectory to allow for quicker analysis.

Output API

There is now an Output API which can be used to more easily post process the output from a run. For more information please see the *Output API* documentation.

1.2.2 Developers

Framework

Imports

To distinguish between the different versions of WA, WA3's package name has been renamed to `wa`. This means that all the old `wlauto` imports will need to be updated. For more information please see the corresponding section in the *developer reference section*

Asset Deployment

WA3 now contains a generic assets deployment and clean up mechanism so if a workload was previously doing this in an ad-hoc manner this should be updated to utilize the new functionality. To make use of this functionality a list of assets should be set as the workload `deployable_assets` attribute, these will be automatically retrieved via WA's resource getters and deployed either to the targets working directory or a custom directory specified as the workloads `assets_directory` attribute. If a custom implementation is required the `deploy_assets` method should be overridden inside the workload. To allow for the removal of the additional assets any additional file paths should be added to the `self.deployed_assets` list which is used to keep track of any assets that have been deployed for the workload. This is what is used by the generic `remove_assets` method to clean up any files deployed to the target. Optionally if the file structure of the deployed assets requires additional logic then the `remove_assets` method can be overridden for a particular workload as well.

Workloads

Python Workload Structure

- The `update_results` method has been split out into 2 stages. There is now `extract_results` and `update_output` which should be used for extracting any results from the target back to the host system and to update the output with any metrics or artefacts for the specific workload iteration respectively.
- WA now features *execution decorators* which can be used to allow for more efficient binary deployment and that they are only installed to the device once per run. For more information of implementing this please see *deploying executables to a target*.

APK Functionality

All apk functionality has re-factored into an APKHandler object which is available as the apk attribute of the workload. This means that for example `self.launchapplication()` would now become `self.apk.start_activity()`

UiAutomator Java Structure

Instead of a single `runUiAutomation` method to perform all of the UiAutomation, the structure has been refactored into 5 methods that can optionally be overridden. The available methods are `initialize`, `setup`, `runWorkload`, `extactResults` and `teardown` to better mimic the different stages in the python workload.

- `initialize` should be used to retrieve and set any relevant parameters required during the workload.
- `setup` should be used to perform any setup required for the workload, for example dismissing popups or configuring and required settings.
- `runWorkload` should be used to perform the actual measurable work of the workload.
- `extractResults` should be used to extract any relevant results from the target after the workload has been completed.
- `teardown` should be used to perform any final clean up of the workload on the target.

Note: The `initialize` method should have the `@Before` tag attached to the method which will cause it to be ran before each of the stages of the workload. The remaining method should all have the `@Test` tag attached to the method to indicate that this is a test stage that should be called at the appropriate time.

GUI Functionality

For UI based applications all UI functionality has been re-factored to into a `gui` attribute which currently will be either a `UiAutomatorGUI` object or a `ReventGUI` depending on the workload type. This means that for example if you wish to pass parameters to a UiAuotmator workload you will now need to use `self.gui.uiauto_params['Parameter Name'] = value`

Attributes

- The old `package` attribute has been replaced by `package_names` which expects a list of strings which allows for multiple package names to be specified if required. It is also no longer required to explicitly state the launch-able activity, this will be automatically discovered from the apk so this workload attribute can be removed.
- The `device` attribute of the workload is now a devlib `target`. Some of the command names remain the same, however there will be differences. The API can be found at http://devlib.readthedocs.io/en/latest/target.html however some of the more common changes can be found below:

Original Method	New Method
`self.device.pull_file(file)`	`self.target.pull(file)`
`self.device.push_file(file)`	`self.target.push(file)`
`self.device.install_executable(file)`	`self.target.install(file)`
`self.device.execute(cmd, background=True)`	`self.target.background(cmd)`

CHAPTER 2

User Information

This section lists general usage documentation. If you're new to WA3, it is recommended you start with the *User Guide* page. This section also contains installation and configuration guides.

2.1 User Information

Contents

2.1.1 Installation

Contents

This page describes the 3 methods of installing Workload Automation 3. The first option is to use *pip* which will install the latest release of WA, the latest development version from *github* or via a *Dockerfile*.

Prerequisites

Operating System

WA runs on a native Linux install. It was tested with Ubuntu 14.04, but any recent Linux distribution should work. It should run on either 32-bit or 64-bit OS, provided the correct version of Android (see below) was installed. Officially, **other environments are not supported**. WA has been known to run on Linux Virtual machines and in Cygwin environments, though additional configuration may be required in both cases (known issues include makings sure USB/serial connections are passed to the VM, and wrong python/pip binaries being picked up in Cygwin). WA *should* work on other Unix-based systems such as BSD or Mac OS X, but it has not been tested in those environments. WA *does not* run on Windows (though it should be possible to get limited functionality with minimal porting effort).

Note: If you plan to run Workload Automation on Linux devices only, SSH is required, and Android SDK is optional if you wish to run WA on Android devices at a later time. Then follow the steps to install the necessary python packages to set up WA.

However, you would be starting off with a limited number of workloads that will run on Linux devices.

Android SDK

You need to have the Android SDK with at least one platform installed. To install it, download the ADT Bundle from here. Extract it and add `<path_to_android_sdk>/sdk/platform-tools` and `<path_to_android_sdk>/sdk/tools` to your `PATH`. To test that you've installed it properly, run `adb version`. The output should be similar to this:

```
adb version
Android Debug Bridge version 1.0.39
```

Once that is working, run

```
android update sdk
```

This will open up a dialog box listing available android platforms and corresponding API levels, e.g. `Android 4.3 (API 18)`. For WA, you will need at least API level 18 (i.e. Android 4.3), though installing the latest is usually the best bet.

Optionally (but recommended), you should also set `ANDROID_HOME` to point to the install location of the SDK (i.e. `<path_to_android_sdk>/sdk`).

Python

Workload Automation 3 currently supports both Python 2.7 and Python 3.

pip

pip is the recommended package manager for Python. It is not part of standard Python distribution and would need to be installed separately. On Ubuntu and similar distributions, this may be done with APT:

```
sudo apt-get install python-pip
```

Note: Some versions of pip (in particluar v1.5.4 which comes with Ubuntu 14.04) are know to set the wrong permissions when installing packages, resulting in WA failing to import them. To avoid this it is recommended that you update pip and setuptools before proceeding with installation:

```
sudo -H pip install --upgrade pip
sudo -H pip install --upgrade setuptools
```

If you do run into this issue after already installing some packages, you can resolve it by running

```
sudo chmod -R a+r /usr/local/lib/python2.7/dist-packagessudo
find /usr/local/lib/python2.7/dist-packages -type d -exec chmod a+x {} \;
```

(The paths above will work for Ubuntu; they may need to be adjusted for other distros).

Python Packages

Note: pip should automatically download and install missing dependencies, so if you're using pip, you can skip this section. However some packages the will be installed have C plugins and will require Python development headers to install. You can get those by installing `python-dev` package in apt on Ubuntu (or the equivalent for your distribution).

Workload Automation 3 depends on the following additional libraries:

- pexpect
- docutils
- pySerial
- pyYAML
- python-dateutil
- louie
- pandas
- devlib
- wrapt
- requests
- colorama
- future

You can install these with pip:

```
sudo -H pip install pexpect
sudo -H pip install pyserial
sudo -H pip install pyyaml
sudo -H pip install docutils
sudo -H pip install python-dateutil
```

(continues on next page)

(continued from previous page)

```
sudo -H pip install devlib
sudo -H pip install pandas
sudo -H pip install louie
sudo -H pip install wrapt
sudo -H pip install requests
sudo -H pip install colorama
sudo -H pip install future
```

Some of these may also be available in your distro's repositories, e.g.

```
sudo apt-get install python-serial
```

Distro package versions tend to be older, so pip installation is recommended. However, pip will always download and try to build the source, so in some situations distro binaries may provide an easier fall back. Please also note that distro package names may differ from pip packages.

Optional Python Packages

Note: Unlike the mandatory dependencies in the previous section, pip will *not* install these automatically, so you will have to explicitly install them if/when you need them.

In addition to the mandatory packages listed in the previous sections, some WA functionality (e.g. certain plugins) may have additional dependencies. Since they are not necessary to be able to use most of WA, they are not made mandatory to simplify initial WA installation. If you try to use an plugin that has additional, unmet dependencies, WA will tell you before starting the run, and you can install it then. They are listed here for those that would rather install them upfront (e.g. if you're planning to use WA to an environment that may not always have Internet access).

- nose
- PyDAQmx
- pymongo
- jinja2

Installing

Installing the latest released version from PyPI (Python Package Index):

```
sudo -H pip install wa
```

This will install WA along with its mandatory dependencies. If you would like to install all optional dependencies at the same time, do the following instead:

```
sudo -H pip install wa[all]
```

Alternatively, you can also install the latest development version from GitHub (you will need git installed for this to work):

```
git clone git@github.com:ARM-software/workload-automation.git workload-automation
cd workload-automation
sudo -H python setup.py install
```

Note: Please note that if using pip to install from github this will most likely result in an older and incompatible version of devlib being installed alongside WA. If you wish to use pip please also manually install the latest version of devlib.

If the above succeeds, try

```
wa --version
```

Hopefully, this should output something along the lines of

```
"Workload Automation version $version".
```

Dockerfile

As an alternative we also provide a Dockerfile that will create an image called wadocker, and is preconfigured to run WA and devlib. Please note that the build process automatically accepts the licenses for the Android SDK, so please be sure that you are willing to accept these prior to building and running the image in a container.

The Dockerfile can be found in the "extras" directory or online at https://github.com/ARM-software/workload-automation/blob/next/extras/Dockerfile which contains additional information about how to build and to use the file.

(Optional) Post Installation

Some WA plugins have additional dependencies that need to be satisfied before they can be used. Not all of these can be provided with WA and so will need to be supplied by the user. They should be placed into `~/.workload_automation/dependencies/<extension name>` so that WA can find them (you may need to create the directory if it doesn't already exist). You only need to provide the dependencies for workloads you want to use.

APK Files

APKs are application packages used by Android. These are necessary to install on a device when running an *ApkWorkload* or derivative. Please see the workload description using the *show* command to see which version of the apk the UI automation has been tested with and place the apk in the corresponding workloads dependency directory. Automation may also work with other versions (especially if it's only a minor or revision difference – major version differences are more likely to contain incompatible UI changes) but this has not been tested. As a general rule we do not guarantee support for the latest version of an app and they are updated on an as needed basis. We do however attempt to support backwards compatibility with previous major releases however beyond this support will likely be dropped.

Gaming Workloads

Some workloads (games, demos, etc) cannot be automated using Android's UIAutomator framework because they render the entire UI inside a single OpenGL surface. For these, an interaction session needs to be recorded so that it can be played back by WA. These recordings are device-specific, so they would need to be done for each device you're planning to use. The tool for doing is `revent` and it is packaged with WA. You can find instructions on how to use it in the *How To* section.

This is the list of workloads that rely on such recordings:

angrybirds_rio
templerun2

Maintaining Centralized Assets Repository

If there are multiple users within an organization that may need to deploy assets for WA plugins, that organization may wish to maintain a centralized repository of assets that individual WA installs will be able to automatically retrieve asset files from as they are needed. This repository can be any directory on a network filer that mirrors the structure of `~/.workload_automation/dependencies`, i.e. has a subdirectories named after the plugins which assets they contain. Individual WA installs can then set `remote_assets_path` setting in their config to point to the local mount of that location.

(Optional) Uninstalling

If you have installed Workload Automation via `pip` and wish to remove it, run this command to uninstall it:

```
sudo -H pip uninstall wa
```

Note: This will *not* remove any user configuration (e.g. the ~/.workload_automation directory)

(Optional) Upgrading

To upgrade Workload Automation to the latest version via `pip`, run:

```
sudo -H pip install --upgrade --no-deps wa
```

2.1.2 User Guide

This guide will show you how to quickly start running workloads using Workload Automation 3.

Contents

Install

Note: This is a quick summary. For more detailed instructions, please see the *Installation* section.

Make sure you have Python 2.7 or Python 3 and a recent Android SDK with API level 18 or above installed on your system. A complete install of the Android SDK is required, as WA uses a number of its utilities, not just adb. For the SDK, make sure that either `ANDROID_HOME` environment variable is set, or that `adb` is in your `PATH`.

Note: If you plan to run Workload Automation on Linux devices only, SSH is required, and Android SDK is optional if you wish to run WA on Android devices at a later time.

However, you would be starting off with a limited number of workloads that will run on Linux devices.

In addition to the base Python install, you will also need to have `pip` (Python's package manager) installed as well. This is usually a separate package.

Once you have those, you can install WA with:

```
sudo -H pip install wlauto
```

This will install Workload Automation on your system, along with its mandatory dependencies.

Alternatively we provide a Dockerfile that which can be used to create a Docker image for running WA along with its dependencies. More information can be found in the *Installation* section.

(Optional) Verify installation

Once the tarball has been installed, try executing

```
wa -h
```

You should see a help message outlining available subcommands.

(Optional) APK files

A large number of WA workloads are installed as APK files. These cannot be distributed with WA and so you will need to obtain those separately.

For more details, please see the *installation* section.

List Command

In order to get started with using WA we first we need to find out what is available to use. In order to do this we can use the *list* command followed by the type of plugin that you wish to see.

For example to see what workloads are available along with a short description of each you run:

```
wa list workloads
```

Which will give an output in the format of:

```
   adobereader:    The Adobe Reader workflow carries out the following typical
                   productivity tasks.
    androbench:    Executes storage performance benchmarks
angrybirds_rio:    Angry Birds Rio game.
        antutu:    Executes Antutu 3D, UX, CPU and Memory tests
     applaunch:    This workload launches and measures the launch time of applications
                   for supporting workloads.
   benchmarkpi:    Measures the time the target device takes to run and complete the
                   Pi calculation algorithm.
     dhrystone:    Runs the Dhrystone benchmark.
     exoplayer:    Android ExoPlayer
     geekbench:    Geekbench provides a comprehensive set of benchmarks engineered to
                   quickly and accurately measure
                   processor and memory performance.
  #..
```

The same syntax can be used to display `commands`, `energy_instrument_backends`, `instruments`, `output_processors`, `resource_getters`, `targets`. Once you have found the plugin you are looking for you can use the *show* command to display more detailed information. Alternatively please see the *Plugin Reference* for an online version.

Show Command

If you want to learn more information about a particular plugin, such as the parameters it supports, you can use the "show" command:

```
wa show dhrystone
```

If you have `pandoc` installed on your system, this will display man page-like description of the plugin, and the parameters it supports. If you do not have `pandoc`, you will instead see the same information as raw restructured text.

Configure Your Device

There are multiple options for configuring your device depending on your particular use case.

You can either add your configuration to the default configuration file `config.yaml`, under the `$WA_USER_HOME/` directory or you can specify it in the `config` section of your agenda directly.

Alternatively if you are using multiple devices, you may want to create separate config files for each of your devices you will be using. This allows you to specify which device you would like to use for a particular run and pass it as an argument when invoking with the `-c` flag.

```
wa run dhrystone -c my_device.yaml
```

By default WA will use the "most specific" configuration available for example any configuration specified inside an agenda will override a passed configuration file which will in turn overwrite the default configuration file.

Note: For a more information about configuring your device please see *Setting Up A Device*.

Android

By default, the device WA will use is set to 'generic_android'. WA is configured to work with a generic Android device through `adb`. If you only have one device listed when you execute `adb devices`, and your device has a standard Android configuration, then no extra configuration is required.

However, if your device is connected via network, you will have to manually execute `adb connect <device ip>` (or specify this in your *agenda*) so that it appears in the device listing.

If you have multiple devices connected, you will need to tell WA which one you want it to use. You can do that by setting `device` in the device_config section.

```
# ...

device_config:
        device: 'abcdef0123456789'
        # ...
# ...
```

Linux

First, set the device to 'generic_linux'

```
# ...
  device: 'generic_linux'
# ...
```

Find the device_config section and add these parameters

```
# ...

device_config:
        host: '192.168.0.100'
        username: 'root'
        password: 'password'
        # ...
# ...
```

Parameters:

- Host is the IP of your target Linux device
- Username is the user for the device
- Password is the password for the device

Enabling and Disabling Augmentations

Augmentations are the collective name for "instruments" and "output processors" in WA3.

Some augmentations are enabled by default after your initial install of WA, which are specified in the `config.yaml` file located in your `WA_USER_DIRECTORY`, typically `~/.workload_autoamation`.

Note: Some Linux devices may not be able to run certain augmentations provided by WA (e.g. cpufreq is disabled or unsupported by the device).

```
# ...

augmentations:
    # Records the time it took to run the workload
    - execution_time

    # Collects /proc/interrupts before and after execution and does a diff.
    - interrupts

    # Collects the contents of/sys/devices/system/cpu before and after
    # execution and does a diff.
    - cpufreq

    # Generate a txt file containing general status information about
    # which runs failed and which were successful.
    - status

    # ...
```

If you only wanted to keep the 'execution_time' instrument enabled, you can comment out the rest of the list augmentations to disable them.

This should give you basic functionality. If you are working with a development board or you need some advanced functionality additional configuration may be required. Please see the *device setup* section for more details.

Note: In WA2 'Instrumentation' and 'Result Processors' were divided up into their own sections in the agenda. In WA3 they now fall under the same category of 'augmentations'. For compatibility the old naming structure is still valid however using the new entry names is recommended.

Running Your First Workload

The simplest way to run a workload is to specify it as a parameter to WA `run` *run* sub-command:

```
wa run dhrystone
```

You will see INFO output from WA as it executes each stage of the run. A completed run output should look something like this:

```
INFO     Creating output directory.
INFO     Initializing run
INFO     Connecting to target
INFO     Setting up target
INFO     Initializing execution context
INFO     Generating jobs
INFO         Loading job wk1 (dhrystone) [1]
INFO     Installing instruments
INFO     Installing output processors
INFO     Starting run
INFO     Initializing run
INFO         Initializing job wk1 (dhrystone) [1]
INFO     Running job wk1
INFO         Configuring augmentations
INFO         Configuring target for job wk1 (dhrystone) [1]
INFO         Setting up job wk1 (dhrystone) [1]
INFO         Running job wk1 (dhrystone) [1]
INFO         Tearing down job wk1 (dhrystone) [1]
INFO         Completing job wk1
INFO     Job completed with status OK
INFO     Finalizing run
INFO         Finalizing job wk1 (dhrystone) [1]
INFO     Done.
INFO     Run duration: 9 seconds
INFO     Ran a total of 1 iterations: 1 OK
INFO     Results can be found in wa_output
```

Once the run has completed, you will find a directory called `wa_output` in the location where you have invoked `wa run`. Within this directory, you will find a "results.csv" file which will contain results obtained for dhrystone, as well as a "run.log" file containing detailed log output for the run. You will also find a sub-directory called 'wk1-dhrystone-1' that contains the results for that iteration. Finally, you will find various additional information in the `wa_output/__meta` subdirectory for example information extracted from the target and a copy of the agenda file. The contents of iteration-specific subdirectories will vary from workload to workload, and, along with the contents of the main output directory, will depend on the augmentations that were enabled for that run.

The `run` sub-command takes a number of options that control its behaviour, you can view those by executing `wa run -h`. Please see the *Commands* section for details.

Create an Agenda

Simply running a single workload is normally of little use. Typically, you would want to specify several workloads, setup the device state and, possibly, enable additional augmentations. To do this, you would need to create an "agenda" for the run that outlines everything you want WA to do.

Agendas are written using YAML markup language. A simple agenda might look like this:

```
config:
        augmentations:
            - ~execution_time
            - json
        iterations: 2
workloads:
        - memcpy
        - name: dhrystone
          params:
```

(continues on next page)

(continued from previous page)

```
                mloops: 5
                threads: 1
```

This agenda:

- Specifies two workloads: memcpy and dhrystone.
- Specifies that dhrystone should run in one thread and execute five million loops.
- Specifies that each of the two workloads should be run twice.
- Enables json output processor, in addition to the output processors enabled in the config.yaml.
- Disables execution_time instrument, if it is enabled in the config.yaml

An agenda can be created using WA's `create` *command* or in a text editor and saved as a YAML file.

For more options please see the *Defining Experiments With an Agenda* documentation.

Using the Create Command

The easiest way to create an agenda is to use the 'create' command. For more in-depth information please see the *Create Command* documentation.

In order to populate the agenda with relevant information you can supply all of the plugins you wish to use as arguments to the command, for example if we want to create an agenda file for running `dhystrone` on a 'generic android' device and we want to enable the `execution_time` and `trace-cmd` instruments and display the metrics using the `csv` output processor. We would use the following command:

```
wa create agenda generic_android dhrystone execution_time trace-cmd csv -o my_agenda.
↪yaml
```

This will produce a *my_agenda.yaml* file containing all the relevant configuration for the specified plugins along with their default values as shown below:

```
config:
    augmentations:
    - execution_time
    - trace-cmd
    - csv
    iterations: 1
    device: generic_android
    device_config:
        adb_server: null
        big_core: null
        core_clusters: null
        core_names: null
        device: null
        disable_selinux: true
        executables_directory: null
        load_default_modules: true
        logcat_poll_period: null
        model: null
        modules: null
        package_data_directory: /data/data
        shell_prompt: !<tag:wa:regex> '8:^.*(shell|root)@.*:/\S* [#$] '
        working_directory: null
```

(continues on next page)

(continued from previous page)

```
    execution_time: {}
    trace-cmd:
        buffer_size: null
        buffer_size_step: 1000
        events:
        - sched*
        - irq*
        - power*
        - thermal*
        functions: null
        no_install: false
        report: true
        report_on_target: false
    csv:
        extra_columns: null
        use_all_classifiers: false
workloads:
-   name: dhrystone
    params:
        cleanup_assets: true
        delay: 0
        duration: 0
        mloops: 0
        taskset_mask: 0
        threads: 4
```

Run Command

These examples show some useful options that can be used with WA's `run` command.

Once we have created an agenda to use it with WA we can pass it as a argument to the run command e.g.:

```
wa run <path/to/agenda> (e.g. wa run ~/myagenda.yaml)
```

By default WA will use the "wa_output" directory to stores its output however to redirect the output to a different directory we can use:

```
wa run dhrystone -d my_output_directory
```

We can also tell WA to use additional config files by supplying it with the `-c` argument. One use case for passing additional config files is if you have multiple devices you wish test with WA, you can store the relevant device configuration in individual config files and then pass the file corresponding to the device you wish to use for that particular test.

Note: As previously mentioned, any more specific configuration present in the agenda file will overwrite the corresponding config parameters specified in the config file(s).

```
wa run -c myconfig.yaml ~/myagenda.yaml
```

To use the same output directory but override the existing contents to store new dhrystone results we can specify the `-f` argument:

```
wa run -f dhrystone
```

To display verbose output while running memcpy:

```
wa run --verbose memcpy
```

Output

The output directory will contain subdirectories for each job that was run, which will in turn contain the generated metrics and artifacts for each job. The directory will also contain a `run.log` file containing the complete log output for the run, and a `__meta` directory with the configuration and metadata for the run. Metrics are serialized inside `result.json` files inside each job's subdirectory. There may also be a `__failed` directory containing failed attempts for jobs that have been re-run.

Augmentations may add additional files at the run or job directory level. The default configuration has `status` and `csv` augmentations enabled which generate a `status.txt` containing status summary for the run and individual jobs, and a `results.csv` containing metrics from all jobs in a CSV table, respectively.

See *Output Directory Structure* for more information.

In order to make it easier to access WA results from scripts, WA provides an API that parses the contents of the output directory:

```
>>> from wa import RunOutput
>>> ro = RunOutput('./wa_output')
>>> for job in ro.jobs:
...     if job.status != 'OK':
...         print 'Job "{}" did not complete successfully: {}'.format(job, job.status)
...         continue
...     print 'Job "{}":'.format(job)
...     for metric in job.metrics:
...         if metric.units:
...             print '\t{}: {} {}'.format(metric.name, metric.value, metric.units)
...         else:
...             print '\t{}: {}'.format(metric.name, metric.value)
...
Job "wk1-dhrystone-1":
        thread 0 score: 20833333
        thread 0 DMIPS: 11857
        thread 1 score: 24509804
        thread 1 DMIPS: 13950
        thread 2 score: 18011527
        thread 2 DMIPS: 10251
        thread 3 score: 26371308
        thread 3 DMIPS: 15009
        time: 1.001251 seconds
        total DMIPS: 51067
        total score: 89725972
        execution_time: 1.4834280014 seconds
```

See *Output* for details.

Uninstall

If you have installed Workload Automation via `pip`, then run this command to uninstall it:

```
sudo pip uninstall wa
```

Note: It will *not* remove any user configuration (e.g. the ~/.workload_automation directory).

Upgrade

To upgrade Workload Automation to the latest version via `pip`, run:

```
sudo pip install --upgrade --no-deps wa
```

2.1.3 How Tos

Contents

Defining Experiments With an Agenda

An agenda specifies what is to be done during a Workload Automation run, including which workloads will be run, with what configuration, which augmentations will be enabled, etc. Agenda syntax is designed to be both succinct and expressive.

Agendas are specified using YAML notation. It is recommended that you familiarize yourself with the linked page.

Specifying which workloads to run

The central purpose of an agenda is to specify what workloads to run. A minimalist agenda contains a single entry at the top level called "workloads" that maps onto a list of workload names to run:

```
workloads:
        - dhrystone
        - memcpy
        - rt_app
```

This specifies a WA run consisting of `dhrystone` followed by `memcpy`, followed by `rt_app` workloads, and using the augmentations specified in config.yaml (see *Configuration* section).

Note: If you're familiar with YAML, you will recognize the above as a single-key associative array mapping onto a list. YAML has two notations for both associative arrays and lists: block notation (seen above) and also in-line notation. This means that the above agenda can also be written in a single line as

```
workloads: [dhrystone, memcpy, rt-app]
```

(with the list in-lined), or

```
{workloads: [dhrystone, memcpy, rt-app]}
```

(with both the list and the associative array in-line). WA doesn't care which of the notations is used as they all get parsed into the same structure by the YAML parser. You can use whatever format you find easier/clearer.

Note: WA plugin names are case-insensitive, and dashes (-) and underscores (_) are treated identically. So all of the following entries specify the same workload: `rt_app`, `rt-app`, `RT-app`.

Multiple iterations

There will normally be some variability in workload execution when running on a real device. In order to quantify it, multiple iterations of the same workload are usually performed. You can specify the number of iterations for each workload by adding `iterations` field to the workload specifications (or "specs"):

```
workloads:
        - name: dhrystone
          iterations: 5
        - name: memcpy
          iterations: 5
        - name: cyclictest
          iterations: 5
```

Now that we're specifying both the workload name and the number of iterations in each spec, we have to explicitly name each field of the spec.

It is often the case that, as in in the example above, you will want to run all workloads for the same number of iterations. Rather than having to specify it for each and every spec, you can do with a single entry by adding *iterations* to your `config` section in your agenda:

```
config:
        iterations: 5
workloads:
        - dhrystone
        - memcpy
        - cyclictest
```

If the same field is defined both in config section and in a spec, then the value in the spec will overwrite the value. For example, suppose we wanted to run all our workloads for five iterations, except cyclictest which we want to run for ten (e.g. because we know it to be particularly unstable). This can be specified like this:

```
config:
        iterations: 5
workloads:
        - dhrystone
        - memcpy
        - name: cyclictest
          iterations: 10
```

Again, because we are now specifying two fields for cyclictest spec, we have to explicitly name them.

Configuring Workloads

Some workloads accept configuration parameters that modify their behaviour. These parameters are specific to a particular workload and can alter the workload in any number of ways, e.g. set the duration for which to run, or specify a media file to be used, etc. The vast majority of workload parameters will have some default value, so it is only necessary to specify the name of the workload in order for WA to run it. However, sometimes you want more control over how a workload runs.

For example, by default, dhrystone will execute 10 million loops across four threads. Suppose your device has six cores available and you want the workload to load them all. You also want to increase the total number of loops accordingly to 15 million. You can specify this using dhrystone's parameters:

```
config:
        iterations: 5
workloads:
        - name: dhrystone
          params:
                threads: 6
                mloops: 15
        - memcpy
        - name: cyclictest
          iterations: 10
```

Note: You can find out what parameters a workload accepts by looking it up in the *Workloads* section or using WA itself with "show" command:

```
wa show dhrystone
```

see the *Commands* section for details.

In addition to configuring the workload itself, we can also specify configuration for the underlying device which can be done by setting runtime parameters in the workload spec. Explicit runtime parameters have been exposed for configuring cpufreq, hotplug and cpuidle. For more detailed information on Runtime Parameters see the *runtime parameters* section. For example, suppose we want to ensure the maximum score for our benchmarks, at the expense of power consumption so we want to set the cpufreq governor to "performance" and enable all of the cpus on the device, (assuming there are 8 cpus available), which can be done like this:

```
config:
        iterations: 5
workloads:
        - name: dhrystone
          runtime_params:
                governor: performance
                num_cores: 8
          workload_params:
                threads: 6
                mloops: 15
        - memcpy
        - name: cyclictest
          iterations: 10
```

I've renamed `params` to `workload_params` for clarity, but that wasn't strictly necessary as `params` is interpreted as `workload_params` inside a workload spec.

Runtime parameters do not automatically reset at the end of workload spec execution, so all subsequent iterations will also be affected unless they explicitly change the parameter (in the example above, performance governor will also be used for `memcpy` and `cyclictest`. There are two ways around this: either set `reboot_policy` WA setting (see *Configuration* section) such that the device gets rebooted between job executions, thus being returned to its initial state, or set the default runtime parameter values in the `config` section of the agenda so that they get set for every spec that doesn't explicitly override them.

If additional configuration of the device is required which are not exposed via the built in runtime parameters, you can write a value to any file exposed on the device using `sysfile_values`, for example we could have also performed

the same configuration manually (assuming we have a big.LITTLE system and our cores 0-3 and 4-7 are in 2 separate DVFS domains and so setting the governor for cpu0 and cpu4 will affect all our cores) e.g.

```
config:
        iterations: 5
workloads:
        - name: dhrystone
        runtime_params:
                sysfile_values:
                    /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor: performance
                    /sys/devices/system/cpu/cpu4/cpufreq/scaling_governor: performance
                    /sys/devices/system/cpu/cpu0/online: 1
                    /sys/devices/system/cpu/cpu1/online: 1
                    /sys/devices/system/cpu/cpu2/online: 1
                    /sys/devices/system/cpu/cpu3/online: 1
                    /sys/devices/system/cpu/cpu4/online: 1
                    /sys/devices/system/cpu/cpu5/online: 1
                    /sys/devices/system/cpu/cpu6/online: 1
                    /sys/devices/system/cpu/cpu7/online: 1
        workload_params:
                threads: 6
                mloops: 15
    - memcpy
    - name: cyclictest
        iterations: 10
```

Here, we're specifying a `sysfile_values` runtime parameter for the device. For more information please see *setting sysfiles*.

APK Workloads

WA has various resource getters that can be configured to locate APK files but for most people APK files should be kept in the `$WA_USER_DIRECTORY/dependencies/SOME_WORKLOAD/` directory. (by default `~/.workload_automation/dependencies/SOME_WORKLOAD/`). The `WA_USER_DIRECTORY` environment variable can be used to change the location of this directory. The APK files need to be put into the corresponding directories for the workload they belong to. The name of the file can be anything but as explained below may need to contain certain pieces of information.

All ApkWorkloads have parameters that affect the way in which APK files are resolved, `exact_abi`, `force_install` and `prefer_host_package`. Their exact behaviours are outlined below.

exact_abi If this setting is enabled WA's resource resolvers will look for the devices ABI with any native code present in the apk. By default this setting is disabled since most apks will work across all devices. You may wish to enable this feature when working with devices that support multiple ABI's (like 64-bit devices that can run 32-bit APK files) and are specifically trying to test one or the other.

force_install If this setting is enabled WA will *always* use the APK file on the host, and re-install it on every iteration. If there is no APK on the host that is a suitable version and/or ABI for the workload WA will error when `force_install` is enabled.

prefer_host_package This parameter is used to specify a preference over host or target versions of the app. When set to `True` WA will prefer the host side version of the APK. It will check if the host has the APK and whether it meets the version requirements of the workload. If so, and the target also already has same version nothing will be done, otherwise WA will overwrite the targets installed application with the host version. If the host is missing the APK or it does not meet version requirements WA will fall back to the app on the target if present and is a suitable version. When

this parameter is set to `False` WA will prefer to use the version already on the target if it meets the workloads version requirements. If it does not it will fall back to searching the host for the correct version. In both modes if neither the host nor target have a suitable version, WA will produce and error and will not run the workload.

version This parameter is used to specify which version of uiautomation for the workload is used. In some workloads e.g. `geekbench` multiple versions with drastically different UI's are supported. A APKs version will be automatically extracted therefore it is possible to have multiple apks for different versions of a workload present on the host and select between which is used for a particular job by specifying the relevant version in your *agenda*.

variant_name Some workloads use variants of APK files, this is usually the case with web browser APK files, these work in exactly the same way as the version.

IDs and Labels

It is possible to list multiple specs with the same workload in an agenda. You may wish to do this if you want to run a workload with different parameter values or under different runtime configurations of the device. The workload name therefore does not uniquely identify a spec. To be able to distinguish between different specs (e.g. in reported results), each spec has an ID which is unique to all specs within an agenda (and therefore with a single WA run). If an ID isn't explicitly specified using `id` field (note that the field name is in lower case), one will be automatically assigned to the spec at the beginning of the WA run based on the position of the spec within the list. The first spec *without an explicit ID* will be assigned ID `wk1`, the second spec *without an explicit ID* will be assigned ID `wk2`, and so forth.

Numerical IDs aren't particularly easy to deal with, which is why it is recommended that, for non-trivial agendas, you manually set the ids to something more meaningful (or use labels – see below). An ID can be pretty much anything that will pass through the YAML parser. The only requirement is that it is unique to the agenda. However, is usually better to keep them reasonably short (they don't need to be *globally* unique), and to stick with alpha-numeric characters and underscores/dashes. While WA can handle other characters as well, getting too adventurous with your IDs may cause issues further down the line when processing WA output (e.g. when uploading them to a database that may have its own restrictions).

In addition to IDs, you can also specify labels for your workload specs. These are similar to IDs but do not have the uniqueness restriction. If specified, labels will be used by some output processes instead of (or in addition to) the workload name. For example, the `csv` output processor will put the label in the "workload" column of the CSV file.

It is up to you how you chose to use IDs and labels. WA itself doesn't expect any particular format (apart from uniqueness for IDs). Below is the earlier example updated to specify explicit IDs and label dhrystone spec to reflect parameters used.

```
config:
        iterations: 5
workloads:
        - id: 01_dhry
          name: dhrystone
          label: dhrystone_15over6
          runtime_params:
                cpu0_governor: performance
          workload_params:
                threads: 6
                mloops: 15
        - id: 02_memc
          name: memcpy
        - id: 03_cycl
          name: cyclictest
          iterations: 10
```

Classifiers

Classifiers can be used in 2 distinct ways, the first use is being supplied in an agenda as a set of key-value pairs which can be used to help identify sub-tests of a run, for example if you have multiple sections in your agenda running your workloads at different frequencies you might want to set a classifier specifying which frequencies are being used. These can then be utilized later, for example with the `csv` *output processor* with `use_all_classifiers` set to `True` and this will add additional columns to the output file for each of the classifier keys that have been specified allowing for quick comparison.

An example agenda is shown here:

```
config:
    augmentations:
        - csv
    iterations: 1
    device: generic_android
    csv:
        use_all_classifiers: True
sections:
    - id: max_speed
      runtime_parameters:
          frequency: 1700000
      classifiers:
          freq: 1700000
    - id: min_speed
      runtime_parameters:
          frequency: 200000
      classifiers:
          freq: 200000
workloads:
-   name: recentfling
```

The other way that they can used is by being automatically added by some workloads to identify their results metrics and artifacts. For example some workloads perform multiple tests with the same execution run and therefore will use metrics to differentiate between them, e.g. the `recentfling` workload will use classifiers to distinguish between which loop a particular result is for or whether it is an average across all loops ran.

The output from the agenda above will produce a csv file similar to what is shown below. Some columns have been omitted for clarity however as can been seen the custom **frequency** classifier column has been added and populated, along with the **loop** classifier added by the workload.

```
id              | workload      | metric                    | freq      | loop    |␣
↪value ||
max_speed-wk1   | recentfling   | 90th Percentile           | 1700000   | 1       | 8␣
↪    ||
max_speed-wk1   | recentfling   | 95th Percentile           | 1700000   | 1       | 9␣
↪    ||
max_speed-wk1   | recentfling   | 99th Percentile           | 1700000   | 1       |␣
↪16    ||
max_speed-wk1   | recentfling   | Jank                      | 1700000   | 1       |␣
↪11    ||
max_speed-wk1   | recentfling   | Jank%                     | 1700000   | 1       | 1␣
↪    ||
# ...
max_speed-wk1   | recentfling   | Jank                      | 1700000   | 3       | 1␣
↪    ||
max_speed-wk1   | recentfling   | Jank%                     | 1700000   | 3       | 0␣
↪    ||
```

(continues on next page)

(continued from previous page)

```
max_speed-wk1   | recentfling   | Average 90th Percentqile  | 1700000   | Average | 7␣
↪    ||
max_speed-wk1   | recentfling   | Average 95th Percentile   | 1700000   | Average | 8␣
↪    ||
max_speed-wk1   | recentfling   | Average 99th Percentile   | 1700000   | Average |␣
↪14    ||
max_speed-wk1   | recentfling   | Average Jank              | 1700000   | Average | 6␣
↪    ||
max_speed-wk1   | recentfling   | Average Jank%             | 1700000   | Average | 0␣
↪    ||
min_speed-wk1   | recentfling   | 90th Percentile           | 200000    | 1       | 7␣
↪    ||
min_speed-wk1   | recentfling   | 95th Percentile           | 200000    | 1       | 8␣
↪    ||
min_speed-wk1   | recentfling   | 99th Percentile           | 200000    | 1       |␣
↪14    ||
min_speed-wk1   | recentfling   | Jank                      | 200000    | 1       | 5␣
↪    ||
min_speed-wk1   | recentfling   | Jank%                     | 200000    | 1       | 0␣
↪    ||
# ...
min_speed-wk1   | recentfling   | Jank                      | 200000    | 3       | 5␣
↪    ||
min_speed-wk1   | recentfling   | Jank%                     | 200000    | 3       | 0␣
↪    ||
min_speed-wk1   | recentfling   | Average 90th Percentile   | 200000    | Average | 7␣
↪    ||
min_speed-wk1   | recentfling   | Average 95th Percentile   | 200000    | Average | 8␣
↪    ||
min_speed-wk1   | recentfling   | Average 99th Percentile   | 200000    | Average |␣
↪13    ||
min_speed-wk1   | recentfling   | Average Jank              | 200000    | Average | 4␣
↪    ||
min_speed-wk1   | recentfling   | Average Jank%             | 200000    | Average | 0␣
↪    ||
```

Sections

It is a common requirement to be able to run the same set of workloads under different device configurations. E.g. you may want to investigate the impact of changing a particular setting to different values on the benchmark scores, or to quantify the impact of enabling a particular feature in the kernel. WA allows this by defining "sections" of configuration with an agenda.

For example, suppose that we want to measure the impact of using 3 different cpufreq governors on 2 benchmarks. We could create 6 separate workload specs and set the governor runtime parameter for each entry. However, this introduces a lot of duplication; and what if we want to change spec configuration? We would have to change it in multiple places, running the risk of forgetting one.

A better way is to keep the two workload specs and define a section for each governor:

```
config:
        iterations: 5
        augmentations:
            - ~cpufreq
            - csv
```

(continues on next page)

(continued from previous page)

```
        sysfs_extractor:
                paths: [/proc/meminfo]
        csv:
            use_all_classifiers: True
sections:
        - id: perf
          runtime_params:
                cpu0_governor: performance
        - id: inter
          runtime_params:
                cpu0_governor: interactive
        - id: sched
          runtime_params:
                cpu0_governor: sched
workloads:
        - id: 01_dhry
          name: dhrystone
          label: dhrystone_15over6
          workload_params:
                threads: 6
                mloops: 15
        - id: 02_memc
          name: memcpy
          augmentations: [sysfs_extractor]
```

A section, just like an workload spec, needs to have a unique ID. Apart from that, a "section" is similar to the `config` section we've already seen – everything that goes into a section will be applied to each workload spec. Workload specs defined under top-level `workloads` entry will be executed for each of the sections listed under `sections`.

Note: It is also possible to have a `workloads` entry within a section, in which case, those workloads will only be executed for that specific section.

In order to maintain the uniqueness requirement of workload spec IDs, they will be namespaced under each section by prepending the section ID to the spec ID with a dash. So in the agenda above, we no longer have a workload spec with ID `01_dhry`, instead there are two specs with IDs `perf-01-dhry` and `inter-01_dhry`.

Note that the `config` section still applies to every spec in the agenda. So the precedence order is – spec settings override section settings, which in turn override global settings.

Section Groups

Section groups are a way of grouping sections together and are used to produce a cross product of each of the different groups. This can be useful when you want to run a set of experiments with all the available combinations without having to specify each combination manually.

For example if we want to investigate the differences between running the maximum and minimum frequency with both the maximum and minimum number of cpus online, we can create an agenda as follows:

```
sections:
  - id: min_freq
  runtime_parameters:
      freq: min
  group: frequency
 - id: max_freq
```

(continues on next page)

(continued from previous page)

```
  runtime_parameters:
      freq: max
  group: frequency

 - id: min_cpus
   runtime_parameters:
      cpus: 1
  group: cpus
 - id: max_cpus
   runtime_parameters:
      cpus: 8
  group: cpus

workloads:
-  dhrystone
```

This will results in 8 jobs being generated for each of the possible combinations.

```
min_freq-min_cpus-wk1 (dhrystone)
min_freq-max_cpus-wk1 (dhrystone)
max_freq-min_cpus-wk1 (dhrystone)
max_freq-max_cpus-wk1 (dhrystone)
min_freq-min_cpus-wk1 (dhrystone)
min_freq-max_cpus-wk1 (dhrystone)
max_freq-min_cpus-wk1 (dhrystone)
max_freq-max_cpus-wk1 (dhrystone)
```

Each of the generated jobs will have *classifiers* for each group and the associated id automatically added.

```
# ...
print('Job ID: {}'.format(job.id))
print('Classifiers:')
for k, v in job.classifiers.items():
    print('  {}: {}'.format(k, v))

Job ID: min_freq-min_cpus-no_idle-wk1
Classifiers:
    frequency: min_freq
    cpus: min_cpus
```

Augmentations

Augmentations are plugins that augment the execution of workload jobs with additional functionality; usually, that takes the form of generating additional metrics and/or artifacts, such as traces or logs. There are two types of augmentations:

Instruments These "instrument" a WA run in order to change it's behaviour (e.g. introducing delays between successive job executions), or collect additional measurements (e.g. energy usage). Some instruments may depend on particular features being enabled on the target (e.g. cpufreq), or on additional hardware (e.g. energy probes).

Output processors These post-process metrics and artifacts generated by workloads or instruments, as well as target metadata collected by WA, in order to generate additional metrics and/or artifacts (e.g. generating statistics or reports). Output processors are also used to export WA output externally (e.g. upload to a database).

The main practical difference between instruments and output processors, is that the former rely on an active connection to the target to function, where as the latter only operated on previously collected results and metadata. This

means that output processors can run "off-line" using `wa process` command.

Both instruments and output processors are configured in the same way in the agenda, which is why they are grouped together into "augmentations". Augmentations are enabled by listing them under `augmentations` entry in a config file or `config` section of the agenda.

```
config:
        augmentations: [trace-cmd]
```

The code above illustrates an agenda entry to enabled `trace-cmd` instrument.

If your have multiple `augmentations` entries (e.g. both, in your config file and in the agenda), then they will be combined, so that the final set of augmentations for the run will be their union.

Note: WA2 did not have have augmentationts, and instead supported "instrumentation" and "result_processors" as distinct configuration enetries. For compantibility, these entries are still supported in WA3, however they should be considered to be depricated, and their use is discouraged.

Configuring augmentations

Most augmentations will take parameters that modify their behavior. Parameters available for a particular augmentation can be viewed using `wa show <augmentation name>` command. This will also show the default values used. Values for these parameters can be specified by creating an entry with the augmentation's name, and specifying parameter values under it.

```
config:
        augmentations: [trace-cmd]
        trace-cmd:
                events: ['sched*', 'power*', irq]
                buffer_size: 100000
```

The code above specifies values for `events` and `buffer_size` parameters for the `trace-cmd` instrument, as well as enabling it.

You may specify configuration for the same augmentation in multiple locations (e.g. your config file and the config section of the agenda). These entries will be combined to form the final configuration for the augmentation used during the run. If different values for the same parameter are present in multiple entries, the ones "more specific" to a particular run will be used (e.g. values in the agenda will override those in the config file).

Note: Creating an entry for an augmentation alone does not enable it! You **must** list it under `augmentations` in order for it to be enabed for a run. This makes it easier to quickly enabled and diable augmentations with complex configurations, and also allows defining "static" configuation in top-level config, without actually enabling the augmentation for all runs.

Disabling augmentations

Sometimes, you may wish to disable an augmentation for a particular run, but you want to keep it enabled in general. You *could* modify your config file to temporarily disable it. However, you must then remember to re-enable it afterwards. This could be inconvenient and error prone, especially if you're running multiple experiments in parallel and only want to disable the augmentation for one of them.

Instead, you can explicitly disable augmentation by specifying its name prefixed with a tilde (~) inside `augumentations`.

```
config:
        augmentations: [trace-cmd, ~cpufreq]
```

The code above enables `trace-cmd` instrument and disables `cpufreq` instrument (which is enabled in the default config).

If you want to start configuration for an experiment form a "blank slate" and want to disable all previously-enabled augmentations, without necessarily knowing what they are, you can use the special `~~` entry.

```
config:
        augmentations: [~~, trace-cmd, csv]
```

The code above disables all augmentations enabled up to that point, and enabled `trace-cmd` and `csv` for this run.

Note: The `~~` only disables augmentations from previously-processed sources. Its ordering in the list does not matter. For example, specifying `augmentations: [trace-cmd, ~~, csv]` will have exactly the same effect as above – i.e. both trace-cmd *and* csv will be enabled.

Workload-specific augmentation

It is possible to enable or disable (but not configure) augmentations at workload or section level, as well as in the global config, in which case, the augmentations would only be enabled/disabled for that workload/section. If the same augmentation is enabled at one level and disabled at another, as will all WA configuration, the more specific settings will take precedence over the less specific ones (i.e. workloads override sections that, in turn, override global config).

Augmentations Example

```
config:
        augmentations: [~~, fps]
        trace-cmd:
                events: ['sched*', 'power*', irq]
                buffer_size: 100000
        file_poller:
                files:
                        - /sys/class/thermal/thermal_zone0/temp
sections:
        - classifers:
                type: energy
        augmentations: [energy_measurement]
        - classifers:
                type: trace
        augmentations: [trace-cmd, file_poller]
workloads:
        - gmail
        - geekbench
        - googleplaybooks
        - name: dhrystone
          augmentations: [~fps]
```

The example above shows an experiment that runs a number of workloads in order to evaluate their thermal impact and energy usage. All previously-configured augmentations are disabled with `~~`, so that only configuration specified in this agenda is enabled. Since most of the workloads are "productivity" use cases that do not generate their own metrics, `fps` instrument is enabled to get some meaningful performance metrics for them; the only exception is `dhrystone` which is a benchmark that reports its own metrics and has not GUI, so the instrument is disabled for it using `~fps`.

Each workload will be run in two configurations: once, to collect energy measurements, and once to collect thermal data and kernel trace. Trace can give insight into why a workload is using more or less energy than expected, but it can be relatively intrusive and might impact absolute energy and performance metrics, which is why it is collected separately. *Classifiers* are used to separate metrics from the two configurations in the results.

Other Configuration

As mentioned previously, `config` section in an agenda can contain anything that can be defined in `config.yaml`. Certain configuration (e.g. `run_name`) makes more sense to define in an agenda than a config file. Refer to the *Configuration* section for details.

```
config:
        project: governor_comparison
        run_name: performance_vs_interactive

        device: generic_android
        reboot_policy: never

        iterations: 5
        augmentations:
            - ~cpufreq
            - csv
        sysfs_extractor:
                paths: [/proc/meminfo]
        csv:
            use_all_classifiers: True
sections:
        - id: perf
          runtime_params:
                sysfile_values:
                cpu0_governor: performance
        - id: inter
          runtime_params:
                cpu0_governor: interactive
workloads:
        - id: 01_dhry
          name: dhrystone
          label: dhrystone_15over6
          workload_params:
                threads: 6
                mloops: 15
        - id: 02_memc
          name: memcpy
          augmentations: [sysfs_extractor]
        - id: 03_cycl
          name: cyclictest
          iterations: 10
```

Setting Up A Device

WA should work with most Android devices out-of-the box, as long as the device is discoverable by `adb` (i.e. gets listed when you run `adb devices`). For USB-attached devices, that should be the case; for network devices, `adb connect` would need to be invoked with the IP address of the device. If there is only one device connected to the host running WA, then no further configuration should be necessary (though you may want to *tweak some Android settings*).

If you have multiple devices connected, have a non-standard Android build (e.g. on a development board), or want to use of the more advanced WA functionality, further configuration will be required.

Android

General Device Setup

You can specify the device interface by setting `device` setting in a `config` file or section. Available interfaces can be viewed by running `wa list targets` command. If you don't see your specific platform listed (which is likely unless you're using one of the Arm-supplied platforms), then you should use `generic_android` interface (this is what is used by the default config).

```
device: generic_android
```

The device interface may be configured through `device_config` setting, who's value is a `dict` mapping setting names to their values. Some of the most common parameters you might want to change are outlined below.

device If you have multiple Android devices connected to the host machine, you will need to set this to indicate to WA which device you want it to use. The will be the adb name the is displayed when running `adb devices`

working_directory WA needs a "working" directory on the device which it will use for collecting traces, caching assets it pushes to the device, etc. By default, it will create one under `/sdcard` which should be mapped and writable on standard Android builds. If this is not the case for your device, you will need to specify an alternative working directory (e.g. under `/data/local`).

modules A list of additional modules to be installed for the target. Devlib implements functionality for particular subsystems as modules. A number of "default" modules (e.g. for cpufreq subsystem) are loaded automatically, unless explicitly disabled. If additional modules need to be loaded, they may be specified using this parameter.

Please see the devlib documentation for information on the available modules.

core_names `core_names` should be a list of core names matching the order in which they are exposed in sysfs. For example, Arm TC2 SoC is a 2x3 big.LITTLE system; its core_names would be `['a7', 'a7', 'a7', 'a15', 'a15']`, indicating that cpu0-cpu2 in cpufreq sysfs structure are A7's and cpu3 and cpu4 are A15's.

Note: This should not usually need to be provided as it will be automatically extracted from the target.

A typical `device_config` inside `config.yaml` may look something like

```
device_config:
        device: 0123456789ABCDEF
# ...
```

or a more specific config could be:

```
device_config:
        device: 0123456789ABCDEF
        working_direcory: '/sdcard/wa-working'
        modules: ['hotplug', 'cpufreq']
        core_names : ['a7', 'a7', 'a7', 'a15', 'a15']
        # ...
```

Configuring Android

There are a few additional tasks you may need to perform once you have a device booted into Android (especially if this is an initial boot of a fresh OS deployment):

- You have gone through FTU (first time usage) on the home screen and in the apps menu.
- You have disabled the screen lock.
- You have set sleep timeout to the highest possible value (30 mins on most devices).
- You have set the locale language to "English" (this is important for some workloads in which UI automation looks for specific text in UI elements).

Juno Setup

Note: At the time of writing, the Android software stack on Juno was still very immature. Some workloads may not run, and there maybe stability issues with the device.

The full software stack can be obtained from Linaro:

https://releases.linaro.org/android/images/lcr-reference-juno/latest/

Please follow the instructions on the "Binary Image Installation" tab on that page. More up-to-date firmware and kernel may also be obtained by registered members from ARM Connected Community: http://www.arm.com/community/ (though this is not guaranteed to work with the Linaro file system).

UEFI

Juno uses UEFI to boot the kernel image. UEFI supports multiple boot configurations, and presents a menu on boot to select (in default configuration it will automatically boot the first entry in the menu if not interrupted before a timeout). WA will look for a specific entry in the UEFI menu (`'WA'` by default, but that may be changed by setting `uefi_entry` in the `device_config`). When following the UEFI instructions on the above Linaro page, please make sure to name the entry appropriately (or to correctly set the `uefi_entry`).

There are two supported ways for Juno to discover kernel images through UEFI. It can either load them from NOR flash on the board, or from the boot partition on the file system. The setup described on the Linaro page uses the boot partition method.

If WA does not find the UEFI entry it expects, it will create one. However, it will assume that the kernel image resides in NOR flash, which means it will not work with Linaro file system. So if you're replicating the Linaro setup exactly, you will need to create the entry manually, as outline on the above-linked page.

Rebooting

At the time of writing, normal Android reboot did not work properly on Juno Android, causing the device to crash into an irrecoverable state. Therefore, WA will perform a hard reset to reboot the device. It will attempt to do this by toggling the DTR line on the serial connection to the device. In order for this to work, you need to make sure that SW1 configuration switch on the back panel of the board (the right-most DIP switch) is toggled *down*.

Linux

General Device Setup

You can specify the device interface by setting `device` setting in a `config` file or section. Available interfaces can be viewed by running `wa list targets` command. If you don't see your specific platform listed (which is likely unless you're using one of the Arm-supplied platforms), then you should use `generic_linux` interface.

```
device: generic_linux
```

The device interface may be configured through `device_config` setting, who's value is a `dict` mapping setting names to their values. Some of the most common parameters you might want to change are outlined below.

host This should be either the the DNS name or IP address of the device.

username The login name of the user on the device that WA will use. This user should have a home directory (unless an alternative working directory is specified using `working_directory` config – see below), and, for full functionality, the user should have sudo rights (WA will be able to use sudo-less acounts but some instruments or workload may not work).

password Password for the account on the device. Either this of a `keyfile` (see below) must be specified.

keyfile If key-based authentication is used, this may be used to specify the SSH identity file instead of the password.

property_files This is a list of paths that will be pulled for each WA run into the __meta subdirectory in the results. The intention is to collect meta-data about the device that may aid in reporducing the results later. The paths specified do not have to exist on the device (they will be ignored if they do not). The default list is `['/proc/version', '/etc/debian_version', '/etc/lsb-release', '/etc/arch-release']`

In addition, `working_directory`, `core_names`, `modules` etc. can also be specified and have the same meaning as for Android devices (see above).

A typical `device_config` inside `config.yaml` may look something like

```
device_config:
        host: 192.168.0.7
        username: guest
        password: guest
        # ...
```

Chrome OS

General Device Setup

You can specify the device interface by setting `device` setting in a `config` file or section. Available interfaces can be viewed by running `wa list targets` command. If you don't see your specific platform listed (which is likely unless you're using one of the Arm-supplied platforms), then you should use `generic_chromeos` interface.

```
device: generic_chromeos
```

The device interface may be configured through `device_config` setting, who's value is a `dict` mapping setting names to their values. The ChromeOS target is essentially the same as a linux device and requires a similar setup, however it also optionally supports connecting to an android container running on the device which will be automatically detected if present. If the device supports android applications then the android configuration is also supported. In order to support this WA will open 2 connections to the device, one via SSH to the main OS and another via ADB to the android container where a limited subset of functionality can be performed.

In order to distinguish between the two connections some of the android specific configuration has been renamed to reflect the destination.

android_working_directory WA needs a "working" directory on the device which it will use for collecting traces, caching assets it pushes to the device, etc. By default, it will create one under `/sdcard` which should be mapped and writable on standard Android builds. If this is not the case for your device, you will need to specify an alternative working directory (e.g. under `/data/local`).

A typical `device_config` inside `config.yaml` for a ChromeOS device may look something like

```
device_config:
        host: 192.168.0.7
        username: root
        android_working_direcory: '/sdcard/wa-working'
        # ...
```

Note: This assumes that your Chromebook is in developer mode and is configured to run an SSH server with the appropriate ssh keys added to the authorized_keys file on the device.

Related Settings

Reboot Policy

This indicates when during WA execution the device will be rebooted. By default this is set to `as_needed`, indicating that WA will only reboot the device if it becomes unresponsive. Please see `reboot_policy` documentation in *Configuration* for more details.

Execution Order

`execution_order` defines the order in which WA will execute workloads. `by_iteration` (set by default) will execute the first iteration of each spec first, followed by the second iteration of each spec (that defines more than one iteration) and so forth. The alternative will loop through all iterations for the first first spec first, then move on to second spec, etc. Again, please see *Configuration* for more details.

Adding a new target interface

If you are working with a particularly unusual device (e.g. a early stage development board) or need to be able to handle some quirk of your Android build, configuration available in `generic_android` interface may not be enough for you. In that case, you may need to write a custom interface for your device. A device interface is an `Extension` (a plug-in) type in WA and is implemented similar to other extensions (such as workloads or instruments). Pleaser refer to the *adding a custom target* section for information on how this may be done.

Automating GUI Interactions With Revent

Overview and Usage

The revent utility can be used to record and later play back a sequence of user input events, such as key presses and touch screen taps. This is an alternative to Android UI Automator for providing automation for workloads.

Using revent with workloads

Some workloads (pretty much all games) rely on recorded revents for their execution. ReventWorkloads will require between 1 and 4 revent files be be ran. There is one mandatory recording `run` for performing the actual execution of the workload and the remaining are optional. `setup` can be used to perform the initial setup (navigating menus, selecting game modes, etc). `extract_results` can be used to perform any actions after the main stage of the workload for example to navigate a results or summary screen of the app. And finally `teardown` can be used to perform any final actions for example exiting the app.

Because revents are very device-specific*[0], these files would need to be recorded for each device.

The files must be called `<device name>.(setup|run|extract_results|teardown).revent` , where `<device name>` is the name of your device (as defined by the `name` attribute of your device's class). WA will look for these files in two places: `<install dir>/wa/workloads/<workload name>/revent_files` and `~/.workload_automation/dependencies/<workload name>`. The first location is primarily intended for revent files that come with WA (and if you did a system-wide install, you'll need sudo to add files there), so it's probably easier to use the second location for the files you record. Also, if revent files for a workload exist in both locations, the files under `~/.workload_automation/dependencies` will be used in favour of those installed with WA.

Recording

WA features a `record` command that will automatically deploy and start revent on the target device.

If you want to simply record a single recording on the device then the following command can be used which will save the recording in the current directory:

```
wa record
```

There is one mandatory stage called 'run' and 3 optional stages: 'setup', 'extract_results' and 'teardown' which are used for playback of a workload. The different stages are distinguished by the suffix in the recording file path. In order to facilitate in creating these recordings you can specify `--setup`, `--extract-results`, `--teardown` or `--all` to indicate which stages you would like to create recordings for and the appropriate file name will be generated.

[0] It's not just about screen resolution – the event codes may be different even if devices use the same screen.

You can also directly specify a workload to create recordings for and WA will walk you through the relevant steps. For example if we waned to create recordings for the Angrybirds Rio workload we can specify the `workload` flag with `-w`. And in this case WA can be used to automatically deploy and launch the workload and record `setup` (`-s`) , `run` (`-r`) and `teardown` (`-t`) stages for the workload. In order to do this we would use the following command with an example output shown below:

```
wa record -srt -w angrybirds_rio
```

```
INFO     Setting up target
INFO     Deploying angrybirds_rio
INFO     Press Enter when you are ready to record SETUP...
[Pressed Enter]
INFO     Press Enter when you have finished recording SETUP...
[Pressed Enter]
INFO     Pulling '<device_model>setup.revent' from device
INFO     Press Enter when you are ready to record RUN...
[Pressed Enter]
INFO     Press Enter when you have finished recording RUN...
[Pressed Enter]
INFO     Pulling '<device_model>.run.revent' from device
INFO     Press Enter when you are ready to record TEARDOWN...
[Pressed Enter]
INFO     Press Enter when you have finished recording TEARDOWN...
[Pressed Enter]
INFO     Pulling '<device_model>.teardown.revent' from device
INFO     Tearing down angrybirds_rio
INFO     Recording(s) are available at: '$WA_USER_DIRECTORY/dependencies/angrybirds_
↪rio/revent_files'
```

Once you have made your desired recordings, you can either manually playback individual recordings using the *replay* command or, with the recordings in the appropriate dependencies location, simply run the workload using the *run* command and then all the available recordings will be played back automatically.

For more information on available arguments please see the *Record* command.

> **Note:** By default revent recordings are not portable across devices and therefore will require recording for each new device you wish to use the workload on. Alternatively a "gamepad" recording mode is also supported. This mode requires a gamepad to be connected to the device when recording but the recordings produced in this mode should be portable across devices.

Replaying

If you want to replay a single recorded file, you can use `wa replay` providing it with the file you want to replay. An example of the command output is shown below:

```
wa replay my_recording.revent
INFO     Setting up target
INFO     Pushing file to target
INFO     Starting replay
INFO     Finished replay
```

If you are using a device that supports android you can optionally specify a package name to launch before replaying the recording.

If you have recorded the required files for your workload and have placed the in the appropriate location (or specified the workload during recording) then you can simply run the relevant workload and your recordings will be replayed at the appropriate times automatically.

For more information run please read *Replay*

Revent vs UiAutomator

In general, Android UI Automator is the preferred way of automating user input for Android workloads because, unlike revent, UI Automator does not depend on a particular screen resolution, and so is more portable across different devices. It also gives better control and can potentially be faster for doing UI manipulations, as input events are scripted based on the available UI elements, rather than generated by human input.

On the other hand, revent can be used to manipulate pretty much any workload, where as UI Automator only works for Android UI elements (such as text boxes or radio buttons), which makes the latter useless for things like games. Recording revent sequence is also faster than writing automation code (on the other hand, one would need maintain a different revent log for each screen resolution).

Note: For ChromeOS targets, UI Automator can only be used with android applications and not the ChomeOS host applications themselves.

2.1.4 User Reference

Contents

Configuration

Agenda

An agenda can be thought of as a way to define an experiment as it specifies what is to be done during a Workload Automation run. This includes which workloads will be run, with what configuration and which augmentations will be enabled, etc. Agenda syntax is designed to be both succinct and expressive and is written using YAML notation.

There are three valid top level entries which are: *config*, *workloads*, *sections*.

An example agenda can be seen here:

```
config:                     # General configuration for the run
    user_directory: ~/.workload_automation/
    default_output_directory: 'wa_output'
    augmentations:          # A list of all augmentations to be enabled and disabled.
    - trace-cmd
    - csv
    - ~dmesg                # Disable the dmseg augmentation

    iterations: 1           # How many iterations to run each workload by default

    device: generic_android
    device_config:
        device: R32C801B8XY # Th adb name of our device we want to run on
        disable_selinux: true
        load_default_modules: true
        package_data_directory: /data/data

    trace-cmd:              # Provide config for the trace-cmd augmentation.
        buffer_size_step: 1000
        events:
        - sched*
        - irq*
        - power*
        - thermal*
        no_install: false
        report: true
        report_on_target: false
    csv:                    # Provide config for the csv augmentation
        use_all_classifiers: true

sections:                   # Configure what sections we want and their settings
    - id: LITTLES           # Run workloads just on the LITTLE cores
      runtime_parameters:   # Supply RT parameters to be used for this section
            num_little_cores: 4
            num_big_cores: 0

    - id: BIGS               # Run workloads just on the big cores
      runtime_parameters:    # Supply RT parameters to be used for this section
```

(continues on next page)

(continued from previous page)

```
            num_big_cores: 4
            num_little_cores: 0

workloads:                  # List which workloads should be run
-   name: benchmarkpi
    augmentations:
        - ~trace-cmd        # Disable the trace-cmd instrument for this workload
    iterations: 2           # Override the global number of iteration for this␣
↪workload
    params:                 # Specify workload parameters for this workload
        cleanup_assets: true
        exact_abi: false
        force_install: false
        install_timeout: 300
        markers_enabled: false
        prefer_host_package: true
        strict: false
        uninstall: false
-   dhrystone               # Run the dhrystone workload with all default config
```

This agenda will result in a total of 6 jobs being executed on our Android device, 4 of which running the BenchmarkPi workload with its customized workload parameters and 2 running dhrystone with its default configuration. The first 3 will be running on only the little cores and the latter running on the big cores. For all of the jobs executed the output will be processed by the `csv` processor,(plus any additional processors enabled in the default config file), however trace data will only be collected for the dhrystone jobs.

config

This section is used to provide overall configuration for WA and its run. The `config` section of an agenda will be merged with any other configuration files provided (including the default config file) and merged with the most specific configuration taking precedence (see *Config Merging* for more information. The only restriction is that `run_name` can only be specified in the config section of an agenda as this would not make sense to set as a default.

Within this section there are multiple distinct types of configuration that can be provided. However in addition to the options listed here all configuration that is available for *sections* can also be entered here and will be globally applied.

Configuration

The first is to configure the behaviour of WA and how a run as a whole will behave. The most common options that that you may want to specify are:

device The name of the 'device' that you wish to perform the run on. This name is a combination of a devlib Platform and Target. To see the available options please use `wa list targets`.

device_config The is a dict mapping allowing you to configure which target to connect to (e.g. `host` for an SSH connection or `device` to specific an ADB name) as well as configure other options for the device for example the `working_directory` or the list of `modules` to be loaded onto the device.

execution_order Defines the order in which the agenda spec will be executed.

reboot_policy Defines when during execution of a run a Device will be rebooted.

max_retries The maximum number of times failed jobs will be retried before giving up.

allow_phone_home Prevent running any workloads that are marked with 'phones_home'.

For more information and a full list of these configuration options please see *Run Configuration* and *"Meta Configuration"*.

Plugins

augmentations Specify a list of which augmentations should be enabled (or if prefixed with a ~, disabled).

Note: While augmentations can be enabled and disabled on a per workload basis, they cannot yet be re-configured part way through a run and the configuration provided as part of an agenda config section or separate config file will be used for all jobs in a WA run.

<plugin_name> You can also use this section to supply configuration for specific plugins, such as augmentations, workloads, resource getters etc. To do this the plugin name you wish to configure should be provided as an entry in this section and should contain a mapping of configuration options to their desired settings. If configuration is supplied for a plugin that is not currently enabled then it will simply be ignored. This allows for plugins to be temporarily removed without also having to remove their configuration, or to provide a set of defaults for a plugin which can then be overridden.

<global_alias> Some plugins provide global aliases which can set one or more configuration options at once, and these can also be specified here. For example if you specify a value for the entry `remote_assets_url` this will set the URL the http resource getter will use when searching for any missing assets.

---

workloads

Here you can specify a list of workloads to be run. If you wish to run a workload with all default values then you can specify the workload name directly as an entry, otherwise a dict mapping should be provided. Any settings provided here will be the most specific and therefore override any other more generalised configuration for that particular workload spec. The valid entries are as follows:

workload_name (Mandatory) The name of the workload to be run

iterations Specify how many iterations the workload should be run

label Similar to IDs but do not have the uniqueness restriction. If specified, labels will be used by some output processors instead of (or in addition to) the workload name. For example, the csv output processor will put the label in the "workload" column of the CSV file.

augmentations The instruments and output processors to enable (or disabled using a ~) during this workload.

classifiers Classifiers allow you to tag metrics from this workload spec which are often used to help identify what runtime parameters were used when post processing results.

workload_parameters Any parameters to configure that particular workload in a dict form.

Alias: `workload_params`

Note: You can see available parameters for a given workload with the *show command* or look it up in the *Plugin Reference*.

runtime_parameters A dict mapping of any runtime parameters that should be set for the device for that particular workload. For available parameters please see *runtime parameters*.

Alias: `runtime_parms`

Note: Unless specified elsewhere these configurations will not be undone once the workload has finished. I.e. if the frequency of a core is changed it will remain at that frequency until otherwise changed.

Note: There is also a shorter `params` alias available, however this alias will be interpreted differently depending on whether it is used in workload entry, in which case it will be interpreted as `workload_params`, or at global config or section (see below) level, in which case it will be interpreted as `runtime_params`.

sections

Sections are used for for grouping sets of configuration together in order to reduce the need for duplicated configuration (for more information please see *Sections*). Each section specified will be applied for each entry in the `workloads` section. The valid configuration entries are the same as the `"workloads"` section as mentioned above, except you can additionally specify:

workloads An entry which can be provided with the same configuration entries as the *workloads* top level entry.

Run Configuration

In addition to specifying run execution parameters through an agenda, the behaviour of WA can be modified through configuration file(s). The default configuration file is `~/.workload_automation/config.yaml` (the location can be changed by setting `WA_USER_DIRECTORY` environment variable, see *Environment Variables* section below). This file will be created when you first run WA if it does not already exist. This file must always exist and will always be loaded. You can add to or override the contents of that file on invocation of Workload Automation by specifying an additional configuration file using `--config` option. Variables with specific names will be picked up by the framework and used to modify the behaviour of Workload automation e.g. the `iterations` variable might be specified to tell WA how many times to run each workload.

execution_order: type: `'str'`

Defines the order in which the agenda spec will be executed. At the moment, the following execution orders are supported:

"by_iteration" The first iteration of each workload spec is executed one after the other, so all workloads are executed before proceeding on to the second iteration. E.g. A1 B1 C1 A2 C2 A3. This is the default if no order is explicitly specified.

In case of multiple sections, this will spread them out, such that specs from the same section are further part. E.g. given sections X and Y, global specs A and B, and two iterations, this will run

```
X.A1, Y.A1, X.B1, Y.B1, X.A2, Y.A2, X.B2, Y.B2
```

"by_section" Same as "by_iteration", however this will group specs from the same section together, so given sections X and Y, global specs A and B, and two iterations, this will run

```
X.A1, X.B1, Y.A1, Y.B1, X.A2, X.B2, Y.A2, Y.B2
```

"by_workload" All iterations of the first spec are executed before moving on to the next spec. E.g:

```
X.A1, X.A2, Y.A1, Y.A2, X.B1, X.B2, Y.B1, Y.B2
```

"random" Execution order is entirely random.

allowed values: 'by_iteration', 'by_section', 'by_workload', 'random'

default: 'by_iteration'

reboot_policy: type: 'RebootPolicy'

This defines when during execution of a run the Device will be rebooted. The possible values are:

"as_needed" The device will only be rebooted if the need arises (e.g. if it becomes unresponsive.

"never" The device will never be rebooted.

"initial" The device will be rebooted when the execution first starts, just before executing the first workload spec.

"each_job" The device will be rebooted before each new job.

"each_spec" The device will be rebooted before running a new workload spec.

Note: this acts the same as each_job when execution order is set to by_iteration

---

allowed values: 'never', 'as_needed', 'initial', 'each_spec', 'each_job'

default: 'as_needed'

device: type: 'str'

This setting defines what specific Device subclass will be used to interact the connected device. Obviously, this must match your setup.

default: 'generic_android'

retry_on_status: type: 'list_of_Enums'

This is list of statuses on which a job will be considered to have failed and will be automatically retried up to max_retries times. This defaults to ["FAILED", "PARTIAL"] if not set. Possible values are:

"OK" This iteration has completed and no errors have been detected

"PARTIAL" One or more instruments have failed (the iteration may still be running).

"FAILED" The workload itself has failed.

"ABORTED" The user interrupted the workload.

allowed values: `RUNNING`, `OK`, `PARTIAL`, `FAILED`, `ABORTED`, `SKIPPED`

default: `['FAILED', 'PARTIAL']`

max_retries: type: `'integer'`

The maximum number of times failed jobs will be retried before giving up.

Note: This number does not include the original attempt

---

default: `2`

bail_on_init_failure: type: `'boolean'`

When jobs fail during their main setup and run phases, WA will continue attempting to run the remaining jobs. However, by default, if they fail during their early initialization phase, the entire run will end without continuing to run jobs. Setting this to `False` means that WA will instead skip all the jobs from the job spec that failed, but continue attempting to run others.

default: `True`

allow_phone_home: type: `'boolean'`

Setting this to `False` prevents running any workloads that are marked with 'phones_home', meaning they are at risk of exposing information about the device to the outside world. For example, some benchmark applications upload device data to a database owned by the maintainers.

This can be used to minimise the risk of accidentally running such workloads when testing confidential devices.

default: `True`

Meta Configuration

There are also a couple of settings are used to provide additional metadata for a run. These may get picked up by instruments or output processors to attach context to results.

user_directory: type: `'expanded_path'`

Path to the user directory. This is the location WA will look for user configuration, additional plugins and plugin dependencies.

default: `'~/.workload_automation'`

assets_repository: type: `'str'`

The local mount point for the filer hosting WA assets.

logging: type: `'LoggingConfig'`

WA logging configuration. This should be a dict with a subset of the following keys:

```
:normal_format: Logging format used for console output
:verbose_format: Logging format used for verbose console output
:file_format: Logging format used for run.log
:color: If ``True`` (the default), console logging output will
        contain bash color escape codes. Set this to ``False`` if
        console output will be piped somewhere that does not know
        how to handle those.
```

default:

```
{
    color: True,
    verbose_format: %(asctime)s %(levelname)-8s %(name)s: %(message)s,
    regular_format: %(levelname)-8s %(message)s,
    file_format: %(asctime)s %(levelname)-8s %(name)s: %(message)s
}
```

verbosity: type: `'integer'`

Verbosity of console output.

default_output_directory: type: `'str'`

The default output directory that will be created if not specified when invoking a run.

default: `'wa_output'`

extra_plugin_paths: type: `'list_of_strs'`

A list of additional paths to scan for plugins.

Environment Variables

In addition to standard configuration described above, WA behaviour can be altered through environment variables. These can determine where WA looks for various assets when it starts.

WA_USER_DIRECTORY This is the location WA will look for config.yaml, plugins, dependencies, and it will also be used for local caches, etc. If this variable is not set, the default location is `~/.workload_automation` (this is created when WA is installed).

Note: This location **must** be writable by the user who runs WA.

WA_LOG_BUFFER_CAPACITY Specifies the capacity (in log records) for the early log handler which is used to buffer log records until a log file becomes available. If the is not set, the default value of `1000` will be used. This should sufficient for most scenarios, however this may need to be increased, e.g. if plugin loader scans a very large number of locations; this may also be set to a lower value to reduce WA's memory footprint on memory-constrained hosts.

Runtime Parameters

Contents

- *Setting Sysfiles*

Runtime parameters are options that can be specified to automatically configure device at runtime. They can be specified at the global level in the agenda or for individual workloads.

Example

Say we want to perform an experiment on an Android big.LITTLE devices to compare the power consumption between the big and LITTLE clusters running the dhrystone and benchmarkpi workloads. Assuming we have additional instrumentation active for this device that can measure the power the device is consuming, to reduce external factors we want to ensure that the device is in airplane mode turned on for all our tests and the screen is off only for our dhrystone run. We will then run 2 *sections* will each enable a single cluster on the device, set the cores to their maximum frequency and disable all available idle states.

```
config:
    runtime_parameters:
          airplane_mode: true
#..
workloads:
        - name: dhrystone
          iterations: 1
          runtime_parameters:
                screen_on: false
        - name: benchmarkpi
          iterations: 1
sections:
        - id: LITTLES
          runtime_parameters:
                num_little_cores: 4
                little_governor: userspace
                little_frequency: max
                little_idle_states: none
                num_big_cores: 0

        - id: BIGS
          runtime_parameters:
                num_big_cores: 4
                big_governor: userspace
                big_frequency: max
                big_idle_states: none
                num_little_cores: 0
```

HotPlug

Parameters:

num_cores An `int` that specifies the total number of cpu cores to be online.

num_<core_name>_cores An `int` that specifies the total number of that particular core to be online, the target will be queried and if the core_names can be determine a parameter for each of the unique core names will be available.

cpu<core_no>_online A `boolean` that specifies whether that particular cpu, e.g. cpu0 will be online.

If big.LITTLE is detected for the device and additional 2 parameters are available:

num_big_cores An `int` that specifies the total number of *big* cpu cores to be online.

num_little_cores An `int` that specifies the total number of *little* cpu cores to be online.

Note: Please note that if the device in question is operating its own dynamic hotplugging then WA may be unable to set the CPU state or will be overridden. Unfortunately the method of disabling dynamic hot plugging will vary from device to device.

CPUFreq

frequency An `int` that can be used to specify a frequency for all cores if there are common frequencies available.

Note: When settings the frequency, if the governor is not set to userspace then WA will attempt to set the maximum and minimum frequencies to mimic the desired behaviour.

---

max_frequency An `int` that can be used to specify a maximum frequency for all cores if there are common frequencies available.

min_frequency An `int` that can be used to specify a minimum frequency for all cores if there are common frequencies available.

governor A `string` that can be used to specify the governor for all cores if there are common governors available.

governor A `string` that can be used to specify the governor for all cores if there are common governors available.

gov_tunables A `dict` that can be used to specify governor tunables for all cores, unlike the other common parameters these are not validated at the beginning of the run therefore incorrect values will cause an error during runtime.

<core_name>_frequency An `int` that can be used to specify a frequency for cores of a particular type e.g. 'A72'.

<core_name>_max_frequency An `int` that can be used to specify a maximum frequency for cores of a particular type e.g. 'A72'.

<core_name>_min_frequency An `int` that can be used to specify a minimum frequency for cores of a particular type e.g. 'A72'.

<core_name>_governor A `string` that can be used to specify the governor for cores of a particular type e.g. 'A72'.

<core_name>_governor A `string` that can be used to specify the governor for cores of a particular type e.g. 'A72'.

<core_name>_gov_tunables A `dict` that can be used to specify governor tunables for cores of a particular type e.g. 'A72', these are not validated at the beginning of the run therefore incorrect values will cause an error during runtime.

cpu<no>_frequency An `int` that can be used to specify a frequency for a particular core e.g. 'cpu0'.

cpu<no>_max_frequency An `int` that can be used to specify a maximum frequency for a particular core e.g. 'cpu0'.

cpu<no>_min_frequency An `int` that can be used to specify a minimum frequency for a particular core e.g. 'cpu0'.

cpu<no>_governor A `string` that can be used to specify the governor for a particular core e.g. 'cpu0'.

cpu<no>_governor A `string` that can be used to specify the governor for a particular core e.g. 'cpu0'.

cpu<no>_gov_tunables A `dict` that can be used to specify governor tunables for a particular core e.g. 'cpu0', these are not validated at the beginning of the run therefore incorrect values will cause an error during runtime.

If big.LITTLE is detected for the device an additional set of parameters are available:

big_frequency An `int` that can be used to specify a frequency for the big cores.

big_max_frequency An `int` that can be used to specify a maximum frequency for the big cores.

big_min_frequency An `int` that can be used to specify a minimum frequency for the big cores.

big_governor A `string` that can be used to specify the governor for the big cores.

big_governor A `string` that can be used to specify the governor for the big cores.

big_gov_tunables A `dict` that can be used to specify governor tunables for the big cores, these are not validated at the beginning of the run therefore incorrect values will cause an error during runtime.

little_frequency An `int` that can be used to specify a frequency for the little cores.

little_max_frequency An `int` that can be used to specify a maximum frequency for the little cores.

little_min_frequency An `int` that can be used to specify a minimum frequency for the little cores.

little_governor A `string` that can be used to specify the governor for the little cores.

little_governor A `string` that can be used to specify the governor for the little cores.

little_gov_tunables A `dict` that can be used to specify governor tunables for the little cores, these are not validated at the beginning of the run therefore incorrect values will cause an error during runtime.

CPUIdle

idle_states A `string` or list of strings which can be used to specify what idles states should be enabled for all cores if there are common idle states available. 'all' and 'none' are also valid entries as a shorthand

<core_name>_idle_states A `string` or list of strings which can be used to specify what idles states should be enabled for cores of a particular type e.g. 'A72'. 'all' and 'none' are also valid entries as a shorthand

cpu<no>_idle_states A `string` or list of strings which can be used to specify what idles states should be enabled for a particular core e.g. 'cpu0'. 'all' and 'none' are also valid entries as a shorthand

If big.LITTLE is detected for the device and additional set of parameters are available:

big_idle_states A `string` or list of strings which can be used to specify what idles states should be enabled for the big cores. 'all' and 'none' are also valid entries as a shorthand

little_idle_states A `string` or list of strings which can be used to specify what idles states should be enabled for the little cores. 'all' and 'none' are also valid entries as a shorthand.

Android Specific Runtime Parameters

brightness An `int` between 0 and 255 (inclusive) to specify the brightness the screen should be set to. Defaults to `127`.

airplane_mode A `boolean` to specify whether airplane mode should be enabled for the device.

rotation A `String` to specify the screen orientation for the device. Valid entries are `NATURAL`, `LEFT`, `INVERTED`, `RIGHT`.

screen_on A `boolean` to specify whether the devices screen should be turned on. Defaults to `True`.

Setting Sysfiles

In order to perform additional configuration of a target the `sysfile_values` runtime parameter can be used. The value for this parameter is a mapping (an associative array, in YAML) of file paths onto values that should be written into those files. `sysfile_values` is the only runtime parameter that is available for any (Linux) device. Other runtime parameters will depend on the specifics of the device used (e.g. its CPU cores configuration) as detailed above.

Note: By default WA will attempt to verify that the any sysfile values were written correctly by reading the node back and comparing the two values. If you do not wish this check to happen, for example the node you are writing to is write only, you can append an `!` to the file path to disable this verification.

For example the following configuration could be used to enable and verify that cpu0 is online, however will not attempt to check that its governor have been set to userspace:

```
- name: dhrystone
runtime_params:
      sysfile_values:
            /sys/devices/system/cpu/cpu0/online: 1
            /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor!: userspace
```

Configuration Merging

WA configuration can come from various sources of increasing priority, as well as being specified in a generic and specific manner. For example WA's global config file would be considered the least specific vs the parameters of a workload in an agenda which would be the most specific. WA has two rules for the priority of configuration:

- Configuration from higher priority sources overrides configuration from lower priority sources.
- More specific configuration overrides less specific configuration.

There is a situation where these two rules come into conflict. When a generic configuration is given in config source of high priority and a specific configuration is given in a config source of lower priority. In this situation it is not possible to know the end users intention and WA will error.

This functionality allows for defaults for plugins, targets etc. to be configured at a global level and then seamless overridden without the need to remove the high level configuration.

Dependent on specificity, configuration parameters from different sources will have different inherent priorities. Within an agenda, the configuration in "workload" entries will be more specific than "sections" entries, which in turn are more specific than parameters in the "config" entry.

Configuration Includes

It is possible to include other files in your config files and agendas. This is done by specifying `include#` (note the trailing hash) as a key in one of the mappings, with the value being the path to the file to be included. The path must be either absolute, or relative to the location of the file it is being included from (*not* to the current working directory). The path may also include ~ to indicate current user's home directory.

The include is performed by removing the `include#` loading the contents of the specified into the mapping that contained it. In cases where the mapping already contains the key to be loaded, values will be merged using the usual merge method (for overwrites, values in the mapping take precedence over those from the included files).

Below is an example of an agenda that includes other files. The assumption is that all of those files are in one directory

```
# agenda.yaml
config:
   augmentations: [trace-cmd]
   include#: ./my-config.yaml
sections:
   - include#: ./section1.yaml
   - include#: ./section2.yaml
include#: ./workloads.yaml
```

```
# my-config.yaml
augmentations: [cpufreq]
```

```
# section1.yaml
runtime_parameters:
   frequency: max
```

```
# section2.yaml
runtime_parameters:
   frequency: min
```

```
# workloads.yaml
workloads:
   - dhrystone
   - memcpy
```

The above is equivalent to having a single file like this:

```
# agenda.yaml
config:
   augmentations: [cpufreq, trace-cmd]
sections:
   - runtime_parameters:
       frequency: max
   - runtime_parameters:
       frequency: min
workloads:
   - dhrystone
   - memcpy
```

Some additional details about the implementation and its limitations:

- The `include#` *must* be a key in a mapping, and the contents of the included file *must* be a mapping as well; it is not possible to include a list (e.g. in the examples above `workload:` part *must* be in the included file.
- Being a key in a mapping, there can only be one `include#` entry per block.

- The included file *must* have a `.yaml` extension.
- Nested inclusions *are* allowed. I.e. included files may themselves include files; in such cases the included paths must be relative to *that* file, and not the "main" file.

Commands

Installing the wa package will add `wa` command to your system, which you can run from anywhere. This has a number of sub-commands, which can be viewed by executing

```
wa -h
```

Individual sub-commands are discussed in detail below.

Run

The most common sub-command you will use is `run`. This will run the specified workload(s) and process its resulting output. This takes a single mandatory argument which specifies what you want WA to run. This could be either a workload name, or a path to an agenda" file that allows to specify multiple workloads as well as a lot additional configuration (see *Defining Experiments With an Agenda* section for details). Executing

```
wa run -h
```

Will display help for this subcommand that will look something like this:

```
usage: wa run [-h] [-c CONFIG] [-v] [--version] [-d DIR] [-f] [-i ID]
      [--disable INSTRUMENT]
      AGENDA

Execute automated workloads on a remote device and process the resulting
output.

positional arguments:
  AGENDA                Agenda for this workload automation run. This defines
                        which workloads will be executed, how many times, with
                        which tunables, etc. See example agendas in
                        /usr/local/lib/python2.7/dist-packages/wa for an
                        example of how this file should be structured.

optional arguments:
  -h, --help            show this help message and exit
  -c CONFIG, --config CONFIG
                        specify an additional config.yaml
  -v, --verbose         The scripts will produce verbose output.
  --version             show program's version number and exit
  -d DIR, --output-directory DIR
                        Specify a directory where the output will be
                        generated. If the directory already exists, the script
                        will abort unless -f option (see below) is used, in
                        which case the contents of the directory will be
                        overwritten. If this option is not specified, then
                        wa_output will be used instead.
  -f, --force           Overwrite output directory if it exists. By default,
```

(continues on next page)

(continued from previous page)

```
                        the script will abort in this situation to prevent
                        accidental data loss.
  -i ID, --id ID        Specify a workload spec ID from an agenda to run. If
                        this is specified, only that particular spec will be
                        run, and other workloads in the agenda will be
                        ignored. This option may be used to specify multiple
                        IDs.
  --disable INSTRUMENT  Specify an instrument or output processor to disable
                        from the command line. This equivalent to adding
                        "~{metavar}" to the instruments list in the
                        agenda. This can be used to temporarily disable a
                        troublesome instrument for a particular run without
                        introducing permanent change to the config (which one
                        might then forget to revert). This option may be
                        specified multiple times.
```

List

This lists all plugins of a particular type. For example

```
wa list instruments
```

will list all instruments currently included in WA. The list will consist of plugin names and short descriptions of the functionality they offer e.g.

```
#..
           cpufreq:    Collects dynamic frequency (DVFS) settings before and after
                       workload execution.
             dmesg:    Collected dmesg output before and during the run.
energy_measurement:    This instrument is designed to be used as an interface to
                       the various energy measurement instruments located
                       in devlib.
    execution_time:    Measure how long it took to execute the run() methods of
                       a Workload.
       file_poller:    Polls the given files at a set sample interval. The values
                       are output in CSV format.
               fps:    Measures Frames Per Second (FPS) and associated metrics for
                       a workload.
#..
```

You can use the same syntax to quickly display information about `commands`, `energy_instrument_backends`, `instruments`, `output_processors`, `resource_getters`, `targets` and `workloads`

Show

This will show detailed information about an plugin (workloads, targets, instruments etc.), including a full description and any relevant parameters/configuration that are available. For example executing

```
wa show benchmarkpi
```

will produce something like:

```
benchmarkpi
-----------

Measures the time the target device takes to run and complete the Pi
calculation algorithm.

http://androidbenchmark.com/howitworks.php

from the website:

The whole idea behind this application is to use the same Pi calculation
algorithm on every Android Device and check how fast that process is.
Better calculation times, conclude to faster Android devices. This way you
can also check how lightweight your custom made Android build is. Or not.

As Pi is an irrational number, Benchmark Pi does not calculate the actual Pi
number, but an approximation near the first digits of Pi over the same
calculation circles the algorithms needs.

So, the number you are getting in milliseconds is the time your mobile device
takes to run and complete the Pi calculation algorithm resulting in a
approximation of the first Pi digits.

parameters
~~~~~~~~~~

cleanup_assets : boolean
    If ``True``, if assets are deployed as part of the workload they
    will be removed again from the device as part of finalize.

    default: ``True``

package_name : str
    The package name that can be used to specify
    the workload apk to use.

install_timeout : integer
    Timeout for the installation of the apk.

    constraint: ``value > 0``

    default: ``300``

version : str
    The version of the package to be used.

variant : str
    The variant of the package to be used.

strict : boolean
    Whether to throw an error if the specified package cannot be found
    on host.

force_install : boolean
    Always re-install the APK, even if matching version is found already installed
    on the device.
```

(continues on next page)

(continued from previous page)

```
uninstall : boolean
    If ``True``, will uninstall workload's APK as part of teardown.'

exact_abi : boolean
    If ``True``, workload will check that the APK matches the target
    device ABI, otherwise any suitable APK found will be used.

markers_enabled : boolean
    If set to ``True``, workloads will insert markers into logs
    at various points during execution. These markers may be used
    by other plugins or post-processing scripts to provide
    measurements or statistics for specific parts of the workload
    execution.
```

Note: You can also use this command to view global settings by using `wa show settings`

Create

This aids in the creation of new WA-related objects for example agendas and workloads. For more detailed information on creating workloads please see the *adding a workload* section for more details.

As an example to create an agenda that will run the dhrystone and memcpy workloads that will use the status and hwmon augmentations, run each test 3 times and save into the file `my_agenda.yaml` the following command can be used:

```
wa create agenda dhrystone memcpy status hwmon -i 3 -o my_agenda.yaml
```

Which will produce something like:

```
config:
    augmentations:
    - status
    - hwmon
    status: {}
    hwmon: {}
    iterations: 3
workloads:
-   name: dhrystone
    params:
        cleanup_assets: true
        delay: 0
        duration: 0
        mloops: 0
        taskset_mask: 0
        threads: 4
-   name: memcpy
    params:
        buffer_size: 5242880
        cleanup_assets: true
        cpus: null
        iterations: 1000
```

This will be populated with default values which can then be customised for the particular use case.

Additionally the create command can be used to initialize (and update) a Postgres database which can be used by the `postgres` output processor.

The most of database connection parameters have a default value however they can be overridden via command line arguments. When initializing the database WA will also save the supplied parameters into the default user config file so that they do not need to be specified time the output processor is used.

As an example if we had a database server running on at 10.0.0.2 using the standard port we could use the following command to initialize a database for use with WA:

```
wa create database -a 10.0.0.2 -u my_username -p Pa55w0rd
```

This will log into the database server with the supplied credentials and create a database (defaulting to 'wa') and will save the configuration to the `~/.workload_automation/config.yaml` file.

With updates to WA there may be changes to the database schema used. In this case the create command can also be used with the `-U` flag to update the database to use the new schema as follows:

```
wa create database -a 10.0.0.2 -u my_username -p Pa55w0rd -U
```

This will upgrade the database sequentially until the database schema is using the latest version.

Process

This command allows for output processors to be ran on data that was produced by a previous run.

There are 2 ways of specifying which processors you wish to use, either passing them directly as arguments to the process command with the `--processor` argument or by providing an additional config file with the `--config` argument. Please note that by default the process command will not rerun processors that have already been ran during the run, in order to force a rerun of the processors you can specific the `--force` argument.

Additionally if you have a directory containing multiple run directories you can specify the `--recursive` argument which will cause WA to walk the specified directory processing all the WA output sub-directories individually.

As an example if we had performed multiple experiments and have the various WA output directories in our `my_experiments` directory, and we now want to process the outputs with a tool that only supports CSV files. We can easily generate CSV files for all the runs contained in our directory using the CSV processor by using the following command:

```
wa process -r -p csv my_experiments
```

Record

This command simplifies the process of recording revent files. It will automatically deploy revent and has options to automatically open apps and record specified stages of a workload. Revent allows you to record raw inputs such as screen swipes or button presses. This can be useful for recording inputs for workloads such as games that don't have XML UI layouts that can be used with UIAutomator. As a drawback from this, revent recordings are specific to the device type they were recorded on. WA uses two parts to the names of revent recordings in the format, `{device_name}.{suffix}.revent`. - device_name can either be specified manually with the `-d` argument or it can be automatically determined. On Android device it will be obtained from `build.prop`, on Linux devices it is obtained from `/proc/device-tree/model`. - suffix is used by WA to determine which part of the app execution the recording is for, currently these are either `setup`, `run`, `extract_results` or `teardown`. All stages except `run` are optional for playback and to specify which stages should be recorded the `-s`, `-r`, `-e` or `-t` arguments respectively, or optionally `-a` to indicate all stages should be recorded.

The full set of options for this command are:

```
usage: wa record [-h] [-c CONFIG] [-v] [--version] [-d DEVICE] [-o FILE] [-s]
                 [-r] [-e] [-t] [-a] [-C] [-p PACKAGE | -w WORKLOAD]

optional arguments:
  -h, --help            show this help message and exit
  -c CONFIG, --config CONFIG
                        specify an additional config.yaml
  -v, --verbose         The scripts will produce verbose output.
  --version             show program's version number and exit
  -d DEVICE, --device DEVICE
                        Specify the device on which to run. This will take
                        precedence over the device (if any) specified in
                        configuration.
  -o FILE, --output FILE
                        Specify the output file
  -s, --setup           Record a recording for setup stage
  -r, --run             Record a recording for run stage
  -e, --extract_results Record a recording for extract_results stage
  -t, --teardown        Record a recording for teardown stage
  -a, --all             Record recordings for available stages
  -C, --clear           Clear app cache before launching it
  -p PACKAGE, --package PACKAGE
                        Android package to launch before recording
  -w WORKLOAD, --workload WORKLOAD
                        Name of a revent workload (mostly games)
```

For more information please see *Revent Recording*.

Replay

Alongside `record` wa also has a command to playback a single recorded revent file. It behaves similar to the `record` command taking a subset of the same options allowing you to automatically launch a package on the device

```
usage: wa replay [-h] [-c CONFIG] [-v] [--debug] [--version] [-p PACKAGE] [-C]
                 revent

positional arguments:
  revent                The name of the file to replay

optional arguments:
  -h, --help            show this help message and exit
  -c CONFIG, --config CONFIG
                        specify an additional config.py
  -v, --verbose         The scripts will produce verbose output.
  --debug               Enable debug mode. Note: this implies --verbose.
  --version             show program's version number and exit
  -p PACKAGE, --package PACKAGE
                        Package to launch before recording
  -C, --clear           Clear app cache before launching it
```

For more information please see *Revent Replaying*.

Output Directory Structure

This is an overview of WA output directory structure.

Note: In addition to files and subdirectories described here, other content may present in the output directory for a run, depending on the enabled augmentations.

Overview

The output directory will contain a subdirectory for every job iteration that was run, as well as some additional entries. The following diagram illustrates the typical structure of WA output directory:

```
wa_output/
├── __meta/
│   ├── config.json
│   ├── jobs.json
│   ├── raw_config
│   │   ├── cfg0-config.yaml
│   │   └── agenda.yaml
│   ├── run_info.json
│   └── target_info.json
├── __failed/
│   └── wk1-dhrystone-1-attempt1
├── wk1-dhrystone-1/
│   └── result.json
├── wk1-dhrystone-2/
│   └── result.json
├── wk2-memcpy-1/
│   └── result.json
├── wk2-memcpy-2/
│   └── result.json
├── result.json
└── run.log
```

This is the directory structure that would be generated after running two iterations each of `dhrystone` and `memcpy` workloads with no augmentations enabled, and with the first attempt at the first iteration of dhrystone having failed.

You may notice that a number of directories named `wk*-x-x` were generated in the output directory structure. Each of these directories represents a *job*. The name of the output directory is as stated *here*.

Output Directory Entries

result.json Contains a JSON structure describing the result of the execution, including collected metrics and artifacts. There will be one for each job execution, and one for the overall run. The run `result.json` will only contain metrics/artifacts for the run as a whole, and will not contain results for individual jobs.

You typically would not access `result.json` files directly. Instead you would either enable augmentations to format the results in easier to manage form (such as CSV table), or use *Output* to access the results from scripts.

run.log This is a log of everything that happened during the run, including all interactions with the target, and all the decisions made by the framework. The output is equivalent to what you would see on the console when running with `--verbose` option.

Note: WA source contains a syntax file for Vim that will color the initial part of each log line, in a similar way to what you see on the console. This may be useful for quickly spotting error and warning messages when scrolling through the log.

https://github.com/ARM-software/workload-automation/blob/next/extras/walog.vim

__meta This directory contains configuration and run metadata. See *Configuration and Metadata* below for details.

__failed This directory will only be present if one or more job executions has failed and were re-run. This directory contains output directories for the failed attempts.

job execution output subdirectory Each subdirectory will be named `<job id>_<workload label>_<iteration number>`, and will, at minimum, contain a `result.json` (see above). Additionally, it may contain raw output from the workload, and any additional artifacts (e.g. traces) generated by augmentations. Finally, if workload execution has failed, WA may gather some additional logging (such as the UI state at the time of failure) and place it here.

Configuration and Metadata

As stated above, the `__meta` directory contains run configuration and metadata. Typically, you would not access these files directly, but would use the *Output* to query the metadata.

For more details about WA configuration see *Configuration*.

config.json Contains the overall run configuration, such as target interface configuration, and job execution order, and various "meta-configuration" settings, such as default output path, verbosity level, and logging formatting.

jobs.json Final configuration for all jobs, including enabled augmentations, workload and runtime parameters, etc.

raw_config This directory contains copies of config file(s) and the agenda that were parsed in order to generate configuration for this run. Each config file is prefixed with `cfg<N>-`, where `<N>` is the number indicating the order (with respect to the other other config files) in which it was parsed, e.g. `cfg0-config.yaml` is always a copy of `$WA_USER_DIRECTORY/config.yaml`. The one file without a prefix is the agenda.

run_info.json Run metadata, e.g. duration, start/end timestamps and duration.

target_info.json Extensive information about the target. This includes information about the target's CPUS configuration, kernel and userspace versions, etc. The exact content will vary depending on the target type (Android vs Linux) and what could accessed on a particular device (e.g. if `/proc/config.gz` exists on the target, the kernel config will be included).

CHAPTER 3

Developer Information

This section contains more advanced topics, such how to write your own Plugins and detailed descriptions of how WA functions under the hood.

3.1 Developer Information

Contents

3.1.1 Developer Guide

Writing Plugins

Workload Automation offers several plugin points (or plugin types). The most interesting of these are

workloads These are the tasks that get executed and measured on the device. These can be benchmarks, high-level use cases, or pretty much anything else.

targets These are interfaces to the physical devices (development boards or end-user devices, such as smartphones) that use cases run on. Typically each model of a physical device would require its own interface class (though some functionality may be reused by subclassing from an existing base).

instruments Instruments allow collecting additional data from workload execution (e.g. system traces). Instruments are not specific to a particular workload. Instruments can hook into any stage of workload execution.

output processors These are used to format the results of workload execution once they have been collected. Depending on the callback used, these will run either after each iteration and/or at the end of the run, after all of the results have been collected.

You can create a plugin by subclassing the appropriate base class, defining appropriate methods and attributes, and putting the .py file containing the class into the "plugins" subdirectory under `~/.workload_automation` (or equivalent) where it will be automatically picked up by WA.

Plugin Basics

This sub-section covers things common to implementing plugins of all types. It is recommended you familiarize yourself with the information here before proceeding onto guidance for specific plugin types.

Dynamic Resource Resolution

The idea is to decouple resource identification from resource discovery. Workloads/instruments/devices/etc state *what* resources they need, and not *where* to look for them – this instead is left to the resource resolver that is part of the

execution context. The actual discovery of resources is performed by resource getters that are registered with the resolver.

A resource type is defined by a subclass of `wa.framework.resource.Resource`. An instance of this class describes a resource that is to be obtained. At minimum, a `Resource` instance has an owner (which is typically the object that is looking for the resource), but specific resource types may define other parameters that describe an instance of that resource (such as file names, URLs, etc).

An object looking for a resource invokes a resource resolver with an instance of `Resource` describing the resource it is after. The resolver goes through the getters registered for that resource type in priority order attempting to obtain the resource; once the resource is obtained, it is returned to the calling object. If none of the registered getters could find the resource, `NotFoundError` is raised (or `None` is returned instead, if invoked with `strict=False`).

The most common kind of object looking for resources is a `Workload`, and the `Workload` class defines `wa.framework.workload.Workload.init_resources()` method, which may be overridden by subclasses to perform resource resolution. For example, a workload looking for an executable file would do so like this:

```
from wa import Workload
from wa.import Executable

class MyBenchmark(Workload):

    # ...

    def init_resources(self, resolver):
        resource = Executable(self, self.target.abi, 'my_benchmark')
        host_exe = resolver.get(resource)

    # ...
```

Currently available resource types are defined in `wa.framework.resources`.

Deploying executables to a target

Some targets may have certain restrictions on where executable binaries may be placed and how they should be invoked. To ensure your plugin works with as wide a range of targets as possible, you should use WA APIs for deploying and invoking executables on a target, as outlined below.

As with other resources, host-side paths to the executable binary to be deployed should be obtained via the *resource resolver*. A special resource type, `Executable` is used to identify a binary to be deployed. This is similar to the regular `File` resource, however it takes an additional parameter that specifies the ABI for which the executable was compiled for.

In order for the binary to be obtained in this way, it must be stored in one of the locations scanned by the resource resolver in a directory structure `<root>/bin/<abi>/<binary>` (where `root` is the base resource location to be searched, e.g. `~/.workload_automation/dependencies/<plugin name>`, and `<abi>` is the ABI for which the executable has been compiled, as returned by `self.target.abi`).

Once the path to the host-side binary has been obtained, it may be deployed using one of two methods from a Target instance – `install` or `install_if_needed`. The latter will check a version of that binary has been previously deployed by WA and will not try to re-install.

```
from wa import Executable

host_binary = context.get(Executable(self, self.target.abi, 'some_binary'))
target_binary = self.target.install_if_needed(host_binary)
```

Note: Please also note that the check is done based solely on the binary name. For more information please see the devlib documentation.

Both of the above methods will return the path to the installed binary on the target. The executable should be invoked *only* via that path; do **not** assume that it will be in `PATH` on the target (or that the executable with the same name in `PATH` is the version deployed by WA.

For more information on how to implement this, please see the *how to guide*.

Deploying assets

WA provides a generic mechanism for deploying assets during workload initialization. WA will automatically try to retrieve and deploy each asset to the target's working directory that is contained in a workloads `deployable_assets` attribute stored as a list.

If the parameter `cleanup_assets` is set then any asset deployed will be removed again and the end of the run.

If the workload requires a custom deployment mechanism the `deploy_assets` method can be overridden for that particular workload, in which case, either additional assets should have their on target paths added to the workload's `deployed_assests` attribute or the corresponding `remove_assets` method should also be implemented.

Adding an Instrument

Instruments can be used to collect additional measurements during workload execution (e.g. collect power readings). An instrument can hook into almost any stage of workload execution. Any new instrument should be a subclass of Instrument and it must have a name. When a new instrument is added to Workload Automation, the methods of the new instrument will be found automatically and hooked up to the supported signals. Once a signal is broadcasted, the corresponding registered method is invoked.

Each method in `Instrument` must take two arguments, which are `self` and `context`. Supported methods and their corresponding signals can be found in the *Signals Documentation*. To make implementations easier and common, the basic steps to add new instrument is similar to the steps to add new workload and an example can be found in the *How To* section.

To implement your own instrument the relevant methods of the interface shown below should be implemented:

name The name of the instrument, this must be unique to WA.

description A description of what the instrument can be used for.

parameters A list of additional `Parameters` the instrument can take.

initialize(context) This method will only be called once during the workload run therefore operations that only need to be performed initially should be performed here for example pushing the files to the target device, installing them.

setup(context) This method is invoked after the workload is setup. All the necessary setup should go inside this method. Setup, includes operations like clearing logs, additional configuration etc.

start(context) It is invoked just before the workload start execution. Here is where instrument measurement start being registered/taken.

stop(context) It is invoked just after the workload execution stops and where the measurements should stop being taken/registered.

update_output(context) This method is invoked after the workload updated its result and where the taken measures should be added to the result so it can be processed by WA.

teardown(context) It is invoked after the workload is torn down. It is a good place to clean any logs generated by the instrument.

finalize(context) This method is the complement to the initialize method and will also only be called once so should be used to deleting/uninstalling files pushed to the device.

This is similar to a `Workload`, except all methods are optional. In addition to the workload-like methods, instruments can define a number of other methods that will get invoked at various points during run execution. The most useful of which is perhaps `initialize` that gets invoked after the device has been initialised for the first time, and can be used to perform one-time setup (e.g. copying files to the device – there is no point in doing that for each iteration). The full list of available methods can be found in *Signals Documentation*.

Prioritization

Callbacks (e.g. `setup()` methods) for all instruments get executed at the same point during workload execution, one after another. The order in which the callbacks get invoked should be considered arbitrary and should not be relied on (e.g. you cannot expect that just because instrument A is listed before instrument B in the config, instrument A's callbacks will run first).

In some cases (e.g. in `start()` and `stop()` methods), it is important to ensure that a particular instrument's callbacks run a closely as possible to the workload's invocations in order to maintain accuracy of readings; or, conversely, that a callback is executed after the others, because it takes a long time and may throw off the accuracy of other instruments. You can do this by using decorators on the appropriate methods. The available decorators are: `very_slow`, `slow`, `normal`, `fast`, `very_fast`, with `very_fast` running closest to the workload invocation and `very_slow` running furtherest away. For example:

```
from wa import very_fast
# ..

class PreciseInstrument(Instrument)

    # ...
    @very_fast
    def start(self, context):
        pass

    @very_fast
    def stop(self, context):
        pass

    # ...
```

`PreciseInstrument` will be started after all other instruments (i.e. *just* before the workload runs), and it will stopped before all other instruments (i.e. *just* after the workload runs).

If more than one active instrument has specified fast (or slow) callbacks, then their execution order with respect to each other is not guaranteed. In general, having a lot of instruments enabled is going to negatively affect the readings. The best way to ensure accuracy of measurements is to minimize the number of active instruments (perhaps doing several identical runs with different instruments enabled).

Example

Below is a simple instrument that measures the execution time of a workload:

```
class ExecutionTimeInstrument(Instrument):
    """
    Measure how long it took to execute the run() methods of a Workload.

    """

    name = 'execution_time'

    def initialize(self, context):
        self.start_time = None
        self.end_time = None

    @very_fast
    def start(self, context):
        self.start_time = time.time()

    @very_fast
    def stop(self, context):
        self.end_time = time.time()

    def update_output(self, context):
        execution_time = self.end_time - self.start_time
        context.add_metric('execution_time', execution_time, 'seconds')
```

Instrumentation Signal-Method Mapping

Instrument methods get automatically hooked up to signals based on their names. Mostly, the method name corresponds to the name of the signal, however there are a few convenience aliases defined (listed first) to make easier to relate instrumentation code to the workload execution model. For an overview on when these signals are dispatched during execution please see the *Developer Reference*.

method name	signal
initialize	run-initialized
setup	before-workload-setup
start	before-workload-execution
stop	after-workload-execution
process_workload_output	successful-workload-output-update
update_output	after-workload-output-update
teardown	after-workload-teardown
finalize	run-finalized
on_run_start	run-started
on_run_end	run-completed
on_job_start	job-started
on_job_restart	job-restarted
on_job_end	job-completed
on_job_failure	job-failed
on_job_abort	job-aborted
before_job_queue_execution	before-job-queue-execution
on_successful_job_queue_exection	successful-job-queue-execution
after_job_queue_execution	after-job-queue-execution
before_job	before-job
on_successful_job	successful-job
after_job	after-job
before_processing_job_output	before-job-output-processed
on_successfully_processing_job	successful-job-output-processed
after_processing_job_output	after-job-output-processed
before_reboot	before-reboot
on_successful_reboot	successful-reboot
after_reboot	after-reboot
on_error	error-logged
on_warning	warning-logged

The methods above may be decorated with on the listed decorators to set the priority of the Instrument method relative to other callbacks registered for the signal (within the same priority level, callbacks are invoked in the order they were registered). The table below shows the mapping of the decorator to the corresponding priority:

decorator	priority
extremely_low	-30
very_low	-20
low	-10
normal	0
high	10
very_high	20
extremely_high	30

Unresponsive Targets

If a target is believed to be unresponsive, instrument callbacks will be disabled to prevent a cascade of errors and potential corruptions of state, as it is generally assumed that instrument callbacks will want to do something with the target.

If your callback only does something with the host, and does not require an active target connection, you can decorate it with `@hostside` decorator to ensure it gets invoked even if the target becomes unresponsive.

Adding an Output processor

A output processor is responsible for processing the results. This may involve formatting and writing them to a file, uploading them to a database, generating plots, etc. WA comes with a few output processors that output results in a few common formats (such as csv or JSON).

You can add your own output processors by creating a Python file in `~/.workload_automation/plugins` with a class that derives from `wa.OutputProcessor`, and should implement the relevant methods shown below, for more information and please see the *Adding an Output Processor* section.

name The name of the output processor, this must be unique to WA.

description A description of what the output processor can be used for.

parameters A list of additional `Parameters` the output processor can take.

initialize(context) This method will only be called once during the workload run therefore operations that only need to be performed initially should be performed here.

process_job_output(output, target_info, run_ouput) This method should be used to perform the processing of the output from an individual job output. This is where any additional artifacts should be generated if applicable.

export_job_output(output, target_info, run_ouput) This method should be used to perform the exportation of the existing data collected/generated for an individual job. E.g. uploading them to a database etc.

process_run_output(output, target_info) This method should be used to perform the processing of the output from the run as a whole. This is where any additional artifacts should be generated if applicable.

export_run_output(output, target_info) This method should be used to perform the exportation of the existing data collected/generated for the run as a whole. E.g. uploading them to a database etc.

finalize(context) This method is the complement to the initialize method and will also only be called once.

The method names should be fairly self-explanatory. The difference between "process" and "export" methods is that export methods will be invoked after process methods for all output processors have been generated. Process methods may generate additional artifacts (metrics, files, etc.), while export methods should not – they should only handle existing results (upload them to a database, archive on a filer, etc).

The output object passed to job methods is an instance of `wa.framework.output.JobOutput`, the output object passed to run methods is an instance of `wa.RunOutput`.

Adding a Resource Getter

A resource getter is a plugin that is designed to retrieve a resource (binaries, APK files or additional workload assets). Resource getters are invoked in priority order until one returns the desired resource.

If you want WA to look for resources somewhere it doesn't by default (e.g. you have a repository of APK files), you can implement a getter for the resource and register it with a higher priority than the standard WA getters, so that it gets invoked first.

Instances of a resource getter should implement the following interface:

```
class ResourceGetter(Plugin):

    name = None

    def register(self, resolver):
        raise NotImplementedError()
```

The getter should define a name for itself (as with all plugins), in addition it should implement the `register` method. This involves registering a method with the resolver that should used to be called when trying to retrieve a resource (typically `get`) along with it's priority (see *Getter Prioritization* below. That method should return an instance of the resource that has been discovered (what "instance" means depends on the resource, e.g. it could be a file path), or `None` if this getter was unable to discover that resource.

Getter Prioritization

A priority is an integer with higher numeric values indicating a higher priority. The following standard priority aliases are defined for getters:

preferred Take this resource in favour of the environment resource.

local Found somewhere under ~/.workload_automation/ or equivalent, or from environment variables, external configuration files, etc. These will override resource supplied with the package.

lan Resource will be retrieved from a locally mounted remote location (such as samba share)

remote Resource will be downloaded from a remote location (such as an HTTP server)

package Resource provided with the package.

These priorities are defined as class members of `wa.framework.resource.SourcePriority`, e.g. `SourcePriority.preferred`.

Most getters in WA will be registered with either `local` or `package` priorities. So if you want your getter to override the default, it should typically be registered as `preferred`.

You don't have to stick to standard priority levels (though you should, unless there is a good reason). Any integer is a valid priority. The standard priorities range from 0 to 40 in increments of 10.

Example

The following is an implementation of a getter that searches for files in the users dependencies directory, typically `~/.workload_automation/dependencies/<workload_name>` It uses the `get_from_location` method to filter the available files in the provided directory appropriately:

```
import sys

from wa import settings,
from wa.framework.resource import ResourceGetter, SourcePriority
from wa.framework.getters import get_from_location
from wa.utils.misc import ensure_directory_exists as _d

class UserDirectory(ResourceGetter):

    name = 'user'
```

(continues on next page)

(continued from previous page)

```
    def register(self, resolver):
        resolver.register(self.get, SourcePriority.local)

    def get(self, resource):
        basepath = settings.dependencies_directory
        directory = _d(os.path.join(basepath, resource.owner.name))
        return get_from_location(directory, resource)
```

Adding a Target

In WA3, a 'target' consists of a platform and a devlib target. The implementations of the targets are located in `devlib`. WA3 will instantiate a devlib target passing relevant parameters parsed from the configuration. For more information about devlib targets please see the documentation.

The currently available platforms are:

generic The 'standard' platform implementation of the target, this should work for the majority of use cases.

juno A platform implementation specifically for the juno.

tc2 A platform implementation specifically for the tc2.

gem5 A platform implementation to interact with a gem5 simulation.

The currently available targets from devlib are:

linux A device running a Linux based OS.

android A device running Android OS.

local Used to run locally on a linux based host.

chromeos A device running ChromeOS, supporting an android container if available.

For an example of adding you own customized version of an existing devlib target, please see the how to section *Adding a Custom Target*.

Other Plugin Types

In addition to plugin types covered above, there are few other, more specialized ones. They will not be covered in as much detail. Most of them expose relatively simple interfaces with only a couple of methods and it is expected that if the need arises to extend them, the API-level documentation that accompanies them, in addition to what has been outlined here, should provide enough guidance.

commands This allows extending WA with additional sub-commands (to supplement exiting ones outlined in the *Commands* section).

modules Modules are "plugins for plugins". They can be loaded by other plugins to expand their functionality (for example, a flashing module maybe loaded by a device in order to support flashing).

Packaging Your Plugins

If your have written a bunch of plugins, and you want to make it easy to deploy them to new systems and/or to update them on existing systems, you can wrap them in a Python package. You can use `wa create package` command

to generate appropriate boiler plate. This will create a `setup.py` and a directory for your package that you can place your plugins into.

For example, if you have a workload inside `my_workload.py` and an output processor in `my_output_processor.py`, and you want to package them as `my_wa_exts` package, first run the create command

```
wa create package my_wa_exts
```

This will create a `my_wa_exts` directory which contains a `my_wa_exts/setup.py` and a subdirectory `my_wa_exts/my_wa_exts` which is the package directory for your plugins (you can rename the top-level `my_wa_exts` directory to anything you like – it's just a "container" for the setup.py and the package directory). Once you have that, you can then copy your plugins into the package directory, creating `my_wa_exts/my_wa_exts/my_workload.py` and `my_wa_exts/my_wa_exts/my_output_processor.py`. If you have a lot of plugins, you might want to organize them into subpackages, but only the top-level package directory is created by default, and it is OK to have everything in there.

Note: When discovering plugins through this mechanism, WA traverses the Python module/submodule tree, not the directory structure, therefore, if you are going to create subdirectories under the top level directory created for you, it is important that your make sure they are valid Python packages; i.e. each subdirectory must contain a __init__.py (even if blank) in order for the code in that directory and its subdirectories to be discoverable.

At this stage, you may want to edit `params` structure near the bottom of the `setup.py` to add correct author, license and contact information (see "Writing the Setup Script" section in standard Python documentation for details). You may also want to add a README and/or a COPYING file at the same level as the setup.py. Once you have the contents of your package sorted, you can generate the package by running

```
cd my_wa_exts
python setup.py sdist
```

This will generate `my_wa_exts/dist/my_wa_exts-0.0.1.tar.gz` package which can then be deployed on the target system with standard Python package management tools, e.g.

```
sudo pip install my_wa_exts-0.0.1.tar.gz
```

As part of the installation process, the setup.py in the package, will write the package's name into `~/.workoad_automation/packages`. This will tell WA that the package contains plugin and it will load them next time it runs.

Note: There are no uninstall hooks in `setuputils`, so if you ever uninstall your WA plugins package, you will have to manually remove it from `~/.workload_automation/packages` otherwise WA will complain about a missing package next time you try to run it.

3.1.2 How Tos

Contents

Deploying Executables

Installing binaries for a particular plugin should generally only be performed once during a run. This should typically be done in the `initialize` method, if the only functionality performed in the method is to install the required binaries then the `initialize` method should be decorated with the `@once` *decorator* otherwise this should be placed into a dedicated method which is decorated instead. Please note if doing this then any installed paths should be added as class attributes rather than instance variables. As a general rule if binaries are installed as part of `initialize` then they should be uninstalled in the complementary `finalize` method.

Part of an example workload demonstrating this is shown below:

```
class MyWorkload(Workload):
      #..
      @once
      def initialize(self, context):
          resource = Executable(self, self.target.abi, 'my_executable')
          host_binary = context.resolver.get(resource)
          MyWorkload.target_binary = self.target.install(host_binary)
      #..

      def setup(self, context):
          self.command = "{} -a -b -c".format(self.target_binary)
          self.target.execute(self.command)
      #..

      @once
      def finalize(self, context):
          self.target.uninstall('my_executable')
```

Adding a Workload

The easiest way to create a new workload is to use the *create* command. `wa create workload <args>`. This will use predefined templates to create a workload based on the options that are supplied to be used as a starting point

for the workload. For more information on using the create workload command see `wa create workload -h`

The first thing to decide is the type of workload you want to create depending on the OS you will be using and the aim of the workload. The are currently 6 available workload types to choose as detailed in the *Developer Reference*.

Once you have decided what type of workload you wish to choose this can be specified with `-k <workload_kind>` followed by the workload name. This will automatically generate a workload in the your `WA_CONFIG_DIR/plugins`. If you wish to specify a custom location this can be provided with `-p <path>`

Adding a Basic Workload

To add a basic workload you can simply use the command:

```
wa create workload basic
```

This will generate a very basic workload with dummy methods for the workload interface and it is left to the developer to add any required functionality to the workload.

Not all the methods are required to be implemented, this example shows how a subset might be used to implement a simple workload that times how long it takes to compress a file of a particular size on the device.

Note: This is intended as an example of how to implement the Workload *interface*. The methodology used to perform the actual measurement is not necessarily sound, and this Workload should not be used to collect real measurements.

The first step is to subclass our desired *workload type* depending on the purpose of our workload, in this example we are implementing a very simple workload and do not require any additional feature so shall inherit directly from the the base `Workload` class. We then need to provide a `name` for our workload which is what will be used to identify your workload for example in an agenda or via the show command.

```
import os
from wa import Workload, Parameter

class ZipTestWorkload(Workload):

    name = 'ziptest'
```

The `description` attribute should be a string in the structure of a short summary of the purpose of the workload, and will be shown when using the *list command*, followed by a more in- depth explanation separated by a new line.

```
description = '''
              Times how long it takes to gzip a file of a particular size on a device.

              This workload was created for illustration purposes only. It should not␣
↪be
              used to collect actual measurements.
              '''
```

In order to allow for additional configuration of the workload from a user a list of *parameters* can be supplied. These can be configured in a variety of different ways. For example here we are ensuring that the value of the parameter is an integer and larger than 0 using the `kind` and `constraint` options, also if no value is provided we are providing a `default` value of 2000000. These parameters will automatically have their value set as an attribute of the workload so later on we will be able to use the value provided here as `self.file_size`.

```
parameters = [
        Parameter('file_size', kind=int, default=2000000,
```

(continues on next page)

(continued from previous page)

```
                    constraint=lambda x: 0 < x,
                    description='Size of the file (in bytes) to be gzipped.')
]
```

Next we will implement our `setup` method. This is where we do any preparation that is required before the workload is ran, this is usually things like setting up required files on the device and generating commands from user input. In this case we will generate our input file on the host system and then push it to a known location on the target for use in the 'run' stage.

```
def setup(self, context):
    super(ZipTestWorkload, self).setup(context)
    # Generate a file of the specified size containing random garbage.
    host_infile = os.path.join(context.output_directory, 'infile')
    command = 'openssl rand -base64 {} > {}'.format(self.file_size, host_infile)
    os.system(command)
    # Set up on-device paths
    devpath = self.target.path  # os.path equivalent for the target
    self.target_infile = devpath.join(self.target.working_directory, 'infile')
    self.target_outfile = devpath.join(self.target.working_directory, 'outfile')
    # Push the file to the target
    self.target.push(host_infile, self.target_infile)
```

The `run` method is where the actual 'work' of the workload takes place and is what is measured by any instrumentation. So for this example this is the execution of creating the zip file on the target.

```
def run(self, context):
    cmd = 'cd {} && (time gzip {}) &>> {}'
    self.target.execute(cmd.format(self.target.working_directory,
                                   self.target_infile,
                                   self.target_outfile))
```

The `extract_results` method is used to extract any results from the target for example we want to pull the file containing the timing information that we will use to generate metrics for our workload and then we add this file as an artifact with a 'raw' kind, which means once WA has finished processing it will allow it to decide whether to keep the file or not.

```
def extract_results(self, context):
    super(ZipTestWorkload, self).extract_results(context)
    # Pull the results file to the host
    self.host_outfile = os.path.join(context.output_directory, 'timing_results')
    self.target.pull(self.target_outfile, self.host_outfile)
    context.add_artifact('ziptest-results', host_output_file, kind='raw')
```

The `update_output` method we can do any generation of metrics that we wish to for our workload. In this case we are going to simply convert the times reported into seconds and add them as 'metrics' to WA which can then be displayed to the user along with any others in a format dependant on which output processors they have enabled for the run.

```
def update_output(self, context):
    super(ZipTestWorkload, self).update_output(context)
    # Extract metrics form the file's contents and update the result
    # with them.
    content = iter(open(self.host_outfile).read().strip().split())
    for value, metric in zip(content, content):
        mins, secs = map(float, value[:-1].split('m'))
        context.add_metric(metric, secs + 60 * mins, 'seconds')
```

Finally in the `teardown` method we will perform any required clean up for the workload so we will delete the input and output files from the device.

```
def teardown(self, context):
    super(ZipTestWorkload, self).teardown(context)
    self.target.remove(self.target_infile)
    self.target.remove(self.target_outfile)
```

The full implementation of this workload would look something like:

```
import os
from wa import Workload, Parameter

class ZipTestWorkload(Workload):

    name = 'ziptest'

    description = '''
                  Times how long it takes to gzip a file of a particular size on a␣
↪device.

                  This workload was created for illustration purposes only. It should␣
↪not be
                  used to collect actual measurements.
                  '''

    parameters = [
            Parameter('file_size', kind=int, default=2000000,
                      constraint=lambda x: 0 < x,
                      description='Size of the file (in bytes) to be gzipped.')
    ]

    def setup(self, context):
        super(ZipTestWorkload, self).setup(context)
        # Generate a file of the specified size containing random garbage.
        host_infile = os.path.join(context.output_directory, 'infile')
        command = 'openssl rand -base64 {} > {}'.format(self.file_size, host_infile)
        os.system(command)
        # Set up on-device paths
        devpath = self.target.path  # os.path equivalent for the target
        self.target_infile = devpath.join(self.target.working_directory, 'infile')
        self.target_outfile = devpath.join(self.target.working_directory, 'outfile')
        # Push the file to the target
        self.target.push(host_infile, self.target_infile)

    def run(self, context):
        cmd = 'cd {} && (time gzip {}) &>> {}'
        self.target.execute(cmd.format(self.target.working_directory,
                                       self.target_infile,
                                       self.target_outfile))
    def extract_results(self, context):
        super(ZipTestWorkload, self).extract_results(context)
        # Pull the results file to the host
        self.host_outfile = os.path.join(context.output_directory, 'timing_results')
        self.target.pull(self.target_outfile, self.host_outfile)
        context.add_artifact('ziptest-results', host_output_file, kind='raw')

    def update_output(self, context):
```

(continues on next page)

(continued from previous page)

```
        super(ZipTestWorkload, self).update_output(context)
        # Extract metrics form the file's contents and update the result
        # with them.
        content = iter(open(self.host_outfile).read().strip().split())
        for value, metric in zip(content, content):
            mins, secs = map(float, value[:-1].split('m'))
            context.add_metric(metric, secs + 60 * mins, 'seconds')

    def teardown(self, context):
        super(ZipTestWorkload, self).teardown(context)
        self.target.remove(self.target_infile)
        self.target.remove(self.target_outfile)
```

Adding a ApkUiAutomator Workload

If we wish to create a workload to automate the testing of the Google Docs android app, we would choose to perform the automation using UIAutomator and we would want to automatically deploy and install the apk file to the target, therefore we would choose the *ApkUiAutomator workload* type with the following command:

```
$ wa create workload -k apkuiauto google_docs
Workload created in $WA_USER_DIRECTORY/plugins/google_docs
```

From here you can navigate to the displayed directory and you will find your `__init__.py` and a `uiauto` directory. The former is your python WA workload and will look something like this. For an example of what should be done in each of the main method please see *adding a basic example* above.

```
from wa import Parameter, ApkUiautoWorkload
class GoogleDocs(ApkUiautoWorkload):
    name = 'google_docs'
    description = "This is an placeholder description"
    # Replace with a list of supported package names in the APK file(s).
    package_names = ['package_name']

    parameters = [
     # Workload parameters go here e.g.
     Parameter('example_parameter', kind=int, allowed_values=[1,2,3],
               default=1, override=True, mandatory=False,
               description='This is an example parameter')
    ]

    def __init__(self, target, **kwargs):
     super(GoogleDocs, self).__init__(target, **kwargs)
     # Define any additional attributes required for the workload

    def init_resources(self, resolver):
     super(GoogleDocs, self).init_resources(resolver)
     # This method may be used to perform early resource discovery and
     # initialization. This is invoked during the initial loading stage and
     # before the device is ready, so cannot be used for any device-dependent
     # initialization. This method is invoked before the workload instance is
     # validated.

    def initialize(self, context):
     super(GoogleDocs, self).initialize(context)
```

(continues on next page)

(continued from previous page)

```
        # This method should be used to perform once-per-run initialization of a
        # workload instance.

    def validate(self):
        super(GoogleDocs, self).validate()
        # Validate inter-parameter assumptions etc

    def setup(self, context):
        super(GoogleDocs, self).setup(context)
        # Perform any necessary setup before starting the UI automation

    def extract_results(self, context):
        super(GoogleDocs, self).extract_results(context)
        # Extract results on the target

    def update_output(self, context):
        super(GoogleDocs, self).update_output(context)
        # Update the output within the specified execution context with the
        # metrics and artifacts form this workload iteration.

    def teardown(self, context):
        super(GoogleDocs, self).teardown(context)
        # Perform any final clean up for the Workload.
```

Depending on the purpose of your workload you can choose to implement which methods you require. The main things that need setting are the list of `package_names` which must be a list of strings containing the android package name that will be used during resource resolution to locate the relevant apk file for the workload. Additionally the the workload parameters will need to updating to any relevant parameters required by the workload as well as the description.

The latter will contain a framework for performing the UI automation on the target, the files you will be most interested in will be `uiauto/app/src/main/java/arm/wa/uiauto/UiAutomation.java` which will contain the actual code of the automation and will look something like:

```
package com.arm.wa.uiauto.google_docs;

import android.app.Activity;
import android.os.Bundle;
import org.junit.Test;
import org.junit.runner.RunWith;
import android.support.test.runner.AndroidJUnit4;

import android.util.Log;
import android.view.KeyEvent;

// Import the uiautomator libraries
import android.support.test.uiautomator.UiObject;
import android.support.test.uiautomator.UiObjectNotFoundException;
import android.support.test.uiautomator.UiScrollable;
import android.support.test.uiautomator.UiSelector;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;

import com.arm.wa.uiauto.BaseUiAutomation;
```

(continues on next page)

(continued from previous page)

```
@RunWith(AndroidJUnit4.class)
public class UiAutomation extends BaseUiAutomation {

    protected Bundle parameters;
    protected int example_parameter;

    public static String TAG = "google_docs";

    @Before
    public void initilize() throws Exception {
        // Perform any parameter initialization here
        parameters = getParams(); // Required to decode passed parameters.
        packageID = getPackageID(parameters);
        example_parameter = parameters.getInt("example_parameter");
    }

    @Test
    public void setup() throws Exception {
        // Optional: Perform any setup required before the main workload
        // is ran, e.g. dismissing welcome screens
    }

    @Test
    public void runWorkload() throws Exception {
            // The main UI Automation code goes here
    }

    @Test
    public void extractResults() throws Exception {
        // Optional: Extract any relevant results from the workload,
    }

    @Test
    public void teardown() throws Exception {
        // Optional: Perform any clean up for the workload
    }
}
```

A few items to note from the template:

- Each of the stages of execution for example `setup`, `runWorkload` etc are decorated with the `@Test` decorator, this is important to allow these methods to be called at the appropriate time however any additional methods you may add do not require this decorator.
- The `initialize` method has the `@Before` decorator, this is there to ensure that this method is called before executing any of the workload stages and therefore is used to decode and initialize any parameters that are passed in.
- The code currently retrieves the `example_parameter` that was provided to the python workload as an Integer, there are similar calls to retrieve parameters of different types e.g. `getString`, `getBoolean`, `getDouble` etc.

Once you have implemented your java workload you can use the file `uiauto/build.sh` to compile your automation into an apk file to perform the automation. The generated apk will be generated with the package name `com.arm.wa.uiauto.<workload_name>` which when running your workload will be automatically detected by the resource getters and deployed to the device.

Adding a ReventApk Workload

If we wish to create a workload to automate the testing of a UI based workload that we cannot / do not wish to use UiAutomator then we can perform the automation using revent. In this example we would want to automatically deploy and install an apk file to the target, therefore we would choose the *ApkRevent workload* type with the following command:

```
$ wa create workload -k apkrevent my_game
Workload created in $WA_USER_DIRECTORY/plugins/my_game
```

This will generate a revent based workload you will end up with a very similar python file as to the one outlined in generating a *UiAutomator based workload* however without the accompanying java automation files.

The main difference between the two is that this workload will subclass `ApkReventWorkload` instead of `ApkUiautomatorWorkload` as shown below.

```
from wa import ApkReventWorkload

class MyGame(ApkReventWorkload):

    name = 'mygame'
    package_names = ['com.mylogo.mygame']

    # ..
```

Adding an Instrument

This is an example of how we would create a instrument which will trace device errors using a custom "trace" binary file. For more detailed information please see the *Instrument Reference*. The first thing to do is to subclass `Instrument`, overwrite the variable name with what we want our instrument to be called and locate our binary for our instrument.

```
class TraceErrorsInstrument(Instrument):

    name = 'trace-errors'

    def __init__(self, target):
        super(TraceErrorsInstrument, self).__init__(target)
        self.binary_name = 'trace'
        self.binary_file = os.path.join(os.path.dirname(__file__), self.binary_name)
        self.trace_on_target = None
```

We then declare and implement the required methods as detailed in the *Instrument API*. For the `initialize` method, we want to install the executable file to the target so we can use the target's `install` method which will try to copy the file to a location on the device that supports execution, change the file mode appropriately and return the file path on the target.

```
def initialize(self, context):
    self.trace_on_target = self.target.install(self.binary_file)
```

Then we implemented the start method, which will simply run the file to start tracing. Supposing that the call to this binary requires some overhead to begin collecting errors we might want to decorate the method with the `@slow` decorator to try and reduce the impact on other running instruments. For more information on prioritization please see the *Developer Reference*.

```
@slow
def start(self, context):
    self.target.execute('{} start'.format(self.trace_on_target))
```

Lastly, we need to stop tracing once the workload stops and this happens in the stop method, assuming stopping the collection also require some overhead we have again decorated the method.

```
@slow
def stop(self, context):
    self.target.execute('{} stop'.format(self.trace_on_target))
```

Once we have generated our result data we need to retrieve it from the device for further processing or adding directly to WA's output for that job. For example for trace data we will want to pull it to the device and add it as a *artifact* to WA's *context* as shown below:

```
def extract_results(self, context):
    # pull the trace file from the target
    self.result = os.path.join(self.target.working_directory, 'trace.txt')
    self.target.pull(self.result, context.working_directory)
    context.add_artifact('error_trace', self.result, kind='export')
```

Once we have retrieved the data we can now do any further processing and add any relevant *Metrics* to the *context*. For this we will use the the `add_metric` method to add the results to the final output for that workload. The method can be passed 4 params, which are the metric *key*, *value*, *unit* and *lower_is_better*.

```
def update_output(self, context):
    # parse the file if needs to be parsed, or add result directly to
    # context.

    metric = # ..
    context.add_metric('number_of_errors', metric, lower_is_better=True
```

At the end of each job we might want to delete any files generated by the instruments and the code to clear these file goes in teardown method.

```
def teardown(self, context):
    self.target.remove(os.path.join(self.target.working_directory, 'trace.txt'))
```

At the very end of the run we would want to uninstall the binary we deployed earlier.

```
def finalize(self, context):
    self.target.uninstall(self.binary_name)
```

So the full example would look something like:

```
class TraceErrorsInstrument(Instrument):

    name = 'trace-errors'

    def __init__(self, target):
        super(TraceErrorsInstrument, self).__init__(target)
        self.binary_name = 'trace'
        self.binary_file = os.path.join(os.path.dirname(__file__), self.binary_name)
        self.trace_on_target = None

    def initialize(self, context):
        self.trace_on_target = self.target.install(self.binary_file)
```

(continues on next page)

(continued from previous page)

```
    @slow
    def start(self, context):
        self.target.execute('{} start'.format(self.trace_on_target))

    @slow
    def stop(self, context):
        self.target.execute('{} stop'.format(self.trace_on_target))

    def extract_results(self, context):
        self.result = os.path.join(self.target.working_directory, 'trace.txt')
        self.target.pull(self.result, context.working_directory)
        context.add_artifact('error_trace', self.result, kind='export')

    def update_output(self, context):
        metric = # ..
        context.add_metric('number_of_errors', metric, lower_is_better=True

    def teardown(self, context):
        self.target.remove(os.path.join(self.target.working_directory, 'trace.txt'))

    def finalize(self, context):
        self.target.uninstall(self.binary_name)
```

Adding an Output Processor

This is an example of how we would create an output processor which will format the run metrics as a column-aligned table. The first thing to do is to subclass `OutputProcessor` and overwrite the variable name with what we want our processor to be called and provide a short description.

Next we need to implement any relevant methods, (please see *adding an output processor* for all the available methods). In this case we only want to implement the `export_run_output` method as we are not generating any new artifacts and we only care about the overall output rather than the individual job outputs. The implementation is very simple, it just loops through all the available metrics for all the available jobs and adds them to a list which is written to file and then added as an *artifact* to the *context*.

```
import os
from wa import OutputProcessor
from wa.utils.misc import write_table


class Table(OutputProcessor):

    name = 'table'
    description = 'Generates a text file containing a column-aligned table of run␣
↪results.'

    def export_run_output(self, output, target_info):
        rows = []

        for job in output.jobs:
            for metric in job.metrics:
                rows.append([metric.name, str(metric.value), metric.units or '',
                             metric.lower_is_better  and '-' or '+'])
```

(continues on next page)

(continued from previous page)

```
        outfile =  output.get_path('table.txt')
        with open(outfile, 'w') as wfh:
            write_table(rows, wfh)
        output.add_artifact('results_table', 'table.txt', 'export')
```

Adding a Custom Target

This is an example of how we would create a customised target, this is typically used where we would need to augment the existing functionality for example on development boards where we need to perform additional actions to implement some functionality. In this example we are going to assume that this particular device is running Android and requires a special "wakeup" command to be sent before it can execute any other command.

To add a new target to WA we will first create a new file in `$WA_USER_DIRECTORY/plugins/example_target.py`. In order to facilitate with creating a new target WA provides a helper function to create a description for the specified target class, and specified components. For components that are not explicitly specified it will attempt to guess sensible defaults based on the target class' bases.

```
# Import our helper function
from wa import add_description_for_target

# Import the Target that our custom implementation will be based on
from devlib import AndroidTarget

class ExampleTarget(AndroidTarget):
    # Provide the name that will be used to identify your custom target
    name = 'example_target'

    # Override our custom method(s)
    def execute(self, *args, **kwargs):
        super(ExampleTarget, self).execute('wakeup', check_exit_code=False)
        return super(ExampleTarget, self).execute(*args, **kwargs)


description = '''An Android target which requires an explicit "wakeup" command
                 to be sent before accepting any other command'''
# Call the helper function with our newly created function and its description.
add_description_for_target(ExampleTarget, description)
```

Processing WA Output

This section will illustrate the use of WA's *output processing API* by creating a simple ASCII report generator. To make things concrete, this how-to will be processing the output from running the following agenda:

```
sections:
    - runtime_params:
        frequency: min
      classifiers:
          frequency: min
    - runtime_params:
        frequency: max
      classifiers:
          frequency: max
workloads:
```

(continues on next page)

(continued from previous page)

```
    - sysbench
    - deepbench
```

This runs two workloads under two different configurations each – once with CPU frequency fixed to max, and once with CPU frequency fixed to min. Classifiers are used to indicate the configuration in the output.

First, create the *RunOutput* object, which is the main interface for interacting with WA outputs. Or alternatively a *RunDatabaseOutput* if storing your results in a postgres database.

```
import sys

from wa import RunOutput

# Path to the output directory specified in the first argument
ro = RunOutput(sys.argv[1])
```

Run Info

Next, we're going to print out an overall summary of the run.

```
from __future__ import print_function   # for Python 2 compat.

from wa.utils.misc import format_duration

print('-'*20)
print('Run ID:', ro.info.uuid)
print('Run status:', ro.status)
print('Run started at:', ro.info.start_time.isoformat())
print('Run completed at:', ro.info.end_time.isoformat())
print('Run duration:', format_duration(ro.info.duration))
print('Ran', len(ro.jobs), 'jobs')
print('-'*20)
print()
```

`RunOutput.info` is an instance of `RunInfo` which encapsulates Overall-run metadata, such as the duration.

Target Info

Next, some information about the device the results where collected on.

```
print('    Target Information     ')
print('    -------------------    ')
print('hostname:', ro.target_info.hostname)
if ro.target_info.os == 'android':
    print('Android ID:', ro.target_info.android_id)
else:
    print('host ID:', ro.target_info.hostid)
print('CPUs:', ', '.join(cpu.name for cpu in ro.target_info.cpus))
print()

print('OS:', ro.target_info.os)
print('ABI:', ro.target_info.abi)
print('rooted:', ro.target_info.is_rooted)
```

(continues on next page)

(continued from previous page)

```
print('kernel version:', ro.target_info.kernel_version)
print('os version:')
for k, v in ro.target_info.os_version.items():
    print('\t', k+':', v)
print()
print('-'*27)
print()
```

`RunOutput.target_info` is an instance of `TargetInfo` that contains information collected from the target during the run.

Jobs Summary

Next, show a summary of executed jobs.

```
from wa.utils.misc import write_table

print('           Jobs            ')
print('           ----            ')
print()
rows = []
for job in ro.jobs:
    rows.append([job.id, job.label, job.iteration, job.status])
write_table(rows, sys.stdout, align='<<><',
        headers=['ID', 'LABEL', 'ITER.', 'STATUS'])
print()
print('-'*27)
print()
```

`RunOutput.jobs` is a list of `JobOutput` objects. These contain information about that particular job, including its execution status, and *Metrics* and *Artifacts* generated by the job.

Compare Metrics

Finally, collect metrics, sort them by the "frequency" classifier. Classifiers that are present in the metric but not its job have been added by the workload. For the purposes of this report, they will be used to augment the metric's name.

```
from collections import defaultdict

print()
print('    Metrics Comparison     ')
print('    ------------------     ')
print()
scores = defaultdict(lambda: defaultdict(lambda: defaultdict()))
for job in ro.jobs:
    for metric in job.metrics:
        workload = job.label
        name = metric.name
        freq = job.classifiers['frequency']
        for cname, cval in sorted(metric.classifiers.items()):
            if cname not in job.classifiers:
                # was not propagated from the job, therefore was
                # added by the workload
```

(continues on next page)

(continued from previous page)

```
                name += '/{}={}'.format(cname, cval)

        scores[workload][name][freq] = metric
```

Once the metrics have been sorted, generate the report showing the delta between the two configurations (indicated by the "frequency" classifier) and highlight any unexpected deltas (based on the `lower_is_better` attribute of the metric). (In practice, you will want to run multiple iterations of each configuration, calculate averages and standard deviations, and only highlight statically significant deltas.)

```
rows = []
for workload in sorted(scores.keys()):
    wldata = scores[workload]

    for name in sorted(wldata.keys()):
        min_score = wldata[name]['min'].value
        max_score = wldata[name]['max'].value
        delta =  max_score - min_score
        units = wldata[name]['min'].units or ''
        lib = wldata[name]['min'].lower_is_better

        warn = ''
        if (lib and delta > 0) or (not lib and delta < 0):
            warn = '!!!'

        rows.append([workload, name,
        '{:.3f}'.format(min_score), '{:.3f}'.format(max_score),
        '{:.3f}'.format(delta), units, warn])

    # separate workloads with a blank row
    rows.append(['', '', '', '', '', '', ''])


write_table(rows, sys.stdout, align='<<>>><<',
        headers=['WORKLOAD', 'METRIC', 'MIN.', 'MAX', 'DELTA', 'UNITS', ''])
print()
print('-'*27)
```

This concludes this how-to. For more information, please see *output processing API documentation*.

Complete Example

Below is the complete example code, and a report it generated for a sample run.

```
from __future__ import print_function   # for Python 2 compat.
import sys
from collections import defaultdict

from wa import RunOutput
from wa.utils.misc import format_duration, write_table



# Path to the output directory specified in the first argument
ro = RunOutput(sys.argv[1])
```

(continues on next page)

(continued from previous page)

```
print('-'*27)
print('Run ID:', ro.info.uuid)
print('Run status:', ro.status)
print('Run started at:', ro.info.start_time.isoformat())
print('Run completed at:', ro.info.end_time.isoformat())
print('Run duration:', format_duration(ro.info.duration))
print('Ran', len(ro.jobs), 'jobs')
print('-'*27)
print()

print('    Target Information     ')
print('    ------------------    ')
print('hostname:', ro.target_info.hostname)
if ro.target_info.os == 'android':
    print('Android ID:', ro.target_info.android_id)
else:
    print('host ID:', ro.target_info.hostid)
print('CPUs:', ', '.join(cpu.name for cpu in ro.target_info.cpus))
print()

print('OS:', ro.target_info.os)
print('ABI:', ro.target_info.abi)
print('rooted:', ro.target_info.is_rooted)
print('kernel version:', ro.target_info.kernel_version)
print('OS version:')
for k, v in ro.target_info.os_version.items():
    print('\t', k+':', v)
print()
print('-'*27)
print()

print('           Jobs            ')
print('           ----            ')
print()
rows = []
for job in ro.jobs:
    rows.append([job.id, job.label, job.iteration, job.status])
write_table(rows, sys.stdout, align='<<><',
        headers=['ID', 'LABEL', 'ITER.', 'STATUS'])
print()
print('-'*27)

print()
print('    Metrics Comparison     ')
print('    ------------------     ')
print()
scores = defaultdict(lambda: defaultdict(lambda: defaultdict()))
for job in ro.jobs:
    for metric in job.metrics:
        workload = job.label
        name = metric.name
        freq = job.classifiers['frequency']
        for cname, cval in sorted(metric.classifiers.items()):
            if cname not in job.classifiers:
                # was not propagated from the job, therefore was
                # added by the workload
                name += '/{}={}'.format(cname, cval)
```

(continues on next page)

(continued from previous page)

```
    scores[workload][name][freq] = metric

rows = []
for workload in sorted(scores.keys()):
    wldata = scores[workload]

    for name in sorted(wldata.keys()):
        min_score = wldata[name]['min'].value
        max_score = wldata[name]['max'].value
        delta =  max_score - min_score
        units = wldata[name]['min'].units or ''
        lib = wldata[name]['min'].lower_is_better

        warn = ''
        if (lib and delta > 0) or (not lib and delta < 0):
            warn = '!!!'

        rows.append([workload, name,
        '{:.3f}'.format(min_score), '{:.3f}'.format(max_score),
        '{:.3f}'.format(delta), units, warn])

    # separate workloads with a blank row
    rows.append(['', '', '', '', '', '', ''])


write_table(rows, sys.stdout, align='<<>>><<',
        headers=['WORKLOAD', 'METRIC', 'MIN.', 'MAX', 'DELTA', 'UNITS', ''])
print()
print('-'*27)
```

Sample output:

```
---------------------------
Run ID: 78aef931-cd4c-429b-ac9f-61f6893312e6
Run status: OK
Run started at: 2018-06-27T12:55:23.746941
Run completed at: 2018-06-27T13:04:51.067309
Run duration: 9 minutes 27 seconds
Ran 4 jobs
---------------------------

Target Information
-------------------
hostname: localhost
Android ID: b9d1d8b48cfba007
CPUs: A53, A53, A53, A53, A73, A73, A73, A73

OS: android
ABI: arm64
rooted: True
kernel version: 4.9.75-04208-g2c913991a83d-dirty 114 SMP PREEMPT Wed May 9 10:33:36␣
↪BST 2018
OS version:
        all_codenames: O
        base_os:
        codename: O
```

(continues on next page)

(continued from previous page)

```
        incremental: eng.valsch.20170517.180115
        preview_sdk: 0
        release: O
        sdk: 25
        security_patch: 2017-04-05

----------------------------

        Jobs
        ----

ID      LABEL     ITER. STATUS
--      -----     ----- ------
s1-wk1 sysbench      1 OK
s1-wk2 deepbench     1 OK
s2-wk1 sysbench      1 OK
s2-wk2 deepbench     1 OK

----------------------------

Metrics Comparison
------------------

WORKLOAD  METRIC                                          MIN.       MAX    DELTA␣
↪UNITS
--------  ------                                          ----       ---    ----- --
↪---
deepbench GOPS/a_t=n/b_t=n/k=1024/m=128/n=1              0.699     0.696   -0.003  ␣
↪       !!!
deepbench GOPS/a_t=n/b_t=n/k=1024/m=3072/n=1             0.471     0.715    0.244
deepbench GOPS/a_t=n/b_t=n/k=1024/m=3072/n=1500         23.514    36.432   12.918
deepbench GOPS/a_t=n/b_t=n/k=1216/m=64/n=1               0.333     0.333   -0.000  ␣
↪       !!!
deepbench GOPS/a_t=n/b_t=n/k=128/m=3072/n=1              0.405     1.073    0.668
deepbench GOPS/a_t=n/b_t=n/k=128/m=3072/n=1500          19.914    34.966   15.052
deepbench GOPS/a_t=n/b_t=n/k=128/m=4224/n=1              0.232     0.486    0.255
deepbench GOPS/a_t=n/b_t=n/k=1280/m=128/n=1500          20.721    31.654   10.933
deepbench GOPS/a_t=n/b_t=n/k=1408/m=128/n=1              0.701     0.702    0.001
deepbench GOPS/a_t=n/b_t=n/k=1408/m=176/n=1500          19.902    29.116    9.214
deepbench GOPS/a_t=n/b_t=n/k=176/m=4224/n=1500          26.030    39.550   13.519
deepbench GOPS/a_t=n/b_t=n/k=2048/m=35/n=700            10.884    23.615   12.731
deepbench GOPS/a_t=n/b_t=n/k=2048/m=5124/n=700          26.740    37.334   10.593
deepbench execution_time                               318.758   220.629  -98.129␣
↪seconds !!!
deepbench time (msec)/a_t=n/b_t=n/k=1024/m=128/n=1       0.375     0.377    0.002  ␣
↪       !!!
deepbench time (msec)/a_t=n/b_t=n/k=1024/m=3072/n=1     13.358     8.793   -4.565
deepbench time (msec)/a_t=n/b_t=n/k=1024/m=3072/n=1500 401.338   259.036 -142.302
deepbench time (msec)/a_t=n/b_t=n/k=1216/m=64/n=1        0.467     0.467    0.000  ␣
↪       !!!
deepbench time (msec)/a_t=n/b_t=n/k=128/m=3072/n=1       1.943     0.733   -1.210
deepbench time (msec)/a_t=n/b_t=n/k=128/m=3072/n=1500   59.237    33.737  -25.500
deepbench time (msec)/a_t=n/b_t=n/k=128/m=4224/n=1       4.666     2.224   -2.442
deepbench time (msec)/a_t=n/b_t=n/k=1280/m=128/n=1500   23.721    15.528   -8.193
deepbench time (msec)/a_t=n/b_t=n/k=1408/m=128/n=1       0.514     0.513   -0.001
deepbench time (msec)/a_t=n/b_t=n/k=1408/m=176/n=1500   37.354    25.533  -11.821
deepbench time (msec)/a_t=n/b_t=n/k=176/m=4224/n=1500   85.679    56.391  -29.288
```

(continues on next page)

(continued from previous page)

```
deepbench time (msec)/a_t=n/b_t=n/k=2048/m=35/n=700          9.220     4.249   -4.970
deepbench time (msec)/a_t=n/b_t=n/k=2048/m=5124/n=700      549.413   393.517 -155.896

sysbench  approx.  95 percentile                             3.800     1.450   -2.350 ms
sysbench  execution_time                                     1.790     1.437   -0.353␣
↪seconds !!!
sysbench  response time avg                                  1.400     1.120   -0.280 ms
sysbench  response time max                                 40.740    42.760    2.020␣
↪ms       !!!
sysbench  response time min                                  0.710     0.710    0.000 ms
sysbench  thread fairness events avg                      1250.000  1250.000    0.000
sysbench  thread fairness events stddev                    772.650   213.040 -559.610
sysbench  thread fairness execution time avg                 1.753     1.401   -0.352  ␣
↪       !!!
sysbench  thread fairness execution time stddev              0.000     0.000    0.000
sysbench  total number of events                         10000.000 10000.000    0.000
sysbench  total time                                         1.761     1.409   -0.352 s


---------------------------
```

3.1.3 Developer Reference

Framework Overview

Execution Model

At the high level, the execution model looks as follows:

After some initial setup, the framework initializes the device, loads and initialized instruments and output processors

and begins executing jobs defined by the workload specs in the agenda. Each job executes in basic stages:

initialize Perform any once-per-run initialization of a workload instance, i.e. binary resource resolution.

setup Initial setup for the workload is performed. E.g. required assets are deployed to the devices, required services or applications are launched, etc. Run time configuration of the device for the workload is also performed at this time.

setup_rerun (apk based workloads only) For some apk based workloads the application is required to be started twice. If the `requires_rerun` attribute of the workload is set to `True` then after the first setup method is called the application will be killed and then restarted. This method can then be used to perform any additional setup required.

run This is when the workload actually runs. This is defined as the part of the workload that is to be measured. Exactly what happens at this stage depends entirely on the workload.

extract results Extract any results that have been generated during the execution of the workload from the device and back to that target. Any files pulled from the devices should be added as artifacts to the run context.

update output Perform any required parsing and processing of any collected results and add any generated metrics to the run context.

teardown Final clean up is performed, e.g. applications may closed, files generated during execution deleted, etc.

Signals are dispatched (see *below*) at each stage of workload execution, which installed instruments can hook into in order to collect measurements, alter workload execution, etc. Instruments implementation usually mirrors that of workloads, defining initialization, setup, teardown and output processing stages for a particular instrument. Instead of a `run` method instruments usually implement `start` and `stop` methods instead which triggered just before and just after a workload run. However, the signal dispatch mechanism gives a high degree of flexibility to instruments allowing them to hook into almost any stage of a WA run (apart from the very early initialization).

Metrics and artifacts generated by workloads and instruments are accumulated by the framework and are then passed to active output processors. This happens after each individual workload execution and at the end of the run. A output processor may chose to act at either or both of these points.

Control Flow

This section goes into more detail explaining the relationship between the major components of the framework and how control passes between them during a run. It will only go through the major transitions and interactions and will not attempt to describe every single thing that happens.

Note: This is the control flow for the `wa run` command which is the main functionality of WA. Other commands are much simpler and most of what is described below does not apply to them.

---

1. `wa.framework.entrypoint` parses the command from the arguments, creates a `wa.framework.configuration.execution.ConfigManager` and executes the run command (`wa.commands.run.RunCommand`) passing it the ConfigManger.
2. Run command initializes the output directory and creates a `wa.framework.configuration.parsers.AgendaParser` and will parser an agenda and populate the ConfigManger based on the command line arguments. Finally it instantiates a `wa.framework.execution.Executor` and passes it the completed ConfigManager.
3. The Executor uses the ConfigManager to create a `wa.framework.configuration.core.RunConfiguration` and fully defines the configuration for the run (which will be serialised into `__meta` subdirectory under the output directory).

4. The Executor proceeds to instantiate a TargetManager, used to handle the device connection and configuration, and a `wa.framework.execution.ExecutionContext` which is used to track the current state of the run execution and also serves as a means of communication between the core framework and plugins. After this any required instruments and output processors are initialized and installed.
5. Finally, the Executor instantiates a `wa.framework.execution.Runner`, initializes its job queue with workload specs from the RunConfiguration, and kicks it off.
6. The Runner performs the run time configuration of the device and goes through the workload specs (in the order defined by `execution_order` setting), running each spec according to the execution model described in the previous section and sending signals (see below) at appropriate points during execution.
7. At the end of the run, the control is briefly passed back to the Executor, which outputs a summary for the run.

Signal Dispatch

WA uses the louie (formerly, pydispatcher) library for signal dispatch. Callbacks can be registered for signals emitted during the run. WA uses a version of louie that has been modified to introduce *priority* to registered callbacks (so that callbacks that are know to be slow can be registered with a lower priority and therefore do not interfere with other callbacks).

This mechanism is abstracted for instruments. Methods of an `wa.framework.Instrument` subclass automatically get hooked to appropriate signals based on their names when the instrument is "installed" for the run. Priority can then be specified by adding `extremely_fast`, `very_fast`, `fast` , `slow`, `very_slow` or `extremely_slow` *decorators* to the method definitions.

The full list of method names and the signals they map to may be seen at the *instrument method map*.

Signal dispatching mechanism may also be used directly, for example to dynamically register callbacks at runtime or allow plugins other than `Instruments` to access stages of the run they are normally not aware of.

Signals can be either paired or non paired signals. Non paired signals are one off signals that are sent to indicate special events or transitions in execution stages have occurred for example `TARGET_CONNECTED`. Paired signals are used to signify the start and end of a particular event. If the start signal has been sent the end signal is guaranteed to also be sent, whether the operation was a successes or not, however in the case of correct operation an additional success signal will also be sent. For example in the event of a successful reboot of the the device, the following signals will be sent `BEFORE_REBOOT`, `SUCCESSFUL_REBOOT` and `AFTER_REBOOT`.

An overview of what signals are sent at which point during execution can be seen below. Most of the paired signals have been removed from the diagram for clarity and shown as being dispatched from a particular stage of execution, however in reality these signals will be sent just before and just after these stages are executed. As mentioned above for each of these signals there will be at least 2 and up to 3 signals sent. If the "BEFORE_X" signal (sent just before the stage is ran) is sent then the "AFTER_X" (sent just after the stage is ran) signal is guaranteed to also be sent, and under normal operation a "SUCCESSFUL_X" signal is also sent just after stage has been completed. The diagram also lists the conditional signals that can be sent at any time during execution if something unexpected happens, for example an error occurs or the user aborts the run.

For more information see *Instrumentation Signal-Method Mapping*.

Plugins

Workload Automation offers several plugin points (or plugin types). The most interesting of these are

workloads These are the tasks that get executed and measured on the device. These can be benchmarks, high-level use cases, or pretty much anything else.

targets These are interfaces to the physical devices (development boards or end-user devices, such as smartphones) that use cases run on. Typically each model of a physical device would require its own interface class (though some functionality may be reused by subclassing from an existing base).

instruments Instruments allow collecting additional data from workload execution (e.g. system traces). Instruments are not specific to a particular workload. Instruments can hook into any stage of workload execution.

output processors These are used to format the results of workload execution once they have been collected. Depending on the callback used, these will run either after each iteration and/or at the end of the run, after all of the results have been collected.

You can create a plugin by subclassing the appropriate base class, defining appropriate methods and attributes, and putting the .py file containing the class into the "plugins" subdirectory under `~/.workload_automation` (or equivalent) where it will be automatically picked up by WA.

Plugin Basics

This section contains reference information common to plugins of all types.

The Context

Note: For clarification on the meaning of "workload specification" ("spec"), "job" and "workload" and the distiction between them, please see the *glossary*.

The majority of methods in plugins accept a context argument. This is an instance of `wa.framework.execution.ExecutionContext`. It contains information about the current state of execution of WA and keeps track of things like which workload is currently running.

Notable methods of the context are:

context.get_resource(resource, strict=True) This method should be used to retrieve a resource using the resource getters rather than using the ResourceResolver directly as this method additionally record any found resources hash in the output metadata.

context.add_artifact(name, host_file_path, kind, description=None, classifier=None) Plugins can add *artifacts* of various kinds to the run output directory for WA and associate them with a description and/or *classifier*.

context.add_metric(name, value, units=None, lower_is_better=False, classifiers=None) This method should be used to add *metrics* that have been generated from a workload, this will allow WA to process the results accordingly depending on which output processors are enabled.

Notable attributes of the context are:

context.workload `wa.framework.workload` object that is currently being executed.

context.tm This is the target manager that can be used to access various information about the target including initialization parameters.

context.current_job This is an instance of `wa.framework.job.Job` and contains all the information relevant to the workload job currently being executed.

context.current_job.spec The current workload specification being executed. This is an instance of `wa.framework.configuration.core.JobSpec` and defines the workload and the parameters under which it is being executed.

context.current_job.current_iteration The current iteration of the spec that is being executed. Note that this is the iteration for that spec, i.e. the number of times that spec has been run, *not* the total number of all iterations have been executed so far.

context.job_output This is the output object for the current iteration which is an instance of `wa.framework.output.JobOutput`. It contains the status of the iteration as well as the metrics and artifacts generated by the job.

In addition to these, context also defines a few useful paths (see below).

Paths

You should avoid using hard-coded absolute paths in your plugins whenever possible, as they make your code too dependent on a particular environment and may mean having to make adjustments when moving to new (host and/or device) platforms. To help avoid hard-coded absolute paths, WA defines a number of standard locations. You should strive to define your paths relative to one of these.

On the host

Host paths are available through the context object, which is passed to most plugin methods.

context.run_output_directory This is the top-level output directory for all WA results (by default, this will be "wa_output" in the directory in which WA was invoked.

context.output_directory This is the output directory for the current iteration. This will an iteration-specific subdirectory under the main results location. If there is no current iteration (e.g. when processing overall run results) this will point to the same location as `root_output_directory`.

Additionally, the global `wa.settings` object exposes on other location:

settings.dependency_directory this is the root directory for all plugin dependencies (e.g. media files, assets etc) that are not included within the plugin itself.

As per Python best practice, it is recommended that methods and values in `os.path` standard library module are used for host path manipulation.

On the target

Workloads and instruments have a `target` attribute, which is an interface to the target used by WA. It defines the following location:

target.working_directory This is the directory for all WA-related files on the target. All files deployed to the target should be pushed to somewhere under this location (the only exception being executables installed with `target.install` method).

Since there could be a mismatch between path notation used by the host and the target, the `os.path` modules should *not* be used for on-target path manipulation. Instead target has an equipment module exposed through `target.path` attribute. This has all the same attributes and behaves the same way as `os.path`, but is guaranteed to produce valid paths for the target, irrespective of the host's path notation. For example:

```
result_file = self.target.path.join(self.target.working_directory, "result.txt")
self.command = "{} -a -b -c {}".format(target_binary, result_file)
```

Note: Output processors, unlike workloads and instruments, do not have their own target attribute as they are designed to be able to be run offline.

Parameters

All plugins can be parametrized. Parameters are specified using `parameters` class attribute. This should be a list of `wa.framework.plugin.Parameter` instances. The following attributes can be specified on parameter creation:

name This is the only mandatory argument. The name will be used to create a corresponding attribute in the plugin instance, so it must be a valid Python identifier.

kind This is the type of the value of the parameter. This must be an callable. Normally this should be a standard Python type, e.g. `int` or `float`, or one the types defined in `wa.utils.types`. If not explicitly specified, this will default to `str`.

Note: Irrespective of the `kind` specified, `None` is always a valid value for a parameter. If you don't want to allow `None`, then set `mandatory` (see below) to `True`.

allowed_values A list of the only allowed values for this parameter.

Note: For composite types, such as `list_of_strings` or `list_of_ints` in `wa.utils.types`, each element of the value will be checked against `allowed_values` rather than the composite value itself.

default The default value to be used for this parameter if one has not been specified by the user. Defaults to `None`.

mandatory A `bool` indicating whether this parameter is mandatory. Setting this to `True` will make `None` an illegal value for the parameter. Defaults to `False`.

Note: Specifying a `default` will mean that this parameter will, effectively, be ignored (unless the user sets the param to `None`).

Note: Mandatory parameters are *bad*. If at all possible, you should strive to provide a sensible `default` or to make do without the parameter. Only when the param is absolutely necessary, and there really is no sensible default that could be given (e.g. something like login credentials), should you consider making it mandatory.

constraint This is an additional constraint to be enforced on the parameter beyond its type or fixed allowed values set. This should be a predicate (a function that takes a single argument – the user-supplied value – and returns a `bool` indicating whether the constraint has been satisfied).

override A parameter name must be unique not only within an plugin but also with that plugin's class hierarchy. If you try to declare a parameter with the same name as already exists, you will get an error. If you do want to override a parameter from further up in the inheritance hierarchy, you can indicate that by setting `override` attribute to `True`.

When overriding, you do not need to specify every other attribute of the parameter, just the ones you what to override. Values for the rest will be taken from the parameter in the base class.

Validation and cross-parameter constraints

A plugin will get validated at some point after construction. When exactly this occurs depends on the plugin type, but it *will* be validated before it is used.

You can implement `validate` method in your plugin (that takes no arguments beyond the `self`) to perform any additional *internal* validation in your plugin. By "internal", I mean that you cannot make assumptions about the surrounding environment (e.g. that the device has been initialized).

The contract for `validate` method is that it should raise an exception (either `wa.framework.exception.ConfigError` or plugin-specific exception type – see further on this page) if some validation condition has not, and cannot, been met. If the method returns without raising an exception, then the plugin is in a valid internal state.

Note that `validate` can be used not only to verify, but also to impose a valid internal state. In particular, this where cross-parameter constraints can be resolved. If the `default` or `allowed_values` of one parameter depend on another parameter, there is no way to express that declaratively when specifying the parameters. In that case the dependent attribute should be left unspecified on creation and should instead be set inside `validate`.

Logging

Every plugin class has it's own logger that you can access through `self.logger` inside the plugin's methods. Generally, a `Target` will log everything it is doing, so you shouldn't need to add much additional logging for device actions. However you might what to log additional information, e.g. what settings your plugin is using, what it is doing on the host, etc. (Operations on the host will not normally be logged, so your plugin should definitely log what it is doing on the host). One situation in particular where you should add logging is before doing something that might take a significant amount of time, such as downloading a file.

Documenting

All plugins and their parameter should be documented. For plugins themselves, this is done through `description` class attribute. The convention for an plugin description is that the first paragraph should be a short summary description of what the plugin does and why one would want to use it (among other things, this will get extracted and used by `wa list` command). Subsequent paragraphs (separated by blank lines) can then provide a more detailed description, including any limitations and setup instructions.

For parameters, the description is passed as an argument on creation. Please note that if `default`, `allowed_values`, or `constraint`, are set in the parameter, they do not need to be explicitly mentioned in the description (wa documentation utilities will automatically pull those). If the `default` is set in `validate` or additional cross-parameter constraints exist, this *should* be documented in the parameter description.

Both plugins and their parameters should be documented using reStructureText markup (standard markup for Python documentation). See:

http://docutils.sourceforge.net/rst.html

Aside from that, it is up to you how you document your plugin. You should try to provide enough information so that someone unfamiliar with your plugin is able to use it, e.g. you should document all settings and parameters your plugin expects (including what the valid values are).

Error Notification

When you detect an error condition, you should raise an appropriate exception to notify the user. The exception would typically be `ConfigError` or (depending the type of the plugin)

`WorkloadError`/`DeviceError`/`InstrumentError`/`OutputProcessorError`. All these errors are defined in `wa.framework.exception` module.

A `ConfigError` should be raised where there is a problem in configuration specified by the user (either through the agenda or config files). These errors are meant to be resolvable by simple adjustments to the configuration (and the error message should suggest what adjustments need to be made. For all other errors, such as missing dependencies, mis-configured environment, problems performing operations, etc., the plugin type-specific exceptions should be used.

If the plugin itself is capable of recovering from the error and carrying on, it may make more sense to log an ERROR or WARNING level message using the plugin's logger and to continue operation.

Metrics

This is what WA uses to store a single metric collected from executing a workload.

name the name of the metric. Uniquely identifies the metric within the results.

value The numerical value of the metric for this execution of a workload. This can be either an int or a float.

units Units for the collected value. Can be None if the value has no units (e.g. it's a count or a standardised score).

lower_is_better Boolean flag indicating where lower values are better than higher ones. Defaults to False.

classifiers A set of key-value pairs to further classify this metric beyond current iteration (e.g. this can be used to identify sub-tests).

Metrics can be added to WA output via the *context*:

```
context.add_metric("score", 9001)
context.add_metric("time", 2.35, "seconds", lower_is_better=True)
```

You only need to specify the name and the value for the metric. Units and classifiers are optional, and, if not specified otherwise, it will be assumed that higher values are better (`lower_is_better=False`).

The metric will be added to the result for the current job, if there is one; otherwise, it will be added to the overall run result.

Artifacts

This is an artifact generated during execution/post-processing of a workload. Unlike *metrics*, this represents an actual artifact, such as a file, generated. This may be "output", such as trace, or it could be "meta data" such as logs. These are distinguished using the `kind` attribute, which also helps WA decide how it should be handled. Currently supported kinds are:

log A log file. Not part of the "output" as such but contains information about the run/workload execution that be useful for diagnostics/meta analysis.

meta A file containing metadata. This is not part of the "output", but contains information that may be necessary to reproduce the results (contrast with `log` artifacts which are *not* necessary).

data This file contains new data, not available otherwise and should be considered part of the "output" generated by WA. Most traces would fall into this category.

export Exported version of results or some other artifact. This signifies that this artifact does not contain any new data that is not available elsewhere and that it may be safely discarded without losing information.

raw Signifies that this is a raw dump/log that is normally processed to extract useful information and is then discarded. In a sense, it is the opposite of `export`, but in general may also be discarded.

Note: whether a file is marked as `log`/`data` or `raw` depends on how important it is to preserve this file, e.g. when archiving, vs how much space it takes up. Unlike `export` artifacts which are (almost) always ignored by other exporters as that would never result in data loss, `raw` files *may* be processed by exporters if they decided that the risk of losing potentially (though unlikely) useful data is greater than the time/space cost of handling the artifact (e.g. a database uploader may choose to ignore `raw` artifacts, whereas a network filer archiver may choose to archive them).

As with *Metrics*, artifacts are added via the *context*:

```
context.add_artifact("benchmark-output", "bech-out.txt", kind="raw",
                     description="stdout from running the benchmark")
```

Note: The file *must* exist on the host by the point at which the artifact is added, otherwise an error will be raised.

The artifact will be added to the result of the current job, if there is one; otherwise, it will be added to the overall run result. In some situations, you may wish to add an artifact to the overall run while being inside a job context, this can be done with `add_run_artifact`:

```
context.add_run_artifact("score-summary", "scores.txt", kind="export",
       description="""
       Summary of the scores so far. Updated after
       every job.
       """)
```

In this case, you also need to make sure that the file represented by the artifact is written to the output directory for the run and not the current job.

Metadata

There may be additional data collected by your plugin that you want to record as part of the result, but that does not fall under the definition of a "metric". For example, you may want to record the version of the binary you're executing. You can do this by adding a metadata entry:

```
context.add_metadata("exe-version", 1.3)
```

Metadata will be added either to the current job result, or to the run result, depending on the current context. Metadata values can be scalars or nested structures of dicts/sequences; the only constraint is that all constituent objects of the value must be POD (Plain Old Data) types – see *WA POD types*.

There is special support for handling metadata entries that are dicts of values. The following call adds a metadata entry `"versions"` who's value is `{"my_exe": 1.3}`:

```
context.add_metadata("versions", "my_exe", 1.3)
```

If you attempt to add a metadata entry that already exists, an error will be raised, unless `force=True` is specified, in which case, it will be overwritten.

Updating an existing entry whose value is a collection can be done with `update_metadata`:

```
context.update_metadata("ran_apps", "my_exe")
context.update_metadata("versions", "my_other_exe", "2.3.0")
```

The first call appends `"my_exe"` to the list at metadata entry `"ran_apps"`. The second call updates the `"versions"` dict in the metadata with an entry for `"my_other_exe"`.

If an entry does not exit, `update_metadata` will create it, so it's recommended to always use that for non-scalar entries, unless the intention is specifically to ensure that the entry does not exist at the time of the call.

Classifiers

Classifiers are key-value pairs of tags that can be attached to metrics, artifacts, jobs, or the entire run. Run and job classifiers get propagated to metrics and artifacts. Classifier keys should be strings, and their values should be simple scalars (i.e. strings, numbers, or bools).

Classifiers can be thought of as "tags" that are used to annotate metrics and artifacts, in order to make it easier to sort through them later. WA itself does not do anything with them, however output processors will augment the output they generate with them (for example, `csv` processor can add additional columns for classifier keys).

Classifiers are typically added by the user to attach some domain-specific information (e.g. experiment configuration identifier) to the results, see *using classifiers*. However, plugins can also attach additional classifiers, by specifying them in `add_metric()` and `add_artifacts()` calls.

Metadata vs Classifiers

Both metadata and classifiers are sets of essentially opaque key-value pairs that get included in WA output. While they may seem somewhat similar and interchangeable, they serve different purposes and are handled differently by the framework.

Classifiers are used to annotate generated metrics and artifacts in order to assist post-processing tools in sorting through them. Metadata is used to record additional information that is not necessary for processing the results, but that may be needed in order to reproduce them or to make sense of them in a grander context.

These are specific differences in how they are handled:

- Classifiers are often provided by the user via the agenda (though can also be added by plugins). Metadata in only created by the framework and plugins.
- Classifier values must be simple scalars; metadata values can be nested collections, such as lists or dicts.
- Classifiers are used by output processors to augment the output the latter generated; metadata typically isn't.
- Classifiers are essentially associated with the individual metrics and artifacts (though in the agenda they're specified at workload, section, or global run levels); metadata is associated with a particular job or run, and not with metrics or artifacts.

Execution Decorators

The following decorators are available for use in order to control how often a method should be able to be executed.

For example, if we want to ensure that no matter how many iterations of a particular workload are ran, we only execute the initialize method for that instance once, we would use the decorator as follows:

```
from wa.utils.exec_control import once

@once
def initialize(self, context):
    # Perform one time initialization e.g. installing a binary to target
    # ..
```

@once_per_instance

The specified method will be invoked only once for every bound instance within the environment.

@once_per_class

The specified method will be invoked only once for all instances of a class within the environment.

@once

The specified method will be invoked only once within the environment.

Warning: If a method containing a super call is decorated, this will also cause stop propagation up the hierarchy, unless this is the desired effect, additional functionality should be implemented in a separate decorated method which can then be called allowing for normal propagation to be retained.

Utils

Workload Automation defines a number of utilities collected under `wa.utils` subpackage. These utilities were created to help with the implementation of the framework itself, but may be also be useful when implementing plugins.

Workloads

All of the type inherit from the same base `Workload` and its API can be seen in the *API* section.

Workload methods (except for `validate`) take a single argument that is a `wa.framework.execution.ExecutionContext` instance. This object keeps track of the current execution state (such as the current workload, iteration number, etc), and contains, among other things, a `wa.framework.output.JobOutput` instance that should be populated from the `update_output` method with the results of the execution. For more information please see *the context* documentation.

```
# ...

def update_output(self, context):
    # ...
    context.add_metric('energy', 23.6, 'Joules', lower_is_better=True)

# ...
```

Workload Types

There are multiple workload types that you can inherit from depending on the purpose of your workload, the different types along with an output of their intended use cases are outlined below.

Basic (`wa.Workload`)

This type of the workload is the simplest type of workload and is left the to developer to implement its full functionality.

Apk (`wa.ApkWorkload`)

This workload will simply deploy and launch an android app in its basic form with no UI interaction.

UiAuto (`wa.UiautoWorkload`)

This workload is for android targets which will use UiAutomator to interact with UI elements without a specific android app, for example performing manipulation of android itself. This is the preferred type of automation as the results are more portable and reproducible due to being able to wait for UI elements to appear rather than having to rely on human recordings.

ApkUiAuto (`wa.ApkUiautoWorkload`)

The is the same as the UiAuto workload however it is also associated with an android app e.g. AdobeReader and will automatically deploy and launch the android app before running the automation.

Revent (`wa.ReventWorkload`)

Revent workloads are designed primarily for games as these are unable to be automated with UiAutomator due to the fact that they are rendered within a single UI element. They require a recording to be performed manually and currently will need re-recording for each different device. For more information on revent workloads been please see *Automating GUI Interactions With Revent*

APKRevent (`wa.ApkReventWorkload`)

The is the same as the Revent workload however it is also associated with an android app e.g. AngryBirds and will automatically deploy and launch the android app before running the automation.

Revent Recordings

Convention for Naming revent Files for Revent Workloads

There is a convention for naming revent files which you should follow if you want to record your own revent files. Each revent file must start with the device name(case sensitive) then followed by a dot '.' then the stage name then '.revent'. All your custom revent files should reside at `'~/.workload_automation/dependencies/WORKLOAD NAME/'`. These are the current supported stages:

setup This stage is where the application is loaded (if present). It is a good place to record an revent here to perform any tasks to get ready for the main part of the workload to start.

run This stage is where the main work of the workload should be performed. This will allow for more accurate results if the revent file for this stage only records the main actions under test.

extract_results This stage is used after the workload has been completed to retrieve any metrics from the workload e.g. a score.

teardown This stage is where any final actions should be performed to clean up the workload.

Only the run stage is mandatory, the remaining stages will be replayed if a recording is present otherwise no actions will be performed for that particular stage.

For instance, to add a custom revent files for a device named "mydevice" and a workload name "myworkload", you need to add the revent files to the directory `/home/$WA_USER_HOME/dependencies/myworkload/revent_files` creating it if necessary.

```
mydevice.setup.revent
mydevice.run.revent
mydevice.extract_results.revent
mydevice.teardown.revent
```

Any revent file in the dependencies will always overwrite the revent file in the workload directory. So for example it is possible to just provide one revent for setup in the dependencies and use the run.revent that is in the workload directory.

File format of revent recordings

You do not need to understand recording format in order to use revent. This section is intended for those looking to extend revent in some way, or to utilize revent recordings for other purposes.

Format Overview

Recordings are stored in a binary format. A recording consists of three sections:

```
+-+-+-+-+-+-+-+-+-+-+-+-+
|       Header          |
+-+-+-+-+-+-+-+-+-+-+-+-+
|                       |
|  Device Description   |
|                       |
+-+-+-+-+-+-+-+-+-+-+-+-+
|                       |
|                       |
```

(continues on next page)

(continued from previous page)

```
|     Event Stream    |
|                     |
|                     |
+-+-+-+-+-+-+-+-+-+-+-+
```

The header contains metadata describing the recording. The device description contains information about input devices involved in this recording. Finally, the event stream contains the recorded input events.

All fields are either fixed size or prefixed with their length or the number of (fixed-sized) elements.

Note: All values below are little endian

Recording Header

An revent recoding header has the following structure

- It starts with the "magic" string `REVENT` to indicate that this is an revent recording.
- The magic is followed by a 16 bit version number. This indicates the format version of the recording that follows. Current version is `2`.
- The next 16 bits indicate the type of the recording. This dictates the structure of the Device Description section. Valid values are:

  0 This is a general input event recording. The device description contains a list of paths from which the events where recorded.

  1 This a gamepad recording. The device description contains the description of the gamepad used to create the recording.

- The header is zero-padded to 128 bits.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       'R'       |       'E'       |       'V'       |       'E'       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       'N'       |       'T'       |             Version             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Mode               |             PADDING             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             PADDING                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

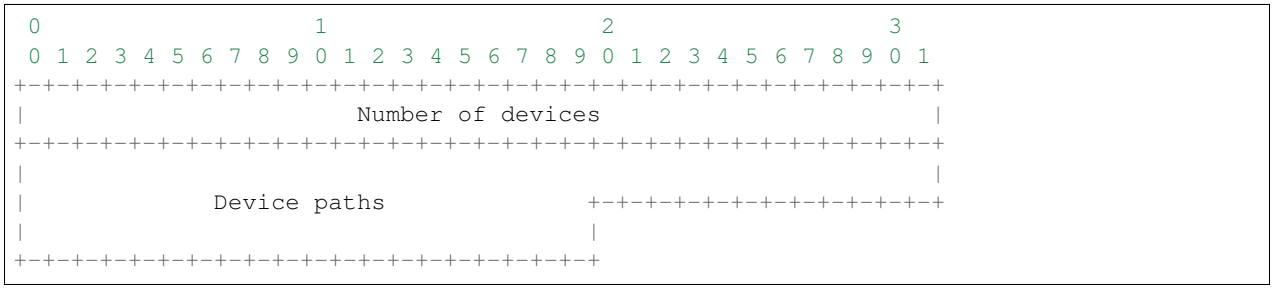
Device Description

This section describes the input devices used in the recording. Its structure is determined by the value of `Mode` field in the header.

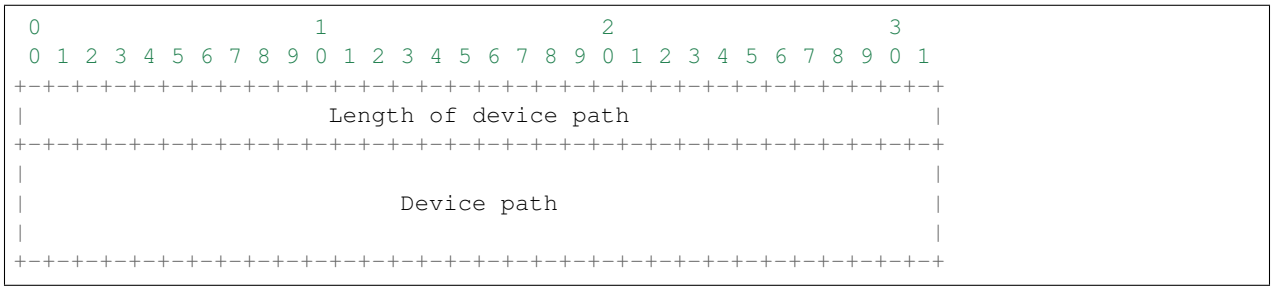
General Recording

Note: This is the only format supported prior to version 2.

The recording has been made from all available input devices. This section contains the list of /dev/input paths for the devices, prefixed with total number of the devices recorded.

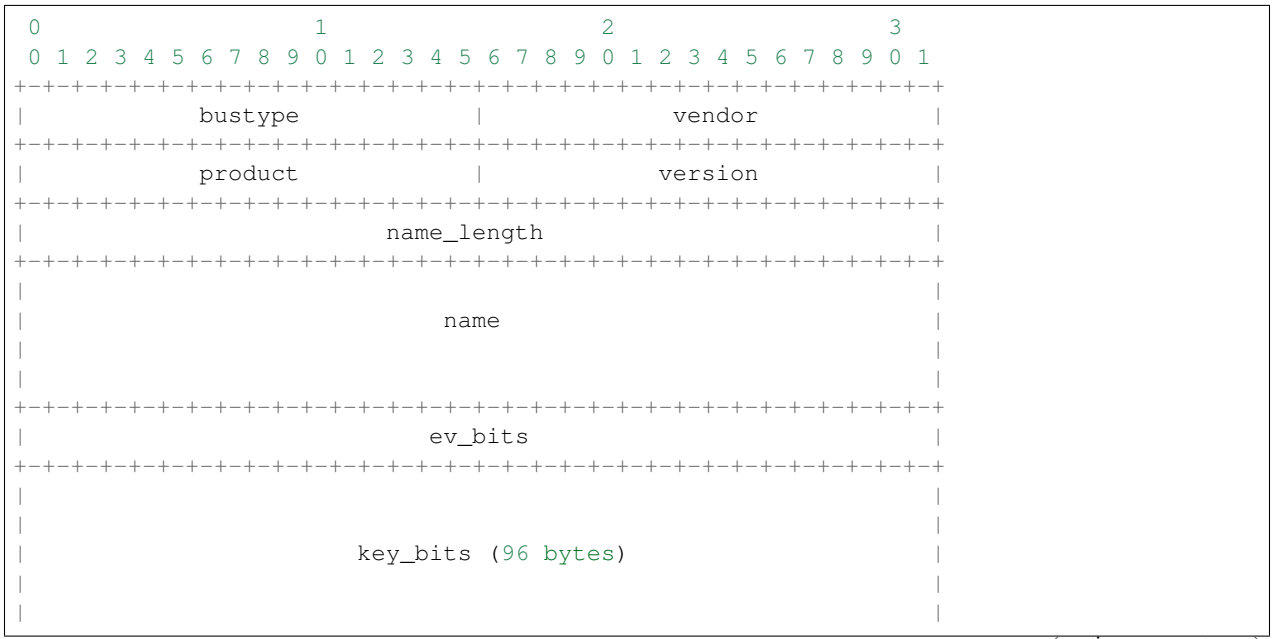


Similarly, each device path is a length-prefixed string. Unlike C strings, the path is *not* NULL-terminated.



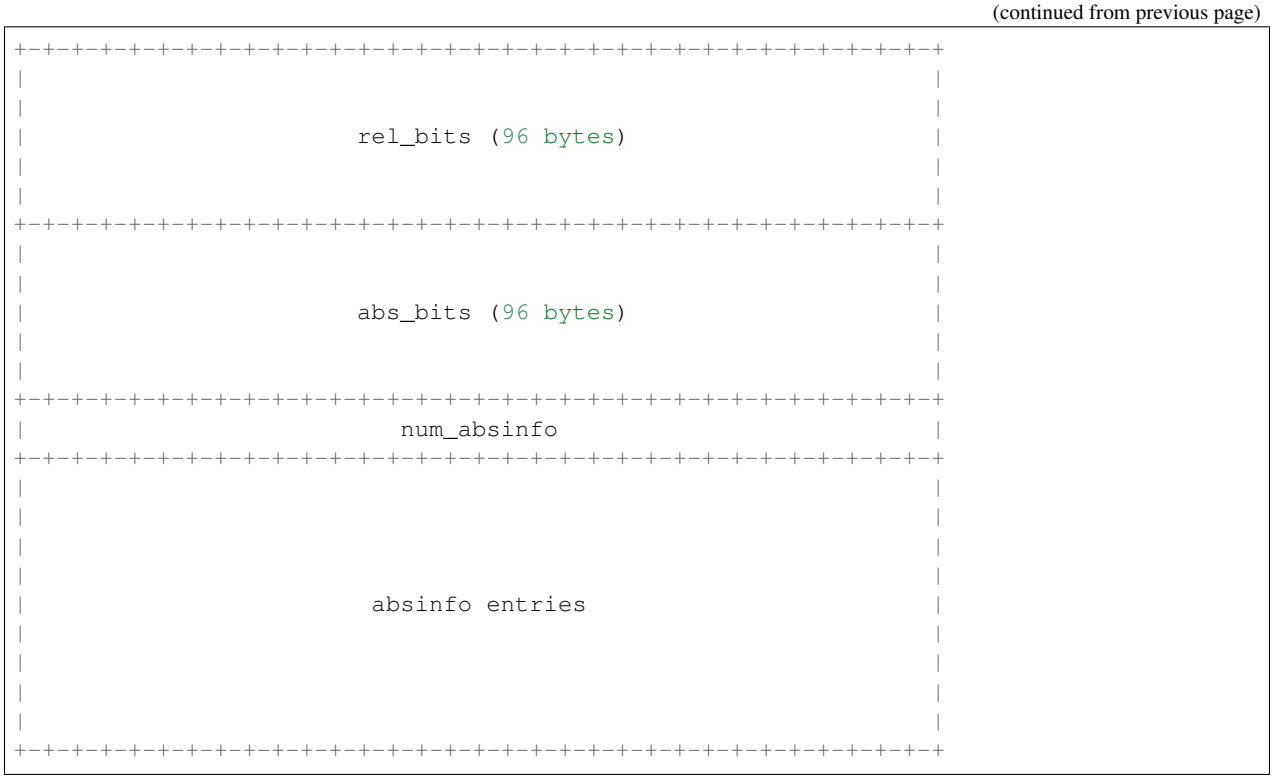
Gamepad Recording

The recording has been made from a specific gamepad. All events in the stream will be for that device only. The section describes the device properties that will be used to create a virtual input device using /dev/uinput. Please see linux/input.h header in the Linux kernel source for more information about the fields in this section.

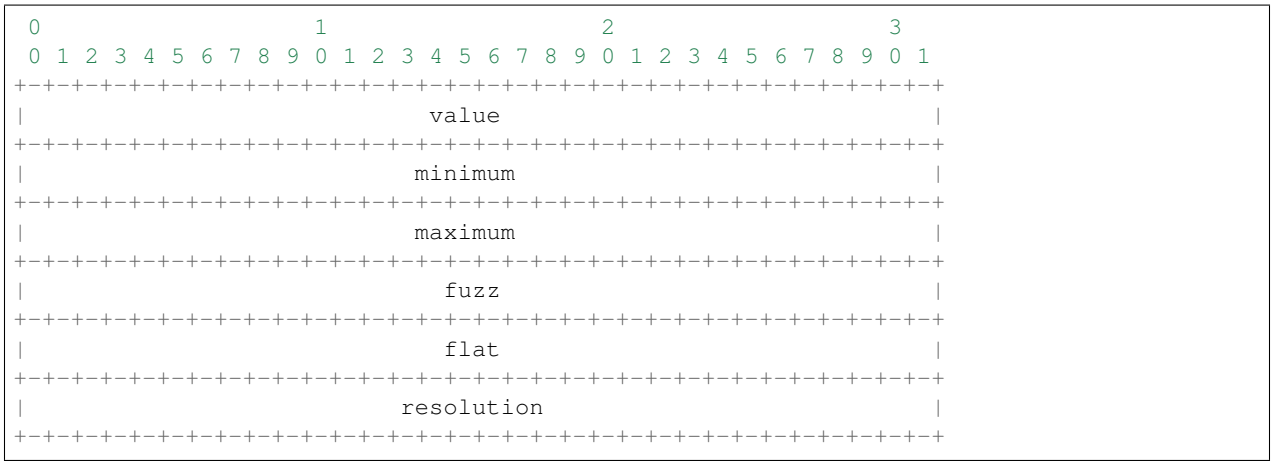


(continues on next page)

(continued from previous page)

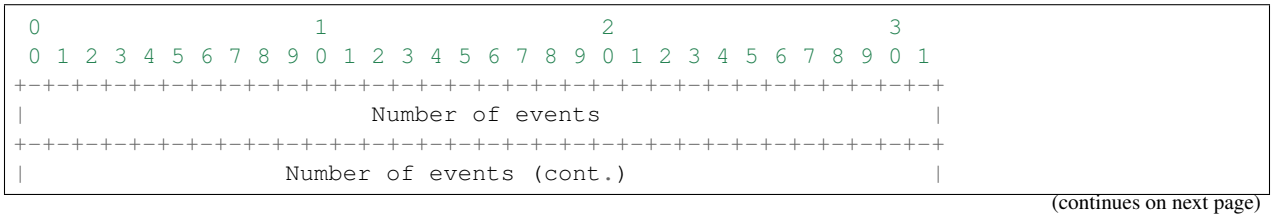


Each `absinfo` entry consists of six 32 bit values. The number of entries is determined by the `abs_bits` field.



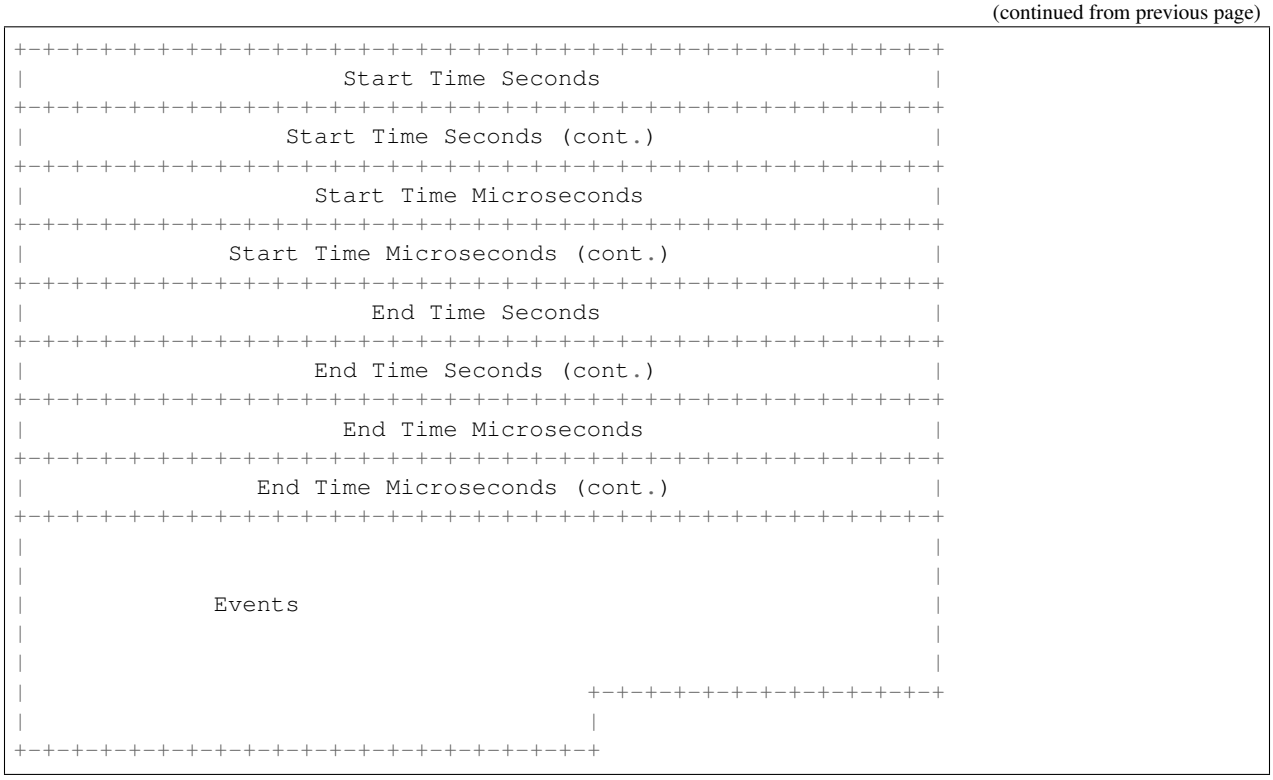
Event Stream

The majority of an revent recording will be made up of the input events that were recorded. The event stream is prefixed with the number of events in the stream, and start and end times for the recording.



(continues on next page)

(continued from previous page)

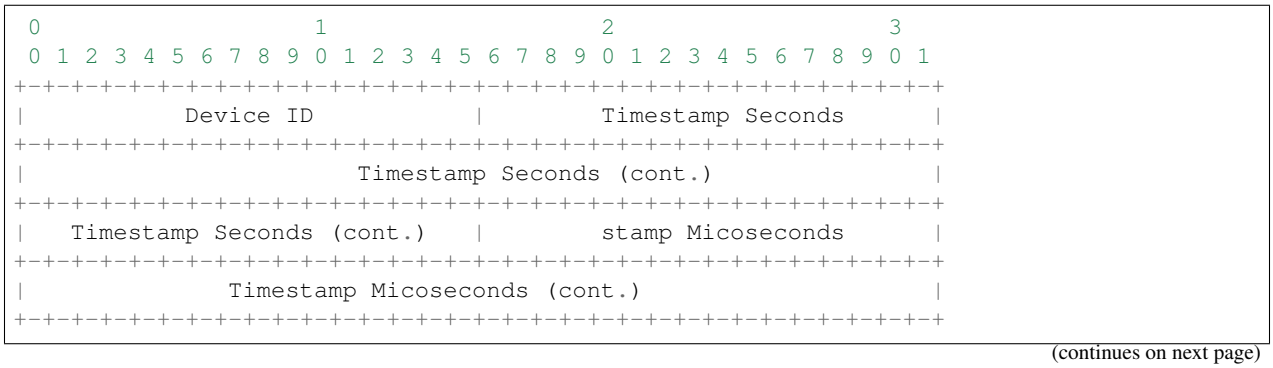


Event Structure

Each event entry structured as follows:

- An unsigned short integer representing which device from the list of device paths this event is for (zero indexed). E.g. Device ID = 3 would be the 4th device in the list of device paths.
- A unsigned long integer representing the number of seconds since "epoch" when the event was recorded.
- A unsigned long integer representing the microseconds part of the timestamp.
- An unsigned integer representing the event type
- An unsigned integer representing the event code
- An unsigned integer representing the event value

For more information about the event type, code and value please read: https://www.kernel.org/doc/Documentation/input/event-codes.txt



(continues on next page)

(continued from previous page)

```
| Timestamp Micoseconds (cont.) |          Event Type           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Event Code           |          Event Value          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       Event Value (cont.)     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Parser

WA has a parser for revent recordings. This can be used to work with revent recordings in scripts. Here is an example:

```
from wa.utils.revent import ReventRecording

with ReventRecording('/path/to/recording.revent') as recording:
    print "Recording: {}".format(recording.filepath)
    print "There are {} input events".format(recording.num_events)
    print "Over a total of {} seconds".format(recording.duration)
```

Serialization

Overview of Serialization

WA employs a serialization mechanism in order to store some of its internal structures inside the output directory. Serialization is performed in two stages:

1. A serializable object is converted into a POD (Plain Old Data) structure consisting of primitive Python types, and a few additional types (see *WA POD Types* below).
2. The POD structure is serialized into a particular format by a generic parser for that format. Currently, *yaml* and *json* are supported.

Deserialization works in reverse order – first the serialized text is parsed into a POD, which is then converted to the appropriate object.

Implementing Serializable Objects

In order to be considered serializable, an object must either be a POD, or it must implement the `to_pod()` method and `from_pod` static/class method, which will perform the conversion to/form pod.

As an example, below as a (somewhat trimmed) implementation of the `Event` class:

```
class Event(object):

    @staticmethod
    def from_pod(pod):
        instance = Event(pod['message'])
        instance.timestamp = pod['timestamp']
        return instance

    def __init__(self, message):
```

(continues on next page)

(continued from previous page)

```
        self.timestamp = datetime.utcnow()
        self.message = message

    def to_pod(self):
        return dict(
            timestamp=self.timestamp,
            message=self.message,
        )
```

Serialization API

read_pod(*source*, *fmt=None*)

write_pod(*pod*, *dest*, *fmt=None*)

These read and write PODs from a file. The format will be inferred, if possible, from the extension of the file, or it may be specified explicitly with `fmt`. `source` and `dest` can be either strings, in which case they will be interpreted as paths, or they can be file-like objects.

is_pod(*obj*)

Returns `True` if `obj` is a POD, and `False` otherwise.

dump(*o*, *wfh*, *fmt='json'*, **args*, ***kwargs*)

load(*s*, *fmt='json'*, **args*, ***kwargs*)

These implment an altenative serialization interface, which matches the interface exposed by the parsers for the supported formats.

WA POD Types

POD types are types that can be handled by a serializer directly, without a need for any additional information. These consist of the build-in python types

```
list
tuple
dict
set
str
unicode
int
float
bool
```

. . . the standard library types

```
OrderedDict
datetime
```

. . . and the WA-defined types

```
regex_type
none_type
level
cpu_mask
```

Any structure consisting entirely of these types is a POD and can be serialized and then deserialized without losing information. It is important to note that only these specific types are considered POD, their subclasses are *not*.

Note: `dicts` get deserialized as `OrderedDicts`.

Serialization Formats

WA utilizes two serialization formats: YAML and JSON. YAML is used for files intended to be primarily written and/or read by humans; JSON is used for files intended to be primarily written and/or read by WA and other programs.

The parsers and serializers for these formats used by WA have been modified to handle additional types (e.g. regular expressions) that are typically not supported by the formats. This was done in such a way that the resulting files are still valid and can be parsed by any parser for that format.

Contributing

Code

We welcome code contributions via GitHub pull requests. To help with maintainability of the code line we ask that the code uses a coding style consistent with the rest of WA code. Briefly, it is

- PEP8 with line length and block comment rules relaxed (the wrapper for PEP8 checker inside `dev_scripts` will run it with appropriate configuration).
- Four-space indentation (*no tabs!*).
- Title-case for class names, underscore-delimited lower case for functions, methods, and variables.
- Use descriptive variable names. Delimit words with `'_'` for readability. Avoid shortening words, skipping vowels, etc (common abbreviations such as "stats" for "statistics", "config" for "configuration", etc are OK). Do *not* use Hungarian notation (so prefer `birth_date` over `dtBirth`).

New extensions should also follow implementation guidelines specified in the *Writing Plugins* section of the documentation.

We ask that the following checks are performed on the modified code prior to submitting a pull request:

Note: You will need pylint and pep8 static checkers installed:

```
pip install pep8
pip install pylint
```

It is recommended that you install via pip rather than through your distribution's package manager because the latter is likely to contain out-of-date version of these tools.

- `./dev_scripts/pylint` should be run without arguments and should produce no output (any output should be addressed by making appropriate changes in the code or adding a pylint ignore directive, if there is a good reason for keeping the code as is).
- `./dev_scripts/pep8` should be run without arguments and should produce no output (any output should be addressed by making appropriate changes in the code).

- If the modifications touch core framework (anything under `wa/framework`), unit tests should be run using `nosetests`, and they should all pass.
  - If significant additions have been made to the framework, unit tests should be added to cover the new functionality.
- If modifications have been made to documentation (this includes description attributes for Parameters and Extensions), documentation should be built to make sure no errors or warning during build process, and a visual inspection of new/updated sections in resulting HTML should be performed to ensure everything renders as expected.

Once you have your contribution is ready, please follow instructions in GitHub documentation to create a pull request.

Documentation

Headings

To allow for consistent headings to be used through out the document the following character sequences should be used when creating headings

```
=========
Heading 1
=========

Only used for top level headings which should also have an entry in the
navigational side bar.

*********
Heading 2
*********

Main page heading used for page title, should not have a top level entry in the
side bar.

Heading 3
==========

Regular section heading.

Heading 4
---------

Sub-heading.

Heading 5
~~~~~~~~~

Heading 6
^^^^^^^^^

Heading 7
"""""""""
```

Configuration Listings

To keep a consistent style for presenting configuration options, the preferred style is to use a *Field List*.

(See: http://docutils.sourceforge.net/docs/user/rst/quickref.html#field-lists)

Example:

```
:parameter: My Description
```

Will render as:

parameter My Description

API Style

When documenting an API the currently preferred style is to provide a short description of the class, followed by the attributes of the class in a *Definition List* followed by the methods using the *method* directive.

(See: http://docutils.sourceforge.net/docs/user/rst/quickref.html#definition-lists)

Example:

```
API
===

:class:`MyClass`
----------------

:class:`MyClass` is an example class to demonstrate API documentation.

``attribute1``
    The first attribute of the example class.

``attribute2``
    Another attribute example.

methods
"""""""

.. method:: MyClass.retrieve_output(name)

    Retrieve the output for ``name``.

    :param name:  The output that should be returned.
    :return: An :class:`Output` object for ``name``.
    :raises NotFoundError: If no output can be found.
```

Will render as:

`MyClass` is an example class to demonstrate API documentation.

`attribute1` The first attribute of the example class.

`attribute2` Another attribute example.

methods

MyClass.**retrieve_output**(*name*)

Retrieve the output for `name`.

Parameters **name** – The output that should be returned.

Returns An `Output` object for `name`.

Raises **NotFoundError** – If no output can be found.

CHAPTER 4

Plugin Reference

4.1 Plugin Reference

This section lists Plugins that currently come with WA3. Each package below represents a particular type of extension (e.g. a workload); each sub-package of that package is a particular instance of that extension (e.g. the Andebench workload). Clicking on a link will show what the individual extension does, what configuration parameters it takes, etc.

For how to implement you own Plugins, please refer to the guides in the *writing plugins* section.

4.1.1 Workloads

- *geekbench-corporate*
- *gfxbench-corporate*
- *glbenchmark*
- *gmail*
- *googlemaps*
- *googlephotos*
- *googleplaybooks*
- *googleslides*
- *hackbench*
- *homescreen*
- *hwuitest*
- *idle*
- *jankbench*
- *lmbench*
- *manual*
- *meabo*
- *memcpy*
- *mongoperf*
- *openssl*
- *pcmark*
- *recentfling*
- *rt-app*
- *shellscript*
- *speedometer*
- *stress-ng*
- *sysbench*
- *templerun2*
- *thechase*
- *vellamo*
- *youtube*
- *youtube_playback*

adobereader

The Adobe Reader workflow carries out the following typical productivity tasks.

Test description:

1. Open a local file on the device
2. **Gestures test:** 2.1. Swipe down across the central 50% of the screen in 200 x 5ms steps 2.2. Swipe up across the central 50% of the screen in 200 x 5ms steps 2.3. Swipe right from the edge of the screen in 50 x 5ms steps 2.4. Swipe left from the edge of the screen in 50 x 5ms steps 2.5. Pinch out 50% in 100 x 5ms steps 2.6. Pinch In 50% in 100 x 5ms steps
3. **Search test:** Search `document_name` for each string in the `search_string_list`
4. Close the document

Known working APK version: 16.1

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

package_name: type: `'str'`

The package name that can be used to specify the workload apk to use.

install_timeout: type: `'integer'`

Timeout for the installation of the apk.

constraint: `value > 0`

default: `300`

version: type: `'str'`

The version of the package to be used.

variant: type: `'str'`

The variant of the package to be used.

strict: type: `'boolean'`

Whether to throw an error if the specified package cannot be found on host.

force_install: type: `'boolean'`

Always re-install the APK, even if matching version is found already installed on the device.

uninstall: type: `'boolean'`

If `True`, will uninstall workload's APK as part of teardown.'

exact_abi: type: `'boolean'`

If `True`, workload will check that the APK matches the target device ABI, otherwise any suitable APK found will be used.

prefer_host_package: type: `'boolean'`

If `True` then a package found on the host will be preferred if it is a valid version and ABI, if not it will fall back to the version on the target if available. If `False` then the version on the target is preferred instead.

aliases: `'check_apk'`

default: `True`

view: type: `'str'`

Manually override the 'View' of the workload for use with instruments such as the `fps` instrument. If not specified, a workload dependant 'View' will be automatically generated.

markers_enabled: type: `'boolean'`

If set to `True`, workloads will insert markers into logs at various points during execution. These markers may be used by other plugins or post-processing scripts to provide measurements or statistics for specific parts of the workload execution.

document_name: type: `'str'`

The document name to use for the Gesture and Search test.

default: `'uxperf_test_doc.pdf'`

search_string_list: type: `'list_of_strs'`

For each string in the list, a document search is performed using the string as the search term. At least one must be provided.

constraint: `len(value) > 0`

default: `['The quick brown fox jumps over the lazy dog', 'TEST_SEARCH_STRING']`

---

androbench

Executes storage performance benchmarks

The Androbench workflow carries out the following typical productivity tasks. 1. Open Androbench application 2. Execute all memory benchmarks

Known working APK version: 5.0.1

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

package_name: type: `'str'`

The package name that can be used to specify the workload apk to use.

install_timeout: type: `'integer'`

Timeout for the installation of the apk.

constraint: `value > 0`

default: `300`

version: type: `'str'`

The version of the package to be used.

variant: type: `'str'`

The variant of the package to be used.

strict: type: `'boolean'`

Whether to throw an error if the specified package cannot be found on host.

force_install: type: `'boolean'`

Always re-install the APK, even if matching version is found already installed on the device.

uninstall: type: `'boolean'`

If `True`, will uninstall workload's APK as part of teardown.'

exact_abi: type: `'boolean'`

If `True`, workload will check that the APK matches the target device ABI, otherwise any suitable APK found will be used.

prefer_host_package: type: `'boolean'`

If `True` then a package found on the host will be preferred if it is a valid version and ABI, if not it will fall back to the version on the target if available. If `False` then the version on the target is preferred instead.

aliases: `'check_apk'`

default: `True`

view: type: `'str'`

Manually override the 'View' of the workload for use with instruments such as the `fps` instrument. If not specified, a workload dependant 'View' will be automatically generated.

markers_enabled: type: `'boolean'`

If set to `True`, workloads will insert markers into logs at various points during execution. These markers may be used by other plugins or post-processing scripts to provide measurements or statistics for specific parts of the workload execution.

---

angrybirds_rio

Angry Birds Rio game.

The sequel to the very popular Android 2D game.

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

package_name: type: `'str'`

The package name that can be used to specify the workload apk to use.

install_timeout: type: `'integer'`

Timeout for the installation of the apk.

constraint: `value > 0`

default: `300`

version: type: `'str'`

The version of the package to be used.

variant: type: `'str'`

The variant of the package to be used.

strict: type: `'boolean'`

Whether to throw an error if the specified package cannot be found on host.

force_install: type: `'boolean'`

Always re-install the APK, even if matching version is found already installed on the device.

uninstall: type: `'boolean'`

If `True`, will uninstall workload's APK as part of teardown.'

exact_abi: type: `'boolean'`

If `True`, workload will check that the APK matches the target device ABI, otherwise any suitable APK found will be used.

prefer_host_package: type: `'boolean'`

If `True` then a package found on the host will be preferred if it is a valid version and ABI, if not it will fall back to the version on the target if available. If `False` then the version on the target is preferred instead.

aliases: `'check_apk'`

default: `True`

view: type: `'str'`

Manually override the 'View' of the workload for use with instruments such as the `fps` instrument. If not specified, a workload dependant 'View' will be automatically generated.

---

antutu

Executes Antutu 3D, UX, CPU and Memory tests

Test description: 1. Open Antutu application 2. Execute Antutu benchmark

Known working APK version: 7.0.4

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

package_name: type: `'str'`

The package name that can be used to specify the workload apk to use.

install_timeout: type: `'integer'`

Timeout for the installation of the apk.

constraint: `value > 0`

default: `300`

version: type: `'str'`

The version of the package to be used.

variant: type: `'str'`

The variant of the package to be used.

strict: type: `'boolean'`

Whether to throw an error if the specified package cannot be found on host.

force_install: type: `'boolean'`

Always re-install the APK, even if matching version is found already installed on the device.

uninstall: type: `'boolean'`

If `True`, will uninstall workload's APK as part of teardown.'

exact_abi: type: `'boolean'`

If `True`, workload will check that the APK matches the target device ABI, otherwise any suitable APK found will be used.

prefer_host_package: type: `'boolean'`

If `True` then a package found on the host will be preferred if it is a valid version and ABI, if not it will fall back to the version on the target if available. If `False` then the version on the target is preferred instead.

aliases: `'check_apk'`

default: `True`

view: type: `'str'`

Manually override the 'View' of the workload for use with instruments such as the `fps` instrument. If not specified, a workload dependant 'View' will be automatically generated.

markers_enabled: type: `'boolean'`

If set to `True`, workloads will insert markers into logs at various points during execution. These markers may be used by other plugins or post-processing scripts to provide measurements or statistics for specific parts of the workload execution.

---

apache

Load-test an apache installation by issueing parallel requests with ab.

Run ab, the Apache benchmark on the host, directed at the target as the server.

Note: It is assumed that Apache is already running on target.

Note: Current implmentation only supports a very basic use of the benchmark.

---

aliases

ab

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

port: type: `'integer'`

Port on which Apache is running.

default: `80`

path: type: `'str'`

Path to request.

default: `'/'`

parallel_requests: type: `'integer'`

The number of parallel requests at a time.

default: `350`

total_requests: type: `'integer'`

The total number of parallel requests.

default: `100000`

---

applaunch

This workload launches and measures the launch time of applications for supporting workloads.

Currently supported workloads are the ones that implement `ApplaunchInterface`. For any workload to support this workload, it should implement the `ApplaunchInterface`. The corresponding java file of the workload associated with the application being measured is executed during the run. The application that needs to be measured is passed as a parameter `workload_name`. The parameters required for that workload have to be passed as a dictionary which is captured by the parameter `workload_params`. This information can be obtained by inspecting the workload details of the specific workload.

The workload allows to run multiple iterations of an application launch in two modes:

1. Launch from background
2. Launch from long-idle

These modes are captured as a parameter applaunch_type.

`launch_from_background` Launches an application after the application is sent to background by pressing Home button.

`launch_from_long-idle` Launches an application after killing an application process and clearing all the caches.

Test Description:

- During the initialization and setup, the application being launched is launched for the first time. The jar file of the workload of the application is moved to device at the location `workdir` which further implements the methods needed to measure the application launch time.
- **Run phase calls the UiAutomator of the applaunch which runs in two subphases.**
    A. **Applaunch Setup Run:** During this phase, welcome screens and dialogues during the first launch of the instrumented application are cleared.
    B. **Applaunch Metric Run:** During this phase, the application is launched multiple times determined by the iteration number specified by the parameter `applaunch_iterations`. Each of these iterations are instrumented to capture the launch time taken and the values are recorded as UX-PERF marker values in logfile.

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

package_name: type: `'str'`

The package name that can be used to specify the workload apk to use.

install_timeout: type: `'integer'`

Timeout for the installation of the apk.

constraint: `value > 0`

default: `300`

version: type: `'str'`

The version of the package to be used.

variant: type: `'str'`

The variant of the package to be used.

strict: type: `'boolean'`

Whether to throw an error if the specified package cannot be found on host.

force_install: type: `'boolean'`

Always re-install the APK, even if matching version is found already installed on the device.

uninstall: type: `'boolean'`

If `True`, will uninstall workload's APK as part of teardown.'

exact_abi: type: `'boolean'`

If `True`, workload will check that the APK matches the target device ABI, otherwise any suitable APK found will be used.

prefer_host_package: type: `'boolean'`

If `True` then a package found on the host will be preferred if it is a valid version and ABI, if not it will fall back to the version on the target if available. If `False` then the version on the target is preferred instead.

aliases: `'check_apk'`

default: `True`

view: type: `'str'`

Manually override the 'View' of the workload for use with instruments such as the `fps` instrument. If not specified, a workload dependant 'View' will be automatically generated.

markers_enabled: type: `'boolean'`

If set to `True`, workloads will insert markers into logs at various points during execution. These markers may be used by other plugins or post-processing scripts to provide measurements or statistics for specific parts of the workload execution.

workload_name: type: `'str'`

Name of the uxperf workload to launch

default: `'gmail'`

workload_params: type: `'OrderedDict'`

parameters of the uxperf workload whose application launch time is measured

applaunch_type: type: `'str'`

Choose launch_from_long-idle for measuring launch time from long-idle. These two types are described in the workload description.

allowed values: `'launch_from_background'`, `'launch_from_long-idle'`

default: `'launch_from_background'`

applaunch_iterations: type: `'integer'`

Number of iterations of the application launch

default: `1`

---

benchmarkpi

Measures the time the target device takes to run and complete the Pi calculation algorithm.

http://androidbenchmark.com/howitworks.php

from the website:

The whole idea behind this application is to use the same Pi calculation algorithm on every Android Device and check how fast that process is. Better calculation times, conclude to faster Android devices. This way you can also check how lightweight your custom made Android build is. Or not.

As Pi is an irrational number, Benchmark Pi does not calculate the actual Pi number, but an approximation near the first digits of Pi over the same calculation circles the algorithms needs.

So, the number you are getting in milliseconds is the time your mobile device takes to run and complete the Pi calculation algorithm resulting in a approximation of the first Pi digits.

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

package_name: type: `'str'`

The package name that can be used to specify the workload apk to use.

install_timeout: type: `'integer'`

Timeout for the installation of the apk.

constraint: `value > 0`

default: `300`

version: type: `'str'`

The version of the package to be used.

variant: type: `'str'`

The variant of the package to be used.

strict: type: `'boolean'`

Whether to throw an error if the specified package cannot be found on host.

force_install: type: `'boolean'`

Always re-install the APK, even if matching version is found already installed on the device.

uninstall: type: `'boolean'`

If `True`, will uninstall workload's APK as part of teardown.'

exact_abi: type: `'boolean'`

If `True`, workload will check that the APK matches the target device ABI, otherwise any suitable APK found will be used.

prefer_host_package: type: `'boolean'`

If `True` then a package found on the host will be preferred if it is a valid version and ABI, if not it will fall back to the version on the target if available. If `False` then the version on the target is preferred instead.

aliases: `'check_apk'`

default: `True`

view: type: `'str'`

Manually override the 'View' of the workload for use with instruments such as the `fps` instrument. If not specified, a workload dependant 'View' will be automatically generated.

markers_enabled: type: `'boolean'`

If set to `True`, workloads will insert markers into logs at various points during execution. These markers may be used by other plugins or post-processing scripts to provide measurements or statistics for specific parts of the workload execution.

---

chrome

A workload to perform standard Web browsing tasks with Google Chrome. The workload carries out a number of typical Web-based tasks, navigating through a handful of Wikipedia pages in multiple browser tabs.

To run the workload in offline mode, a `pages.tar` archive and an `OfflinePages.db` file are required. For users wishing to generate these files themselves, Chrome should first be operated from an Internet-connected environment and the following Wikipedia pages should be downloaded for offline use within Chrome:

- https://en.m.wikipedia.org/wiki/Main_Page
- https://en.m.wikipedia.org/wiki/United_States
- https://en.m.wikipedia.org/wiki/California

Following this, the files of interest for viewing these pages offline can be found in the `/data/data/com.android.chrome/app_chrome/Default/Offline Pages` directory. The `OfflinePages.db` file can be copied from the 'metadata' subdirectory, while the `*.mhtml` files that should make up the `pages.tar` file can be found in the 'archives' subdirectory. These page files can then be archived to produce a tarball using a command such as `tar -cvf pages.tar -C /path/to/archives .`. Both this and `OfflinePages.db` should then be placed in the `~/.workload_automation/dependencies/chrome/` directory on your local machine, creating this if it does not already exist.

Known working APK version: 65.0.3325.109

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

package_name: type: `'str'`

The package name that can be used to specify the workload apk to use.

install_timeout: type: `'integer'`

Timeout for the installation of the apk.

constraint: `value > 0`

default: `300`

version: type: `'str'`

The version of the package to be used.

variant: type: `'str'`

The variant of the package to be used.

strict: type: `'boolean'`

Whether to throw an error if the specified package cannot be found on host.

force_install: type: `'boolean'`

Always re-install the APK, even if matching version is found already installed on the device.

uninstall: type: `'boolean'`

If `True`, will uninstall workload's APK as part of teardown.'

exact_abi: type: `'boolean'`

If `True`, workload will check that the APK matches the target device ABI, otherwise any suitable APK found will be used.

prefer_host_package: type: `'boolean'`

If `True` then a package found on the host will be preferred if it is a valid version and ABI, if not it will fall back to the version on the target if available. If `False` then the version on the target is preferred instead.

aliases: `'check_apk'`

default: `True`

view: type: `'str'`

Manually override the 'View' of the workload for use with instruments such as the `fps` instrument. If not specified, a workload dependant 'View' will be automatically generated.

markers_enabled: type: `'boolean'`

If set to `True`, workloads will insert markers into logs at various points during execution. These markers may be used by other plugins or post-processing scripts to provide measurements or statistics for specific parts of the workload execution.

offline_mode: type: `'boolean'`

If set to `True`, the workload will execute in offline mode. This mode requires root and makes use of a tarball of *.mhtml files 'pages.tar' and an metadata database 'OfflinePages.db'. The tarball is extracted directly to the application's offline pages 'archives' directory, while the database is copied to the offline pages 'metadata' directory.

---

deepbench

Benchmarks operations that are important to deep learning. Including GEMM and convolution.

The benchmark and its documentation are available here:

https://github.com/baidu-research/DeepBench

Note: parameters of matrices used in each sub-test are added as classifiers to the metrics. See the benchmark documentation for the explanation of the various parameters

---

Note: at the moment only the "Arm Benchmarks" subset of DeepBench is supported.

---

aliases

deep-gemm test=‘‘'gemm'‘‘

deep-conv test=‘‘'conv'‘‘

deep-sparse test=‘‘'sparse'‘‘

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

test: type: `'str'`

Specifies which of the available benchmarks will be run.

gemm Performs GEneral Matrix Multiplication of dense matrices of varying sizes.

conv Performs convolutions on inputs in NCHW format.

sparse Performs GEneral Matrix Multiplication of sparse matrices of varying sizes, and compares them to corresponding dense operations.

allowed values: `'gemm'`, `'conv'`, `'sparse'`

default: `'gemm'`

---

dhrystone

Runs the Dhrystone benchmark.

Original source from:

```
http://classes.soe.ucsc.edu/cmpe202/benchmarks/standard/dhrystone.c
```

This version has been modified to configure duration and the number of threads used.

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

duration: type: `'integer'`

The duration, in seconds, for which dhrystone will be executed. Either this or `mloops` should be specified but not both.

mloops: type: `'integer'`

Millions of loops to run. Either this or `duration` should be specified, but not both. If neither is specified, this will default ‘ to `100`

threads: type: `'integer'`

The number of separate dhrystone “threads” that will be forked.

default: `4`

delay: type: `'integer'`

The delay, in seconds, between kicking off of dhrystone threads (if `threads` > 1).

cpus: type: `'cpu_mask'`

The processes spawned by dhrystone will be pinned to cores as specified by this parameter. The mask can be specified directly as a mask, as a list of cpus or a sysfs- style string

aliases: `'taskset_mask'`

---

exoplayer

Android ExoPlayer

ExoPlayer is the basic video player library that is used by the YouTube android app. The aim of this workload is to test a proxy for YouTube performance on targets where running the real YouTube app is not possible due its dependencies.

ExoPlayer sources: https://github.com/google/ExoPlayer

The 'demo' application is used by this workload. It can easily be built by loading the ExoPlayer sources into Android Studio.

Version r2.4.0 built from commit d979469 is known to work

Produces a metric 'exoplayer_dropped_frames' - this is the count of frames that Exoplayer itself reports as dropped. This is not the same thing as the dropped frames reported by gfxinfo.

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

package_name: type: `'str'`

The package name that can be used to specify the workload apk to use.

install_timeout: type: `'integer'`

Timeout for the installation of the apk.

constraint: `value > 0`

default: `300`

version: type: `'str'`

The version of the package to be used.

allowed values: `'2.4'`, `'2.5'`, `'2.6'`

variant: type: `'str'`

The variant of the package to be used.

strict: type: `'boolean'`

Whether to throw an error if the specified package cannot be found on host.

force_install: type: `'boolean'`

Always re-install the APK, even if matching version is found already installed on the device.

uninstall: type: `'boolean'`

If `True`, will uninstall workload's APK as part of teardown.'

exact_abi: type: 'boolean'

If True, workload will check that the APK matches the target device ABI, otherwise any suitable APK found will be used.

prefer_host_package: type: 'boolean'

If True then a package found on the host will be preferred if it is a valid version and ABI, if not it will fall back to the version on the target if available. If False then the version on the target is preferred instead.

aliases: 'check_apk'

default: True

view: type: 'str'

Manually override the 'View' of the workload for use with instruments such as the fps instrument. If not specified, a workload dependant 'View' will be automatically generated.

duration: type: 'integer'

Playback duration of the video file. This becomes the duration of the workload. If provided must be shorter than the length of the media.

default: 20

format: type: 'str'

Specifies which format video file to play. Default is mov_720p

allowed values: 'ogg_128kbps', 'mov_480p', 'mov_720p', 'mp4_1080p'

filename: type: 'str'

The name of the video file to play. This can be either a path to the file anywhere on your file system, or it could be just a name, in which case, the workload will look for it in /home/docs/.workload_automation/dependencies/exoplayer *Note*: either format or filename should be specified, but not both!

force_dependency_push: type: 'boolean'

If true, video will always be pushed to device, regardless of whether the file is already on the device. Default is False.

landscape: type: 'boolean'

Configure the screen in landscape mode, otherwise ensure portrait orientation by default. Default is False.

---

geekbench

Geekbench provides a comprehensive set of benchmarks engineered to quickly and accurately measure processor and memory performance.

http://www.primatelabs.com/geekbench/ From the website: Designed to make benchmarks easy to run and easy to understand, Geekbench takes the guesswork out of producing robust and reliable benchmark results. Geekbench scores are calibrated against a baseline score of 1,000 (which is the score of a single-processor Power Mac G5 @ 1.6GHz). Higher scores are better, with double the score indicating double the performance.

The benchmarks fall into one of four categories:

- integer performance.
- floating point performance.

- memory performance.
- stream performance.

Geekbench benchmarks: http://www.primatelabs.com/geekbench/doc/benchmarks.html Geekbench scoring methedology: http://support.primatelabs.com/kb/geekbench/interpreting-geekbench-scores

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

package_name: type: `'str'`

The package name that can be used to specify the workload apk to use.

install_timeout: type: `'integer'`

Timeout for the installation of the apk.

constraint: `value > 0`

default: `300`

version: type: `'str'`

Specifies which version of the workload should be run.

allowed values: `'2'`, `'3.0.0'`, `'3.4.1'`, `'4.0.1'`, `'4.2.0'`, `'4.3.1'`

variant: type: `'str'`

The variant of the package to be used.

strict: type: `'boolean'`

Whether to throw an error if the specified package cannot be found on host.

force_install: type: `'boolean'`

Always re-install the APK, even if matching version is found already installed on the device.

uninstall: type: `'boolean'`

If `True`, will uninstall workload's APK as part of teardown.'

exact_abi: type: `'boolean'`

If `True`, workload will check that the APK matches the target device ABI, otherwise any suitable APK found will be used.

prefer_host_package: type: `'boolean'`

If `True` then a package found on the host will be preferred if it is a valid version and ABI, if not it will fall back to the version on the target if available. If `False` then the version on the target is preferred instead.

aliases: `'check_apk'`

default: `True`

view: type: `'str'`

Manually override the 'View' of the workload for use with instruments such as the `fps` instrument. If not specified, a workload dependant 'View' will be automatically generated.

markers_enabled: type: `'boolean'`

If set to `True`, workloads will insert markers into logs at various points during execution. These markers may be used by other plugins or post-processing scripts to provide measurements or statistics for specific parts of the workload execution.

loops: type: `'integer'`

Specfies the number of times the benchmark will be run in a "tight loop", i.e. without performaing setup/teardown inbetween.

aliases: `'times'`

default: `1`

timeout: type: `'integer'`

Timeout for a single iteration of the benchmark. This value is multiplied by `times` to calculate the overall run timeout.

default: `3600`

disable_update_result: type: `'boolean'`

If `True` the results file will not be pulled from the targets `/data/data/com.primatelabs.geekbench` folder. This allows the workload to be run on unrooted targets and the results extracted manually later.

---

geekbench-corporate

Geekbench provides a comprehensive set of benchmarks engineered to quickly and accurately measure processor and memory performance.

http://www.primatelabs.com/geekbench/ From the website: Designed to make benchmarks easy to run and easy to understand, Geekbench takes the guesswork out of producing robust and reliable benchmark results. Geekbench scores are calibrated against a baseline score of 1,000 (which is the score of a single-processor Power Mac G5 @ 1.6GHz). Higher scores are better, with double the score indicating double the performance.

The benchmarks fall into one of four categories:

- integer performance.
- floating point performance.
- memory performance.
- stream performance.

Geekbench benchmarks: http://www.primatelabs.com/geekbench/doc/benchmarks.html Geekbench scoring methedology: http://support.primatelabs.com/kb/geekbench/interpreting-geekbench-scores

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: 'clean_up'

global alias: 'cleanup_assets'

default: True

package_name: type: 'str'

The package name that can be used to specify the workload apk to use.

install_timeout: type: 'integer'

Timeout for the installation of the apk.

constraint: value > 0

default: 300

version: type: 'str'

Specifies which version of the workload should be run.

allowed values: '4.1.0', '5.0.0'

variant: type: 'str'

The variant of the package to be used.

strict: type: 'boolean'

Whether to throw an error if the specified package cannot be found on host.

force_install: type: 'boolean'

Always re-install the APK, even if matching version is found already installed on the device.

uninstall: type: 'boolean'

If True, will uninstall workload's APK as part of teardown.'

exact_abi: type: 'boolean'

If True, workload will check that the APK matches the target device ABI, otherwise any suitable APK found will be used.

prefer_host_package: type: 'boolean'

If True then a package found on the host will be preferred if it is a valid version and ABI, if not it will fall back to the version on the target if available. If False then the version on the target is preferred instead.

aliases: 'check_apk'

default: True

view: type: 'str'

Manually override the 'View' of the workload for use with instruments such as the fps instrument. If not specified, a workload dependant 'View' will be automatically generated.

markers_enabled: type: 'boolean'

If set to True, workloads will insert markers into logs at various points during execution. These markers may be used by other plugins or post-processing scripts to provide measurements or statistics for specific parts of the workload execution.

loops: type: 'integer'

Specfies the number of times the benchmark will be run in a "tight loop", i.e. without performaing setup/teardown inbetween.

aliases: `'times'`

default: `1`

timeout: type: `'integer'`

Timeout for a single iteration of the benchmark. This value is multiplied by `times` to calculate the overall run timeout.

default: `3600`

disable_update_result: type: `'boolean'`

If `True` the results file will not be pulled from the targets `/data/data/com.primatelabs.geekbench` folder. This allows the workload to be run on unrooted targets and the results extracted manually later.

---

gfxbench-corporate

Execute a subset of graphical performance benchmarks

Test description: 1. Open the gfxbench application 2. Execute Car Chase, Manhattan and Tessellation benchmarks

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

package_name: type: `'str'`

The package name that can be used to specify the workload apk to use.

install_timeout: type: `'integer'`

Timeout for the installation of the apk.

constraint: `value > 0`

default: `300`

version: type: `'str'`

The version of the package to be used.

variant: type: `'str'`

The variant of the package to be used.

strict: type: `'boolean'`

Whether to throw an error if the specified package cannot be found on host.

force_install: type: `'boolean'`

Always re-install the APK, even if matching version is found already installed on the device.

uninstall: type: `'boolean'`

If `True`, will uninstall workload's APK as part of teardown.'

exact_abi: type: `'boolean'`

If `True`, workload will check that the APK matches the target device ABI, otherwise any suitable APK found will be used.

prefer_host_package: type: `'boolean'`

If `True` then a package found on the host will be preferred if it is a valid version and ABI, if not it will fall back to the version on the target if available. If `False` then the version on the target is preferred instead.

aliases: `'check_apk'`

default: `True`

view: type: `'str'`

Manually override the 'View' of the workload for use with instruments such as the `fps` instrument. If not specified, a workload dependant 'View' will be automatically generated.

markers_enabled: type: `'boolean'`

If set to `True`, workloads will insert markers into logs at various points during execution. These markers may be used by other plugins or post-processing scripts to provide measurements or statistics for specific parts of the workload execution.

timeout: type: `'integer'`

Timeout for an iteration of the benchmark.

default: `3600`

---

glbenchmark

Measures the graphics performance of Android devices by testing the underlying OpenGL (ES) implementation.

http://gfxbench.com/about-gfxbench.jsp

From the website:

> The benchmark includes console-quality high-level 3D animations (T-Rex HD and Egypt HD) and low-level graphics measurements.
>
> With high vertex count and complex effects such as motion blur, parallax mapping and particle systems, the engine of GFXBench stresses GPUs in order provide users a realistic feedback on their device.

aliases

glbench

egypt use_case='egypt'

t-rex use_case='t-rex'

egypt_onscreen use_case='egypt', type='onscreen'

t-rex_onscreen use_case='t-rex', type='onscreen'

egypt_offscreen use_case='egypt', type='offscreen'

t-rex_offscreen use_case=‘‘’t-rex’, `type`=’offscreen’‘‘

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

package_name: type: `'str'`

The package name that can be used to specify the workload apk to use.

install_timeout: type: `'integer'`

Timeout for the installation of the apk.

constraint: `value > 0`

default: `300`

version: type: `'str'`

Specifies which version of the benchmark to run (different versions support different use cases).

allowed values: `'2.7'`, `'2.5'`

variant: type: `'str'`

The variant of the package to be used.

strict: type: `'boolean'`

Whether to throw an error if the specified package cannot be found on host.

force_install: type: `'boolean'`

Always re-install the APK, even if matching version is found already installed on the device.

uninstall: type: `'boolean'`

If `True`, will uninstall workload’s APK as part of teardown.’

exact_abi: type: `'boolean'`

If `True`, workload will check that the APK matches the target device ABI, otherwise any suitable APK found will be used.

prefer_host_package: type: `'boolean'`

If `True` then a package found on the host will be preferred if it is a valid version and ABI, if not it will fall back to the version on the target if available. If `False` then the version on the target is preferred instead.

aliases: `'check_apk'`

default: `True`

view: type: `'str'`

Manually override the ‘View’ of the workload for use with instruments such as the `fps` instrument. If not specified, a workload dependant ‘View’ will be automatically generated.

markers_enabled: type: 'boolean'

If set to True, workloads will insert markers into logs at various points during execution. These markers may be used by other plugins or post-processing scripts to provide measurements or statistics for specific parts of the workload execution.

use_case: type: 'str'

Specifies which usecase to run, as listed in the benchmark menu; e.g. 'GLBenchmark 2.5 Egypt HD'. For convenience, two aliases are provided for the most common use cases: 'egypt' and 't-rex'. These could be use instead of the full use case title. For version '2.7' it defaults to 't-rex', for version '2.5' it defaults to 'egypt-classic'.

type: type: 'str'

Specifies which type of the use case to run, as listed in the benchmarks menu (small text underneath the use case name); e.g. 'C24Z16 Onscreen Auto'. For convenience, two aliases are provided for the most common types: 'onscreen' and 'offscreen'. These may be used instead of full type names.

default: 'onscreen'

timeout: type: 'integer'

Specifies how long, in seconds, UI automation will wait for results screen to appear before assuming something went wrong.

default: 200

---

gmail

A workload to perform standard productivity tasks within Gmail. The workload carries out various tasks, such as creating new emails, attaching images and sending them.

Test description: 1. Open Gmail application 2. Click to create New mail 3. Attach an image from the local images folder to the email 4. Enter recipient details in the To field 5. Enter text in the Subject field 6. Enter text in the Compose field 7. Click the Send mail button

To run the workload in offline mode, a 'mailstore.tar' file is required. In order to generate such a file, Gmail should first be operated from an Internet-connected environment. After this, the relevant database files can be found in the '/data/data/com.google.android.gm/databases' directory. These files can then be archived to produce a tarball using a command such as tar -cvf mailstore.tar -C /path/to/databases .. The result should then be placed in the '~/.workload_automation/dependencies/gmail/' directory on your local machine, creating this if it does not already exist.

Known working APK version: 7.11.5.176133587

parameters

cleanup_assets: type: 'boolean'

If True, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: 'clean_up'

global alias: 'cleanup_assets'

default: True

package_name: type: 'str'

The package name that can be used to specify the workload apk to use.

install_timeout: type: 'integer'

Timeout for the installation of the apk.

constraint: value > 0

default: 300

version: type: 'str'

The version of the package to be used.

variant: type: 'str'

The variant of the package to be used.

strict: type: 'boolean'

Whether to throw an error if the specified package cannot be found on host.

force_install: type: 'boolean'

Always re-install the APK, even if matching version is found already installed on the device.

uninstall: type: 'boolean'

If True, will uninstall workload's APK as part of teardown.'

exact_abi: type: 'boolean'

If True, workload will check that the APK matches the target device ABI, otherwise any suitable APK found will be used.

prefer_host_package: type: 'boolean'

If True then a package found on the host will be preferred if it is a valid version and ABI, if not it will fall back to the version on the target if available. If False then the version on the target is preferred instead.

aliases: 'check_apk'

default: True

view: type: 'str'

Manually override the 'View' of the workload for use with instruments such as the fps instrument. If not specified, a workload dependant 'View' will be automatically generated.

markers_enabled: type: 'boolean'

If set to True, workloads will insert markers into logs at various points during execution. These markers may be used by other plugins or post-processing scripts to provide measurements or statistics for specific parts of the workload execution.

recipient: type: 'str'

The email address of the recipient. Setting a void address will stop any mesage failures clogging up your device inbox

default: 'wa-devnull@mailinator.com'

test_image: type: 'str'

An image to be copied onto the device that will be attached to the email

default: 'uxperf_1600x1200.jpg'

offline_mode: type: `'boolean'`

If set to `True`, the workload will execute in offline mode. This mode requires root and makes use of a tarball of email database files 'mailstore.tar' for the email account to be used. This file is extracted directly to the application's 'databases' directory at '/data/data/com.google.android.gm/databases'.

---

googlemaps

A workload to perform standard navigation tasks with Google Maps. This workload searches for known locations, pans and zooms around the map, and follows driving directions along a route.

To run the workload in offline mode, `databases.tar` and `files.tar` archives are required. In order to generate these files, Google Maps should first be operated from an Internet-connected environment, and a region around Cambridge, England should be downloaded for offline use. This region must include the landmarks used in the UIAutomator program, which include Cambridge train station and Corpus Christi college.

Following this, the files of interest can be found in the `databases` and `files` subdirectories of the `/data/data/com.google.android.apps.maps/` directory. The contents of these subdirectories can be archived into tarballs using commands such as `tar -cvf databases.tar -C /path/to/databases .`. These `databases.tar` and `files.tar` archives should then be placed in the `~/.workload_automation/dependencies/googlemaps` directory on your local machine, creating this if it does not already exist.

Known working APK version: 9.72.2

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

package_name: type: `'str'`

The package name that can be used to specify the workload apk to use.

install_timeout: type: `'integer'`

Timeout for the installation of the apk.

constraint: `value > 0`

default: `300`

version: type: `'str'`

The version of the package to be used.

variant: type: `'str'`

The variant of the package to be used.

strict: type: `'boolean'`

Whether to throw an error if the specified package cannot be found on host.

force_install: type: 'boolean'

Always re-install the APK, even if matching version is found already installed on the device.

uninstall: type: 'boolean'

If True, will uninstall workload's APK as part of teardown.'

exact_abi: type: 'boolean'

If True, workload will check that the APK matches the target device ABI, otherwise any suitable APK found will be used.

prefer_host_package: type: 'boolean'

If True then a package found on the host will be preferred if it is a valid version and ABI, if not it will fall back to the version on the target if available. If False then the version on the target is preferred instead.

aliases: 'check_apk'

default: True

view: type: 'str'

Manually override the 'View' of the workload for use with instruments such as the fps instrument. If not specified, a workload dependant 'View' will be automatically generated.

markers_enabled: type: 'boolean'

If set to True, workloads will insert markers into logs at various points during execution. These markers may be used by other plugins or post-processing scripts to provide measurements or statistics for specific parts of the workload execution.

offline_mode: type: 'boolean'

If set to True, the workload will execute in offline mode. This mode requires root and makes use of a tarball of database files databases.tar and a tarball of auxiliary files files.tar. These tarballs are extracted directly to the application's databases and files directories respectively in /data/data/com.google.android.apps.maps/.

---

googlephotos

A workload to perform standard productivity tasks with Google Photos. The workload carries out various tasks, such as browsing images, performing zooms, and post-processing the image.

Test description:

1. Four images are copied to the target
2. The application is started in offline access mode
3. Gestures are performed to pinch zoom in and out of the selected image
4. The colour of a selected image is edited by selecting the colour menu, incrementing the colour, resetting the colour and decrementing the colour using the seek bar.
5. A crop test is performed on a selected image. UiAutomator does not allow the selection of the crop markers so the image is tilted positively, reset and then tilted negatively to get a similar cropping effect.
6. A rotate test is performed on a selected image, rotating anticlockwise 90 degrees, 180 degrees and 270 degrees.

Known working APK version: 4.0.0.212659618

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

package_name: type: `'str'`

The package name that can be used to specify the workload apk to use.

install_timeout: type: `'integer'`

Timeout for the installation of the apk.

constraint: `value > 0`

default: `300`

version: type: `'str'`

The version of the package to be used.

variant: type: `'str'`

The variant of the package to be used.

strict: type: `'boolean'`

Whether to throw an error if the specified package cannot be found on host.

force_install: type: `'boolean'`

Always re-install the APK, even if matching version is found already installed on the device.

uninstall: type: `'boolean'`

If `True`, will uninstall workload's APK as part of teardown.'

exact_abi: type: `'boolean'`

If `True`, workload will check that the APK matches the target device ABI, otherwise any suitable APK found will be used.

prefer_host_package: type: `'boolean'`

If `True` then a package found on the host will be preferred if it is a valid version and ABI, if not it will fall back to the version on the target if available. If `False` then the version on the target is preferred instead.

aliases: `'check_apk'`

default: `True`

view: type: `'str'`

Manually override the 'View' of the workload for use with instruments such as the `fps` instrument. If not specified, a workload dependant 'View' will be automatically generated.

markers_enabled: type: `'boolean'`

If set to `True`, workloads will insert markers into logs at various points during execution. These markers may be used by other plugins or post-processing scripts to provide measurements or statistics for specific parts of the workload execution.

test_images: type: `'list_of_strs'`

A list of four JPEG and/or PNG files to be pushed to the target. Absolute file paths may be used but tilde expansion must be escaped.

constraint: `len(unique(value)) == 4`

default: `['uxperf_1200x1600.png', 'uxperf_1600x1200.jpg', 'uxperf_2448x3264.png', 'uxperf_3264x2448.jpg']`

---

googleplaybooks

A workload to perform standard productivity tasks with googleplaybooks. This workload performs various tasks, such as searching for a book title online, browsing through a book, adding and removing notes, word searching, and querying information about the book.

Test description: 1. Open Google Play Books application 2. Dismisses sync operation (if applicable) 3. Searches for a book title 4. Adds books to library if not already present 5. Opens 'My Library' contents 6. Opens selected book 7. Gestures are performed to swipe between pages and pinch zoom in and out of a page 8. Selects a specified chapter based on page number from the navigation view 9. Selects a word in the centre of screen and adds a test note to the page 10. Removes the test note from the page (clean up) 11. Searches for the number of occurrences of a common word throughout the book 12. Switches page styles from 'Day' to 'Night' to 'Sepia' and back to 'Day' 13. Uses the 'About this book' facility on the currently selected book

NOTE: This workload requires a network connection (ideally, wifi) to run, a Google account to be setup on the device, and payment details for the account. Free books require payment details to have been setup otherwise it fails. Tip: Install the 'Google Opinion Rewards' app to bypass the need to enter valid card/bank detail.

Known working APK version: 3.15.5

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

package_name: type: `'str'`

The package name that can be used to specify the workload apk to use.

install_timeout: type: `'integer'`

Timeout for the installation of the apk.

constraint: `value > 0`

default: `300`

version: type: `'str'`

The version of the package to be used.

variant: type: 'str'

The variant of the package to be used.

strict: type: 'boolean'

Whether to throw an error if the specified package cannot be found on host.

force_install: type: 'boolean'

Always re-install the APK, even if matching version is found already installed on the device.

uninstall: type: 'boolean'

If True, will uninstall workload's APK as part of teardown.'

exact_abi: type: 'boolean'

If True, workload will check that the APK matches the target device ABI, otherwise any suitable APK found will be used.

prefer_host_package: type: 'boolean'

If True then a package found on the host will be preferred if it is a valid version and ABI, if not it will fall back to the version on the target if available. If False then the version on the target is preferred instead.

aliases: 'check_apk'

default: True

view: type: 'str'

Manually override the 'View' of the workload for use with instruments such as the fps instrument. If not specified, a workload dependant 'View' will be automatically generated.

markers_enabled: type: 'boolean'

If set to True, workloads will insert markers into logs at various points during execution. These markers may be used by other plugins or post-processing scripts to provide measurements or statistics for specific parts of the workload execution.

search_book_title: type: 'str'

The book title to search for within Google Play Books archive. The book must either be already in the account's library, or free to purchase.

default: 'Nikola Tesla: Imagination and the Man That Invented the 20th Century'

library_book_title: type: 'str'

The book title to search for within My Library. The Library name can differ (usually shorter) to the Store name. If left blank, the search_book_title will be used.

default: 'Nikola Tesla'

select_chapter_page_number: type: 'integer'

The Page Number to search for within a selected book's Chapter list. Note: Accepts integers only.

default: 4

search_word: type: 'str'

The word to search for within a selected book. Note: Accepts single words only.

default: 'the'

account: type: `'str'`

If you are running this workload on a device which has more than one Google account setup, then this parameter is used to select which account to select when prompted. The account requires the book to have already been purchased or payment details already associated with the account. If omitted, the first account in the list will be selected if prompted.

---

googleslides

A workload to perform standard productivity tasks with Google Slides. The workload carries out various tasks, such as creating a new presentation, adding text, images, and shapes, as well as basic editing and playing a slideshow. This workload should be able to run without a network connection.

There are two main scenarios:

1. create test: a presentation is created in-app and some editing done on it,
2. load test: a pre-existing PowerPoint file is copied onto the device for testing.

— create — Create a new file in the application and perform basic editing on it. This test also requires an image file specified by the param `test_image` to be copied onto the device.

Test description:

1. Start the app and skip the welcome screen. Dismiss the work offline banner if present.
2. Go to the app settings page and enables PowerPoint compatibility mode. This allows PowerPoint files to be created inside Google Slides.
3. Create a new PowerPoint presentation in the app (PPT compatibility mode) with a title slide and save it to device storage.
4. Insert another slide and to it insert the pushed image by picking it from the gallery.
5. Insert a final slide and add a shape to it. Resize and drag the shape to modify it.
6. Finally, navigate back to the documents list.

— load — Copy a PowerPoint presentation onto the device to test slide navigation. The PowerPoint file to be copied is given by `test_file`.

Test description:

1. From the documents list (following the create test), open the specified PowerPoint by navigating into device storage and wait for it to be loaded.
2. A navigation test is performed while the file is in editing mode (i.e. not slideshow). swiping forward to the next slide until `slide_count` swipes are performed.
3. While still in editing mode, the same action is done in the reverse direction back to the first slide.
4. Enter presentation mode by selecting to play the slideshow.
5. Swipe forward to play the slideshow, for a maximum number of `slide_count` swipes.
6. Finally, repeat the previous step in the reverse direction while still in presentation mode, navigating back to the first slide.

NOTE: There are known issues with the reliability of this workload on some targets. It MAY NOT ALWAYS WORK on your device. If you do run into problems, it might help to set `do_text_entry` parameter to `False`.

Known working APK version: 1.7.032.06

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

package_name: type: `'str'`

The package name that can be used to specify the workload apk to use.

install_timeout: type: `'integer'`

Timeout for the installation of the apk.

constraint: `value > 0`

default: `300`

version: type: `'str'`

The version of the package to be used.

variant: type: `'str'`

The variant of the package to be used.

strict: type: `'boolean'`

Whether to throw an error if the specified package cannot be found on host.

force_install: type: `'boolean'`

Always re-install the APK, even if matching version is found already installed on the device.

uninstall: type: `'boolean'`

If `True`, will uninstall workload's APK as part of teardown.'

exact_abi: type: `'boolean'`

If `True`, workload will check that the APK matches the target device ABI, otherwise any suitable APK found will be used.

prefer_host_package: type: `'boolean'`

If `True` then a package found on the host will be preferred if it is a valid version and ABI, if not it will fall back to the version on the target if available. If `False` then the version on the target is preferred instead.

aliases: `'check_apk'`

default: `True`

view: type: `'str'`

Manually override the 'View' of the workload for use with instruments such as the `fps` instrument. If not specified, a workload dependant 'View' will be automatically generated.

markers_enabled: type: `'boolean'`

If set to `True`, workloads will insert markers into logs at various points during execution. These markers may be used by other plugins or post-processing scripts to provide measurements or statistics for specific parts of the workload execution.

test_image: type: 'str'

An image to be copied onto the device that will be embedded in the PowerPoint file as part of the test.

default: 'uxperf_1600x1200.jpg'

test_file: type: 'str'

If specified, the workload will copy the PowerPoint file to be used for testing onto the device. Otherwise, a file will be created inside the app.

default: 'uxperf_test_doc.pptx'

slide_count: type: 'integer'

Number of slides in aforementioned local file. Determines number of swipe actions when playing slide show.

default: 5

do_text_entry: type: 'boolean'

If set to True, will attempt to enter text in the first slide as part of the test. Currently seems to be problematic on some devices, most notably Samsung devices.

default: True

---

hackbench

Hackbench runs a series of tests for the Linux scheduler.

For details, go to: https://github.com/linux-test-project/ltp/

parameters

cleanup_assets: type: 'boolean'

If True, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: 'clean_up'

global alias: 'cleanup_assets'

default: True

timeout: type: 'integer'

Expected test duration in seconds.

aliases: 'duration'

default: 30

datasize: type: 'integer'

Message size in bytes.

default: 100

groups: type: 'integer'

Number of groups.

default: 10

loops: type: `'integer'`

Number of loops.

default: `100`

fds: type: `'integer'`

Number of file descriptors.

default: `40`

extra_params: type: `'str'`

Extra parameters to pass in. See the hackbench man page or type *hackbench –help* for list of options.

---

homescreen

A workload that goes to the home screen and idles for the the specified duration.

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

duration: type: `'integer'`

Specifies the duration, in seconds, of this workload.

default: `20`

---

hwuitest

Tests UI rendering latency on Android devices.

The binary for this workload is built as part of AOSP's frameworks/base/libs/hwui component.

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

test: type: `'caseless_string'`

The test to run:

- `'shadowgrid'`: creates a grid of rounded rects that cast shadows, high CPU & GPU load
- `'rectgrid'`: creates a grid of 1x1 rects
- `'oval'`: draws 1 oval

allowed values: `'shadowgrid'`, `'rectgrid'`, `'oval'`

default: `'shadowgrid'`

loops: type: `'integer'`

The number of test iterations.

default: `3`

frames: type: `'integer'`

The number of frames to run the test over.

default: `150`

---

idle

Do nothing for the specified duration.

On android devices, this may optionally stop the Android run time, if `stop_android` is set to `True`.

Note: This workload requires the device to be rooted.

---

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

duration: type: `'integer'`

Specifies the duration, in seconds, of this workload.

default: `20`

screen_off: type: `'boolean'`

Ensure that the screen is off before idling.

Note: Make sure screen lock is disabled on the target!

---

stop_android: type: `'boolean'`

Specifies whether the Android run time should be stopped. (Can be set only for Android devices).

---

jankbench

Internal Google benchmark for evaluating jank on Android.

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

package_name: type: `'str'`

The package name that can be used to specify the workload apk to use.

install_timeout: type: `'integer'`

Timeout for the installation of the apk.

constraint: `value > 0`

default: `300`

version: type: `'str'`

The version of the package to be used.

variant: type: `'str'`

The variant of the package to be used.

strict: type: `'boolean'`

Whether to throw an error if the specified package cannot be found on host.

force_install: type: `'boolean'`

Always re-install the APK, even if matching version is found already installed on the device.

uninstall: type: `'boolean'`

If `True`, will uninstall workload's APK as part of teardown.'

exact_abi: type: `'boolean'`

If `True`, workload will check that the APK matches the target device ABI, otherwise any suitable APK found will be used.

prefer_host_package: type: `'boolean'`

If `True` then a package found on the host will be preferred if it is a valid version and ABI, if not it will fall back to the version on the target if available. If `False` then the version on the target is preferred instead.

aliases: `'check_apk'`

default: `True`

view: type: `'str'`

Manually override the 'View' of the workload for use with instruments such as the `fps` instrument. If not specified, a workload dependant 'View' will be automatically generated.

test_ids: type: `'list_or_string'`

ID of the jankbench test to be run.

allowed values: `'list_view'`, `'image_list_view'`, `'shadow_grid'`, `'low_hitrate_text'`, `'high_hitrate_text'`, `'edit_text'`, `'overdraw_test'`

loops: type: `'integer'`

Specifies the number of times the benchmark will be run in a "tight loop", i.e. without performaing setup/teardown inbetween.

aliases: `'reps'`

constraint: `value > 0`

default: `1`

pull_results_db: type: `'boolean'`

Secifies whether an sqlite database with detailed results should be pulled from benchmark app's data. This requires the device to be rooted.

This defaults to `True` for rooted devices and `False` otherwise.

timeout: type: `'integer'`

Time out for workload execution. The workload will be killed if it hasn't completed within this period.

aliases: `'run_timeout'`

default: `600`

---

lmbench

Run a subtest from lmbench, a suite of portable ANSI/C microbenchmarks for UNIX/POSIX. In general, lmbench measures two key features: latency and bandwidth. This workload supports a subset of lmbench tests. lat_mem_rd can be used to measure latencies to memory (including caches). bw_mem can be used to measure bandwidth to/from memory over a range of operations. Further details, and source code are available from:

http://sourceforge.net/projects/lmbench/.

See lmbench/bin/README for license details.

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: 'cleanup_assets'

default: True

test: type: 'str'

Specifies an lmbench test to run.

allowed values: 'lat_mem_rd', 'bw_mem'

default: 'lat_mem_rd'

stride: type: 'list_or_type'

Stride for lat_mem_rd test. Workload will iterate over one or more integer values.

default: [128]

thrash: type: 'boolean'

Sets -t flag for lat_mem_rd_test

default: True

size: type: 'list_or_string'

Data set size for lat_mem_rd bw_mem tests.

default: '4m'

mem_category: type: 'list_or_string'

List of memory catetories for bw_mem test.

default: ('rd', 'wr', 'cp', 'frd', 'fwr', 'fcp', 'bzero', 'bcopy')

parallelism: type: 'integer'

Parallelism flag for tests that accept it.

warmup: type: 'integer'

Warmup flag for tests that accept it.

repetitions: type: 'integer'

Repetitions flag for tests that accept it.

force_abi: type: 'str'

Override device abi with this value. Can be used to force arm32 on 64-bit devices.

run_timeout: type: 'integer'

Timeout for execution of the test.

default: 900

loops: type: 'integer'

Specifies the number of times the benchmark will be run in a "tight loop", i.e. without performaing setup/teardown inbetween. This parameter is distinct from "repetitions", as the latter takes place within the benchmark and produces a single result.

constraint: value > 0

default: 1

cpus: type: `'cpu_mask'`

Specifies the CPU mask the benchmark process will be pinned to.

aliases: `'taskset_mask'`

---

manual

Yields control to the user, either for a fixed period or based on user input, to perform custom operations on the device, which workload automation does not know of.

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

duration: type: `'integer'`

Control of the devices is yielded for the duration (in seconds) specified. If not specified, `user_triggered` is assumed.

user_triggered: type: `'boolean'`

If `True`, WA will wait for user input after starting the workload; otherwise fixed duration is expected. Defaults to `True` if `duration` is not specified, and `False` otherwise.

view: type: `'str'`

Specifies the View of the workload. This enables instruments that require a View to be specified, such as the `fps` instrument. This is required for using "SurfaceFlinger" to collect FPS statistics and is primarily used on devices pre API level 23.

default: `'SurfaceView'`

package: type: `'str'`

Specifies the package name of the workload. This enables instruments that require a Package to be specified, such as the `fps` instrument. This allows for "gfxinfo" to be used and is the preferred method of collection for FPS statistics on devices API level 23+.

---

meabo

A multi-phased multi-purpose micro-benchmark. The micro-benchmark is composed of 10 phases that perform various generic calculations (from memory to compute intensive).

It is a highly configurable tool which can be used for energy efficiency studies, ARM big.LITTLE Linux scheduler analysis and DVFS studies. It can be used for other benchmarking as well.

All floating-point calculations are double-precision.

Phase 1: Floating-point & integer computations with good data locality
Phase 2: Vector multiplication & addition, 1 level of indirection in 1
source vector
Phase 3: Vector scalar addition and reductions
Phase 4: Vector addition
Phase 5: Vector addition, 1 level of indirection in both source vectors
Phase 6: Sparse matrix-vector multiplication
Phase 7: Linked-list traversal
Phase 8: Electrostatic force calculations
Phase 9: Palindrome calculations
Phase 10: Random memory accesses

For more details and benchmark source, see:

https://github.com/ARM-software/meabo

Note: current implementation of automation relies on the executable to be either statically linked or for all necessary depencies to be installed on the target.

---

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

array_size: type: `'integer'`

Size of arrays used in Phases 1, 2, 3, 4 and 5.

constraint: `value > 0`

default: `1048576`

num_rows: type: `'integer'`

Number of rows for the sparse matrix used in Phase 6.

aliases: `'nrow'`

constraint: `value > 0`

default: `16384`

num_cols: type: `'integer'`

Number of columns for the sparse matrix used in Phase 6.

aliases: `'ncol'`

constraint: `value > 0`

default: `16384`

loops: type: `'integer'`

Number of iterations that core loop is executed.

aliases: `'num_iterations'`

constraint: `value > 0`

default: `1000`

block_size: type: `'integer'`

Block size used in Phase 1.

constraint: `value > 0`

default: `8`

num_cpus: type: `'integer'`

Number of total CPUs that the application can bind threads to.

constraint: `value > 0`

default: `6`

per_phase_cpu_ids: type: `'list_of_ints'`

Sets which cores each phase is run on.

constraint: `all(v >= -1 for v in value)`

default: `[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1]`

num_hwcntrs: type: `'integer'`

Only available when using PAPI. Controls how many hardware counters PAPI will get access to.

constraint: `value >= 0`

default: `7`

run_phases: type: `'list_of_ints'`

Controls which phases to run.

constraint: `all(0 < v <= 10 for v in value)`

default: `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`

threads: type: `'integer'`

Controls how many threads the application will be using.

aliases: `'num_threads'`

constraint: `value >= 0`

bind_to_cpu_set: type: `'integer'`

Controls whether threads will be bound to a core set, or each individual thread will be bound to a specific core within the core set.

constraint: `0 <= value <= 1`

default: `1`

llist_size: type: `'integer'`

Size of the linked list available for each thread.

constraint: `value > 0`

default: `16777216`

num_particles: type: `'integer'`

Number of particles used in Phase 8.

constraint: `value > 0`

default: `1048576`

num_palindromes: type: `'integer'`

Number of palindromes used in Phase 9.

constraint: `value > 0`

default: `1024`

num_randomloc: type: `'integer'`

Number of random memory locations accessed in Phase 10.

constraint: `value > 0`

default: `2097152`

timeout: type: `'integer'`

Timeout for execution of the test.

aliases: `'run_timeout'`

constraint: `value > 0`

default: `2700`

---

memcpy

Runs memcpy in a loop.

This will run memcpy in a loop for a specified number of times on a buffer of a specified size. Additionally, the affinity of the test can be set to one or more specific cores.

This workload is single-threaded. It generates no scores or metrics by itself.

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

buffer_size: type: `'integer'`

Specifies the size, in bytes, of the buffer to be copied.

default: `5242880`

loops: type: `'integer'`

Specfies the number of iterations that will be performed.

aliases: `'iterations'`

default: `1000`

cpus: type: `'cpu_mask'`

The cpus for which the affinity of the test process should be set, specified as a mask, as a list of cpus or a sysfs-style string. If not specified, all available cores will be used.

---

mongoperf

A utility to check disk I/O performance independently of MongoDB.

It times tests of random disk I/O and presents the results. You can use mongoperf for any case apart from MongoDB. The mmf true mode is completely generic.

Note: `mongoperf` seems to ramp up threads in powers of two over a period of tens of seconds (there doesn't appear to be a way to change that). Bear this in mind when setting the `duration`.

---

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

duration: type: `'integer'`

Duration of of the workload.

default: `300`

threads: type: `'integer'`

Defines the number of threads mongoperf will use in the test. To saturate the system storage system you will need multiple threads.

default: `16`

file_size_mb: type: `'integer'`

Test file size in MB.

default: `1`

sleep_micros: type: `'integer'`

mongoperf will pause for this number of microseconds divided by the the number of threads between each operation.

mmf: type: `'boolean'`

When `True`, use memory mapped files for the tests. Generally:

- when mmf is `False`, mongoperf tests direct, physical, I/O, without caching. Use a large file size to test heavy random I/O load and to avoid I/O coalescing.
- when mmf is `True`, mongoperf runs tests of the caching system, and can use normal file system cache. Use mmf in this mode to test file system cache behavior with memory mapped files.

default: `True`

read: type: `'boolean'`

When `True`, perform reads as part of the test. Either `read` or `write` must be `True`.

aliases: `'r'`

default: `True`

write: type: `'boolean'`

When `True`, perform writes as part of the test. Either `read` or `write` must be `True`.

aliases: `'w'`

default: `True`

rec_size_kb: type: `'integer'`

The size of each write operation

default: `4`

sync_delay: type: `'integer'`

Seconds between disk flushes. Only use this if `mmf` is set to `True`.

---

openssl

Benchmark Openssl algorithms using Openssl's speed command.

The command tests how long it takes to perfrom typical SSL operations using a range of supported algorithms and ciphers.

By defalt, this workload will use openssl installed on the target, however it is possible to provide an alternative binary as a workload resource.

aliases

ossl-aes-128-cbc algorithm=‘‘’aes-128-cbc’‘‘

ossl-aes-192-cbc algorithm=‘‘’aes-192-cbc’‘‘

ossl-aes-256-cbc algorithm=‘‘’aes-256-cbc’‘‘

ossl-aes-128-gcm algorithm=‘‘’aes-128-gcm’‘‘

ossl-aes-192-gcm algorithm=‘‘’aes-192-gcm’‘‘

ossl-aes-256-gcm algorithm=‘‘’aes-256-gcm’‘‘

ossl-sha1 algorithm=‘‘’sha1’‘‘

ossl-sha256 algorithm=‘‘’sha256’‘‘

ossl-sha384 algorithm=‘‘’sha384’‘‘

ossl-sha512 algorithm=‘‘’sha512’‘‘

ossl-rsa algorithm=‘‘’rsa’‘‘

ossl-dsa algorithm=‘‘’dsa’‘‘

ossl-ecdh algorithm=‘‘’ecdh’‘‘

ossl-ecdsa algorithm=‘‘’ecdsa’‘‘

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

algorithm: type: `'str'`

Algorithm to benchmark.

allowed values: `'aes-128-cbc'`, `'aes-192-cbc'`, `'aes-256-cbc'`, `'aes-128-gcm'`, `'aes-192-gcm'`, `'aes-256-gcm'`, `'sha1'`, `'sha256'`, `'sha384'`, `'sha512'`, `'rsa'`, `'dsa'`, `'ecdh'`, `'ecdsa'`

default: `'aes-256-cbc'`

threads: type: `'integer'`

The number of threads to use

default: `1`

use_system_binary: type: `'boolean'`

If `True`, the system Openssl binary will be used. Otherwise, use the binary provided in the workload resources.

default: `True`

---

pcmark

A workload to execute the Work 2.0 benchmarks within PCMark - https://www.futuremark.com/benchmarks/pcmark-android

Test description: 1. Open PCMark application 2. Swipe right to the Benchmarks screen 3. Select the Work 2.0 benchmark 4. Install the Work 2.0 benchmark 5. Execute the Work 2.0 benchmark

Known working APK version: 2.0.3716

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

package_name: type: `'str'`

The package name that can be used to specify the workload apk to use.

install_timeout: type: `'integer'`

Timeout for the installation of the apk.

constraint: `value > 0`

default: `300`

version: type: `'str'`

The version of the package to be used.

variant: type: `'str'`

The variant of the package to be used.

strict: type: `'boolean'`

Whether to throw an error if the specified package cannot be found on host.

force_install: type: `'boolean'`

Always re-install the APK, even if matching version is found already installed on the device.

uninstall: type: `'boolean'`

If `True`, will uninstall workload's APK as part of teardown.'

exact_abi: type: `'boolean'`

If `True`, workload will check that the APK matches the target device ABI, otherwise any suitable APK found will be used.

prefer_host_package: type: `'boolean'`

If `True` then a package found on the host will be preferred if it is a valid version and ABI, if not it will fall back to the version on the target if available. If `False` then the version on the target is preferred instead.

aliases: `'check_apk'`

default: `True`

view: type: `'str'`

Manually override the 'View' of the workload for use with instruments such as the `fps` instrument. If not specified, a workload dependant 'View' will be automatically generated.

markers_enabled: type: `'boolean'`

If set to `True`, workloads will insert markers into logs at various points during execution. These markers may be used by other plugins or post-processing scripts to provide measurements or statistics for specific parts of the workload execution.

recentfling

Tests UI jank on android devices.

For this workload to work, `recentfling.sh` and `defs.sh` must be placed in `~/.workload_automation/dependencies/recentfling/`. These can be found in the AOSP Git repository.

To change the apps that are opened at the start of the workload you will need to modify the `defs.sh` file. You will need to add your app to `dfltAppList` and then add a variable called `{app_name}Activity` with the name of the activity to launch (where `{add_name}` is the name you put into `dfltAppList`).

You can get a list of activities available on your device by running `adb shell pm list packages -f`

parameters

cleanup_assets: type: `'boolean'`

> If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.
>
> aliases: `'clean_up'`
>
> global alias: `'cleanup_assets'`
>
> default: `True`

loops: type: `'integer'`

> The number of test iterations.
>
> default: `3`

start_apps: type: `'boolean'`

> If set to `False`,no apps will be started before flinging through the recent apps list (in which the assumption is there are already recently started apps in the list.
>
> default: `True`

device_name: type: `'str'`

> If set, recentfling will use the fling parameters for this device instead of automatically guessing the device. This can also be used if the device is not supported by recentfling, but its screensize is similar to that of one that is supported.
>
> For possible values, check your recentfling.sh. At the time of writing, valid values are: 'shamu', 'hammerhead', 'angler', 'ariel', 'mtp8996', 'bullhead' or 'volantis'.

---

rt-app

A test application that simulates configurable real-time periodic load.

rt-app is a test application that starts multiple periodic threads in order to simulate a real-time periodic load. It supports SCHED_OTHER, SCHED_FIFO, SCHED_RR as well as the AQuoSA framework and SCHED_DEADLINE.

The load is described using JSON-like config files. Below are a couple of simple examples.

Simple use case which creates a thread that run 1ms then sleep 9ms until the use case is stopped with Ctrl+C:

```
{
    "tasks" : {
        "thread0" : {
            "loop" : -1,
            "run" :   20000,
            "sleep" : 80000
        }
    },
    "global" : {
        "duration" : 2,
        "calibration" : "CPU0",
        "default_policy" : "SCHED_OTHER",
        "pi_enabled" : false,
        "lock_pages" : false,
        "logdir" : "./",
        "log_basename" : "rt-app1",
        "ftrace" : false,
        "gnuplot" : true,
    }
}
```

Simple use case with 2 threads that runs for 10 ms and wake up each other until the use case is stopped with Ctrl+C

```
{
    "tasks" : {
        "thread0" : {
            "loop" : -1,
            "run" :     10000,
            "resume" : "thread1",
            "suspend" : "thread0"
        },
        "thread1" : {
            "loop" : -1,
            "run" :     10000,
            "resume" : "thread0",
            "suspend" : "thread1"
        }
    }
}
```

Please refer to the existing configs in `$WA_ROOT/wa/workloads/rt_app/use_case` for more examples.

The upstream version of rt-app is hosted here:

https://github.com/scheduler-tools/rt-app

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

config: type: `'str'`

Use case configuration file to run with rt-app. This may be either the name of one of the "standard" configurations included with the workload. or a path to a custom JSON file provided by the user. Either way, the ".json" extension is implied and will be added automatically if not specified in the argument.

The following is the list of standard configurations currently included with the workload: browser-long.json, video-long.json, browser-short.json, spreading-tasks.json, mp3-long.json, taskset.json, mp3-short.json, video-short.json, camera-long.json, camera-short.json

default: `'taskset'`

duration: type: `'integer'`

Duration of the workload execution in Seconds. If specified, this will override the corresponding parameter in the JSON config.

cpus: type: `'cpu_mask'`

Constrain execution to specific CPUs.

aliases: `'taskset_mask'`

uninstall_on_exit: type: `'boolean'`

If set to `True`, rt-app binary will be uninstalled from the device at the end of the run.

force_install: type: `'boolean'`

If set to `True`, rt-app binary will always be deployed to the target device at the beginning of the run, regardless of whether it was already installed there.

---

shellscript

Runs an arbitrary shellscript on the target.

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

script_file: (mandatory) type: `'str'`

The path (on the host) to the shell script file. This must be an absolute path (though it may contain ~).

argstring: type: `'str'`

A string that should contain arguments passed to the script.

as_root: type: `'boolean'`

Specify whether the script should be run as root.

timeout: type: `'integer'`

Timeout, in seconds, for the script run time.

default: `60`

---

speedometer

A workload to execute the speedometer web based benchmark

Test description: 1. Open browser application 2. Navigate to the speedometer website - http://browserbench.org/Speedometer/ 3. Execute the benchmark

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

markers_enabled: type: `'boolean'`

If set to `True`, workloads will insert markers into logs at various points during execution. These markers may be used by other plugins or post-processing scripts to provide measurements or statistics for specific parts of the workload execution.

version: type: `'str'`

The speedometer version to be used.

allowed values: `'1.0'`, `'2.0'`

default: `'2.0'`

---

stress-ng

Run the stress-ng benchmark.

stress-ng will stress test a computer system in various selectable ways. It was designed to exercise various physical subsystems of a computer as well as the various operating system kernel interfaces.

stress-ng can also measure test throughput rates; this can be useful to observe performance changes across different operating system releases or types of hardware. However, it has never been intended to be used as a precise benchmark test suite, so do NOT use it in this manner.

The official website for stress-ng is at: http://kernel.ubuntu.com/~cking/stress-ng/

Source code are available from: http://kernel.ubuntu.com/git/cking/stress-ng.git/

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

stressor: type: `'str'`

Stress test case name. The cases listed in allowed values come from the stable release version 0.01.32. The binary included here compiled from dev version 0.06.01. Refer to man page for the definition of each stressor.

allowed values: `'cpu'`, `'io'`, `'fork'`, `'switch'`, `'vm'`, `'pipe'`, `'yield'`, `'hdd'`, `'cache'`, `'sock'`, `'fallocate'`, `'flock'`, `'affinity'`, `'timer'`, `'dentry'`, `'urandom'`, `'sem'`, `'open'`, `'sigq'`, `'poll'`

default: `'cpu'`

extra_args: type: `'str'`

Extra arguments to pass to the workload.

Please note that these are not checked for validity.

threads: type: `'integer'`

The number of workers to run. Specifying a negative or zero value will select the number of online processors.

duration: type: `'integer'`

Timeout for test execution in seconds

default: `60`

---

sysbench

A modular, cross-platform and multi-threaded benchmark tool for evaluating OS parameters that are important for a system running a database under intensive load.

The idea of this benchmark suite is to quickly get an impression about system performance without setting up complex database benchmarks or even without installing a database at all.

Features of SysBench

- file I/O performance
- scheduler performance
- memory allocation and transfer speed
- POSIX threads implementation performance
- database server performance

See: https://github.com/akopytov/sysbench

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

timeout: type: `'integer'`

timeout for workload execution (adjust from default if running on a slow target and/or specifying a large value for `max_requests`

default: `300`

test: type: `'str'`

sysbench test to run

allowed values: `'fileio'`, `'cpu'`, `'memory'`, `'threads'`, `'mutex'`

default: `'cpu'`

threads: type: `'integer'`

The number of threads sysbench will launch.

aliases: `'num_threads'`

default: `8`

max_requests: type: `'integer'`

The limit for the total number of requests.

max_time: type: `'integer'`

The limit for the total execution time. If neither this nor `max_requests` is specified, this will default to 30 seconds.

file_test_mode: type: `'str'`

File test mode to use. This should only be specified if `test` is `"fileio"`; if that is the case and `file_test_mode` is not specified, it will default to `"seqwr"` (please see sysbench documentation for explanation of various modes).

allowed values: `'seqwr'`, `'seqrewr'`, `'seqrd'`, `'rndrd'`, `'rndwr'`, `'rndrw'`

cmd_params: type: `'str'`

Additional parameters to be passed to sysbench as a single string.

cpus: type: `'cpu_mask'`

The processes spawned by sysbench will be pinned to cores as specified by this parameter. Can be provided as a mask, a list of cpus or a sysfs-style string.

aliases: `'taskset_mask'`

---

templerun2

Temple Run 2 game.

Sequel to Temple Run. 3D on-the-rails racer.

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

package_name: type: `'str'`

The package name that can be used to specify the workload apk to use.

install_timeout: type: `'integer'`

Timeout for the installation of the apk.

constraint: `value > 0`

default: `300`

version: type: `'str'`

The version of the package to be used.

variant: type: `'str'`

The variant of the package to be used.

strict: type: `'boolean'`

Whether to throw an error if the specified package cannot be found on host.

force_install: type: `'boolean'`

Always re-install the APK, even if matching version is found already installed on the device.

uninstall: type: `'boolean'`

If `True`, will uninstall workload's APK as part of teardown.'

exact_abi: type: `'boolean'`

If `True`, workload will check that the APK matches the target device ABI, otherwise any suitable APK found will be used.

prefer_host_package: type: `'boolean'`

If `True` then a package found on the host will be preferred if it is a valid version and ABI, if not it will fall back to the version on the target if available. If `False` then the version on the target is preferred instead.

aliases: `'check_apk'`

default: `True`

view: type: `'str'`

Manually override the 'View' of the workload for use with instruments such as the `fps` instrument. If not specified, a workload dependant 'View' will be automatically generated.

---

thechase

The Chase demo showcasing the capabilities of Unity game engine.

This demo, is a static video-like game demo, that demonstrates advanced features of the unity game engine. It loops continuously until terminated.

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

package_name: type: `'str'`

The package name that can be used to specify the workload apk to use.

install_timeout: type: `'integer'`

Timeout for the installation of the apk.

constraint: `value > 0`

default: `300`

version: type: `'str'`

The version of the package to be used.

variant: type: `'str'`

The variant of the package to be used.

strict: type: `'boolean'`

Whether to throw an error if the specified package cannot be found on host.

force_install: type: `'boolean'`

Always re-install the APK, even if matching version is found already installed on the device.

uninstall: type: `'boolean'`

If `True`, will uninstall workload's APK as part of teardown.'

exact_abi: type: `'boolean'`

If `True`, workload will check that the APK matches the target device ABI, otherwise any suitable APK found will be used.

prefer_host_package: type: `'boolean'`

If `True` then a package found on the host will be preferred if it is a valid version and ABI, if not it will fall back to the version on the target if available. If `False` then the version on the target is preferred instead.

aliases: `'check_apk'`

default: `True`

view: type: `'str'`

Manually override the 'View' of the workload for use with instruments such as the `fps` instrument. If not specified, a workload dependant 'View' will be automatically generated.

duration: type: `'integer'`

Duration, in seconds, note that the demo loops the same (roughly) 60 second sceene until stopped.

default: `70`

---

vellamo

Android benchmark designed by Qualcomm.

Vellamo began as a mobile web benchmarking tool that today has expanded to include three primary chapters. The Browser Chapter evaluates mobile web browser performance, the Multicore chapter measures the synergy of multiple CPU cores, and the Metal Chapter measures the CPU subsystem performance of mobile processors. Through click-and-go test suites, organized by chapter, Vellamo is designed to evaluate: UX, 3D graphics, and memory read/write and peak bandwidth performance, and much more!

Note: Vellamo v3.0 fails to run on Juno

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

package_name: type: `'str'`

The package name that can be used to specify the workload apk to use.

install_timeout: type: `'integer'`

Timeout for the installation of the apk.

constraint: `value > 0`

default: `300`

version: type: `'str'`

Specify the version of Vellamo to be run. If not specified, the latest available version will be used.

allowed values: `'3.2.4'`, `'2.0.3'`, `'3.0'`

variant: type: `'str'`

The variant of the package to be used.

strict: type: `'boolean'`

Whether to throw an error if the specified package cannot be found on host.

force_install: type: `'boolean'`

Always re-install the APK, even if matching version is found already installed on the device.

uninstall: type: `'boolean'`

If `True`, will uninstall workload's APK as part of teardown.'

exact_abi: type: `'boolean'`

If `True`, workload will check that the APK matches the target device ABI, otherwise any suitable APK found will be used.

prefer_host_package: type: `'boolean'`

If `True` then a package found on the host will be preferred if it is a valid version and ABI, if not it will fall back to the version on the target if available. If `False` then the version on the target is preferred instead.

aliases: `'check_apk'`

default: `True`

view: type: `'str'`

Manually override the 'View' of the workload for use with instruments such as the `fps` instrument. If not specified, a workload dependant 'View' will be automatically generated.

markers_enabled: type: `'boolean'`

If set to `True`, workloads will insert markers into logs at various points during execution. These markers may be used by other plugins or post-processing scripts to provide measurements or statistics for specific parts of the workload execution.

benchmarks: type: `'list_of_strs'`

Specify which benchmark sections of Vellamo to be run. Only valid on version 3.0 and newer. NOTE: Browser benchmark can be problematic and seem to hang,just wait and it will progress after ~5 minutes

allowed values: `'Browser'`, `'Metal'`, `'Multi'`

default: `['Browser', 'Metal', 'Multi']`

browser: type: `'integer'`

Specify which of the installed browsers will be used for the tests. The number refers to the order in which browsers are listed by Vellamo. E.g. `1` will select the first browser listed, `2` – the second, etc. Only valid for version `3.0`.

default: `1`

---

youtube

A workload to perform standard productivity tasks within YouTube.

The workload plays a video from the app, determined by the `video_source` parameter. While the video is playing, a some common actions are done such as video seeking, pausing playback and navigating the comments section.

Test description: The `video_source` parameter determines where the video to be played will be found in the app. Possible values are `search`, `home`, `my_videos`, and `trending`.

-A. search - Goes to the search view, does a search for the given term, and plays the first video in the results. The parameter `search_term` must also be provided in the agenda for this to work. This is the default mode.

-B. home - Scrolls down once on the app's home page to avoid ads (if present, would be first video), then select and plays the video that appears at the top of the list.

-C. my_videos - Goes to the 'My Videos' section of the user's account page and plays a video from there. The user must have at least one uploaded video for this to work.

-D. trending - Goes to the 'Trending Videos' section of the app, and plays the first video in the trending videos list.

For the selected video source, the following test steps are performed:

1. Navigate to the general app settings page to disable autoplay. This improves test stability and predictability by preventing screen transition to load a new video while in the middle of the test.
2. Select the video from the source specified above, and dismiss any potential embedded advert that may pop-up before the actual video.
3. Let the video play for a few seconds, pause it, then resume.
4. Expand the info card that shows video metadata, then collapse it again.
5. Scroll down to the end of related videos and comments under the info card, and then back up to the start. A maximum of 5 swipe actions is performed in either direction.

Known working APK version: 12.21.57

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

package_name: type: `'str'`

The package name that can be used to specify the workload apk to use.

install_timeout: type: `'integer'`

Timeout for the installation of the apk.

constraint: `value > 0`

default: `300`

version: type: `'str'`

The version of the package to be used.

variant: type: `'str'`

The variant of the package to be used.

strict: type: `'boolean'`

Whether to throw an error if the specified package cannot be found on host.

force_install: type: `'boolean'`

Always re-install the APK, even if matching version is found already installed on the device.

uninstall: type: `'boolean'`

If `True`, will uninstall workload's APK as part of teardown.'

exact_abi: type: `'boolean'`

If `True`, workload will check that the APK matches the target device ABI, otherwise any suitable APK found will be used.

prefer_host_package: type: `'boolean'`

If `True` then a package found on the host will be preferred if it is a valid version and ABI, if not it will fall back to the version on the target if available. If `False` then the version on the target is preferred instead.

aliases: `'check_apk'`

default: `True`

view: type: `'str'`

Manually override the 'View' of the workload for use with instruments such as the `fps` instrument. If not specified, a workload dependant 'View' will be automatically generated.

markers_enabled: type: `'boolean'`

If set to `True`, workloads will insert markers into logs at various points during execution. These markers may be used by other plugins or post-processing scripts to provide measurements or statistics for specific parts of the workload execution.

video_source: type: `'str'`

Determines where to play the video from. This can either be from the YouTube home, my videos section, trending videos or found in search.

allowed values: `'home'`, `'my_videos'`, `'search'`, `'trending'`

default: `'search'`

search_term: type: `'str'`

The search term to use when `video_source` is set to `search`. Ignored otherwise.

default: `'Big Buck Bunny 60fps 4K - Official Blender Foundation Short Film'`

---

youtube_playback

Simple Youtube video playback

This triggers a video streaming playback on Youtube. Unlike the more featureful "youtube" workload, this performs no other action that starting the video via an intent and then waiting for a certain amount of playback time. This is therefore only useful when you are confident that the content on the end of the provided URL is stable - that means the video should have no advertisements attached.

parameters

cleanup_assets: type: `'boolean'`

If `True`, if assets are deployed as part of the workload they will be removed again from the device as part of finalize.

aliases: `'clean_up'`

global alias: `'cleanup_assets'`

default: `True`

package_name: type: `'str'`

The package name that can be used to specify the workload apk to use.

install_timeout: type: `'integer'`

Timeout for the installation of the apk.

constraint: `value > 0`

default: `300`

version: type: `'str'`

The version of the package to be used.

variant: type: `'str'`

The variant of the package to be used.

strict: type: `'boolean'`

Whether to throw an error if the specified package cannot be found on host.

force_install: type: `'boolean'`

Always re-install the APK, even if matching version is found already installed on the device.

uninstall: type: `'boolean'`

If `True`, will uninstall workload's APK as part of teardown.'

exact_abi: type: `'boolean'`

If `True`, workload will check that the APK matches the target device ABI, otherwise any suitable APK found will be used.

prefer_host_package: type: `'boolean'`

If `True` then a package found on the host will be preferred if it is a valid version and ABI, if not it will fall back to the version on the target if available. If `False` then the version on the target is preferred instead.

aliases: `'check_apk'`

default: `True`

view: type: `'str'`

Manually override the 'View' of the workload for use with instruments such as the `fps` instrument. If not specified, a workload dependant 'View' will be automatically generated.

video_url: type: `'str'`

URL of video to play

default: `'https://www.youtube.com/watch?v=YE7VzlLtp-4'`

duration: type: `'integer'`

Number of seconds of video to play

default: `20`

4.1.2 Instruments

---

apk_version

Extracts APK versions for workloads that have them.

---

cpufreq

Collects dynamic frequency (DVFS) settings before and after workload execution.

parameters

paths: type: `'list_of_strs'`

A list of paths to be pulled from the device. These could be directories as well as files.

global alias: `'sysfs_extract_dirs'`

use_tmpfs: type: `'boolean'`

Specifies whether tmpfs should be used to cache sysfile trees and then pull them down as a tarball. This is significantly faster then just copying the directory trees from the device directly, but requires root and may not work on all devices. Defaults to `True` if the device is rooted and `False` if it is not.

tmpfs_mount_point: type: `'str'`

Mount point for tmpfs partition used to store snapshots of paths.

tmpfs_size: type: `'str'`

Size of the tempfs partition.

default: `'32m'`

---

delay

This instrument introduces a delay before beginning a new spec, a new job or before the main execution of a workload.

The delay may be specified as either a fixed period or a temperature threshold that must be reached.

Optionally, if an active cooling solution is available on the device tqgitq speed up temperature drop between runs, it may be controlled using this instrument.

parameters

temperature_file: type: `'str'`

Full path to the sysfile on the target that contains the target's temperature.

global alias: `'thermal_temp_file'`

default: `'/sys/devices/virtual/thermal/thermal_zone0/temp'`

temperature_timeout: type: `'integer'`

The timeout after which the instrument will stop waiting even if the specified threshold temperature is not reached. If this timeout is hit, then a warning will be logged stating the actual temperature at which the timeout has ended.

global alias: `'thermal_timeout'`

default: `600`

temperature_poll_period: type: `'integer'`

How long to sleep (in seconds) between polling current target temperature.

global alias: `'thermal_sleep_time'`

default: `5`

temperature_between_specs: type: `'integer'`

Temperature (in target-specific units) the target must cool down to before the iteration spec will be run.

If this is set to `0` then the devices initial temperature will used as the threshold.

Note: This cannot be specified at the same time as `fixed_between_specs`

---

global alias: 'thermal_threshold_between_specs'

fixed_between_specs: type: 'integer'

How long to sleep (in seconds) before starting a new workload spec.

Note: This cannot be specified at the same time as temperature_between_specs

---

global alias: 'fixed_delay_between_specs'

temperature_between_jobs: type: 'integer'

Temperature (in target-specific units) the target must cool down to before the next job will be run.

If this is set to 0 then the devices initial temperature will used as the threshold.

Note: This cannot be specified at the same time as fixed_between_jobs

---

aliases: 'temperature_between_iterations'

global alias: 'thermal_threshold_between_jobs'

fixed_between_jobs: type: 'integer'

How long to sleep (in seconds) before starting each new job.

Note: This cannot be specified at the same time as temperature_between_jobs

---

aliases: 'fixed_between_iterations'

global alias: 'fixed_delay_between_jobs'

fixed_before_start: type: 'integer'

How long to sleep (in seconds) after setup for an iteration has been performed but before running the workload.

Note: This cannot be specified at the same time as temperature_before_start

---

global alias: 'fixed_delay_before_start'

temperature_before_start: type: 'integer'

Temperature (in device-specific units) the device must cool down to just before the actual workload execution (after setup has been performed).

Note: This cannot be specified at the same time as fixed_between_jobs

---

global alias: 'thermal_threshold_before_start'

active_cooling: type: 'boolean'

This instrument supports an active cooling solution while waiting for the device temperature to drop to the threshold. If you wish to use this feature please ensure the relevant module is installed on the device.

---

dmesg

Collected dmesg output before and during the run.

parameters

loglevel: type: `'integer'`

Set loglevel for console output.

allowed values: `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`

---

energy_measurement

This instrument is designed to be used as an interface to the various energy measurement instruments located in devlib.

This instrument should be used to provide configuration for any of the Energy Instrument Backends rather than specifying configuration directly.

parameters

instrument: (mandatory) type: `'str'`

Specify the energy instruments to be enabled.

allowed values: `'daq'`, `'energy_probe'`, `'acme_cape'`, `'monsoon'`, `'juno_readenergy'`, `'arm_energy_probe'`

instrument_parameters: type: `'OrderedDict'`

Specify the parameters used to initialize the desired instruments. To see parameters available for a particular instrument, run

wa show <instrument name>

See help for `instrument` parameter to see available options for <instrument name>.

sites: type: `'list_or_string'`

Specify which sites measurements should be collected from, if not specified the measurements will be collected for all available sites.

kinds: type: `'list_or_string'`

Specify the kinds of measurements should be collected, if not specified measurements will be collected for all available kinds.

channels: type: `'list_or_string'`

Specify the channels to be collected, if not specified the measurements will be collected for all available channels.

---

execution_time

Measure how long it took to execute the run() methods of a Workload.

---

file_poller

Polls the given files at a set sample interval. The values are output in CSV format.

This instrument places a file called poller.csv in each iterations result directory. This file will contain a timestamp column which will be in uS, the rest of the columns will be the contents of the polled files at that time.

This instrument will strip any commas or new lines for the files' values before writing them.

parameters

sample_interval: type: `'integer'`

The interval between samples in mS.

default: `1000`

files: (mandatory) type: `'list_or_string'`

A list of paths to the files to be polled

labels: type: `'list_or_string'`

A list of lables to be used in the CSV output for the corresponding files. This cannot be used if a * wildcard is used in a path.

align_with_ftrace: type: `'boolean'`

Insert a marker into ftrace that aligns with the first timestamp. During output processing, extract the marker and use it's timestamp to adjust the timestamps in the collected csv so that they align with ftrace.

as_root: type: `'boolean'`

Whether or not the poller will be run as root. This should be used when the file you need to poll can only be accessed by root.

---

fps

Measures Frames Per Second (FPS) and associated metrics for a workload.

Note: This instrument depends on pandas Python library (which is not part of standard WA dependencies), so you will need to install that first, before you can use it.

Android L and below use SurfaceFlinger to calculate the FPS data. Android M and above use gfxinfo to calculate the FPS data.

SurfaceFlinger: The view is specified by the workload as `view` attribute. This defaults to `'SurfaceView'` for game workloads, and `None` for non-game workloads (as for them FPS mesurement usually doesn't make sense). Individual workloads may override this.

gfxinfo: The view is specified by the workload as `package` attribute. This is because gfxinfo already processes for all views in a package.

parameters

drop_threshold: type: `'numeric'`

Data points below this FPS will be dropped as they do not constitute "real" gameplay. The assumption being that while actually running, the FPS in the game will not drop below X frames per second, except on loading screens, menus, etc, which should not contribute to FPS calculation.

default: `5`

keep_raw: type: `'boolean'`

If set to `True`, this will keep the raw dumpsys output in the results directory (this is maily used for debugging) Note: frames.csv with collected frames data will always be generated regardless of this setting.

crash_threshold: type: `'float'`

Specifies the threshold used to decided whether a measured/expected frames ration indicates a content crash. E.g. a value of `0.75` means the number of actual frames counted is a quarter lower than expected, it will treated as a content crash.

If set to zero, no crash check will be performed.

default: `0.7`

period: type: `'float'`

Specifies the time period between polling frame data in seconds when collecting frame data. Using a lower value improves the granularity of timings when recording actions that take a short time to complete. Note, this will produce duplicate frame data in the raw dumpsys output, however, this is filtered out in frames.csv. It may also affect the overall load on the system.

The default value of 2 seconds corresponds with the NUM_FRAME_RECORDS in android/services/surfaceflinger/FrameTracker.h (as of the time of writing currently 128) and a frame rate of 60 fps that is applicable to most devices.

constraint: `value > 0`

default: `2`

force_surfaceflinger: type: `'boolean'`

By default, the method to capture fps data is based on Android version. If this is set to true, force the instrument to use the SurfaceFlinger method regardless of its Android version.

---

hwmon

Hardware Monitor (hwmon) is a generic Linux kernel subsystem, providing access to hardware monitoring components like temperature or voltage/current sensors.

Data from hwmon that are a snapshot of a fluctuating value, such as temperature and voltage, are reported once at the beginning and once at the end of the workload run. Data that are a cumulative total of a quantity, such as energy (which is the cumulative total of power consumption), are reported as the difference between the values at the beginning and at the end of the workload run.

There is currently no functionality to filter sensors: all of the available hwmon data will be reported.

interrupts

Pulls the `/proc/interrupts` file before and after workload execution and diffs them to show what interrupts occurred during that time.

perf

Perf is a Linux profiling with performance counters.

Performance counters are CPU hardware registers that count hardware events such as instructions executed, cache-misses suffered, or branches mispredicted. They form a basis for profiling applications to trace dynamic control flow and identify hotspots.

pref accepts options and events. If no option is given the default '-a' is used. For events, the default events are migrations and cs. They both can be specified in the config file.

Events must be provided as a list that contains them and they will look like this

```
perf_events = ['migrations', 'cs']
```

Events can be obtained by typing the following in the command line on the device

```
perf list
```

Whereas options, they can be provided as a single string as following

```
perf_options = '-a -i'
```

Options can be obtained by running the following in the command line

```
man perf-stat
```

parameters

events: type: `'list_of_strs'`

Specifies the events to be counted.

global alias: `'perf_events'`

constraint: `must not be empty.`

default: `['migrations', 'cs']`

optionstring: type: `'list_or_string'`

Specifies options to be used for the perf command. This may be a list of option strings, in which case, multiple instances of perf will be kicked off – one for each option string. This may be used to e.g. collected different events from different big.LITTLE clusters.

global alias: `'perf_options'`

default: `'-a'`

labels: type: `'list_of_strs'`

Provides labels for pref output. If specified, the number of labels must match the number of `optionstrings`.

global alias: `'perf_labels'`

force_install: type: `'boolean'`

always install perf binary even if perf is already present on the device.

---

screen_capture

A simple instrument which captures the screen on the target devices with a user-specified period.

Please note that if a too short period is specified, then this instrument will capture the screen as fast as possible, rather than at the specified periodicity.

parameters

period: type: `'integer'`

Period (in seconds) at which to capture the screen on the target.

default: `10`

---

serialmon

Records the traffic on a serial connection

The traffic on a serial connection is monitored and logged to a file. In the event that the device is reset, the instrument will stop monitoring during the reset, and will reconnect once the reset has completed. This is to account for devices (i.e., the Juno) which utilise the serial connection to reset the board.

parameters

serial_port: type: `'str'`

The serial device to monitor.

default: `'/dev/ttyS0'`

baudrate: type: `'integer'`

The baud-rate to use when connecting to the serial connection.

default: `115200`

---

sysfs_extractor

Collects the contest of a set of directories, before and after workload execution and diffs the result.

parameters

paths: (mandatory) type: `'list_of_strs'`

A list of paths to be pulled from the device. These could be directories as well as files.

global alias: `'sysfs_extract_dirs'`

use_tmpfs: type: `'boolean'`

Specifies whether tmpfs should be used to cache sysfile trees and then pull them down as a tarball. This is significantly faster then just copying the directory trees from the device directly, but requires root and may not work on all devices. Defaults to `True` if the device is rooted and `False` if it is not.

tmpfs_mount_point: type: `'str'`

Mount point for tmpfs partition used to store snapshots of paths.

tmpfs_size: type: `'str'`

Size of the tempfs partition.

default: `'32m'`

---

trace-cmd

trace-cmd is an instrument which interacts with ftrace Linux kernel internal tracer

From trace-cmd man page:

trace-cmd command interacts with the ftrace tracer that is built inside the Linux kernel. It interfaces with the ftrace specific files found in the debugfs file system under the tracing directory.

trace-cmd reads a list of events it will trace, which can be specified in the config file as follows

```
trace_events = ['irq*', 'power*']
```

If no event is specified, a default set of events that are generally considered useful for debugging/profiling purposes will be enabled.

The list of available events can be obtained by rooting and running the following command line on the device

```
trace-cmd list
```

You may also specify `trace_buffer_size` setting which must be an integer that will be used to set the ftrace buffer size. It will be interpreted as KB:

```
trace_cmd_buffer_size = 8000
```

The maximum buffer size varies from device to device, but there is a maximum and trying to set buffer size beyond that will fail. If you plan on collecting a lot of trace over long periods of time, the buffer size will not be enough and you will only get trace for the last portion of your run. To deal with this you can set the `trace_mode` setting to `'record'` (the default is `'start'`):

```
trace_cmd_mode = 'record'
```

This will cause trace-cmd to trace into file(s) on disk, rather than the buffer, and so the limit for the max size of the trace is set by the storage available on device. Bear in mind that `'record'` mode *is* more intrusive than the default, so if you do not plan on generating a lot of trace, it is best to use the default `'start'` mode.

Note: Mode names correspond to the underlying trace-cmd executable's command used to implement them. You can find out more about what is happening in each case from trace-cmd documentation: https://lwn.net/Articles/341902/.

This instrument comes with an trace-cmd binary that will be copied and used on the device, however post-processing will be, by default, done on-host and you must have trace-cmd installed and in your path. On Ubuntu systems, this may be done with:

```
sudo apt-get install trace-cmd
```

Alternatively, you may set `report_on_target` parameter to `True` to enable on-target processing (this is useful when running on non-Linux hosts, but is likely to take longer and may fail on particularly resource-constrained targets).

parameters

events: type: `'list_of_strs'`

Specifies the list of events to be traced. Each event in the list will be passed to trace-cmd with -e parameter and must be in the format accepted by trace-cmd.

global alias: `'trace_events'`

default: `['sched*', 'irq*', 'power*', 'thermal*']`

functions: type: `'list_of_strs'`

Specifies the list of functions to be traced.

global alias: `'trace_functions'`

buffer_size: type: `'integer'`

Attempt to set ftrace buffer size to the specified value (in KB). Default buffer size may need to be increased for long-running workloads, or if a large number of events have been enabled. Note: there is a maximum size that the buffer can be set, and that varies from device to device. Attempting to set buffer size higher than this will fail. In that case, this instrument will set the size to the highest possible value by going down from the specified size in `buffer_size_step` intervals.

global alias: `'trace_buffer_size'`

buffer_size_step: type: `'integer'`

Defines the decremental step used if the specified `buffer_size` could not be set. This will be subtracted form the buffer size until set succeeds or size is reduced to 1MB.

global alias: `'trace_buffer_size_step'`

default: `1000`

report: type: `'boolean'`

Specifies whether reporting should be performed once the binary trace has been generated.

default: `True`

no_install: type: `'boolean'`

Do not install the bundled trace-cmd and use the one on the device instead. If there is not already a trace-cmd on the device, an error is raised.

report_on_target: type: `'boolean'`

When enabled generation of reports will be done host-side because the generated file is very large. If trace-cmd is not available on the host device this setting can be disabled and the report will be generated on the target device.

Note: This requires the latest version of trace-cmd to be installed on the host (the one in your distribution's repos may be too old).

4.1.3 Energy Instrument Backends

---

acme_cape

BayLibre ACME cape

This backend relies on iio-capture utility:

https://github.com/BayLibre/iio-capture

For more information about ACME cape please see:

https://baylibre.com/acme/

parameters

iio_capture: type: `'str'`

Path to the iio-capture binary will be taken from the environment, if not specfied.

host: type: `'str'`

Host name (or IP address) of the ACME cape board.

default: `'baylibre-acme.local'`

iio_devices: type: `'list_or_string'`

default: `'iio:device0'`

buffer_size: type: `'integer'`

Size of the capture buffer (in KB).

default: `256`

---

arm_energy_probe

Arm Energy Probe arm-probe version

An alternative Arm Energy Probe backend that relies on arm-probe utility:

https://git.linaro.org/tools/arm-probe.git

For more information about Arm Energy Probe please see

https://developer.arm.com/products/software-development-tools/ds-5-development-studio/streamline/arm-energy-probe

parameters

config_file: type: `'str'`

Path to config file of the AEP

---

daq

National Instruments Data Acquisition device

For more information about the device, please see the NI website:

http://www.ni.com/data-acquisition/

This backend has been used with USB-62xx and USB-63xx devices, though other models (e.g. the PCIe variants will most likely also work).

This backend relies on the daq-server running on a machinge connected to a DAQ device:

https://github.com/ARM-software/daq-server

The server is necessary because DAQ devices have drivers only for Windows and very specific (old) Linux kernels, so the machine interfacing with the DAQ is most likely going to be different from the machinge running WA.

parameters

resistor_values: type: `'list_of_numbers'`

The values of resistors (in Ohms) across which the voltages are measured on.

global alias: `'daq_resistor_values'`

labels: type: `'list_of_strs'`

'List of port labels. If specified, the length of the list must match the length of `resistor_values`.

global alias: `'daq_labels'`

host: type: 'str'

The host address of the machine that runs the daq Server which the instrument communicates with.

global alias: 'daq_server_host'

default: 'localhost'

port: type: 'integer'

The port number for daq Server in which daq instrument communicates with.

global alias: 'daq_server_port'

default: 45677

device_id: type: 'str'

The ID under which the DAQ is registered with the driver.

global alias: 'daq_device_id'

default: 'Dev1'

v_range: type: 'str'

Specifies the voltage range for the SOC voltage channel on the DAQ (please refer to daq-server package documentation for details).

global alias: 'daq_v_range'

default: 2.5

dv_range: type: 'str'

Specifies the voltage range for the resistor voltage channel on the DAQ (please refer to daq-server package documentation for details).

global alias: 'daq_dv_range'

default: 0.2

sample_rate_hz: type: 'integer'

Specify the sample rate in Hz.

global alias: 'daq_sampling_rate'

default: 10000

channel_map: type: 'list_of_ints'

Represents mapping from logical AI channel number to physical connector on the DAQ (varies between DAQ models). The default assumes DAQ 6363 and similar with AI channels on connectors 0-7 and 16-23.

global alias: 'daq_channel_map'

default: (0, 1, 2, 3, 4, 5, 6, 7, 16, 17, 18, 19, 20, 21, 22, 23)

---

energy_probe

Arm Energy Probe caiman version

This backend relies on caiman utility:

https://github.com/ARM-software/caiman

For more information about Arm Energy Probe please see

> https://developer.arm.com/products/software-development-tools/ds-5-development-studio/streamline/arm-energy-probe

parameters

resistor_values: type: `'list_of_ints'`

> The values of resistors (in Ohms) across which the voltages are measured on.

labels: type: `'list_of_strs'`

> 'List of port labels. If specified, the length of the list must match the length of `resistor_values`.

device_entry: type: `'str'`

> Path to /dev entry for the energy probe (it should be /dev/ttyACMx)
>
> default: `'/dev/ttyACM0'`

---

juno_readenergy

Arm Juno development board on-board energy meters

For more information about Arm Juno board see:

> https://developer.arm.com/products/system-design/development-boards/juno-development-board

---

monsoon

Monsoon Solutions power monitor

To use this instrument, you need to install the monsoon.py script available from the Android Open Source Project. As of May 2017 this is under the CTS repository:

> https://android.googlesource.com/platform/cts/+/master/tools/utils/monsoon.py

Collects power measurements only, from a selection of two channels, the USB passthrough channel and the main output channel.

parameters

monsoon_bin: type: `'str'`

> Path to monsoon.py executable. If not provided, `PATH` is searched.

tty_device: type: `'str'`

> TTY device to use to communicate with the Power Monitor. If not provided, /dev/ttyACM0 is used.
>
> default: `'/dev/ttyACM0'`

4.1.4 Output Processors

cpustates

Process trace-cmd output to generate timelines and statistics of CPU power state (a.k.a P- and C-state) transitions in the trace.

The results will be written into a subdirectory called "power-stats" under the specified `output_basedir`.

The output directory will contain the following files:

power-state-stats.csv Power state residency statistics for each CPU. Shows the percentage of time a CPU has spent in each of its available power states.

parallel-stats.csv Parallel execution stats for each CPU cluster, and combined stats for the whole system.

power-state-timeline.csv Timeline of CPU power states. Shows which power state each CPU is in at a point in time.

state-transitions-timeline.csv Timeline of CPU power state transitions. Each entry shows a CPU's transition from one power state to another.

utilzation-timeline.csv Timeline of CPU utilizations.

Note: Timeline entries aren't at regular intervals, but at times of power transition events.

Stats are generated by assembling a pipeline consisting of the following stages:

1. Parse trace into trace events
2. Filter trace events into power state transition events
3. Record power state transitions
4. Convert transitions into a power states.
5. Collapse the power states into timestamped `(C state, P state)` tuples for each cpu.
6. Update reporters/stats generators with cpu states.

parameters

use_ratios: type: `'boolean'`

By default proportional values will be reported as percentages, if this flag is enabled, they will be reported as ratios instead.

no_idle: type: `'boolean'`

Indicate that there will be no idle transitions in the trace. By default, a core will be reported as being in an "unknown" state until the first idle transtion for that core. Normally, this is not an issue, as cores are "nudged" as part of the setup to ensure that there is an idle transtion before the meassured region. However, if all idle states for the core have been disabled, or if the kernel does not have cpuidle, the nudge will not result in an idle transition, which would cause the cores to be reported to be in "unknown" state for the entire execution.

If this parameter is set to `True`, the processor will assume that cores are running prior to the begining of the issue, and they will leave unknown state on the first frequency transition.

split_wfi_states: type: `'boolean'`

WFI is a very shallow idle state. The core remains powered on when in this state, which means the power usage while in this state will depend on the current voltage, and therefore current frequency.

Setting this to `True` will track time spent in WFI at each frequency separately, allowing to gain the most accurate picture of energy usage.

---

csv

Creates a `results.csv` in the output directory containing results for all iterations in CSV format, each line containing a single metric.

parameters

use_all_classifiers: type: `'boolean'`

If set to `True`, this will add a column for every classifier that features in at least one collected metric.

Note: This cannot be `True` if `extra_columns` is set.

---

global alias: `'use_all_classifiers'`

extra_columns: type: `'list_of_strs'`

List of classifiers to use as columns.

Note: This cannot be set if `use_all_classifiers` is `True`.

---

postgres

Stores results in a Postgresql database.

The structure of this database can easily be understood by examining the postgres_schema.sql file (the schema used to generate it): /home/docs/checkouts/readthedocs.org/user_builds/workload-automation/checkouts/stable/doc/source/../../wa/utils/../commands/postgres_schemas/postgres_schema.sql

parameters

username: type: `'str'`

This is the username that will be used to connect to the Postgresql database. Note that depending on whether the user has privileges to modify the database (normally only possible on localhost), the user may only be able to append entries.

default: `'postgres'`

password: type: `'str'`

The password to be used to connect to the specified database with the specified username.

dbname: type: `'str'`

Name of the database that will be created or added to. Note, to override this, you can specify a value in your user wa configuration file.

default: `'wa'`

host: type: `'str'`

The host where the Postgresql server is running. The default is localhost (i.e. the machine that wa is running on). This is useful for complex systems where multiple machines may be executing workloads and uploading their results to a remote, centralised database.

default: `'localhost'`

port: type: `'str'`

The port the Postgresql server is running on, on the host. The default is Postgresql's default, so do not change this unless you have modified the default port for Postgresql.

default: `'5432'`

---

sqlite

Stores results in an sqlite database.

This may be used to accumulate results of multiple runs in a single file.

parameters

database: type: `'str'`

Full path to the sqlite database to be used. If this is not specified then a new database file will be created in the output directory. This setting can be used to accumulate results from multiple runs in a single database. If the specified file does not exist, it will be created, however the directory of the file must exist.

Note: The value must resolve to an absolute path, relative paths are not allowed; however the value may contain environment variables and/or the home reference "~".

global alias: `'sqlite_database'`

overwrite: type: `'boolean'`

If `True`, this will overwrite the database file if it already exists. If `False` (the default) data will be added to the existing file (provided schema versions match – otherwise an error will be raised).

global alias: `'sqlite_overwrite'`

---

status

Outputs a txt file containing general status information about which runs failed and which were successful

---

targz

Create a tarball of the output directory.

This will create a gzip-compressed tarball of the output directory. By default, it will be created at the same level and will have the same name as the output directory but with a .tar.gz extensions.

parameters

outfile: type: `'str'`

The name of the output file to be used. If this is not an absolute path, the file will be created realtive to the directory in which WA was invoked. If this contains subdirectories, they must already exist.

The name may contain named format specifiers. Any of the `RunInfo` fields can be named, resulting in the value of that filed (e.g. `'start_time'`) being formatted into the tarball name.

By default, the output file will be created at the same level, share the name of the WA output directory (but with .tar.gz extension).

delete_output: type: `'boolean'`

if set to `True`, WA output directory will be deleted after the tarball is created.

---

uxperf

Parse logcat for UX_PERF markers to produce performance metrics for workload actions using specified instrumentation. An action represents a series of UI interactions to capture. NOTE: The UX_PERF markers are turned off by default and must be enabled in a agenda file by setting `markers_enabled` for the workload to `True`.

4.1.5 Common Targets

This is a list of commonly used targets and their device parameters, to see a complete for a complete reference please use the WA *list command*.

generic_android

Device Parameters:

device: type: `'str'`

ADB device name

aliases: `'adb_name'`

adb_server: type: `'str'`

ADB server to connect to.

core_names: type: `'list_of_strs'`

List of names of CPU cores in the order that they appear to the kernel. If not specified, it will be inferred from the platform.

core_clusters: type: `'list_of_ints'`

Cluster mapping corresponding to the cores in `core_names`. Cluster indexing starts at `0`. If not specified, this will be inferred from `core_names` – consecutive cores with the same name will be assumed to share a cluster.

big_core: type: `'str'`

The name of the big cores in a big.LITTLE system. If not specified, this will be inferred, either from the name (if one of the names in `core_names` matches known big cores), or by assuming that the last cluster is big.

model: type: `'str'`

Hardware model of the platform. If not specified, an attempt will be made to read it from target.

modules: type: `'list'`

An additional list of modules to be loaded into the target.

working_directory: type: `'str'`

On-target working directory that will be used by WA. This directory must be writable by the user WA logs in as without the need for privilege elevation.

executables_directory: type: `'str'`

On-target directory where WA will install its executable binaries. This location must allow execution. This location does *not* need to be writable by unprivileged users or rooted devices (WA will install with elevated privileges as necessary).

modules: type: `'list'`

A list of additional modules to be installed for the target.

`devlib` implements functionality for particular subsystems as modules. A number of "default" modules (e.g. for cpufreq subsystem) are loaded automatically, unless explicitly disabled. If additional modules need to be loaded, they may be specified using this parameter.

Please see `devlib` documentation for information on the available modules.

load_default_modules: type: `'boolean'`

A number of modules (e.g. for working with the cpufreq subsystem) are loaded by default when a Target is instantiated. Setting this to `True` would suppress that, ensuring that only the base Target interface is initialized.

You may want to set this to `False` if there is a problem with one or more default modules on your platform (e.g. your device is unrooted and cpufreq is not accessible to unprivileged users), or if `Target` initialization is taking too long for your platform.

default: `True`

shell_prompt: type: `'regex'`

A regex that matches the shell prompt on the target.

default: `r'^.*(shell|root|juno)@?.*:[/~]\S* *[#$] '`

package_data_directory: type: `'str'`

Directory containing Android data

default: `'/data/data'`

disable_selinux: type: `'boolean'`

If `True`, the default, and the target is rooted, an attempt will be made to disable SELinux by running `setenforce 0` on the target at the beginning of the run.

default: `True`

logcat_poll_period: type: `'integer'`

Polling period for logcat in seconds. If not specified, no polling will be used.

Logcat buffer on android is of limited size and it cannot be adjusted at run time. Depending on the amount of logging activity, the buffer may not be enought to capture comlete trace for a workload execution. For those situations, logcat may be polled periodically during the course of the run and stored in a temporary locaiton on the host. Setting the value of the poll period enables this behavior.

constraint: `value > 0`

generic_chromeos

Device Parameters:

device: type: `'str'`

ADB device name

aliases: `'adb_name'`

adb_server: type: `'str'`

ADB server to connect to.

host: (mandatory) type: `'str'`

Host name or IP address of the target.

username: (mandatory) type: `'str'`

User name to connect with

password: type: `'str'`

Password to use.

keyfile: type: 'str'

Key file to use

port: type: 'integer'

The port SSH server is listening on on the target.

telnet: type: 'boolean'

If set to True, a Telnet connection, rather than SSH will be used.

password_prompt: type: 'str'

Password prompt to expect

original_prompt: type: 'str'

Original shell prompt to expect.

sudo_cmd: type: 'str'

Sudo command to use. Must have {} specified somewhere in the string it indicate where the command to be run via sudo is to go.

default: 'sudo -- sh -c {}'

core_names: type: 'list_of_strs'

List of names of CPU cores in the order that they appear to the kernel. If not specified, it will be inferred from the platform.

core_clusters: type: 'list_of_ints'

Cluster mapping corresponding to the cores in core_names. Cluster indexing starts at 0. If not specified, this will be inferred from core_names – consecutive cores with the same name will be assumed to share a cluster.

big_core: type: 'str'

The name of the big cores in a big.LITTLE system. If not specified, this will be inferred, either from the name (if one of the names in core_names matches known big cores), or by assuming that the last cluster is big.

model: type: 'str'

Hardware model of the platform. If not specified, an attempt will be made to read it from target.

modules: type: 'list'

An additional list of modules to be loaded into the target.

working_directory: type: 'str'

On-target working directory that will be used by WA. This directory must be writable by the user WA logs in as without the need for privilege elevation.

executables_directory: type: 'str'

On-target directory where WA will install its executable binaries. This location must allow execution. This location does *not* need to be writable by unprivileged users or rooted devices (WA will install with elevated privileges as necessary).

modules: type: 'list'

A list of additional modules to be installed for the target.

devlib implements functionality for particular subsystems as modules. A number of "default" modules (e.g. for cpufreq subsystem) are loaded automatically, unless explicitly disabled. If additional modules need to be loaded, they may be specified using this parameter.

Please see `devlib` documentation for information on the available modules.

load_default_modules: type: `'boolean'`

A number of modules (e.g. for working with the cpufreq subsystem) are loaded by default when a Target is instantiated. Setting this to `True` would suppress that, ensuring that only the base Target interface is initialized.

You may want to set this to `False` if there is a problem with one or more default modules on your platform (e.g. your device is unrooted and cpufreq is not accessible to unprivileged users), or if `Target` initialization is taking too long for your platform.

default: `True`

shell_prompt: type: `'regex'`

A regex that matches the shell prompt on the target.

default: `r'^.*(shell|root|juno)@?.*:[/~]\S* *[#$] '`

package_data_directory: type: `'str'`

Directory containing Android data

default: `'/data/data'`

android_working_directory: type: `'str'`

On-target working directory that will be used by WA for the android container. This directory must be writable by the user WA logs in as without the need for privilege elevation.

android_executables_directory: type: `'str'`

On-target directory where WA will install its executable binaries for the android container. This location must allow execution. This location does *not* need to be writable by unprivileged users or rooted devices (WA will install with elevated privileges as necessary). directory must be writable by the user WA logs in as without the need for privilege elevation.

disable_selinux: type: `'boolean'`

If `True`, the default, and the target is rooted, an attempt will be made to disable SELinux by running `setenforce 0` on the target at the beginning of the run.

default: `True`

logcat_poll_period: type: `'integer'`

Polling period for logcat in seconds. If not specified, no polling will be used.

Logcat buffer on android is of limited size and it cannot be adjusted at run time. Depending on the amount of logging activity, the buffer may not be enought to capture comlete trace for a workload execution. For those situations, logcat may be polled periodically during the course of the run and stored in a temporary locaiton on the host. Setting the value of the poll period enables this behavior.

constraint: `value > 0`

generic_linux

Device Parameters:

host: (mandatory) type: `'str'`

Host name or IP address of the target.

username: (mandatory) type: `'str'`

User name to connect with

password: type: `'str'`

Password to use.

keyfile: type: `'str'`

Key file to use

port: type: `'integer'`

The port SSH server is listening on on the target.

telnet: type: `'boolean'`

If set to `True`, a Telnet connection, rather than SSH will be used.

password_prompt: type: `'str'`

Password prompt to expect

original_prompt: type: `'str'`

Original shell prompt to expect.

sudo_cmd: type: `'str'`

Sudo command to use. Must have `{}` specified somewhere in the string it indicate where the command to be run via sudo is to go.

default: `'sudo -- sh -c {}'`

core_names: type: `'list_of_strs'`

List of names of CPU cores in the order that they appear to the kernel. If not specified, it will be inferred from the platform.

core_clusters: type: `'list_of_ints'`

Cluster mapping corresponding to the cores in `core_names`. Cluster indexing starts at `0`. If not specified, this will be inferred from `core_names` – consecutive cores with the same name will be assumed to share a cluster.

big_core: type: `'str'`

The name of the big cores in a big.LITTLE system. If not specified, this will be inferred, either from the name (if one of the names in `core_names` matches known big cores), or by assuming that the last cluster is big.

model: type: `'str'`

Hardware model of the platform. If not specified, an attempt will be made to read it from target.

modules: type: `'list'`

An additional list of modules to be loaded into the target.

working_directory: type: `'str'`

On-target working directory that will be used by WA. This directory must be writable by the user WA logs in as without the need for privilege elevation.

executables_directory: type: `'str'`

On-target directory where WA will install its executable binaries. This location must allow execution. This location does *not* need to be writable by unprivileged users or rooted devices (WA will install with elevated privileges as necessary).

modules: type: 'list'

A list of additional modules to be installed for the target.

devlib implements functionality for particular subsystems as modules. A number of "default" modules (e.g. for cpufreq subsystem) are loaded automatically, unless explicitly disabled. If additional modules need to be loaded, they may be specified using this parameter.

Please see devlib documentation for information on the available modules.

load_default_modules: type: 'boolean'

A number of modules (e.g. for working with the cpufreq subsystem) are loaded by default when a Target is instantiated. Setting this to True would suppress that, ensuring that only the base Target interface is initialized.

You may want to set this to False if there is a problem with one or more default modules on your platform (e.g. your device is unrooted and cpufreq is not accessible to unprivileged users), or if Target initialization is taking too long for your platform.

default: True

shell_prompt: type: 'regex'

A regex that matches the shell prompt on the target.

default: r'^.*(shell|root|juno)@?.*:[/~]\S* *[#$] '

generic_local

Device Parameters:

password: type: 'str'

Password to use for sudo. if not specified, the user will be prompted during intialization.

keep_password: type: 'boolean'

If True (the default), the password will be cached in memory after it is first obtained from the user, so that the user would not be prompted for it again.

default: True

unrooted: type: 'boolean'

Indicate that the target should be considered unrooted; do not attempt sudo or ask the user for their password.

core_names: type: 'list_of_strs'

List of names of CPU cores in the order that they appear to the kernel. If not specified, it will be inferred from the platform.

core_clusters: type: 'list_of_ints'

Cluster mapping corresponding to the cores in core_names. Cluster indexing starts at 0. If not specified, this will be inferred from core_names – consecutive cores with the same name will be assumed to share a cluster.

big_core: type: 'str'

The name of the big cores in a big.LITTLE system. If not specified, this will be inferred, either from the name (if one of the names in core_names matches known big cores), or by assuming that the last cluster is big.

model: type: 'str'

Hardware model of the platform. If not specified, an attempt will be made to read it from target.

modules: type: 'list'

An additional list of modules to be loaded into the target.

working_directory: type: 'str'

On-target working directory that will be used by WA. This directory must be writable by the user WA logs in as without the need for privilege elevation.

executables_directory: type: 'str'

On-target directory where WA will install its executable binaries. This location must allow execution. This location does *not* need to be writable by unprivileged users or rooted devices (WA will install with elevated privileges as necessary).

modules: type: 'list'

A list of additional modules to be installed for the target.

devlib implements functionality for particular subsystems as modules. A number of "default" modules (e.g. for cpufreq subsystem) are loaded automatically, unless explicitly disabled. If additional modules need to be loaded, they may be specified using this parameter.

Please see devlib documentation for information on the available modules.

load_default_modules: type: 'boolean'

A number of modules (e.g. for working with the cpufreq subsystem) are loaded by default when a Target is instantiated. Setting this to True would suppress that, ensuring that only the base Target interface is initialized.

You may want to set this to False if there is a problem with one or more default modules on your platform (e.g. your device is unrooted and cpufreq is not accessible to unprivileged users), or if Target initialization is taking too long for your platform.

default: True

shell_prompt: type: 'regex'

A regex that matches the shell prompt on the target.

default: r'^.*(shell|root|juno)@?.*:[/~]\S* *[#$] '

juno_android

Device Parameters:

device: type: 'str'

ADB device name

aliases: 'adb_name'

adb_server: type: 'str'

ADB server to connect to.

core_names: type: 'list_of_strs'

List of names of CPU cores in the order that they appear to the kernel. If not specified, it will be inferred from the platform.

core_clusters: type: 'list_of_ints'

Cluster mapping corresponding to the cores in core_names. Cluster indexing starts at 0. If not specified, this will be inferred from core_names – consecutive cores with the same name will be assumed to share a cluster.

big_core: type: 'str'

The name of the big cores in a big.LITTLE system. If not specified, this will be inferred, either from the name (if one of the names in core_names matches known big cores), or by assuming that the last cluster is big.

model: type: 'str'

Hardware model of the platform. If not specified, an attempt will be made to read it from target.

modules: type: 'list'

An additional list of modules to be loaded into the target.

serial_port: type: 'str'

The serial device/port on the host for the initial connection to the target (used for early boot, flashing, etc).

baudrate: type: 'integer'

Baud rate for the serial connection.

default: 115200

vemsd_mount: type: 'str'

VExpress MicroSD card mount location. This is a MicroSD card in the VExpress device that is mounted on the host via USB. The card contains configuration files for the platform and firmware and kernel images to be flashed.

default: '/media/JUNO'

bootloader: type: 'str'

Selects the bootloader mechanism used by the board. Depending on firmware version, a number of possible boot mechanisms may be use.

Please see devlib documentation for descriptions.

allowed values: 'uefi', 'uefi-shell', 'u-boot', 'bootmon'

default: 'u-boot'

hard_reset_method: type: 'str'

There are a couple of ways to reset VersatileExpress board if the software running on the board becomes unresponsive. Both require configuration to be enabled (please see devlib documentation).

dtr: toggle the DTR line on the serial connection reboottxt: create reboot.txt in the root of the VEMSD mount.

allowed values: 'dtr', 'reboottxt'

default: 'dtr'

working_directory: type: 'str'

On-target working directory that will be used by WA. This directory must be writable by the user WA logs in as without the need for privilege elevation.

executables_directory: type: 'str'

On-target directory where WA will install its executable binaries. This location must allow execution. This location does *not* need to be writable by unprivileged users or rooted devices (WA will install with elevated privileges as necessary).

modules: type: 'list'

A list of additional modules to be installed for the target.

devlib implements functionality for particular subsystems as modules. A number of "default" modules (e.g. for cpufreq subsystem) are loaded automatically, unless explicitly disabled. If additional modules need to be loaded, they may be specified using this parameter.

Please see devlib documentation for information on the available modules.

load_default_modules: type: 'boolean'

A number of modules (e.g. for working with the cpufreq subsystem) are loaded by default when a Target is instantiated. Setting this to True would suppress that, ensuring that only the base Target interface is initialized.

You may want to set this to False if there is a problem with one or more default modules on your platform (e.g. your device is unrooted and cpufreq is not accessible to unprivileged users), or if Target initialization is taking too long for your platform.

default: True

shell_prompt: type: 'regex'

A regex that matches the shell prompt on the target.

default: r'^.*(shell|root|juno)@?.*:[/~]\S* *[#$] '

package_data_directory: type: 'str'

Directory containing Android data

default: '/data/data'

disable_selinux: type: 'boolean'

If True, the default, and the target is rooted, an attempt will be made to disable SELinux by running setenforce 0 on the target at the beginning of the run.

default: True

logcat_poll_period: type: 'integer'

Polling period for logcat in seconds. If not specified, no polling will be used.

Logcat buffer on android is of limited size and it cannot be adjusted at run time. Depending on the amount of logging activity, the buffer may not be enought to capture comlete trace for a workload execution. For those situations, logcat may be polled periodically during the course of the run and stored in a temporary locaiton on the host. Setting the value of the poll period enables this behavior.

constraint: value > 0

juno_linux

Device Parameters:

host: (mandatory) type: 'str'

Host name or IP address of the target.

username: (mandatory) type: 'str'

User name to connect with

password: type: 'str'

Password to use.

keyfile: type: 'str'

Key file to use

port: type: 'integer'

The port SSH server is listening on on the target.

telnet: type: 'boolean'

If set to True, a Telnet connection, rather than SSH will be used.

password_prompt: type: 'str'

Password prompt to expect

original_prompt: type: 'str'

Original shell prompt to expect.

sudo_cmd: type: 'str'

Sudo command to use. Must have {} specified somewhere in the string it indicate where the command to be run via sudo is to go.

default: 'sudo -- sh -c {}'

core_names: type: 'list_of_strs'

List of names of CPU cores in the order that they appear to the kernel. If not specified, it will be inferred from the platform.

core_clusters: type: 'list_of_ints'

Cluster mapping corresponding to the cores in core_names. Cluster indexing starts at 0. If not specified, this will be inferred from core_names – consecutive cores with the same name will be assumed to share a cluster.

big_core: type: 'str'

The name of the big cores in a big.LITTLE system. If not specified, this will be inferred, either from the name (if one of the names in core_names matches known big cores), or by assuming that the last cluster is big.

model: type: 'str'

Hardware model of the platform. If not specified, an attempt will be made to read it from target.

modules: type: 'list'

An additional list of modules to be loaded into the target.

serial_port: type: 'str'

The serial device/port on the host for the initial connection to the target (used for early boot, flashing, etc).

baudrate: type: 'integer'

Baud rate for the serial connection.

default: 115200

vemsd_mount: type: 'str'

VExpress MicroSD card mount location. This is a MicroSD card in the VExpress device that is mounted on the host via USB. The card contains configuration files for the platform and firmware and kernel images to be flashed.

default: '/media/JUNO'

bootloader: type: 'str'

Selects the bootloader mechanism used by the board. Depending on firmware version, a number of possible boot mechanisms may be use.

Please see devlib documentation for descriptions.

allowed values: 'uefi', 'uefi-shell', 'u-boot', 'bootmon'

default: 'u-boot'

hard_reset_method: type: 'str'

There are a couple of ways to reset VersatileExpress board if the software running on the board becomes unresponsive. Both require configuration to be enabled (please see devlib documentation).

dtr: toggle the DTR line on the serial connection reboottxt: create reboot.txt in the root of the VEMSD mount.

allowed values: 'dtr', 'reboottxt'

default: 'dtr'

working_directory: type: 'str'

On-target working directory that will be used by WA. This directory must be writable by the user WA logs in as without the need for privilege elevation.

executables_directory: type: 'str'

On-target directory where WA will install its executable binaries. This location must allow execution. This location does *not* need to be writable by unprivileged users or rooted devices (WA will install with elevated privileges as necessary).

modules: type: 'list'

A list of additional modules to be installed for the target.

devlib implements functionality for particular subsystems as modules. A number of "default" modules (e.g. for cpufreq subsystem) are loaded automatically, unless explicitly disabled. If additional modules need to be loaded, they may be specified using this parameter.

Please see devlib documentation for information on the available modules.

load_default_modules: type: 'boolean'

A number of modules (e.g. for working with the cpufreq subsystem) are loaded by default when a Target is instantiated. Setting this to True would suppress that, ensuring that only the base Target interface is initialized.

You may want to set this to False if there is a problem with one or more default modules on your platform (e.g. your device is unrooted and cpufreq is not accessible to unprivileged users), or if Target initialization is taking too long for your platform.

default: True

shell_prompt: type: 'regex'

A regex that matches the shell prompt on the target.

default: r'^.*(shell|root|juno)@?.*:[/~]\S* *[#$] '

CHAPTER 5

API

5.1 Workload Automation API

5.1.1 Output

A WA output directory can be accessed via a *RunOutput* object. There are two ways of getting one – either instantiate it with a path to a WA output directory, or use *discover_wa_outputs()* to traverse a directory tree iterating over all WA output directories found.

discover_wa_outputs(*path*)
Recursively traverse `path` looking for WA output directories. Return an iterator over *RunOutput* objects for each discovered output.

Parameters **path** – The directory to scan for WA output

class RunOutput(*path*)
The main interface into a WA output directory.

Parameters **path** – must be the path to the top-level output directory (the one containing `__meta` subdirectory and `run.log`).

WA output stored in a Postgres database by the `Postgres` output processor can be accessed via a *RunDatabaseOutput* which can be initialized as follows:

class RunDatabaseOutput(*password*, *host='localhost'*, *user='postgres'*, *port='5432'*, *dbname='wa'*, *run_uuid=None*, *list_runs=False*)
The main interface into Postgres database containing WA results.

Parameters

- **password** – The password used to authenticate with
- **host** – The database host address. Defaults to `'localhost'`
- **user** – The user name used to authenticate with. Defaults to `'postgres'`
- **port** – The database connection port number. Defaults to `'5432'`

- **dbname** – The database name. Defaults to `'wa'`
- **run_uuid** – The `run_uuid` to identify the selected run
- **list_runs** – Will connect to the database and will print out the available runs with their corresponding run_uuids. Defaults to `False`

Example

See also:

Processing WA Output

To demonstrate how we can use the output API if we have an existing WA output called `wa_output` in the current working directory we can initialize a `RunOutput` as follows:

```
In [1]: from wa import RunOutput
   ...:
   ...: output_directory = 'wa_output'
   ...: run_output = RunOutput(output_directory)
```

Alternatively if the results have been stored in a Postgres database we can initialize a `RunDatabaseOutput` as follows:

```
In [1]: from wa import RunDatabaseOutput
   ...:
   ...: db_settings = {
   ...:                 host: 'localhost',
   ...:                 port: '5432',
   ...:                 dbname: 'wa'
   ...:                 user: 'postgres',
   ...:                 password: 'wa'
   ...:                 }
   ...:
   ...: RunDatabaseOutput(list_runs=True, **db_settings)
Available runs are:
========= ============ ============= ================== ================== ␣
↪====================================
 Run Name      Project Project Stage          Start Time            End Time          ␣
↪                    run_uuid
========= ============ ============= ================== ================== ␣
↪====================================
Test Run    my_project          None 2018-11-29 14:53:08 2018-11-29 14:53:24 aa3077eb-
↪241a-41d3-9610-245fd4e552a9
run_1       my_project          None 2018-11-29 14:53:34 2018-11-29 14:53:37 4c2885c9-
↪2f4a-49a1-bbc5-b010f8d6b12a
========= ============ ============= ================== ================== ␣
↪====================================

In [2]: run_uuid = '4c2885c9-2f4a-49a1-bbc5-b010f8d6b12a'
   ...: run_output = RunDatabaseOutput(run_uuid=run_uuid, **db_settings)
```

From here we can retrieve various information about the run. For example if we want to see what the overall status of the run was, along with the runtime parameters and the metrics recorded from the first job was we can do the following:

```
In [2]: run_output.status
Out[2]: OK(7)
```

(continues on next page)

(continued from previous page)

```
# List all of the jobs for the run
In [3]: run_output.jobs
Out[3]:
[<wa.framework.output.JobOutput at 0x7f70358a1f10>,
 <wa.framework.output.JobOutput at 0x7f70358a1150>,
 <wa.framework.output.JobOutput at 0x7f7035862810>,
 <wa.framework.output.JobOutput at 0x7f7035875090>]

# Examine the first job that was ran
In [4]: job_1 = run_output.jobs[0]

In [5]: job_1.label
Out[5]: u'dhrystone'

# Print out all the runtime parameters and their values for this job
In [6]: for k, v in job_1.spec.runtime_parameters.items():
   ...:     print (k, v)
(u'airplane_mode': False)
(u'brightness': 100)
(u'governor': 'userspace')
(u'big_frequency': 1700000)
(u'little_frequency': 1400000)

# Print out all the metrics available for this job
In [7]: job_1.metrics
Out[7]:
[<thread 0 score: 14423105 (+)>,
 <thread 0 DMIPS: 8209 (+)>,
 <thread 1 score: 14423105 (+)>,
 <thread 1 DMIPS: 8209 (+)>,
 <thread 2 score: 14423105 (+)>,
 <thread 2 DMIPS: 8209 (+)>,
 <thread 3 score: 18292638 (+)>,
 <thread 3 DMIPS: 10411 (+)>,
 <thread 4 score: 17045532 (+)>,
 <thread 4 DMIPS: 9701 (+)>,
 <thread 5 score: 14150917 (+)>,
 <thread 5 DMIPS: 8054 (+)>,
 <time: 0.184497 seconds (-)>,
 <total DMIPS: 52793 (+)>,
 <total score: 92758402 (+)>]

# Load the run results csv file into pandas
In [7]: pd.read_csv(run_output.get_artifact_path('run_result_csv'))
Out[7]:
            id   workload  iteration          metric          value    units
0   450000-wk1  dhrystone          1  thread 0 score   1.442310e+07      NaN
1   450000-wk1  dhrystone          1  thread 0 DMIPS   8.209700e+04      NaN
2   450000-wk1  dhrystone          1  thread 1 score   1.442310e+07      NaN
3   450000-wk1  dhrystone          1  thread 1 DMIPS   8.720900e+04      NaN
...
```

We can also retrieve information about the target that the run was performed on for example:

```
# Print out the target's abi:
In [9]: run_output.target_info.abi
Out[9]: u'arm64'
```

(continues on next page)

(continued from previous page)

```
# The os the target was running
In [9]: run_output.target_info.os
Out[9]: u'android'

# And other information about the os version
In [10]: run_output.target_info.os_version
Out[10]:
OrderedDict([(u'all_codenames', u'REL'),
             (u'incremental', u'3687331'),
             (u'preview_sdk', u'0'),
             (u'base_os', u''),
             (u'release', u'7.1.1'),
             (u'codename', u'REL'),
             (u'security_patch', u'2017-03-05'),
             (u'sdk', u'25')])
```

RunOutput

RunOutput provides access to the output of a WA *run*, including metrics, artifacts, metadata, and configuration. It has the following attributes:

jobs A list of `JobOutput` objects for each job that was executed during the run.

status Run status. This indicates whether the run has completed without problems (`Status.OK`) or if there were issues.

metrics A list of `Metrics` for the run.

> **Note:** these are *overall run* metrics only. Metrics for individual jobs are contained within the corresponding `JobOutputs`.

artifacts A list of `Artifacts` for the run. These are usually backed by a file and can contain traces, raw data, logs, etc.

> **Note:** these are *overall run* artifacts only. Artifacts for individual jobs are contained within the corresponding `JobOutputs`.

info A *RunInfo* object that contains information about the run itself for example it's duration, name, uuid etc.

target_info A *TargetInfo* object which can be used to access various information about the target that was used during the run for example it's `abi`, `hostname`, `os` etc.

run_config A *RunConfiguration* object that can be used to access all the configuration of the run itself, for example the `reboot_policy`, `execution_order`, `device_config` etc.

classifiers *classifiers* defined for the entire run.

metadata *metadata* associated with the run.

events A list of any events logged during the run, that are not associated with a particular job.

event_summary A condensed summary of any events that occurred during the run.

augmentations A list of the *augmentation*s that were enabled during the run (these augmentations may or may not have been active for a particular job).

basepath A (relative) path to the WA output directory backing this object.

methods

RunOutput.**get_artifact**(*name*)
Return the Artifact specified by name. This will only look at the run artifacts; this will not search the artifacts of the individual jobs.

Parameters **name** – The name of the artifact who's path to retrieve.

Returns The Artifact with that name

Raises **HostError** – If the artifact with the specified name does not exist.

RunOutput.**get_artifact_path**(*name*)
Return the path to the file backing the artifact specified by name. This will only look at the run artifacts; this will not search the artifacts of the individual jobs.

Parameters **name** – The name of the artifact who's path to retrieve.

Returns The path to the artifact

Raises **HostError** – If the artifact with the specified name does not exist.

RunOutput.**get_metric**(*name*)
Return the Metric associated with the run (not the individual jobs) with the specified *name*.

Returns The Metric object for the metric with the specified name.

RunOutput.**get_job_spec**(*spec_id*)
Return the JobSpec with the specified *spec_id*. A *spec* describes the job to be executed. Each Job has an associated JobSpec, though a single *spec* can be associated with multiple *job*s (If the *spec* specifies multiple iterations).

RunOutput.**list_workloads**()
List unique workload labels that featured in this run. The labels will be in the order in which they first ran.

Returns A list of *str* labels of workloads that were part of this run.

RunDatabaseOutput

RunDatabaseOutput provides access to the output of a WA *run*, including metrics,artifacts, metadata, and configuration stored in a postgres database. The majority of attributes and methods are the same *RunOutput* however the noticeable differences are:

jobs A list of JobDatabaseOutput objects for each job that was executed during the run.

basepath A representation of the current database and host information backing this object.

methods

RunDatabaseOutput.**get_artifact**(*name*)
Return the Artifact specified by name. This will only look at the run artifacts; this will not search the artifacts of the individual jobs. The *path* attribute of the Artifact will be set to the Database OID of the object.

Parameters **name** – The name of the artifact who's path to retrieve.

Returns The Artifact with that name

Raises **HostError** – If the artifact with the specified name does not exist.

RunDatabaseOutput.**get_artifact_path**(*name*)

Returns a *StringIO* object containing the contents of the artifact specified by `name`. This will only look at the run artifacts; this will not search the artifacts of the individual jobs.

Parameters **name** – The name of the artifact who's path to retrieve.

Returns A *StringIO* object with the contents of the artifact

Raises **HostError** – If the artifact with the specified name does not exist.

JobOutput

`JobOutput` provides access to the output of a single *job* executed during a WA *run*, including metrics, artifacts, metadata, and configuration. It has the following attributes:

status Job status. This indicates whether the job has completed without problems (`Status.OK`) or if there were issues.

Note: Under typical configuration, WA will make a number of attempts to re-run a job in case of issue. This status (and the rest of the output) will represent the the latest attempt. I.e. a `Status.OK` indicates that the latest attempt was successful, but it does mean that there weren't prior failures. You can check the `retry` attribute (see below) to whether this was the first attempt or not.

---

retry Retry number for this job. If a problem is detected during job execution, the job will be re-run up to `max_retries` times. This indicates the final retry number for the output. A value of `0` indicates that the job succeeded on the first attempt, and no retries were necessary.

Note: Outputs for previous attempts are moved into `__failed` subdirectory of WA output. These are currently not exposed via the API.

---

id The ID of the *spec* associated with with job. This ID is unique to the spec, but not necessary to the job – jobs representing multiple iterations of the same spec will share the ID.

iteration The iteration number of this job. Together with the `id` (above), this uniquely identifies a job with a run.

label The workload label associated with this job. Usually, this will be the name or *alias* of the workload, however maybe overwritten by the user in the *agenda*.

metrics A list of `Metrics` for the job.

artifacts A list of `Artifacts` for the job These are usually backed by a file and can contain traces, raw data, logs, etc.

classifiers *classifiers* defined for the job.

metadata *metadata* associated with the job.

events A list of any events logged during the execution of the job.

event_summary A condensed summary of any events that occurred during the execution of the job.

augmentations A list of the *augmentation*s that were enabled for this job. This may be different from overall augmentations specified for the run, as they may be enabled/disabled on per-job basis.

basepath A (relative) path to the WA output directory backing this object.

methods

RunOutput.**get_artifact**(*name*)
Return the Artifact specified by name associated with this job.

Parameters **name** – The name of the artifact to retrieve.

Returns The Artifact with that name

Raises **HostError** – If the artifact with the specified name does not exist.

RunOutput.**get_artifact_path**(*name*)
Return the path to the file backing the artifact specified by name, associated with this job.

Parameters **name** – The name of the artifact who's path to retrieve.

Returns The path to the artifact

Raises **HostError** – If the artifact with the specified name does not exist.

RunOutput.**get_metric**(*name*)
Return the Metric associated with this job with the specified *name*.

Returns The Metric object for the metric with the specified name.

JobDatabaseOutput

JobOutput provides access to the output of a single *job* executed during a WA *run*, including metrics, artifacts, metadata, and configuration stored in a postgres database. The majority of attributes and methods are the same JobOutput however the noticeable differences are:

basepath A representation of the current database and host information backing this object.

methods

JobDatabaseOutput.**get_artifact**(*name*)
Return the Artifact specified by name associated with this job. The *path* attribute of the Artifact will be set to the Database OID of the object.

Parameters **name** – The name of the artifact to retrieve.

Returns The Artifact with that name

Raises **HostError** – If the artifact with the specified name does not exist.

JobDatabaseOutput.**get_artifact_path**(*name*)
Returns a StringIO object containing the contents of the artifact specified by name associated with this job.

Parameters **name** – The name of the artifact who's path to retrieve.

Returns A *StringIO* object with the contents of the artifact

Raises **HostError** – If the artifact with the specified name does not exist.

Metric

A metric represent a single numerical measurement/score collected as a result of running the workload. It would be generated either by the workload or by one of the augmentations active during the execution of the workload.

A Metric has the following attributes:

name The name of the metric.

> **Note:** A name of the metric is not necessarily unique, even for the same job. Some workloads internally run multiple sub-tests, each generating a metric with the same name. In such cases, *classifier*s are used to distinguish between them.

value The value of the metrics collected.

units The units of the metrics. This maybe `None` if the metric has no units.

lower_is_better The default assumption is that higher metric values are better. This may be overridden by setting this to `True`, e.g. if metrics such as "run time" or "latency". WA does not use this internally (at the moment) but this may be used by external parties to sensibly process WA results in a generic way.

classifiers These can be user-defined *classifier*s propagated from the job/run, or they may have been added by the workload to help distinguish between otherwise identical metrics.

label This is a string constructed from the name and classifiers, to provide a more unique identifier, e.g. for grouping values across iterations. The format is in the form `name/cassifier1=value1/classifier2=value2/...`.

Artifact

An artifact is a file that is created on the host as part of executing a workload. This could be trace, logging, raw output, or pretty much anything else. Pretty much every file under WA output directory that is not already represented by some other framework object will have an `Artifact` associated with it.

An `Artifact` has the following attributes:

name The name of this artifact. This will be unique for the job/run (unlike metric names). This is intended as a consistent "handle" for this artifact. The actual file name for the artifact may vary from job to job (e.g. some benchmarks that create files with results include timestamps in the file names), however the name will always be the same.

path Partial path to the file associated with this artifact. Often, this is just the file name. To get the complete path that maybe used to access the file, use `get_artifact_path()` of the corresponding output object.

kind Describes the nature of this artifact to facilitate generic processing. Possible kinds are:

- **log** A log file. Not part of the "output" as such but contains information about the run/workload execution that be useful for diagnostics/meta analysis.
- **meta** A file containing metadata. This is not part of the "output", but contains information that may be necessary to reproduce the results (contrast with `log` artifacts which are *not* necessary).
- **data** This file contains new data, not available otherwise and should be considered part of the "output" generated by WA. Most traces would fall into this category.
- **export** Exported version of results or some other artifact. This signifies that this artifact does not contain any new data that is not available elsewhere and that it may be safely discarded without losing information.
- **raw** Signifies that this is a raw dump/log that is normally processed to extract useful information and is then discarded. In a sense, it is the opposite of `export`, but in general may also be discarded.

  > **Note:** Whether a file is marked as `log`/`data` or `raw` depends on how important it is to preserve this file, e.g. when archiving, vs how much space it takes up. Unlike `export` artifacts which are (almost) always ignored by other exporters as that would never result in data loss, `raw` files *may*

be processed by exporters if they decided that the risk of losing potentially (though unlikely) useful data is greater than the time/space cost of handling the artifact (e.g. a database uploader may choose to ignore `raw` artifacts, where as a network filer archiver may choose to archive them).

Note: The kind parameter is intended to represent the logical function of a particular artifact, not it's intended means of processing – this is left entirely up to the output processors.

description This may be used by the artifact's creator to provide additional free-form information about the artifact. In practice, this is often `None`

classifiers Job- and run-level *classifier*s will be propagated to the artifact.

Additional run info

RunOutput object has `target_info` and `run_info` attributes that contain structures that provide additional information about the run and device.

TargetInfo

The `TargetInfo` class presents various pieces of information about the target device. An instance of this class will be instantiated and populated automatically from the devlib target created during a WA run and serialized to a json file as part of the metadata exported by WA at the end of a run.

The available attributes of the class are as follows:

target The name of the target class that was uised ot interact with the device during the run E.g. `"AndroidTarget"`, `"LinuxTarget"` etc.

cpus A list of `CpuInfo` objects describing the capabilities of each CPU.

os A generic name of the OS the target was running (e.g. `"android"`).

os_version A dict that contains a mapping of OS version elements to their values. This mapping is OS-specific.

abi The ABI of the target device.

hostname The hostname of the the device the run was executed on.

is_rooted A boolean value specifying whether root was detected on the device.

kernel_version The version of the kernel on the target device. This returns a `KernelVersion` instance that has separate version and release fields.

kernel_config A `KernelConfig` instance that contains parsed kernel config from the target device. This may be `None` if the kernel config could not be extracted.

sched_features A list of the available tweaks to the scheduler, if available from the device.

hostid The unique identifier of the particular device the WA run was executed on.

RunInfo

The `RunInfo` provides general run information. It has the following attributes:

uuid A unique identifier for that particular run.

run_name The name of the run (if provided)

project The name of the project the run belongs to (if provided)

project_stage The project stage the run is associated with (if provided)

duration The length of time the run took to complete.

start_time The time the run was stared.

end_time The time at which the run finished.

5.1.2 Workloads

Workload

The base `Workload` interface is as follows, and is the base class for all *workload types*. For more information about to implement your own workload please see the *Developer How Tos*.

All instances of a workload will have the following attributes:

name This identifies the workload (e.g. it is used to specify the workload in the *agenda*).

phones_home This can be set to True to mark that this workload poses a risk of exposing information to the outside world about the device it runs on. For example a benchmark application that sends scores and device data to a database owned by the maintainer.

requires_network Set this to `True` to mark the the workload will fail without a network connection, this enables it to fail early with a clear message.

asset_directory Set this to specify a custom directory for assets to be pushed to, if unset the working directory will be used.

asset_files This can be used to automatically deploy additional assets to the device. If required the attribute should contain a list of file names that are required by the workload which will be attempted to be found by the resource getters

methods

`Workload.`**`init_resources`**(*context*)
This method may be optionally overridden to implement dynamic resource discovery for the workload. This method executes early on, before the device has been initialized, so it should only be used to initialize resources that do not depend on the device to resolve. This method is executed once per run for each workload instance.

Parameters **`context`** – The *Context* for the current run.

`Workload.`**`validate`**(*context*)
This method can be used to validate any assumptions your workload makes about the environment (e.g. that required files are present, environment variables are set, etc) and should raise a `wa.WorkloadError` if that is not the case. The base class implementation only makes sure sure that the name attribute has been set.

Parameters **`context`** – The *Context* for the current run.

`Workload.`**`initialize`**(*context*)
This method is decorated with the `@once_per_instance` decorator, (for more information please see *Execution Decorators*) therefore it will be executed exactly once per run (no matter how many instances of the workload there are). It will run after the device has been initialized, so it may be used to perform device-dependent initialization that does not need to be repeated on each iteration (e.g. as installing executables required by the workload on the device).

Parameters **context** – The *Context* for the current run.

Workload.**setup**(*context*)
Everything that needs to be in place for workload execution should be done in this method. This includes copying files to the device, starting up an application, configuring communications channels, etc.

Parameters **context** – The *Context* for the current run.

Workload.**setup_rerun**(*context*)
Everything that needs to be in place for workload execution should be done in this method. This includes copying files to the device, starting up an application, configuring communications channels, etc.

Parameters **context** – The *Context* for the current run.

Workload.**run**(*context*)
This method should perform the actual task that is being measured. When this method exits, the task is assumed to be complete.

Parameters **context** – The *Context* for the current run.

Note: Instruments are kicked off just before calling this method and disabled right after, so everything in this method is being measured. Therefore this method should contain the least code possible to perform the operations you are interested in measuring. Specifically, things like installing or starting applications, processing results, or copying files to/from the device should be done elsewhere if possible.

Workload.**extract_results**(*context*)
This method gets invoked after the task execution has finished and should be used to extract metrics from the target.

Parameters **context** – The *Context* for the current run.

Workload.**update_output**(*context*)
This method should be used to update the output within the specified execution context with the metrics and artifacts from this workload iteration.

Parameters **context** – The *Context* for the current run.

Workload.**teardown**(*context*)
This could be used to perform any cleanup you may wish to do, e.g. Uninstalling applications, deleting file on the device, etc.

Parameters **context** – The *Context* for the current run.

Workload.**finalize**(*context*)
This is the complement to `initialize`. This will be executed exactly once at the end of the run. This should be used to perform any final clean up (e.g. uninstalling binaries installed in the `initialize`)

Parameters **context** – The *Context* for the current run.

ApkWorkload

The `ApkWorkload` derives from the base `Workload` class however this associates the workload with a package allowing for an apk to be found for the workload, setup and ran on the device before running the workload.

In addition to the attributes mentioned above ApkWorloads this class also features the following attributes however this class does not present any new methods.

loading_time This is the time in seconds that WA will wait for the application to load before continuing with the run. By default this will wait 10 second however if your application under test requires additional time this values should be increased.

package_names This attribute should be a list of Apk packages names that are suitable for this workload. Both the host (in the relevant resource locations) and device will be searched for an application with a matching package name.

supported_versions This attribute should be a list of apk versions that are suitable for this workload, if a specific apk version is not specified then any available supported version may be chosen.

view This is the "view" associated with the application. This is used by instruments like `fps` to monitor the current framerate being generated by the application.

apk The is a `PackageHandler`` which is what is used to store information about the apk and manage the application itself, the handler is used to call the associated methods to manipulate the application itself for example to launch/close it etc.

package This is a more convenient way to access the package name of the Apk that was found and being used for the run.

ApkUiautoWorkload

The `ApkUiautoWorkload` derives from `ApkUIWorkload` which is an intermediate class which in turn inherits from `ApkWorkload`, however in addition to associating an apk with the workload this class allows for automating the application with UiAutomator.

This class define these additional attributes:

gui This attribute will be an instance of a `UiAutmatorGUI` which is used to control the automation, and is what is used to pass parameters to the java class for example `gui.uiauto_params`.

ApkReventWorkload

The `ApkReventWorkload` derives from `ApkUIWorkload` which is an intermediate class which in turn inherits from `ApkWorkload`, however in addition to associating an apk with the workload this class allows for automating the application with *Revent*.

This class define these additional attributes:

gui This attribute will be an instance of a `ReventGUI` which is used to control the automation

setup_timeout This is the time allowed for replaying a recording for the setup stage.

run_timeout This is the time allowed for replaying a recording for the run stage.

extract_results_timeout This is the time allowed for replaying a recording for the extract results stage.

teardown_timeout This is the time allowed for replaying a recording for the teardown stage.

UiautoWorkload

The `UiautoWorkload` derives from `UIWorkload` which is an intermediate class which in turn inherits from `Workload`, however this allows for providing generic automation using UiAutomator without associating a particular application with the workload.

This class define these additional attributes:

gui This attribute will be an instance of a `UiAutmatorGUI` which is used to control the automation, and is what is used to pass parameters to the java class for example `gui.uiauto_params`.

ReventWorkload

The `ReventWorkload` derives from `UIWorkload` which is an intermediate class which in turn inherits from `Workload`, however this allows for providing generic automation using *Revent* without associating with the workload.

This class define these additional attributes:

`gui` This attribute will be an instance of a `ReventGUI` which is used to control the automation

`setup_timeout` This is the time allowed for replaying a recording for the setup stage.

`run_timeout` This is the time allowed for replaying a recording for the run stage.

`extract_results_timeout` This is the time allowed for replaying a recording for the extract results stage.

`teardown_timeout` This is the time allowed for replaying a recording for the teardown stage.

CHAPTER 6

Glossary

6.1 Glossary

Agenda An agenda specifies what is to be done during a Workload Automation run. This includes which workloads will be run, with what configuration and which augmentations will be enabled, etc. (For more information please see the *Agenda Reference*.)

Alias An alias associated with a workload or a parameter. In case of parameters, this is simply an alternative name for a parameter; Usually these are employed to provide backward compatibility for renamed parameters, or in cases where a there are several commonly used terms, each equally valid, for something.

In case of Workloads, aliases can also be merely alternatives to the workload name, however they can also alter the default values for the parameters the Workload is instantiated with. A common scenario is when a single workload can be run under several distinct configurations (e.g. has several alternative tests that might be run) that are configurable via a parameter. An alias may be added for each such configuration. In order to see the available aliases for a workload, one can use *show command*.

See also:

Global Alias

Artifact An artifact is something that was been generated as part of the run for example a file containing output or meta data in the form of log files. WA supports multiple "kinds" of artifacts and will handle them accordingly, for more information please see the *Developer Reference*.

Augmentation Augmentations are plugins that augment the execution of workload jobs with additional functionality; usually, that takes the form of generating additional metrics and/or artifacts, such as traces or logs. For more information please see *augmentations*.

Classifier An arbitrary key-value pair that may associated with a *job*, a *metric*, or an *artifact*. The key must be a string. The value can be any simple scalar type (string, integer, boolean, etc). These have no pre-defined meaning but may be used to aid filtering/grouping of metrics and artifacts during output processing.

See also:

Classifiers.

Global Alias Typically, values for plugin parameters are specified name spaced under the plugin's name in the configuration. A global alias is an alias that may be specified at the top level in configuration.

There two common reasons for this. First, several plugins might specify the same global alias for the same parameter, thus allowing all of them to be configured with one settings. Second, a plugin may not be exposed directly to the user (e.g. resource getters) so it makes more sense to treat its parameters as global configuration values.

See also:

Alias

Instrument A WA "Instrument" can be quite diverse in its functionality, but the majority of those available in are there to collect some kind of additional data (such as trace, energy readings etc.) from the device during workload execution. To see available instruments please use the *list command* or see the *Plugin Reference*.

Job An single execution of a workload. A job is defined by an associated *spec*. However, multiple jobs can share the same spec; E.g. Even if you only have 1 workload to run but wanted 5 iterations then 5 individual jobs will be generated to be run.

Metric A single numeric measurement or score collected during job execution.

Output Processor An "Output Processor" is what is used to process the output generated by a workload. They can simply store the results in a presentable format or use the information collected to generate additional metrics. To see available output processors please use the *list command* or see the *Plugin Reference*.

Run A single execution of *wa run* command. A run consists of one or more *job*s, and results in a single output directory structure containing job results and metadata.

Section A set of configurations for how jobs should be run. The settings in them take less precedence than workload-specific settings. For every section, all jobs will be run again, with the changes specified in the section's agenda entry. Sections are useful for several runs in which global settings change.

Spec A specification of a workload. For example you can have a single workload specification that is then executed multiple times if you desire multiple iterations but the configuration for the workload will remain the same. In WA2 the term "iteration" used to refer to the same underlying idea as spec now does. It should be noted however, that this is no longer the case and an iteration is merely a configuration point in WA3. Spec is to blueprint as job is to product.

WA Workload Automation. The full name of this framework.

Workload A workload is the lowest level specification for tasks that need to be run on a target. A workload can have multiple iterations, and be run additional multiples of times dependent on the number of sections.

CHAPTER 7

FAQ

7.1 FAQ

7.1.1 Q: I receive the error: "<<Workload> file <file_name> file> could not be found."

A: Some workload e.g. AdobeReader, GooglePhotos etc require external asset files. We host some additional workload dependencies in the WA Assets Repo. To allow WA to try and automatically download required assets from the repository please add the following to your configuration:

```
remote_assets_url: https://raw.githubusercontent.com/ARM-software/workload-automation-
↪assets/master/dependencies
```

7.1.2 Q: I receive the error: "No matching package found for workload <workload>"

A: WA cannot locate the application required for the workload. Please either install the application onto the device or source the apk and place into `$WA_USER_DIRECTORY/dependencies/<workload>`

7.1.3 Q: I am trying to set a valid runtime parameters however I still receive the error "Unknown runtime parameter"

A: Please ensure you have the corresponding module loaded on the device. See *Runtime Parameters* for the list of runtime parameters and their containing modules, and the appropriate section in *setting up a device* for ensuring it is installed.

7.1.4 Q: I have a big.LITTLE device but am unable to set parameters corresponding to the big or little core and receive the error "Unknown runtime parameter"

A: Please ensure you have the hot plugging module enabled for your device (Please see question above).

A: This can occur if the device uses dynamic hot-plugging and although WA will try to online all cores to perform discovery sometimes this can fail causing to WA to incorrectly assume that only one cluster is present. To workaround this please set the `core_names` *parameter* in the configuration for your device.

7.1.5 Q: I receive the error Could not find plugin or alias "standard"

A: Upon first use of WA3, your WA2 config file typically located at `$USER_HOME/config.py` will have been converted to a WA3 config file located at `$USER_HOME/config.yaml`. The "standard" output processor, present in WA2, has been merged into the core framework and therefore no longer exists. To fix this error please remove the "standard" entry from the "augmentations" list in the WA3 config file.

7.1.6 Q: My Juno board keeps resetting upon starting WA even if it hasn't crashed.

A Please ensure that you do not have any other terminals (e.g. `screen` sessions) connected to the board's UART. When WA attempts to open the connection for its own use this can cause the board to reset if a connection is already present.

7.1.7 Q: I'm using the FPS instrument but I do not get any/correct results for my workload

A: If your device is running with Android 6.0 + then the default utility for collecting fps metrics will be `gfxinfo` however this does not seem to be able to extract any meaningful information for some workloads. In this case please try setting the `force_surfaceflinger` parameter for the `fps` augmentation to `True`. This will attempt to guess

the "View" for the workload automatically however this is device specific and therefore may need customizing. If this is required please open the application and execute `dumpsys SurfaceFlinger --list` on the device via adb. This will provide a list of all views available for measuring.

As an example, when trying to find the view for the AngryBirds Rio workload you may get something like:

```
...
AppWindowToken{41dfe54 token=Token{77819a7 ActivityRecord{a151266 u0 com.rovio.
↪angrybirdsrio/com.rovio.fusion.App t506}}}#0
a3d001c com.rovio.angrybirdsrio/com.rovio.fusion.App#0
Background for -SurfaceView - com.rovio.angrybirdsrio/com.rovio.fusion.App#0
SurfaceView - com.rovio.angrybirdsrio/com.rovio.fusion.App#0
com.rovio.angrybirdsrio/com.rovio.fusion.App#0
boostedAnimationLayer#0
mAboveAppWindowsContainers#0
...
```

From these `"SurfaceView - com.rovio.angrybirdsrio/com.rovio.fusion.App#0"` is the mostly likely the View that needs to be set as the `view` workload parameter and will be picked up be the `fps` augmentation.

7.1.8 Q: I am getting an error which looks similar to `'CONFIG_SND_BT87X is not exposed in kernel config'...`

A: If you are receiving this under normal operation this can be caused by a mismatch of your WA and devlib versions. Please update both to their latest versions and delete your `$USER_HOME/.workload_automation/cache/targets.json` (or equivalent) file.

Python Module Index

w

Index