1 wrstat

1.1 Description

Wrstat is a modularized performance analysis tool to collect system information of different types and origins. It makes use of the /proc file system or uses third party tools to gather information on CPU, kernel lock and disk usage. It is mostly written in Python because its wide availability, easy and powerful syntax. There are implemented modules for e.g OProfile, iostat and lock_stat. It has also support for sampling, which allows to capture time series data.

1.2 Installation

For a basic installation, you have to perform the following steps

1. Copy the wrstat directory to your Installation path
2. Link wrstat-run to e.g. /usr/local/bin
3. Create wrstat.config based on the given template

1.3 wrstat.config

The wrstat.config file contains the main tool and module configuration. To run wrstat, the following options are required:

modules Space separated list of modules invoked by wrstat. Modules will be called in the same order as supplied, except for the sampling.

runasroot If true, the daemon will be called by sudo to give it root privileges. This is required for some modules as for OProfile and lock_stat.

interval Time span between samples.

1.4 Usage

To run a test program run:

```
wrstat-run path/to/test/directory program args ..
```

While running a test, wrstat-daemon.py will invoke several modules to collect data of different type. The modules to run are specified in wrstat.config.

Wrstat implements the following modules

Name	Description
stat	Handles /proc/stat
diskstats	Handles /proc/diskstats
lock_stat	Handles /proc/lock_stat
oprofile	Wrapper and visualization for OProfile
iostat	Disk I/O throughput (deprecated)

When the test program finished, the collected data will be parsed by wrstart-parser.py and exported to samples.pickle. This data will then be graphed by wrstat-graph.py All steps are triggered by the wrstat-run main script.

1.5 Test Directory Structure

This directory contains all test data and graphs. If the test directory does not exists it will be created, otherwise the results will been overwritten. This directory contains by default the following files:

wrstat.config Snapshot of wrstat.config took at the beginning of a test run. This file can be used by any module to store additional configuration parameters. All modules and scripts will use this file instead of the global one, so later changes for e.g. filtering will not affect all test runs.

cmd The actual command that was evaluated.

output Stdout and stderr of the test command.

time Output of time command.

samples.pickle Parsed data of all modules. This file is created by wrstat-parser.py which contains a dictionary, that maps the module name on its data.

samples/ Directory that will be used for all modules to store raw samples, which are created by sample() method

***.svg** Multiple graphs generated by wrstat-graph.

1.6 Tool Directory Structure

Wrstat consists of the following

wrstat-run test_dir cmd arg1 arg2 ... This is the main program written in bash, that glues all components together. It creates the basic test directory structure, calls the actual test command and initiates pre- and postprocessing.

wrstat-filer-create list of ELF binaries ¿ path/new_filter This script can be used to create custom symbol name filter for OProfile and lock_stat. It uses objdump to extract the symbol table in .text from a list ELF binaries (e.g. .ko files) of interest. This function names can be used to reduce output. To extract symbol names, non stripped binaries are required, but no debug information.

wrstat-daemon.py test_dir This program runs in background as long as the test command. It will first initiate all modules and signals wrstat-run to start the test. As the test runs it will invoke all modules to take samples at the interval specified in the config file and in the end deinitialize all modules. The actual sampling will just take raw snapshots of e.g. /proc files, to avoid lags and unnecessary CPU load. A delayed (queued) copy mechanism from utils.py is used to avoid lags due to IO.

wrstat-parser.py test_dir This component will sanitize and parse the captured data and store the results in samples.pcikle.

wrstat-graph.py test_dir This will create actual graphs from samples.pickle using gnuplot.

graphing.py (Python module) This simple python module has predefined methods for series and histogram graphs based on gnuplot. Per default it will create Scalable Vector Graphics (.svg) output.

utils.py (Python module) This module implements some basic functionality such as config, module loading and the delayed copy.

1.7 Modules

A modules is used to collect data from one specific origin, e.g. /proc/lock_stat or OProfile, that provide the actual functionality. For additional configuration each module can add parameters to wrstat.config file.

Each module is a python script containing the following methods:

def presampling(test_dir): This method is called once at the beginning of a test. It is usually used for initialization.

def sample(test_dir, t): This method is called with the specified interval and is used to take samples for any kind of time series data. To mitigate lag problems this each module has its own sampling thread to ensure stable sampling. The parameter t is the number of the current sample. In case of copying a file it is recommended to use the copy_queued method from utils.py.

def postsampling(test_dir): This method is called once the test is finished. It can by used for any kind of deinitialization or postprocessing.

def parse(test_dir): This method will be called by wrstat-parser.py and will parse and return the data captured by wrstat-daemon.

def plot(test_dir, data, interval): This method will be called by wrstat-graph and will create all needed graphs. The provided data was previously created by parse().

To make a module usable by wrstat, copy the script to the tool directory and add the module name to wrstat.config. The module name is the name of the file without extension.

1.8 Deamon

Foremost the daemon will call each modules presampling() and then create individual threads for each module to avoid interference. To realize timing and thread synchronization the daemon uses a condition variable, which will be used by the the daemon as follows:

def sample(module, modname): This is the actual sampling thread for a single module. It calls the module.sample() method and wait for the condition variable to take the next sample.

def timer(interval): This is the main thread which is responsible to keep a stable sampling rate. It will notify all threads waiting on the condition variable and sleep for the specified interval.

def signal_handler(signal, frame): This method will be called from the main thread, if a SIGINT or SIGTERM is received and signals all sampling threads to finish. To avoid zombie threads that are still waiting on the condition variable, it will notify all threads as long as there are threads alive. Finally int will call each modules postsampling() method.

The implementation of the queued (and threaded) copy from utils.py is similar to the daemon, as it uses an extra thread for dumping the data from queue to disk, that is signaled at any copy.

1.9 samples.pickle

This file contains all data parsed by wrstat and can be loaded by python using pickle for custom postprocessing. Such a file can be loaded (also from python prompt) by the following commands.

```
import pickle #required only once
f = open( "test_dir/samples.pickle", 'r')
samples = pickle.load( f)
f.close()
```

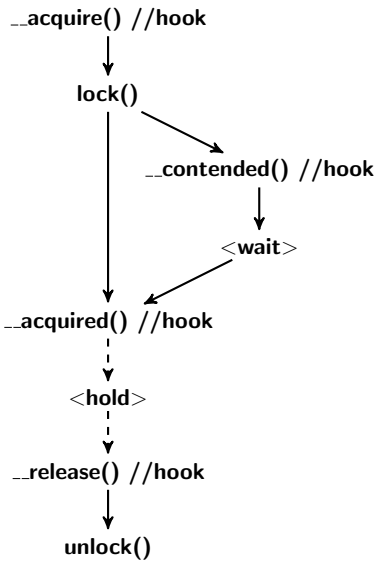
Writing works similar but uses "pickle.dump(samples, f)" instead of the load command. This file contains a nested array and dictionary structure, that is similar to JSON. The top level data structure maps the module name to the data returned by parse() and might look like the following:

```
{
    "stat" : ... ,
    "iostat" : ... ,
    "oprofile" : ...
}
```

2 /proc/lock_stat

The kernel lock usage statistic is available via /proc/lock_stat and has to be enabled manually on the most systems, by recompiling the kernel. The reason for this, that this feature make use of Lockdep, which will hook several kernel lock methods, maps the lock instances to a lock classes of the same type and though create slightly additional load. A lock class is a collection of actual lock instances, that are used in the same context, for example in a file system driver, each inode has its own lock instance, but they are classified as the same lock class as they are used in the same code regions.

According to the documentary the actual hooking mechanism is implemented as follows:



The hooks are used to track the state of lock manipulating methods. A lock is called contended, if an acquisition has to wait. Wait time is defined as the the time between the __contended and __acquired hooks. As we are dealing with multiple lock instances, wait and hold times of more than one second per second is possible. The line of code, that tried to acquire a lock, is contended called contention point. In addition the acquisition, that caused the contention is also tracked. Lockstat will store up to four contention points and points, that caused a contention.

2.1 Wrstat Module

2.1.1 Source Files

The lock_stat module contains the following files:

lock_stat-reset.sh test_directory This script will be called by the pre-sampling method and will reset lock_stat. This done by writing "0" to /proc/lock_stat. According to the documentary this should also disable lock_stat, but this behavior was not observed on any system we have used.

lock_stat.py The actual modules used by wrstat.

2.1.2 Filtering

This module supports filtering for symbol names, but it will not use the actual lock class name, but corresponding contention points. So if a lock class was contended or caused a contention by a method listed in the filter, it will pass. Otherwise it will be discarded. Filters can be passed by the lock_stat_filter option in wrstat.config as space separated list. For each filter separated graphs are created.

2.1.3 Created Files in Test Directory

For the graphs (.svg files) the corresponding patterns are listed. This module will create the following files in the test directory.

samples/lock_stat_*.svg: Snapshot of /proc/lock_stat took at the specified time step.

lock_stat-*-holdtime.svg and lock_stat-*-waittime.svg Stacked bar chart of the locks either for total wait- or holdtime.

lock_stat-*-holdtime-top.svg and lock_stat-*-waittime-top.svg: Time series graph for either wait- or holdtime for the top waiting locks.

lock_stat-*-holdtime-top-*.svg: For each lock class listed in the top-graphs, a corresponding detailed lock view is created. This shows, wait- and holdtime, acquisition and contentions of the lock as time series. In addition there are bar charts for contentions points and points, we are contendning with.

lock_stat-filter-*.svg: Filtered graphs for the given filter, or all filters.

3 OProfile

OProfile is a system wide performance analysis tool, to analyze the runtime of single applications and even of kernel methods. It will periodically issue non maskable interrupts (NMI) and update a counter corresponding to the currently interrupted code.

3.1 Wrstat Module

3.1.1 Source Files

The OProfile module contains the following files:

oprofile-init.sh test_directory This script will initialize OProfile for the specified test directory using either opcontrol or operf. This script will be called by oprofile.py presampling().

oprofile-deinit.sh test_directory This script will deinitialize OProfile and write the opreport output to samples/oprofile.

oprofile.py The actual modules used by this tool.

3.1.2 Created Files in Test Directory

The created graphs will show the CPU usage by applications (actual binaries) or symbol name (methods)

For the graphs (.svg files) the corresponding patterns are listed. This module will create the following files in the test directory.

oprofile_data/ Contains all collected data and will be passed to OProfile as –session-dir argument. It can be used to create custom results with opreport.

samples/oprofile This is the output of opreport -l (list all symbols) and the actual file that is beeing parsed.

oprofile-*-app.svg and oprogile-*-sym.svg Graphs grouped either by symbol or application names.

oprofile-*-aggregated-*.svg and oprofile-*-cpu*-*.svg Are either aggregated or per cpu graphs.

oprofile-filter-all-*.svg and oprofile-filter-path_to_filter-*.svg Graphs filtered by function names for a particuar or all filter in conjunction. If a filter creates an empty data set, its graph will not be created.

oprofile-app-filter-all*.svg and oprofile-app-filter-appname-*.svg Graphs filtered by application names for a particuar or all filter in conjunction. If a filter creates an empty data set, its graph will not be created.

3.1.3 Configuration

There are are the following configuration options available in wrstat.config.

oprofile_use_operf [true,false] If value is true, operf will be used, opcontrol otherwise. Although opcontrol is deprecated, it seems to work more stable than operf.

oprofile_event The default event, that is used by OPRofile for triggering NMI. The default event is

CPU_CLK_UNHALTED:100000:0:1:1

which will trigger a NMI every 100.000 cycles, the CPU is running in kernel or user mode. This can be used to adopt the sampling rate used by OPRofile.

oprofile_vmlinux Path to the current vmlinux ELF binary. It is not possible to use the vmlinuz file as this is zipped and also contains a raw decompressor, that will uncompress the kernel at runtime. Since verion 1.0.0 OProfile does not require the vmlinux image anymore to resolve symbol names, as it will use /proc/kallsyms instead.

oprofile_missing_binaries Comma separated list of paths to search for additional binaries (e.g. .ko files). Since Kernel version 2.6 this is needed to find modules, which are usually located in /lib/modules/version/kernel. It is also possible to pass user space binaries.

oprofile_sym_filter Space separated list of symbol name filters invoked by this module. Those are files containing function names to show and are specified by thier path relative to the tool directory.

oprofile_app_filter Space separated list of applications to filter for.

3.1.4 Filtering

As OProfile is configured to run in a system wide configuration there might be alot of methods listed, that are not of interest and mess up the results. For this reason this module implements filtering for symbol and application names. The most basic filtering method is the application filter, that will create individual plots for a given set of application names provided by OPRofile. Filtering for symbol names allows to create extra graphs for individual subsets of functions. A single filter is a file containing all function names of interest and can be created from a list of non stripped binaries by wrstat-filer-create. This even allows to monitor small subsystems of an application by using object files of specific source file.

3.2 Encountered Problems

3.2.1 opcontrol vs. operf

There are basically two different ways to start the OProfile daemon: operf and opcontrol, which is deprecated since version 0.9.8 and was removed in 1.0.0. Both offer mostly the same functionality but are slightly different in the usage: opcontrol is a tool to configure, start and stop the daemon whereas operf will is a one-line call to start the daemon, with specified options. Since either OProfile or operf could cause problems or is even unavailable (depending on operation system, machine and version), wrstat supports both. To choose one option just modify oprofile_use_operf in wrstat.config.

3.3 Timer Mode

Hardware performance monitor counters (PMC) will count hardware events and issues an interrupt on counter overflow. Per default OPRofile will use events such as CPU_CLK_UNHALTED to periodically interrupt the CPU. On some machines and CPU types (e.g. on virtual machines) PMC may be not available, so usual timer interrupts can be used, by passing the timer=1 to the OPRofile kernel module (prior v1.0.0).

4 /proc/stat

4.1 Description

/proc/stat gives basic information on the cpu usage. It measures the time that the CPU spent various states e.g. user mode, system, idle, virtual machine, etc. It can be used to obtain a standard cpu usage graph. This module requires no further configuration and only consists of one file stat.py.

4.2 Created Files in Test Directory

For the graphs (.svg files) the corresponding patterns are listed. This module will create the following files in the test directory.

samples/stat_* Snapshot of /proc/stat at the given time step.

stat-cpu*.svg and stat-aggregated.svg Per CPU and aggregated time series graphs.

stat-total.svg Average histogram, that show the overall usage for all CPUs in comparison.

5 /proc/diskstats

The file /proc/diskstats gives basic information on disk usage. It will give information on how many sectors read, written and time spent on IO for each device. This module consists only of the file diskstats.py . Even though the sectorsize is a hardcoded constant (512), this values are extracted from /sys/block/DEVICENAME/queue/hw_sector_size to avoid later unexpected issues.

5.1 /proc/diskstats Created Files in Test Directory

For the graphs (.svg files) the corresponding patterns are listed. This module will create the following files in the test directory.

samples/diskstats_* Snapshot of /proc/diskstats at the given time step.

diskstats-*.svg Read and write acces for a specific device.

diskstats-*-time.svg Graph that shows the time spent on reading or writing for a given device.

5.2 iostat

For debug reasons, there is also a module for iostat, which will handle /proc/diskstats and give basic IO throughput, but no further information. So this module is deprecated. This module will create the following files in the test directory. For the graphs (.svg files) the corresponding patterns are listed.

samples/iostat Output of iostat -d at the given interval.

iostat.pid Contains the id of the running iostat process. This is only used to shut down iostat without using e.g. killall.

sectorsizes Sectorsizes for all devices.

iostat-read.svg and iostat-write.svg This graphs will show read or write access for all devices in comparison.

iostat-*.svg Read and write acces for a specific device.