Debian Package usage profiler for Debian based Systems

Spring 2019, Prof. David Irwin

Bharath Honnesara Sreenivasa
bsreenivasa@umass.edu
University of Massachusetts, Amherst

Ajay Rajan
arajan@umass.edu
University of Massachusetts, Amherst

ABSTRACT

The embedded devices of today due to their CPU, RAM capabilities can run various Linux distributions but in most cases they are different from general purpose distributions as they are usually lighter and specific to the needs of that particular system. In this project, we share the problems associated in adopting a fully heavyweight Debian based system like Ubuntu in embedded/automotive platforms and provide solutions to optimize them to identify unused/redundant content in the system. This helps developer to reduce the hefty general purpose distribution to an application specific distribution. The solution involves collecting usage data in the system in a non-invasive manner (to avoid any drop in performance) to suggest users the redundant, unused parts of the system that can be safely removed without impacting the system functionality.

CCS CONCEPTS

• **Computer systems organization → Embedded systems**; *Redundancy*.

KEYWORDS

Linux, Embedded Systems

1 INTRODUCTION

Now-a-days embedded devices like IOT systems, hand-held phones, smart TV devices and automotive platforms have become powerful enough (with capable low-power, high performance CPU's and larger memory) to run a full Linux distribution like Ubuntu. One typical system is ARM (64-bit) architecture powered NVIDIA Tegra SOC platform DRIVE-AGX running Ubuntu based DRIVE SDK for automotive applications.

We see that when deploying OS like Ubuntu to such automotive platforms, the considerations are very different from general purpose desktop computers. Automotive platforms solve a domain-specific problem or may be used to develop software to solve a domain-specific problem but general purpose desktop Ubuntu usage is open-ended. The ecosystem of software in the Ubuntu distribution (provided by Canonical, the company backing Ubuntu) is very well suited for general purpose desktop uses. For embedded/automotive cases, there are two problems with Ubuntu:

- Ubuntu does not have suitable platform-supported images. The Ubuntu distributor expects hardware companies to start with (a very limited seed image called) Ubuntu-base and install software to user's /developer's needs.
- Users/Developers generally do not specifically know the list of software to install and end up installing a larger suite of software (generally by copying desktop configuration) which is much larger than what's required for the platform's use-case.

In order to solve these issues, we are introducing a tool which profiles Debian packages and provides a recommendation to the user based upon a score which informs the user about the unnecessary packages, so that the user can go ahead and remove them. The tool scores all the files present in the system based upon the read and write requests and the run time of the applications using those files. Each of these files are then mapped on to their respective packages to compute a score for each package and a package with higher score is a package which is being referenced more.

2 BACKGROUND AND MOTIVATION

The Ubuntu installation for embedded/automotive systems (as described in previous section) is less than ideal, causing several unnecessary software to get into the system. This additional redundant software in the system have the following problems:

- If its a daemon/service, it keeps running as a background process consuming memory and CPU
- In cases of debugging system issues, additional software exponentially increase number of variables to bisect and root cause the failure
- Unused additional software increases boot-up latency, update latency and latencies in normal workflow (if they are intertwined with user's processes)
- Unused additional software can potentially be doors to attack vectors, leading to a potential security risk

Currently these problems are being managed by the developers deploying the system and there are two ways in which this is being done:

- Get the full desktop level distribution and manually remove the packages which are not required
- Get the minimum distribution and manually install the packages which are required

The problem with the above solutions are that it is cumbersome and requires a high level of technical knowledge and even then many a time developers underestimate or overestimate the usage of many packages. Our method directly informs the users about the packages which are being used in their distribution, so that they can take an informed decision without any expert consultation.

3 ARCHITECTURE

The tool is responsible for scoring the files with regards to open, close, read and write references and with regards to run time and CPU time in case they are being used by some other process and in the main memory. It consists of two parts for capturing the references:

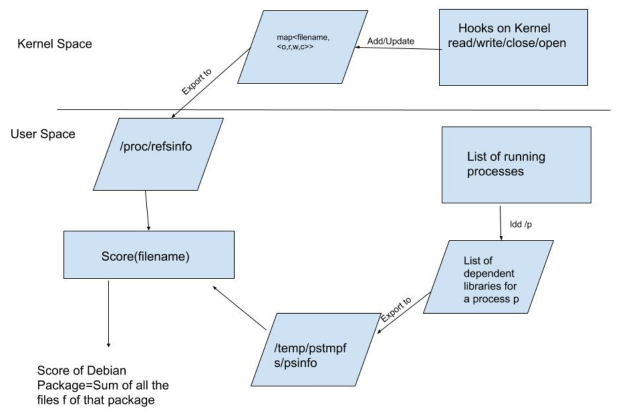
Kernel Space
map<filename, <o,r,w,c>>
Add/Update
Hooks on Kernel read/write/close/open
Export to
User Space
/proc/refsinfo
List of running processes
ldd /p
Score(filename)
List of dependent libraries for a process p
Export to
/temp/pstmpfs/psinfo
Score of Debian Package=Sum of all the files f of that package

Figure 1: Architecture of Scorer, interactions between Kernel and User space

- For the first part, we have modified the Linux Kernel and have added a hook to the open, close and read calls to catch all the open, close and read references to the file and have stored the results into a map which has the following structure *map<filename,<o,r,w,c»* which is used by the main scorer.
- For the second part, we have created a shell script which updates the list of all the running processes and creates a list of all the files which are being used by the process and that list is passed on to scorer.

The scorer gets the list from two programs and merges them using the scoring algorithm to generate a score for each file and in turn for all the packages.

4 DESIGN AND IMPLEMENTATION

The tool is used for scoring the packages and for that the tool needs to log all references to the package files. We are using two methods which together to log all the references.

4.1 Kernel-Space

In the first method we are to tracking all the open,close,read and write references to the files. In order to track all the references we have added hook in the following kernel functions :

- *SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, umode_t, mode)* - This is responsible for opening a file given it filename. We also account for both open and openat syscalls in our profiling.
- *SYSCALL_DEFINE1(close, unsigned int, fd))* - This is responsible for closing a file given it file descriptor. We get the filename from the file descriptor as kernel keep records of opened files.
- *SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf, size_t, count)*- This is responsible for reading a file given its file descriptor. Similarly, we get the filename from the file descriptor as kernel keep records of opened files.

These functions are modified such that every call to these functions are recorded and logged into a file present in */proc* directory. Putting up a hook in the kernel space allows us to capture every reference to a file which would not have been possible if this was done in the user-space.

Algorithm 1 Scorer

```
procedure SCORER                                        ▷ Scorer Program
    $kmap < filename, < nopen, nread, nclose >>$
    $smap < filename, < score1, score2, total >>$
    $pmap < package, score >$
    for Every record in /proc/refsinfo do
        $getcsv <>$
        $filename \leftarrow csv[0]$
        $kmap[filename][0] \leftarrow csv[1]$
        $kmap[filename][1] \leftarrow csv[2]$
        $kmap[filename][2] \leftarrow csv[3]$

    for Every k, v in kmap do
        $filename \leftarrow k$
        $smap[filename][0] \leftarrow score1(v[0], v[1], v[2])$

    for Every record in /tmp/pstmpfs/psinfo do
        $filename, te, tc \leftarrow csv[0], csv[1], csv[3]$
        $smap[filename][1] \leftarrow score2(te, tc)$
        for Each lpath in ldd filename do
            $smap[lpath][1] \leftarrow smap[lpath]+map[filename][1]$

    for Every k in smap do
        $smap[k][2] \leftarrow Wf * smap[k][0] + Wr * smap[k][1]$

    $pkglist \leftarrow shell(dpkglistofinstalledpackagesinthesystem)$
    for Every pkg in pkglist do
        $pmap[pkg] \leftarrow 0$

    for Every path in smap do
        $pkg \leftarrow shell(dpkgpath)$
        $pmap[pkg] \leftarrow pmap[pkg] + smap[path][2]$

    return pmap
```

The following is the implementation in kernel-space

- Add a function to each of read,write, open and close hooks to count on filename
- Add a /proc/refsinfo entry into kernel procfs.
- Keep count of number of open, close and read operations for each filename in a hashmap.
- Implement /proc read driver to expose hashmap data in csv format.

Algorithm 2 Kernel Filesystem Profiling and Exporting to Userspace

```
procedure FS-PROFILING                                ▷ Profiling filesystem
    $kmap < filename, < nopen, nread, nclose >>$
    for Every open to $filename$ do
        Lookup filename in $kmap$
        if filename does not exist then
            $kmap[filename] \leftarrow < 0, 0, 0 >$
        $kmap[filename][0] \leftarrow kmap[filename][0] + 1$

    for Every read to $fd$ do
        Lookup filename from fd to map to $kmap$
        $kmap[filename][1] \leftarrow kmap[filename][1] + 1$

    for Every close to $filename$ do
        Lookup filename from fd to map to $kmap$
        $kmap[filename][2] \leftarrow kmap[filename][2] + 1$

procedure READ /PROC/REFSINFO
    for Every k, v in $kmap$ do
        Convert into csv string "k,v[0],v[1],v[2]"
        output csv string
        output newline
```

4.2 User-Space Profiling

There are scenarios where the files are copied on to primary memory while booting the system up, in order efficiently track those files we are monitoring the processes which are running in the system. In order to monitor all processes we are using a shell script which checks the currently running process every 1 second and updates the a log file which is present in */tmpfs* directory.

The **Scorer** takes the two files i.e. the file generated by the kernel and the file generated by the shell script.For every process in the list we get all the libraries being used by that particular process and maps those files in the result obtained from the first file. So now we have a structure with the information like read/open/close references, CPU time and execution time. We use all this information to compute a score for each file.

4.2.1 Scorer. Scorer consists of algorithms which are used to score every Debian Package and that score is used to profile the packages. A score for a given Debian Package is the sum of scores of all the files contained in the package.

$$S_{File} = S_{FS} + S_{PS} \tag{1}$$

FS = Number I/O references to that file
PS= Running time of the process using that file

$$S_{FS} = B(N_{Open} - N_{Close}) + W_{Open}(N_{Open}) + W_{Read}(N_{Read}) \tag{2}$$

B = Open Bonus
$W_{\text{Open}}, W_{\text{Read}}, W_{\text{Close}}$ = Weight for open, read and close
$N_{\text{Open}}, N_{\text{Read}}, N_{\text{Close}}$ = Number of open, read and close to the file

For S_{FS} we are considering number of opens, reads and close to a file and on top of that we have provided a bonus of 100 points for the files which are still in open state i.e if the file is not closed it is being actively used by some process or system. Default weights for open and read operations are 1 and 5 respectively but they can be configured by the user as per the needs.

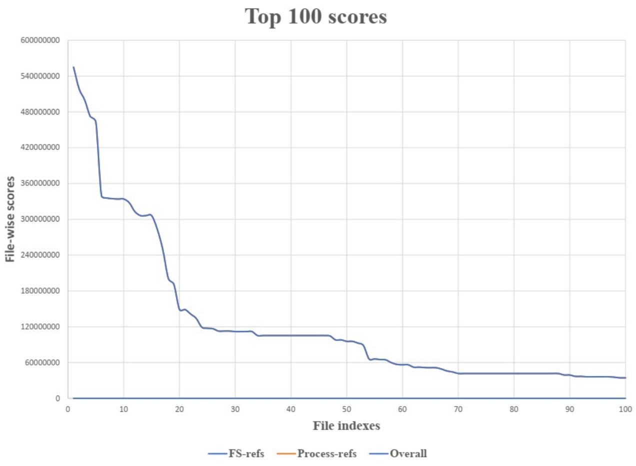

Figure 2: File-wise scores over the system generated by Scorer

$$S_{PS} = W_{Elapsed}(Time_{Elapsed}) + W_{CPU}(Time_{CPU}) \tag{3}$$

$W_{\text{Elapsed}}, W_{\text{CPU}}$ = Weight for Elapsed, CPU Time
$Time_{\text{Elapsed}}, Time_{\text{CPU}}$ = Elapsed and CPU time for a Process
For S_{PS} we are considering the running and CPU times of the process that are using the given file.The Weight given to the CPU time is higher than the weight given to the elapsed time because there can be multiple processes in the primary memory which are just in running state but are not utilizing the CPU.

5 EXPERIMENTS

The kernel-space and user=space components are executed together and data gather on the system w.r.t usage of various files and Debian packages in the system.

5.1 Setup

There two parts of scoring filesystem references are read from /proc/refsinfo and executables and daemons run on the system are accounted through a users-space profiling module written in bash shell scripting which executes ps to return elapsed time since application/daemon has started and the actual CPU time used. We have profiling of filesystem references and processes profiled in our setup: A x86-64 Intel(R) Core(TM) i7-6800K CPU running at 3.40GHz, memory of 32GB, 1TB of SSD and running Ubuntu 16.04 (64-bit). The scorer application is executed to obtain scored metrics for system usage. The system has been running since 85 days.

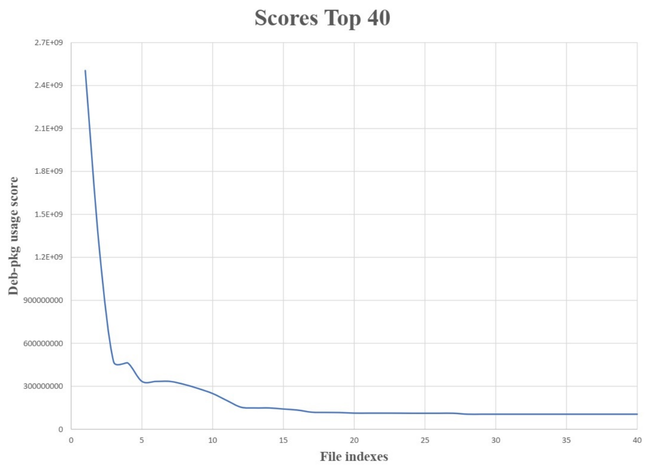

Figure 3: Debian Package-wise scores over the system generated by Scorer

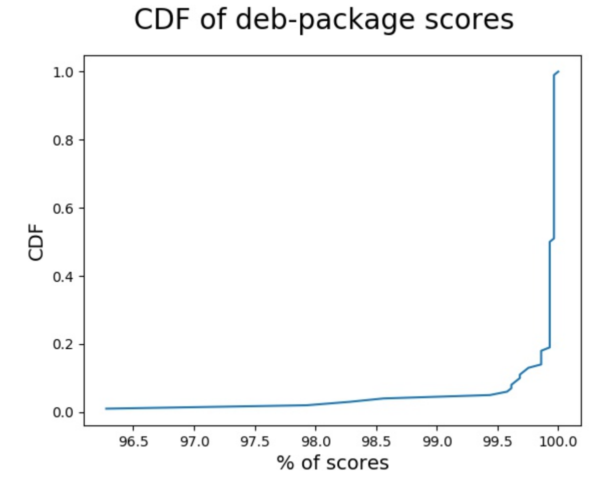

Figure 4: CDF of debian package scores

5.2 Observations

From the list executables and libraries set we see that overall libraries show much more usage and executables. We see top 3 libraries scoring (M - Million, B - Billion) around 500 M are:

- /lib/x86_64-linux-gnu/libc.so.6
- /lib/x86_64-linux-gnu/libdl.so.2
- /lib/x86_64-linux-gnu/libpthread.so.0

The top 3 executables scoring around 7 M are:

- /usr/lib/xorg/Xorg
- /usr/bin/dockerd
- /usr/bin/lxpanel

The decreasing graph trend shows scores decreasing from maximum to one-fifth of maximum value over 20 files in the system.

The scores of debian packages show that top 3 scores for packages are:

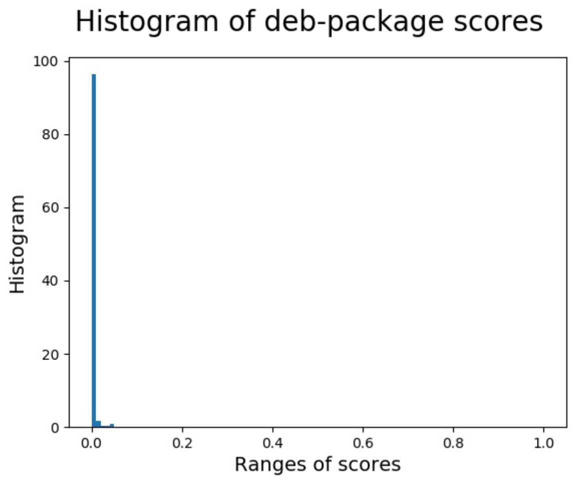

Figure 5: Histogram of scores for debian packages

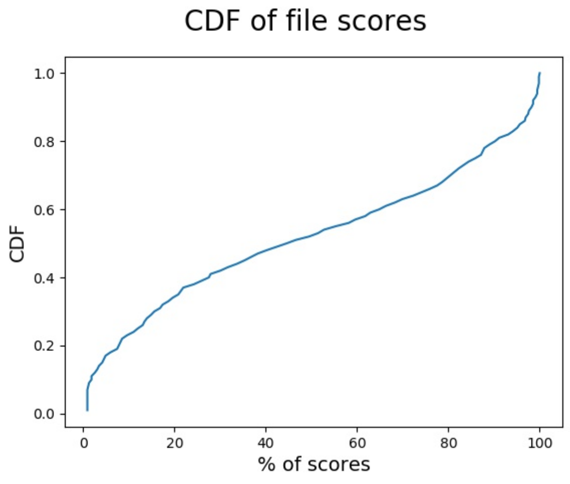

Figure 6: CDF of file-wise scores

- libc6 with score 2 B
- libglib2.0-0 with score 1 B
- libpcre3 with score 500 M

The graph trend for packages show score falls to one-hundredth of maximum within 5 packages.

6 FUTURE WORK

Based on limitations and observations following future improvements can be proposed:

- Model usage of kernel modules using ftrace account back to the current weighted model
- Access to dpkg database for every-file takes lot of I/O operations, this can be optimized
- The current kernel implements bottlenecks any open, read or close to any filesystem in by contending access to hashmap, this should be optimized to keep overheads to a minimum

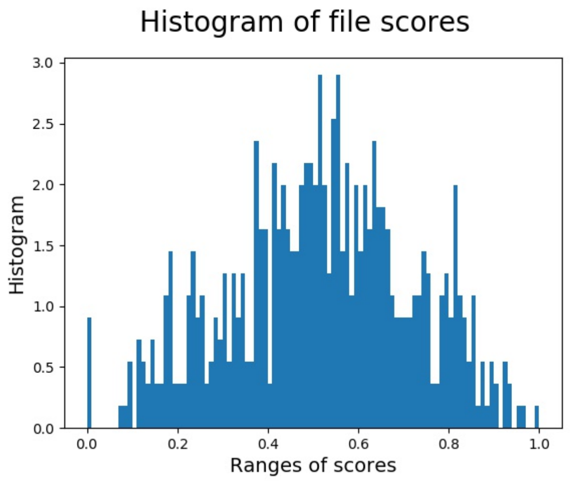

Figure 7: Histogram of file-wise scores

- Analysis to be done to better relate weights for filesystem references, elapsed time and CPU time. This is ensure more uniform scores over the system.

7 CONCLUSIONS

As seen from the data, we have sharp peaks of usage for set of files or set of debian packages which goes down rapidly to small scores. This indicates only a core set of files are extremely important for the system. We also see large number of installed packages with zero scores which show they are not in active use in the system they may have been installed inadvertently by the user or the distribution default configuration.

REFERENCES

[1] http://manpages.ubuntu.com/manpages/xenial/man1/ps.1.html
[2] https://www.enterprisestorageforum.com/storage-hardware/nand-flash-memory.html
[3] https://www.nvidia.com/en-us/self-driving-cars/drive-platform/hardware
[4] http://manpages.ubuntu.com/manpages/xenial/en/man1/dpkg.1.html
[5] https://docs.python.org/3/tutorial/datastructures.html#dictionaries
[6] https://docs.python.org/3/library/subprocess.html#module-subprocess
[7] https://wiki.ubuntu.com/KernelTeam/GitKernelBuild
[8] https://kernelnewbies.org/FAQ/Hashtables