News, tips, partners, and perspectives for the Oracle Linux operating system and upstream Linux kernel work

/ LINUX ... view more

January 11, 2021

A Zoological guide to kernel data structures

Guest Author

Kernel data structures exist in many shapes and sizes, in this blog Oracle Linux kernel engineer Alan Maguire performs a statistical analysis on data structure sizes in the Linux kernel.

Recently I was working on a BPF feature which aimed to provide a mechanism to display any kernel data structure for debugging purposes. As part of that effort, I wondered what the limits are. How big is the biggest kernel data structure? What's the typical kernel data structure size?

The basic questions we will try to answer are

- How many data structures are there, and what patterns can be observed between kernel versions?
- What are the smallest and largest data structures and why?
- What is the overall pattern of structure sizes for a given kernel release? And how does this change between releases?

A lot of the articles we read about the Linux kernel talk about size, but in the context of numbers of lines of code, files, commits and so on. These are interesting metrics, but here we're going to focus on data structures, and we're going to use two fantastic tools in our investigation:

- ***pahole*** (poke-a-hole): as per the manual page, ***pahole*** shows and manipulates data structure layout. This only hints at what it can do though. For our purposes, ***pahole*** can take a kernel image and show us the layout and size of the various structures used within it. ***pahole*** is usually available within a dwarves package; failing that, it can be built from source via https://github.com/acmel/dwarves.

- ***gnuplot***: if you haven't used it before, I hope you'll see how powerful it is. Much more than a plotting tool, we can also use it to fit functions to build models of the data we plot. I'll provide an example of doing that later. See http://www.gnuplot.info for more details.

What we're aiming for is a bird's-eye view of data structure sizes. What are the patterns? Here we're going to confine ourselves to a static analysis; in other words, we won't consider how often particular data structures are used when the kernel is running. A more dynamic-focused analysis would be interesting of course too!

And another rule - to keep things simple (and comparable across kernel releases), we will use "make allyesconfig" to generate the .config file used for kernel build. Clearly selecting more/less features will influence data structure sizes so we'll ignore that wrinkle for now for simplicity's sake.

Here's the kbuild.sh script I used to build vmlinux binaries for 31 released kernels from v4.0-v5.9. This spans a timescale for mid-2015 to October 2020.

The script should be run with no arguments from the toplevel of a kernel source git tree. Assumed is that all the tools and packages are in place to support kernel builds.

```
#!/usr/bin/bash
# Copyright (c) 2020, Oracle and/or its affiliates.
#
# The Universal Permissive License (UPL), Version 1.0
#
# Subject to the condition set forth below, permission is hereby granted to any
# person obtaining a copy of this software, associated documentation and/or data
# (collectively the "Software"), free of charge and under any and all copyright
# rights in the Software, and any and all patent rights owned or freely
# licensable by each licensor hereunder covering either (i) the unmodified
# Software as contributed to or provided by such licensor, or (ii) the Larger
# Works (as defined below), to deal in both
#
# (a) the Software, and
# (b) any piece of software and/or hardware listed in the lrgrwrks.txt file if
# one is included with the Software (each a "Larger Work" to which the Software
# is contributed by such licensors),
#
# without restriction, including without limitation the rights to copy, create
# derivative works of, display, perform, and distribute the Software and make,
# use, sell, offer for sale, import, export, have made, and have sold the
# Software and the Larger Work(s), and to sublicense the foregoing rights on
# either these or other terms.
#
# This license is subject to the following condition:
# The above copyright notice and either this complete permission notice or at
# a minimum a reference to the UPL must be included in all copies or
# substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
# SOFTWARE.
#

set -e

# Extract released tags, not rcs, sorted by version number
TAGS=$(git tag | awk '/^v[45]\.[0-9]+$/ { print $1 }' |sort -V)

# Now build vmlinux for each release, saving it to vmlinux.<release>
CFG=$(pwd)/.config
NPROCS="$(nproc)"
KBUILD="make -j${NPROCS}"

# Add space-separated problematic kernel versions here
SKIPLIST=" "

for TAG in $TAGS ; do
        # In case we need to restart, skip already build vmlinux versions.
        if [[ -f vmlinux.${TAG} ]]; then
                continue
        fi
        if [[ $SKIPLIST =~ "$TAG " ]]; then
                 echo "Skipping $TAG..."
                 continue
        fi
        git checkout $TAG
        $KBUILD allyesconfig
        echo "CONFIG_COMPILE_TEST=n" >> $CFG
        echo "CONFIG_DEBUG_KERNEL=y" >> $CFG
        echo "CONFIG_DEBUG_INFO=y" >>  $CFG
        OLDTARGET=oldconfig
        set +e
        HAVE_OLDDEFCONFIG=$(grep olddefconfig scripts/kconfig/Makefile)
        set -e
        if [[ -n "$HAVE_OLDDEFCONFIG" ]]; then
                OLDTARGET=olddefconfig
        fi
        $KBUILD $OLDTARGET
        $KBUILD
        cp vmlinux vmlinux.${TAG}
        pahole -s vmlinux.${TAG} | sort -nk 2 > struct_sizes.vmlinux.${TAG}
        rm vmlinux.${TAG}
done
```

Here's the patch that may needs to be applied for some kernels, to ensure that the buld process doesn't die due to the kernel image being too big with allyesconfig + debug info. If you see an error message "kernel image bigger than KERNEL_IMAGE_SIZE" apply the following patch to drop the assert, as we don't need to worry about the kernel being usable here:

```
diff --git a/arch/x86/kernel/vmlinux.lds.S b/arch/x86/kernel/vmlinux.lds.S
index 795f3a8..712f670 100644
--- a/arch/x86/kernel/vmlinux.lds.S
+++ b/arch/x86/kernel/vmlinux.lds.S
@@ -393,12 +393,6 @@ SECTIONS
 INIT_PER_CPU(gdt_page);
 INIT_PER_CPU(irq_stack_union);

-/*
- * Build-time check on the image size:
- */
-. = ASSERT((_end - _text <= KERNEL_IMAGE_SIZE),
-          "kernel image bigger than KERNEL_IMAGE_SIZE");
-
 #ifdef CONFIG_SMP
 . = ASSERT((irq_stack_union == 0),
            "irq_stack_union is not at start of per-cpu area");
```

Now we've built our kernels, and got our data structure sizes into a data file per kernel sorted by size; each is called struct_sizes.vmlinux.v{VERSION}.

Now let's start answering some questions about data structures in Linux!

How many data structures?

On the first question, let's get the numbers. These can be gathered by running

```
#  wc -l struct_sizes.vmlinux.v* | egrep -v total | sed 's/struct_sizes.vmlinux.v//' | sort
```

...which

- takes each struct_sizes.vmlinux.vVERSION file containing structure names and their sizes; and
- gets the number of lines in each file via wc, excluding the total (i.e. the number of data structures);
- trims the "struct_sizes.vmlinux.v" prefix; and
- prints an index followed by the number of structures, followed by the version

It's handy to have the data in this form for ***gnuplot***, as an index (rather than version numbers) makes it easier to fit functions.

Now let's plot these using ***gnuplot***.

Here's the total_sizes.plot script. It simply sets style, values, total image size (1600x800) and specifies that the output file is the name of the input file plus a ".png" suffix.

Then the first column in total_sizes.dat is plotted as the X value it's a simple index of data values) and the second is the Y value ((the number of data structures for the release).

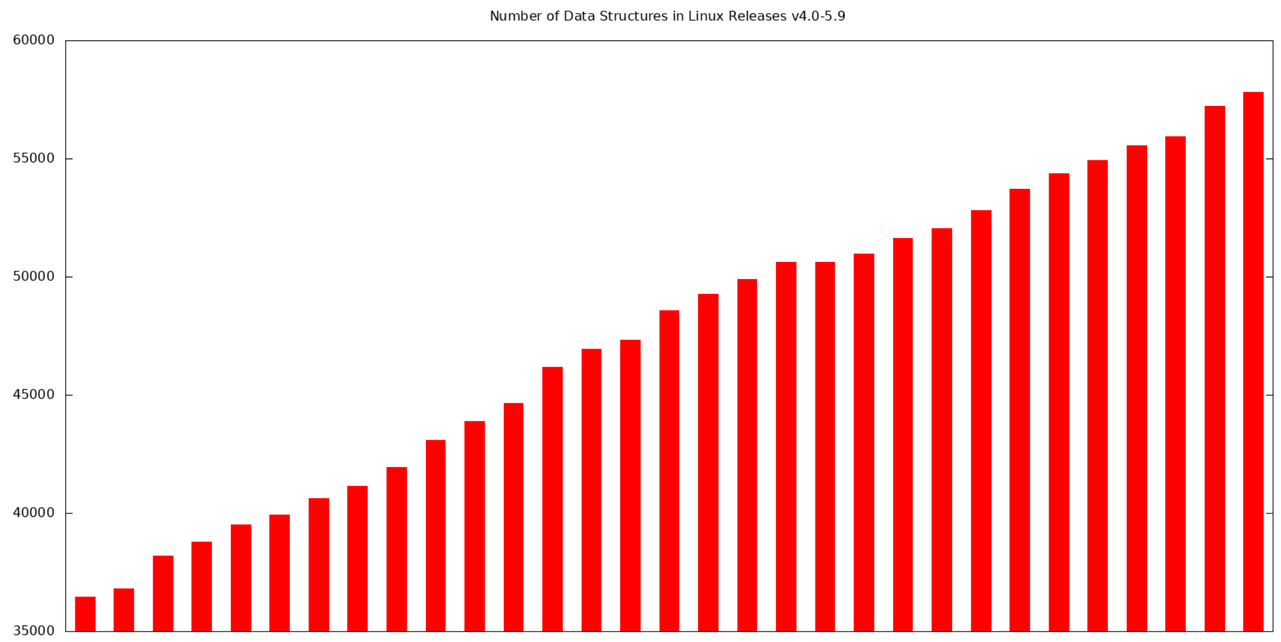
```
set style fill solid
set boxwidth 0.5
set terminal png size 1600,800
set output datafile.'.png'
unset key

set xtics nomirror
unset xtics
set title 'Number of Data Structures in Linux Releases v4.0-5.9'

plot datafile using 1:2 with boxes
```

The above should be run as

```
# gnuplot -e "datafile='total_sizes.dat'" total_sizes.plot
```

...and we will have a total_sizes.dat.png file if all goes well.

It looks like this:



But there's more we can do beyond simply visualizing the data! ***gnuplot*** will also let us fit a model to the data. So let's fit a simple linear model showing how the number of data structures increases between Linux kernel releases.

Here's how we extend our total_sizes.plot to include a linear fit.

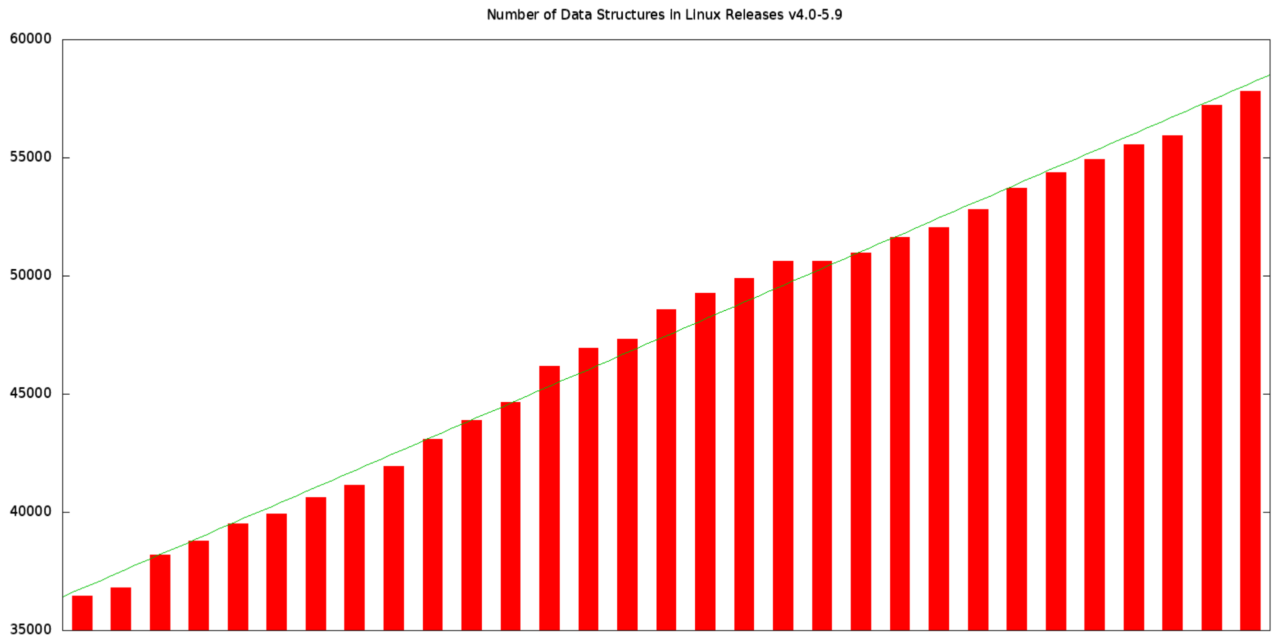
```
set style fill solid
set boxwidth 0.5
set terminal png size 1600,800
set output datafile.'.fit.png'
set multiplot
unset key

set xtics nomirror
```

```
unset xtics
set title 'Number of Data Structures in Linux Releases v4.0-5.9'

plot datafile using 1:2 with boxes

lin(x) = (m*x) + c
fit [1:32] lin(x) datafile using 1:2  via m,c
replot lin(x)
```

The last three lines (and the "set multiplot" to allow multiple plots) are the only difference. And when it is run we see ***gnuplot*** fit the values like this:

```
# gnuplot -e "datafile='total_sizes.dat'" total_sizes_fit.plot
[some fitting output omitted here]
...

Final set of parameters            Asymptotic Standard Error
=======================             ==========================

m               = 712.369           +/- 11.54        (1.62%)
c               = 36072.2           +/- 211.5        (0.5864%)
```

We can see the results here, with the fitted line superimposed on the bar chart.



So this tells us that a linear fit works with a small standard error, and that the slope (m above) is approximately 712. So this allows us to predict how many data structures will be in future releases, presuming the linear fit still applies.

So given that 5.10 would be the 32nd release (x=32), we can calculate

```
lin(x) = (m*x) + c
lin(32) = (712.369*32) + 36072.2 ~= 58868
```

...predicting approximately 58868 data structures in v5.10 (again assuming the linear fit still applies). Note that we get a slightly different answer if we just add the slope value of 712 to the 5.9 value (57827) because in the former case, we're fitting and the line would have predicted a slightly different value for the 5.9 value (58156 in fact).

So using ***pahole*** and ***gnuplot***, we've learnt that data structure growth is approximately 712 new types per release, and is roughly linear.

What are the smallest structures?

On the second question, the smallest data structures - are size 0. Let's look at some examples to see why a 0-sized data structure is useful.

For one case - "struct arch_elf_state" - the structure contains data for some architectures, but is not needed for x86_64. However it needs to be defined, so fs/binfmt_elf.c defines it as an empty struct.

Many other cases are 0-length arrays, for example "struct bpf_raw_tracepoint_args":

```
struct bpf_raw_tracepoint_args {
    __u64 args[0];
};
```

In that case, raw tracepoints have different numbers of arguments, but we'd like to access the u64 values easily. So defining a zero-length array allows a BPF program to utilize the array to get the arguments easily.

What are the largest structures?

The largest data structure across these kernel releases is the enormous "struct rcu_state", weighing in at from 4280320 (v4.0) to 6414336 bytes (v5.9). It stores RCU (Read-Copy-Update, a very clever synchronization mechanism) global state, and the reason for its size is it contains an array of "struct rcu_node".

```
# pahole -C "rcu_state" vmlinux.v5.9
struct rcu_state {
    struct rcu_node            node[521] __attribute__((__aligned__(4096))); /*     0 6402
    /* --- cacheline 100032 boundary (6402048 bytes) --- */
    ...
```

Each struct rcu_node is aligned on a 4k boundary and is padded out to 12k (since that's the closest 4k boundary), an array of 521 of them takes up 6402048 bytes. This accounts for most of the 6414336 bytes of struct rcu_state.

Another notable structure, given its centrality in process management in Linux, is "struct task_struct". 10688 bytes in v4.1, it grows to 16576 bytes by v5.9.

One structure that shrinks over this timescale - albeit slightly - is "struct sk_buff". In v4.0 it is 232 bytes, and by v5.9 it is 224 bytes. Again given that every packet processed by the TCP/IP stack utilizes a "struct sk_buff" makes this structure highly significant; in fact one of the selling points of the eXpress Data Path (XDP) technology is *not* having this per-packet metadata overhead! It's worth also mentioning that shrinkage like this, while appearing modest, requires a huge amount of work, especially as the sk_buff structure is used by a massive range of functionality, so has to carry a representation that covers a hugely diverse set of needs. David Miller has driven a valiant effort to trim its size using some very clever approaches - you can track the results here: http://vger.kernel.org/~davem/skb_size.html.

Next to get an overall view, we're going to plot struct sizes for each kernel release, to get a feel for the distribution of structures of various sizes.

Distribution of structure sizes within releases

One thing we don't have a feel for yet is the overall distribution of data structure sizes. We'd like to see the patterns in data structure sizes within releases, seeing for example if most data structures are small. Again the starting point is to visualize things with ***gnuplot***.

In this case, to aid visualization we will use a frequency plot of values; i.e. how many times are 0-sized data structures seen? 1-sized? etc and the results will be displayed in a bar chart.

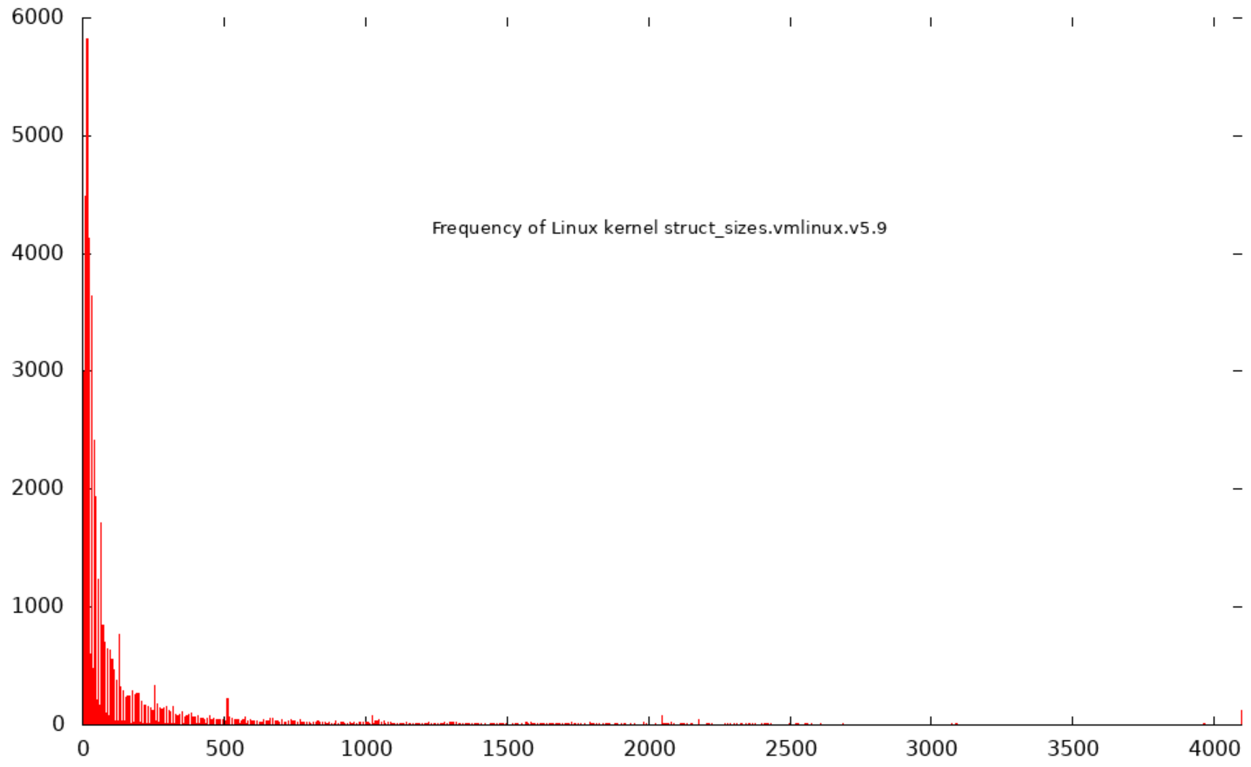
For this we use the "smooth frequency" plot where ***gnuplot*** counts the number of times an x value appears and then sums the associated y-value. Here, we want to add 1 every time a particular struct size is seen so our y value is simply 1. Here's our struct_sizes.plot file:

```
set key off
set border 3
set style fill solid

set xrange [0:4096]
set terminal png size 1000,600
set output datafile.'.png'

set title 'Frequency of Linux kernel '.datafile offset 0,-10 font ",10"

set table 'freq.tmp'
plot datafile using ($2):(1) smooth frequency with boxes
unset table
plot 'freq.tmp' smooth frequency with boxes
```

The above should be run as

```
# gnuplot -e "datafile='struct_sizes.vmlinux.v5.9'" struct_sizes.plot
```

We can see that after an initial early peak, classic exponential decay pattern is observed:



We can examine the temporary file freq.tmp we created above to see that the peak is for 16-byte structures (5825 structures), and that an interesting pattern is observed. Every 4 bytes we get a large value, interspersed by small values. For example, in the following - where the first column is the number of bytes and the second the frequency of occurence - we see:

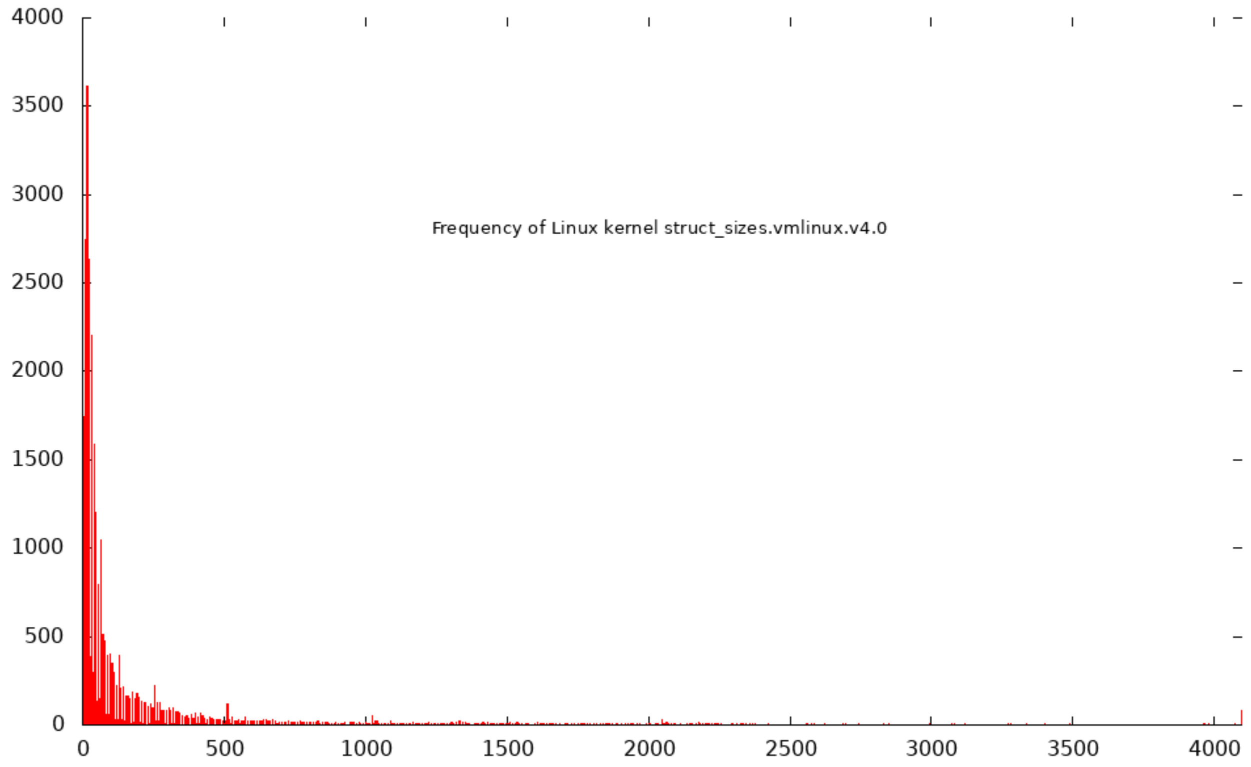
```
8  4488  8  8  i
9  130  9  9  i
10  231  10  10  i
11  73  11  11  i
12  2094  12  12  i
13  64  13  13  i
14  113  14  14  i
```

```
15  55  15  15  i
16  5825  16  16  i
17  60  17  17  i
18  120  18  18  i
19  28  19  19  i
20  1121  20  20  i
21  36  21  21  i
22  55  22  22  i
23  21  23  23  i
24  4125  24  24  i
```

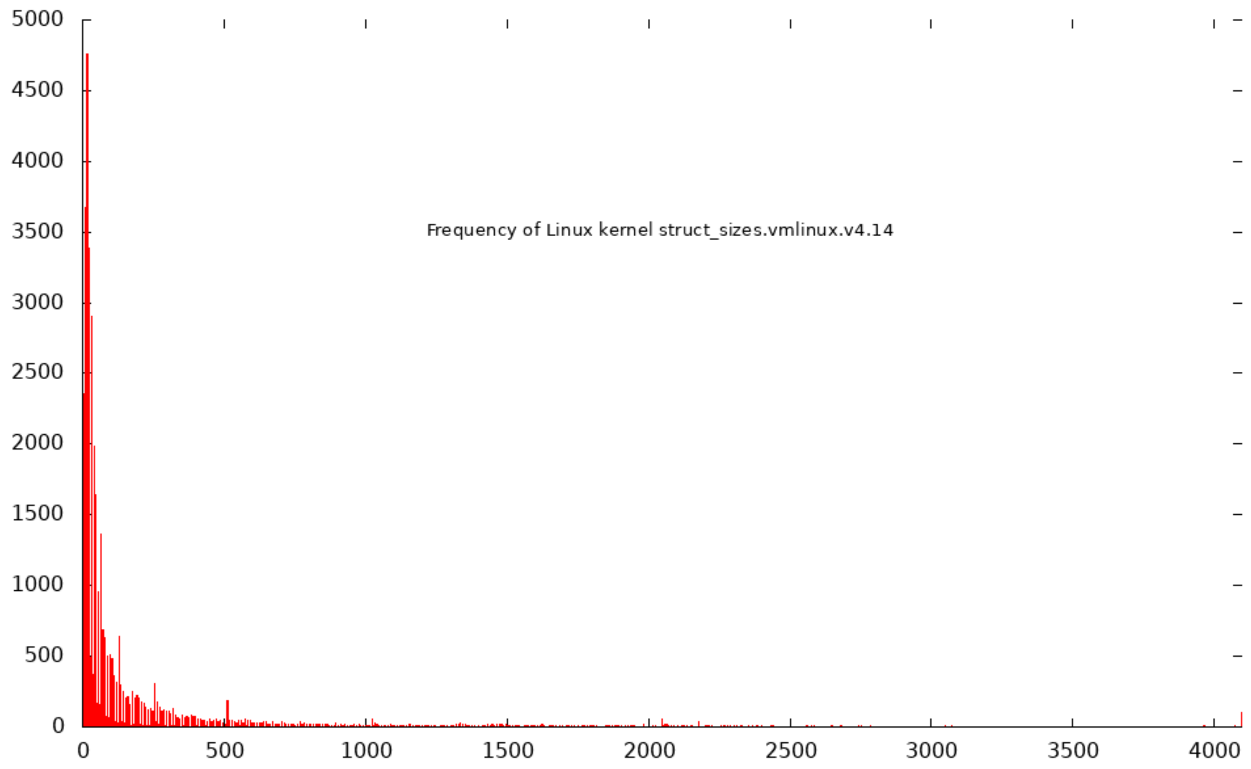
Here we see that sizes divisible by 4 bytes (8,12,16,20,24) all have larger numbers of structures (4488, 2094, 5825, 1121, 4125), while inbetween we have much smaller values. Why would that be? Well the answer is likely that the probability of a structure ending with at least a 4-byte-aligned (e.g. an integer) or an 8-byte-aligned value (e.g. a pointer) is high, given that those values are so commonly used. Also bear in mind that many kernel data structures specify alignment requirements so will pad their structures accordingly. This makes sense since the frequency of structure sizes divisible by 8 tend to be much larger than those divisible by 4. We also see a large number of data structures at 4096 bytes, alignment likely being forced to a 1 page boundary (see the 122 structures of size 4096):

```
4095  1  4095  4095  i
4096  122  4096  4096  i
4098  3  4098  4098  o
```

The same pattern is observed across all kernel releases surveyed, here's 4.0:

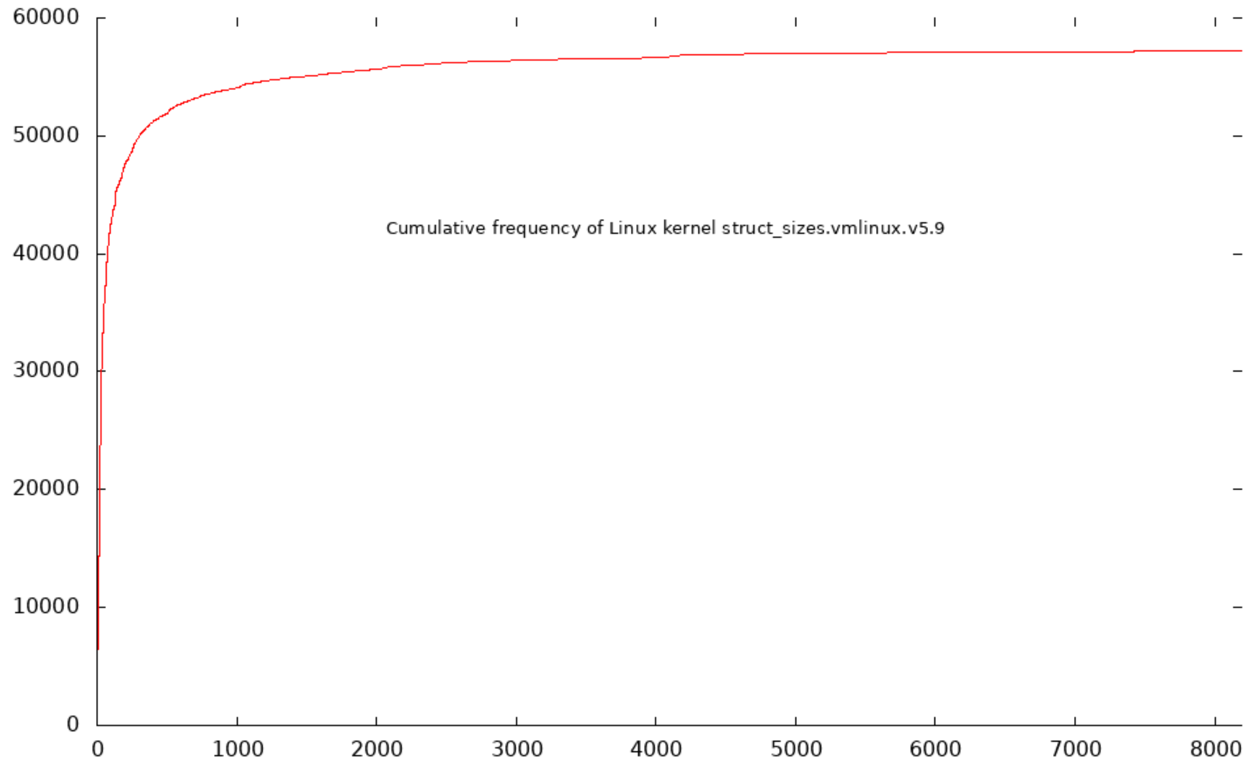

Frequency of Linux kernel struct_sizes.vmlinux.v4.0
4000
3500
3000
2500
2000
1500
1000
500
0
0
500
1000
1500
2000
2500
3000
3500
4000

...and 4.14:



To get a clearer picture of what percentage of total data structures are of a given size or smaller, we can use a cumulative frequency curve, where the height of the curve represents the number of data structures <= the current x value size. This just requires one small tweak to our plot to show a cumulative frequency:

```
set key off
set border 3
set style fill solid
set boxwidth 1 absolute

set xrange [0:8192]
set terminal png size 1000,600
set output datafile.'.cumulative.png'

set title 'Cumulative frequency of Linux kernel '.datafile offset 0,-10 font ",10"

set table 'freq.tmp'
plot datafile using ($2):(1) smooth frequency with boxes
unset table
plot 'freq.tmp' smooth cumulative
```



We can see that for the 5.9 kernel, the vast majority (>50,000 of the total 57827) are under 1000 bytes in size.

Zooming out again, what's interesting about the pattern of structure size frequency is that it seems to reflect the inherent cost of large data structures; they pay a tax in terms of memory utilization, so while we see many small data structures, and the falloff as we approach larger sizes is considerable.

This pattern is observed elsewhere, bringing us back to the zoological title of this post. If we look at the frequency of animal species grouped by their size, we see a similar pattern of exponential decay as we move from smaller to larger species sizes. For more info see https://en.wikipedia.org/wiki/Body_size_and_species_richness. If metabolic cost is a factor in determining this pattern in nature, we can observe a similar "metabolic cost" in memory utilization for larger data structures in the Linux kernel also. A related observation - that smaller species (such as insects) exist in much larger numbers than larger species in nature - would be interesting to investigate for the Linux kernel, but that would require observing data structure utilization in running systems, which is a job for another day!

Be the first to comment

Comments (0)