Tracking of Linux Kernel Regressions

Rafael J. Wysocki
rjw@sisk.pl
Faculty of Physics, University of Warsaw / SUSE Labs, Novell Inc.

November 2, 2010

1 The Purpose

Companies that sell support for enterprise distributions, like RedHat and SUSE/Novell, employ managers who assign priorities to bug reports, so that developers know which bugs are the most important from the company's standpoint and should be taken care of quickly. However, there are no managers in the mainline kernel development community, so it was necessary to find another way of prioritizing reported problems. One thing, though, was quite clear: everyone seemed to agree that regressions were the most annoying type of bugs that ruined the user experience. Moreover, there was a rule made by Linus that mainline kernel developers were not allowed to make regressions happen intentionally and should always do their best to fix all regressions introduced inadvertently before the next major release of the kernel. Thus it was natural to consider regressions as the highest priority bugs and to create a process that would allow kernel developers to learn about reported regressions from the previous major release of the kernel, so that they knew what to focus on first, as far as bug fixing was concerned.

That is why the tracking of kernel regressions was started a few years ago. The basic idea was to prepare and maintain a list of regressions reported to date and post it periodically to the Linux Kernel Mailing List (LKML) and possibly to some other kernel-related development mailing lists. Initially it was done by Adrian Bunk, but he seemed to be disappointed at the effect of the regression lists published by him on the kernel development. When he finally quit after the release of 2.6.21 [1], it had been continued by Michał Piotrowski, who started to use a "known regressions" wiki page editable by other people [2]. Unfortunately, he resigned in the middle of the 2.6.23 development cycle and the task of tracking kernel regressions was taken over by yours truly. It's been being continued since then and nowadays there is a team of three people taking part in it.

2 The Process

Following Ingo Molnar's suggestion the kernel Bugzilla is used for storing reported regressions information. For each major kernel release there is a meta-bug entry in the Bugzilla used for linking bug entries representing reported regressions from that kernel (for example, bug #16444 plays that role for the 2.6.35 kernel). Regression bug entries are linked to these meta-bugs by putting their numbers into the field containing the list of bugs the given meta-bug depends on. The contents of this field represents the current list of regressions from the kernel release associated with the meta-bug containing it. All of the meta-bug entries used for the tracking of kernel regressions are, in turn, linked to bug #15790 in an analogous way. Thus it is possible to see all of the listed kernel regressions at any time by viewing the dependency tree of bug #15790.

Regression reports filed with the kernel Bugzilla can be linked to the appropriate regression meta-bugs easily by putting the meta-bug number into the "Blocks" field of the report's bug entry. Then, the scripts generating the "list of known regressions" periodically sent to the LMKL and other mailing lists can reach into the meta-bugs to retrieve the numbers of bugs they depend on. Next, they can get all of the necessary information (e.g. the name and e-mail address of the reporter, subject, date of submission) from their respective Bugzilla entries. However, for the regressions reported by e-mail that can only work if they are given bug entries in the kernel Bugzilla. Unfortunately, that can't be done automatically in general, so those bug entries need to be

created manually by a human. At the time of this writing Maciej Rutecki takes care of that. To save his time and reduce the level of unuseful noise, Bugzilla entries are only created for the regressions reported by e-mail that are at least one week old and haven't been fixed since reported (i.e. the fixes are not present in the Linus' tree a week after the bug reports have been submitted).

The "list of known regressions" is ususally generated and published after each release of an -rc kernel. Every time that happens the message containing the list is accompanied by a set of automatically generated messages targeted at bug reporters, asking them to verify if they still see the reported issues. This allows us to check if the reporters are still around and whether or not the bugs have been fixed. The feedback we get this way is used for updating the regression Bugzilla entries. In addition to that all of the listed regression entries are periodically browsed to check if patches fixing the bugs are available or if those patches have been included into the Linus' tree. This currently is done by Florian Mickler.

If there is a fix for a given bug, but it hasn't been added to the Linus' tree yet, the bug is marked as "resolved" with resolution "patch already available". At the same time, a `Patch:` tag containing a link to the fix patch is added to its bug entry (usually in a separate comment). This causes the bug to be included into the "regressions with patches" category by the list-generating scripts. In turn, once the patch fixing the bug has been merged by Linus into the mainline kernel tree, the bug is marked as "closed" and is not included into the "list of known regressions" any more. The listed regressions that have been found to be invalid or to be duplicates of other bugs, or that can't be reproduced any more with the current mainline kernel are also marked as "closed", which causes the list-generating scripts to ignore them.

The activities described above are performed for reported regressions from the two latest major releases of the kernel. Every time a new major kernel release is made, the tracking of regressions from the second previous major release is discontinued. For example, after the recent release of the 2.6.36 kernel we have finished the tracking of regressions from 2.6.34, although we are still going to track regressions from 2.6.35 until the future release of 2.6.37. The tracking of regressions from 2.6.36 will start shortly, as the first -rc kernel in the current development cycle, 2.6.37-rc1, is already out. The regression meta-bugs corresponding to the major kernel releases that are not subject to the tracking of regressions any more are still present in the Bugzilla and the related regression bug entries are still linked to them, though, so it always is possible to review the history of kernel regression tracking since the 2.6.23 development cycle.

3 General trends

The regression tracking history records stored in the kernel Bugzilla may be used, among other things, to produce some statistics on the reporting and fixing of kernel regressions. Probably the simplest thing one can do to this end is to use the kernel Bugzilla's feature displaying bugs' dependency trees. Namely, go to bug #15790 and click on the "tree" link. Next, find the kernel of interest, go to its regression meta-bug and click on the "tree" link in there. Then, you will get the list of unresolved bugs the meta-bug depends on, which are listed regressions with no resolution whatsoever, and the number of them. The last (pending) column of Table 1 lists these numbers for the last 10 kernels subject to the tracking of regressions.

To get the number and the list of all tracked regressions from the given kernel, display its meta-bug's dependency tree and ask the Bugzilla to show resolved bugs. You should remember, however, that the list generated as a result of this also contains bugs which have been resolved as invalid or duplicates of other bugs. Since they are not interesting from the statistics point of view, it is better to subtract the number of them from the total number returned by the Bugzilla. The second (reports) column of Table 1 contains the total numbers of regressions from each of the kernels in the first column readjusted by subtracting the numbers of invalid and duplicate reports. The numbers shown in Table 1 were collected on October 28, 2010.

The first observation the data in Table 1 leads to is that the total number of listed regressions from every major kernel release is between 116 and 180. Moreover, from about 10 to about 15 percent of listed regressions remain unresolved for a long time. Also the numbers in the second (reports) column seem to be significanlty lower for the kernels later than 2.6.32, but this is a result of a change in the regression tracking process that occured after the release of 2.6.32. Namely, before that time we had tried to list all regressions reported by e-mail regardless of the time it took to fix them and now we only list regressions reported by e-mail that haven't been fixed within a week since submitted, as stated in Section 2. The reason for the appearance of the much

Ref. kernel	# reports	# pending
2.6.26	180	1
2.6.27	144	4
2.6.28	160	10
2.6.29	136	12
2.6.30	177	21
2.6.31	146	20
2.6.32	133	28
2.6.33	116	18
2.6.34	119	15
2.6.35	63	28
Total	1374	157

Table 1: Listed regressions from kernels between 2.6.26 and 2.6.35.

lower number in the "reports" column for the 2.6.35 kernel is that the tracking of regressions from that kernel release is still in progress and it is expected that many more of them will be reported in the future, which indicates a problem with the numbers in Table 1.

Quite obviously these numbers only reflect the situation at the time they are collected, so they do not provide any information about the dynamics of reporting and fixing kernel regressions. Furthermore, some reports seen by the Bugzilla as "resolved" may have been resolved after the tracking of regressions from the given kernel officially ended, so the numbers in the "pending" column have to be taken with a grain of salt. Fortunately, the Bugzilla's ability to follow and record the changes of the status of a bug may be used for learning what has happened to regression bug reports over time.

It is particularly interesting to see what the situation is at two important moments in each kernel regression tracking cycle. The first of them is when the next major kernel release is made, since at that point developers tend to focus on adding new features rather than on debugging regressions from the previous release and the newly released kernel is now considered as the "current" one. For example, the release of the 2.6.36 kernel has just ended a period in which kernel developers were supposed to focus of fixing regressions from 2.6.35. The second one is the release of the second next major kernel version, since that is the official end of the cycle. For instance, the 2.6.36 release has officially ended the tracking of regressions from 2.6.34.

Ref. kernel	# reports (1)	# pending (1)	# reports (2)	# pending (2)
2.6.30	122	36	170	45
2.6.31	89	31	145	42
2.6.32	101	36	131	45
2.6.33	74	33	114	27
2.6.34	87	31	119	21
2.6.35	61	28		

Table 2: Listed regressions at the time of the next and second next major kernel release.

Table 2 contains numbers corresponding to these events for the kernels between 2.6.30 and 2.6.35 inclusive. Specifically, the columns marked with (1) correspond to the next major kernel release and the ones marked with (2) correspond to the second next major kernel release for each kernel listed in the first column. The "reports" columns contain the total numbers of listed regressions to date and the "pending" columns contain the number of sightings that haven't been resolved yet at those times. The data in Table 2 leads to a few interesting observations.

First, it is evident that the total numbers of listed regressions at the time of the next major kernel release are significantly less than the analogous numbers at the time of the second next major kernel release. This

basically means that many regressions from kernel version 2.6.x are reported after the 2.6.(x+1) kernel has been released which indicates that the testing coverage of the "release candidate" (-rc) kernels is significanlty worse than the testing coverage of major kernel releases. Second, for the kernels between 2.6.30 and 2.6.32 inclusive there is a trend that the number of unresolved regressions at the time of the second next major release is about 20–25% greater than the analogous number at the time of the next major release. If that trend had continued, it would have suggested that kernel developers didn't care enough for regressions from 2.6.x after 2.6.(x+1) had been released. Fortunately, though, for 2.6.33 and 2.6.34 this trend seems to have been reversed and now the number of unresolved regressions at the time of the second next major release tends to be significantly smaller (in paricular, for 2.6.34 the difference is about 30%). One has to hope that this positive trend will continue for the 2.6.35 kernel and there seem to be indications that this actually may be the case, although it obviously is not certain.

4 Dynamics of Reporting and Fixing Regressions

Information extracted from the Bugzilla can be used to investigate the dynamics of reporting regressions and closing regression bug entries with daily resolution. This allows one to plot the difference between the total number of listed regressions and the number of closed regression bug entries against the number of days since the release of the kernel these regressions were from. Such plots for the kernels between 2.6.30 and 2.6.35 inclusive are shown in Figure 1.

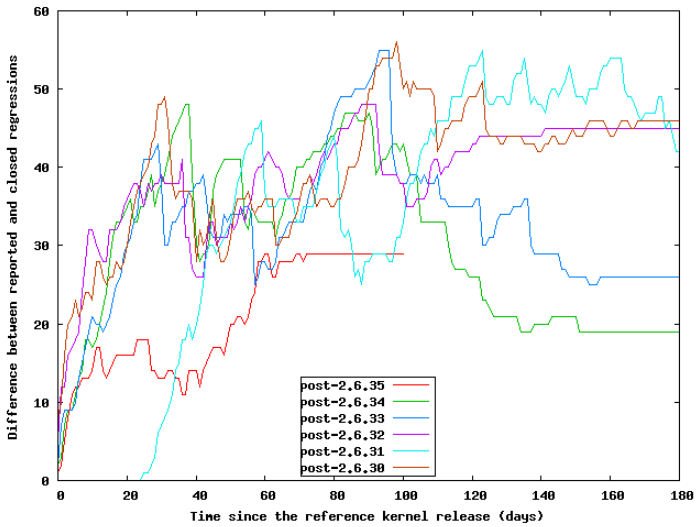

Figure 1: Difference between the number of reported regressions and the number of closed regression bug entries vs the number of days since the kernel release the regressions were from (for 2.6.30–35).

They generally reflect the trends followed by the data in Table 2, but they allow one to make a few interesting observations more. Namely, all of the functions plotted in Figure 1 tend to zero for time tending to zero, they attain a maximum or two somewhere in the middle of the cycle and finally they flatten out at the end of it. Typically, there are two maxima that roughly correspond to the ending of the merge window and the time of the next major kernel release. The 2.6.35 graph is unusual in this respect, because its first maximum it is not very pronounced and there seems to be a plateau instead of the second one. That may simply be a result of a little clash between the 2.6.37 merge window and the time frame of two important conferences (the Kernel Summit and the Linux Plumbers Conference) that probably made people spend relatively less time on testing and debugging during the current development cycle. The 2.6.33 and 2.6.34 graphs, on the other hand, seem to follow the general trend and their asymptotic behavior appears to match the asymptotic behavior of the function

$$f(t) = a\{1 - \exp(-\beta t)\}\exp(-\gamma t)\,. \tag{1}$$

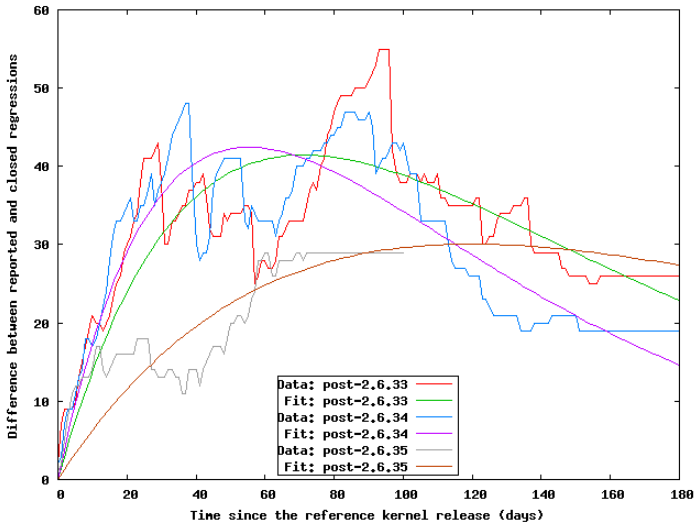

Figure 2: Difference between the number of reported regressions and the number of closed regression bug entries vs the number of days since the kernel release the regressions were from for 2.6.33–35 with fitting functions following Eq (1).

Thus yours truly tried to fit the right-hand side of Eq. (1) to the data points for 2.6.33 and 2.6.34 from Figure 1 and that led to a resonably looking result. Consequently, it appears that $f(t)$ may be regarded as a "big picture" model that reproduces the essential asymptotic features of the graphs in Figure 1, but in reality it is combined with a fluctuating function reflecting, among other things, the way in which regression reports are put into the Bugzilla (e.g. bug entries created for regressions reported by e-mail are usually added to the Bugzilla in packs which causes abrupt jumps to appear in the graphs in Figure 1). Accordingly, it seems reasonable to fit the right-hand side of Eq. (1) to the data points for 2.6.35, although they are incomplete, and try to predict the future shape of the graph for that kernel. The final outcome of it is shown in Figure 2 along with the analogous graphs for 2.6.33 and 2.6.34. It appears to indicate that the number of listed regressions from 2.6.35 will be below 30 when the 2.6.37 kernel is released, but it may be greater than the corresponding numbers for 2.6.33 and 2.6.34.

5 Regression Lifetime

Another question related to the tracking of kernel regressions that one may want to ask is how much time, on the average, it takes to fix a reported regression. It turns out that the numbers extracted from the kernel Bugzilla also may be used to give an answer to it.

For a given regression report whose bug entry has been closed one can compute the number of days it took to resolve the issue (i.e. to debug and presumably fix the problem). This number is the regression's lifetime. Next, given a set of regressions one can count the elements of it whose lifetime was not longer than a given amount of time. If this computation is carried out for times between 0 and 180 days, one can get a graph analogous to those shown in Figure 3.

It only takes a little experience in data analysis to notice that the shapes of the curves in Figure 3 follow the formula known, for example, from the description of radioactive decay [3]:

$$P(t) = a\left\{1 - \exp\left(-\frac{t}{\tau}\right)\right\}\,, \tag{2}$$

where a is related to the total number of regressions in the given set and τ is the mean regression lifetime. Then, the half-life of a given number of regressions, $t_{1/2}$, is related to it via the formula

$$t_{1/2} = \tau \ln 2\,. \tag{3}$$

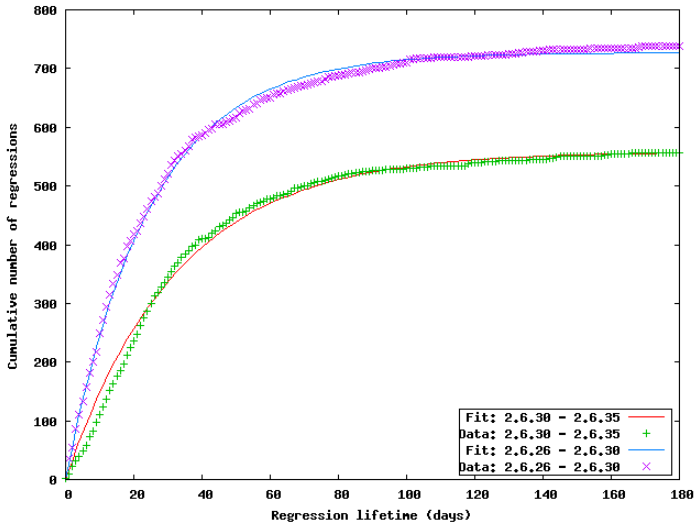

Figure 3: Cumulative number of regressions vs regression lifetime for the kernels 2.6.26–30 and 2.6.30–35.

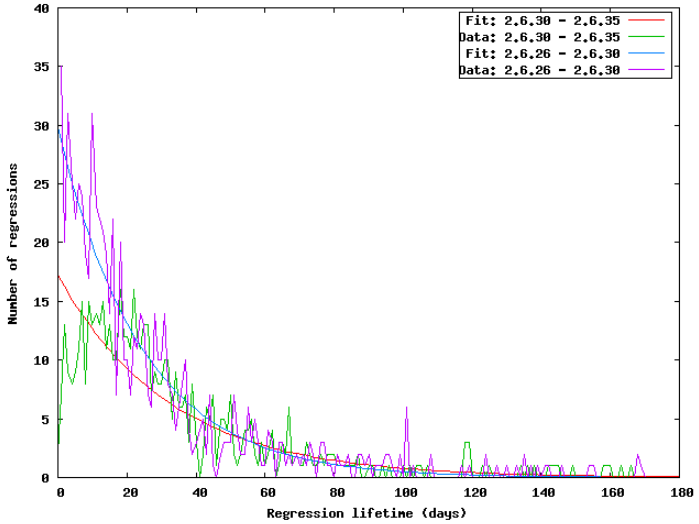

Figure 4: Number of regressions vs regression lifetime for the kernels 2.6.26–30 and 2.6.30–35.

Indeed, if the parameters a and τ on the right-hand side of Eq. (2) are determined from fitting the right-hand side of it to the data points corresponding to listed regressions from the kernels 2.6.26–30 and from the kernels 2.6.30–35, the resulting functions match the data plots almost exactly. In particular, it turns out that $\tau \approx 24.4$ for the kernels between 2.6.26 and 2.6.30 whereas $\tau \approx 32.3$ for the kernels between 2.6.30 and 2.6.35. Thus it seems that currently the average time necessary to debug and fix a kernel regression is somewhat longer that it was before the release of 2.6.32. Still, at least partially that can be explained by the change of the regression tracking proces mentioned in Section 3. Namely, we are now tracking significantly fewer regressions with lifetimes shorter than a week that would contribute to the average and lead to the reduction of τ. Hence, it can be stated that the average lifetime of a kernel regression has been approximately constant for the last two years and amounts to about 24.5 days. Accordingly, the half-life of a set of kernel regressions is approximately equal to 17 days.

Of course, for a given set of closed regression sightings one can plot the number of regressions for every regression lifetime observed in that set. Then, if the right-hand side of Eq. (2) is a good model for the "cumulative" graphs in Figure 3, it should be possible to approximate the analogous plots showing the exact

number of regressions for each regression lifetime by the exponential probability density

$$p(t) = \frac{a}{\tau} \exp\left(-\frac{t}{\tau}\right) . \tag{4}$$

Moreover, the values of a and τ determined by fitting Eq. (2) to the data plots in Figure 3 should be suitable for this purpose. That really turns out to be the case, which is illustrated by the graphs in Figure 4. This figure also shows that the graphs for the kernels 2.6.26–30 and 2.6.30–35 practically overlap each other for times greater than 20 days, which confirms the previous supposition that the difference between the two "fitted" values of τ mentioned above was a result of listing fewer regressions with lifetimes below one week in the more recent period.

6 Subsystems with Many Regression Reports

If you look at the bug lists generated by the kernel Bugzilla from the regression meta-bugs' dependency trees, you will probably notice that for certain kernel subsystems there are more listed regressions than for the others. At least one can consider some broad categories that regressions are reported against and then it turns out that some of them are, so to speak, more "popular".

Category	**2.6.32**	**2.6.33**	**2.6.34**	**2.6.35**	**Total**
DRI (Intel)	20	7	10	12	49
x86	9	13	21	6	49
Filesystems	7	12	8	8	35
DRI (other)	10	7	10	5	32
Network	12	8	6	4	30
Wireless	6	6	11	4	27
Sound	8	9	4	2	23
ACPI	7	9	3	2	21
SCSI & ATA	4	2	2	2	10
MM	2	3	4	0	9
PCI	3	4	1	1	9
Block	2	1	3	2	8
USB	3	0	0	3	6
PM	4	2	0	0	6
Video4Linux	1	3	1	0	5
Other	35	30	35	12	112

Table 3: Listed regressions from the kernels between 2.6.32 and 2.6.35 inclusive divided into categories.

Yours truly identified bug categories with particularly many reported regressions from the kernels between 2.6.32 and 2.6.35 inclusive. They are listed in the first column of Table 3. While some of them are fairly specific, like "DRI (Intel)" that practically only consists of the Intel graphics driver or "x86" that covers features related to the x86 PC platform, the others are very broad, like "Network" that includes all of the "cable" network drivers, network protocols and other things related to networking. Columns 2–4 of Table 3 contain the numbers of listed regressions from 2.6.32–35, respectively, in each category and the last column of it contains the sum of the numbers in each row. It should be noted that these numbers were very difficult to collect, because boundaries between different bug categories usually were not well defined. For instance, the "PM" category only covered problems in the PM core and power management issues whose root causes were unknown. If a driver was known to cause suspend/resume problems to occur, the bug was assigned to the category including that driver (e.g. network, wireless or DRI). Also the bugs need not be assigned correctly in the Bugzilla, so it often was necessary to look into individual bug entries and see how the bugs were fixed to figure out what category to assign them to. Needless to say, that was tedious and prone to mistakes, so the numbers in Table 3

should not be taken too seriously. In fact, they should be regarded as orders of magnitude rather than as accurate data.

It is immediately visible from Table 3 that the numbers of listed regressions for the Intel DRI driver and the x86 platform are greater than for any other category by a wide margin. Moreover, if the two DRI categories ("Intel" and "other") are combined into one, the number of listed regressions in that category will exceed the next one by approximately 30%. Similarly, if "Network" and "Wireless" are combined into one networking category, that new category will easily claim the second place. However, it is instructive to see that the "DRI (other)" and "Wireless" categories hold positions in the upper part of the table on their own.

Another observation following from Table 3 is that nearly all of the categories occupying the 8 topmost rows (i.e. with 20 or more listed regressions), with the exception of filesystems and possibly network to some extent, are closely related to hardware. Thus the bugs in these categories usually manifest themselves by causing some pieces of hardware to be handled incorrectly and the resulting symptoms are readily visible to the affected users. In consequence, they are simply much easier to notice than, for example, subtle scheduler issues leading to performance degradation that's only visible in very specific workloads or scalability problems reproducible only on 48-CPU NUMA machines. Moreover, most probably they cannot be found by means of automatic regression testing with the help of a testing suite like the Linux Test Project (LTP), especially in a virtualized environment where there's no hardware they are associated with.

The existence of so many listed regressions for x86 and DRI drivers that practically also belong to the x86 PC "ecosystem" generally indicates that the testing coverage of the kernel during active development (i.e. before a major release is made) on x86 is much greater than on any other supported architecture. At the same time it need not mean that the x86 and DRI developers are careless and they break things on a whim. The hardware they have to deal with is so complex and there are so many different versions of it that almost every more intrusive change is likely to break some rare variety, but the regressions, once reported, are generally worked on and fixed timely, which follows from the comparison between Table 3 and Table 4 containing the analgous numbers for listed regressions that were unresolved on October 28, 2010.

Category	**2.6.32**	**2.6.33**	**2.6.34**	**2.6.35**	**Total**
DRI (Intel)	1	2	2	5	10
x86	2	2	3	2	9
DRI (other)	1	3	2	3	9
Sound	5	2	0	1	8
Network	2	2	1	2	7
Wireless	1	1	1	2	5
PM	4	1	0	0	5
Filesystems	0	0	0	5	5
Video4Linux	1	3	0	0	4
SCSI + SATA	2	0	1	0	3
MM	1	0	1	0	2
Other	8	2	4	8	22

Table 4: Pending regressions from the kernels between 2.6.32 and 2.6.35 inclusive divided into categories.

The data in Table 4 is much more accurate than the data in Table 3 mostly for the simple reason that the former covers fewer bugs and it takes substantially less time to check what categories they should be assigned to. Evidently, the general trend is that the numbers in Table 4 are much smaller than the corresponding numbers in Table 3. Moreover, some categories present in the latter are not directly included in former, because neary all listed regressions in these categories have been fixed alaredy. Of course, there are exceptions, like the PM category where there are 4 regressions from 2.6.32 whose root causes haven't been identified so far, mostly due to the elusive nature of the observed symptoms. Apart from this, the ordering of the bug categories in Table 4 is slightly different, because, for example, the filesystems developers have been fixing regressions in their code more efficiently than the non-Intel DRI developers. Also the networking people have fixed relatively more

regressions than the others. Notably, the ACPI category that occupies the 8th row in Table 3 with 21 listed regressions is not present in Table 4, which seems to mean that the ACPI developers are very good at fixing regressions.

7 Conclusion

In recent years the tracking of kernel regressions has become an important part of the kernel development process for two basic reasons. First, it tells developers which bugs should be taken care of first so that they know what to focus on. Second, it gives Linus an idea about how many things should ideally be fixed before the next major release of the kernel so that the users for whom the previous kernels worked correctly aren't disappointed.

It is done in such a way that the entire history record associated with it is preserved in the kernel Bugzilla and can be extracted from there at any time. Among other things, this allows one to produce statistics similar to the ones presented in the previous sections and to get some insight into the testers' and developers' activities related to the listed regressions.

The statistics shown above generally indicate that the number of reported regressions from every major release of the kernel remains approximately at the same level, up to some fluctuations in both directions, and nearly all significant differences can be understood as resulting from modifications of the regression tracking process. They also show that the average regression lifetime has been approximately the same for the last two years and indicate that the number of regressions that remain unresolved after each cycle has been decreasing for a few recent cycles. There is hope that this trend will continue in the future.

References

[1] *Linux: Releasing With Known Regressions* (`http://kerneltrap.org/node/8110`).

[2] *Linux: Tracking Regressions* (`http://kerneltrap.org/node/8241`).

[3] *Radioactive decay* (`http://en.wikipedia.org/wiki/Radioactive_decay`).