

# Linus on Linux - The Linus Torvalds Interview With Don Marti (Linux Magazine)

In Part 1, Linus reflects on 18 years of working on Linux, the developer ecosystem and his goal for Linux on the desktop. In Part 2 of our interview, Linus talks about the process of managing kernel developer commits, selecting a revision control system and how he personally uses git.

Linus Torvalds has led the development of the Linux operating system since its inception nearly 20 years ago. In that time Torvalds has had the opportunity not only to witness the positive cultural and economic changes brought about by Linux but has also been a direct participant in making those changes a reality. And though many things have changed greatly since 1991, one thing remains constant: Linus is still at the helm.

In this interview Torvalds looks back on the operating system he created, the impact of new hardware, and the ubiquitous OS on everything from cell phones to desktops to supercomputers.

**Linux Magazine: You've been doing Linux for about 18 years now. That's not a long time by the standards of academic research, but it is a long time by the standards of the software industry. Many of the core contributors have stuck with Linux even as the industry has changed and they have changed employers. Is it good for the project to have the same people able to stick with it? Do you plan to?**

Linus Torvalds: I don't think it's good for a project if it's *only* the same people who stick with it, and I'd be very worried about Linux if we had too much of a "core long-term people" approach. But there really are a lot of developers who are fairly recent, and most importantly there's a really long tail of lots of people who dip their toes into kernel development if only to send in a really small patch. Most of them will never do anything more, but some of them will eventually be major developers. And we need that.

At the same time, I think everybody is also happier with some stability. There's actually a number of people who have been around for quite a long time. People like Ted Ts'o, who showed up very early on and is still involved and still commits code.

So it's not an either-or—we want to have both. And yes, I'll stick with it as long as I think I can do a good job and nobody better comes along (or put another way: "as long as I can subvert whoever is better to work with me" ;)

And by the way, talking about changing employers: one thing I think is *very* healthy is how kernel developers are kernel developers first, and work for some specific company second. Most of the people work for some commercial entity that obviously tends to have its own goals, but I think we've been very good at trusting the people as people, not just some "technical representative of the company" and making it clear to companies too.

The reason I bring that up is that I think that ends up being one of the strengths of open source - with neither the project nor the people being too tied to a particular company effort at any one time.

**LM: Before Linux, nobody would have believed that the same kernel would be running supercomputers and cell phones. Do you think you'll always be able to maintain one code base that works on phones and other tiny devices and on very large servers, and just let people configure it at build time?**

# Linus on Linux - The Linus Torvalds Interview

## With Don Marti

(Linux Magazine)

LT: Personally I wouldn't even say "before Linux". For the longest time "after Linux" I told people inside SGI that they should accept the fact that they'd always have to maintain some extra patches that wouldn't be acceptable to the rest of the Linux crowd just because nobody else cared about scaling quite *that* high up.

So I basically promised them that I'd merge as much infrastructure patches as possible so that their final external maintenance patch-set would be as pain free to maintain as possible. But I didn't really expect that we'd support four-thousand-CPU configurations in the base kernel, simply because I thought it would be too invasive and cause too many problems for the common case.

And the thing is, at the time I thought that, I was probably right. But as time went on, we merged more and more of the support, and cleaned up things so that the code that supports thousands of CPU's would look fine and *also* compile down to something simple and efficient even if you only had a few cores.

So now, of course, I'm really happy that we don't need external patches to cover the whole spectrum from small embedded machines to big thousand-node supercomputers, and I'm very proud of how well the kernel handles it. Some of the last pieces were literally merged fairly recently, because they needed a lot of cleanup and some abstraction changes in order to work well across the board.

And now that I've seen how well it can be done, I'd also hate to do it any other way. So yes, I believe we can continue to maintain a single source base for wildly different targets, ranging from cell phones to supercomputers.

Of course, one of the interesting issues is how even the low end has been growing up. Ten years ago SMP was uncommon on the desktop, these days we're looking at SMP systems even in very tiny embedded environments. So we *have* moved the goalposts up a bit for what we consider "small". Those cell phones tend to have way more computing power than the original PC that I started Linux on had.

**LM: You posted a very positive blog entry about your new Intel SSD. "That thing absolutely rocks." On the other hand, some of the other SSDs on the market don't, and some Linux users have pretty bad taste in hardware. Will the OS be able to get decent write performance and lifespan out of a bad SSD, or are users going to hate life if they buy the wrong one?**

LT: It depends a lot on your usage case. For example, even a bad SSD can work wonderfully well as a secondary drive that gets 99.9% just read activity, since even the bad ones tend to read really well and have low latency and good random-read performance.

Of course, the size and price tends to make that then a hard trade-off to make easily. It's not worth it for big files that you usually just stream, since rotational disks are cheaper and perfectly fine for streaming behavior. Very few among us really know the true access patterns we actually have.

And hey, even the Intel SSDs aren't perfect. If all you do is work with big files and read and write a lot of contiguous data, a regular disk will be much cheaper and bigger, and won't be any slower for those cases.

# Linus on Linux - The Linus Torvalds Interview With Don Marti (Linux Magazine)

But for me, the disk tended to always be the weakest part in the system. I can make up for some of it with just adding more memory, but while caching obviously is a huge issue and hides the disk performance in 95+% of the cases, it just makes the remaining few cases even more noticeable.

Just as an example: I'm used to doing "git grep something" in my kernel tree to find where some function is used, or something similar. It takes me all of half a second, so it's basically instant.

Except when I have just rebooted, or have just done enough other things that my tree isn't in cache any more (ok, so that's pretty rare, but it does happen ;). And then the half second was a *minute* or two with a perfectly reasonably high-end desktop SCSI drive.

So my *average* latency was great. If I get 0.5 seconds 99% of the time, and then very seldom have to wait a minute just because it reads all those small files off the disk, I should be happy, right?

Wrong. The average may be great, but that just makes the bad cases feel even *worse*. I'm used to things being instantaneous, so now that minute feels *really* really bad. And it really is mostly seeking—the median file size in the kernel is about 4kB, so it's reading all those directories and all those 25,000+ small files, and while the total size of it all may be just a few megabytes, because of seek times it takes half a minute.

Enter the Intel SSD, and the cached "git grep" still takes the same half second, but now the *bad* case takes me ten seconds (it used to be less, but those staging drivers really added a *lot* of crap. Some people would blame the Intel SSD degrading, but sadly, it's all my own fault ;)

So my average access time hardly changed, and I can still tell when I'm disk-limited, but oh boy, it makes such a *huge* difference. Now even the slow case is no longer two orders of magnitude slower. Yes, even SSD disks are slower than RAM caches, but they don't have that horrible "fall off the cliff" behavior when having to seek around for the data.

And that's why I dislike a lot of the bad SSD's. They have an even *worse* "fall of the cliff" behavior. It's for a very specific case (random small writes), and people will argue that it's even less common than the case I describe above (random small reads), and it's true. It's not that common. But it's common enough that when you hit it, it just hurts all the more.

This is why I don't like "throughput" measures. You do want throughput, but latency variation is what you notice most. You can get used to slow machines and try to make your workflow match the "Oomph" of the hardware, but you cannot ever get used to fast machines that then occasionally are really slow. Those just drive you wild.

As an aside, that's also very noticeable in CPUs. I had the biggest complaints with Intel's "netburst" (aka "P4") architecture for some rather similar reasons: it had absolutely great "best case" behavior, and then it had some cases that it just stumbled horribly at, and which I happened to care deeply about.

The P4 was like a greased bat out of hell for loads it liked, but when it started missing in its tiny L1 cache, or when you had to serialize the pipeline for locking or for system calls, it turned into something more like a CPU two or three generations old.

# Linus on Linux - The Linus Torvalds Interview

## With Don Marti

(Linux Magazine)

And again—it's actually more irritating to have something that is really good at some things and then really bad at others, than have something that is just consistently middle-of-the-road.

**LM: On a system level, “really good at some things and then really bad at others” sounds like a lot of the Linux-based products out there. Take a workstation and strip off some of the parts to make a dedicated cluster node or a NAS appliance or a PVR. Do you get a good general-purpose kernel by building something that works on the desktop, and letting people configure it to get customized builds for their own needs?**

LT: Yes. To me, Linux on the desktop has always been the most interesting goal. The primary reason for that is simply that it's always been what *I* want (I've never wanted a server OS—I started out writing Linux for my own PC, not to be some file server), but also because all the interesting problems always end up being about desktop uses.

All other uses tend to be very constrained. You have one thing (or a few things) you need to do, and you can just optimize and simplify the problem for those particular issues.

The desktop, in contrast, is all about a wide variety of uses. Huge variety in hardware, huge variety in software, and tons of crazy users doing things that no sane person would ever even *think* of doing. Except, it turns out, those crazy users may be doing odd things, but they do them for (sometimes) good reasons.

So aiming for the desktop always forces you to solve a much more generic problem than any other target would have forced us to look at.

Of course, Linux then becomes extra general-purpose because it's not *just* meant to be a desktop OS. If we *only* cared about the desktop we'd never have worked on other architectures or worried about scalability to thousands of cores. So it's not sufficient to just be a desktop, you do have to also look at other niches, but generally the desktop problems really do get you 90% of the way, and then solving scalability problems etc. is the frosting on the cake.

**LM: Speaking of different platforms, what computers do you have now? Any of them non-x86, or set up as a server, media player, or other special-purpose machine?**

LT: I don't tend to use a lot of computers, actually. I don't like having a “machine room”, and my goal is to have just one primary workstation and do everything on that. And that one has been x86-based for the last few years (basically since I decided that there was no long-term desktop survival for PowerPC - when Apple switched away it became clear that the only thing that could possibly challenge x86 on the all-important desktop was ARM).

I've got a few other machines (mainly laptops) and there's a couple of other machines for the family (one for Tove, one for the kids) but they are also all x86-based. I'm going to be very interested to see if I'll grow an ARM machine this year or the next, but it will require it to be a good netbook platform, and while the potential is there, it's never quite happened yet.

Other architectures tend to be available in form factors that I'm not that interested in (either rack-mountable and often very noisy, or some very embedded thing) so they've never found their way into my home as real computers.

# Linus on Linux - The Linus Torvalds Interview

## With Don Marti

(Linux Magazine)

Of course, I do have a couple of Tivos, and they run Linux, but I don't really think of them like that. I don't tinker with them—that's kind of against the point—and they are just devices. And there's the PS/3, but it's more interesting for games than to use as a computer (I've got faster and better-documented regular computers, thank you).

The most interesting machines I tend to have are pre-release hardware that I can't generally talk about. For example, I had a Nehalem machine before I could talk about it, and I may or may not have another machine I can't talk about right now.

**LM: Is there anything in the pipe from hardware designers that you think will have a major impact on Linux's architecture? Ben Woodard wonders about increasingly complicated memory hierarchies that go beyond just traditional caching and NUMA, as well as newer synchronization primitives such as hardware transactional memory.**

LT: I don't see that being very likely, and one reason is simply that Linux supports so many different platforms, and is very good at abstracting out the details so that 99% of all the kernel code doesn't need to care too deeply about the esoterics of hardware design.

To take the example Ben brought up: transactional memory is unlikely to cause any re-architecting, simply because we would hide it in the locking primitives. We'd likely end up seeing it as a low-cost spinlock, and we might expose it as such, using ("fastlock() / fastunlock()") for short sequences of code that could fit in a transaction.

So we'd never expose transactional memory as such— because even if it were to become common, it wouldn't be ubiquitous.

(In fact, since transactional memory is fundamentally very tied to various micro-architectural limits, if it actually does end up being a success and gets common, I would seriously hope that even hardware never expose it as such, but hide it behind a higher abstraction like a 'spinlock' instruction with transaction failure predictors etc. But that's a whole different discussion).

We'll see. Maybe the hardware people will surprise me with something that really makes a huge architectural difference, but I mostly doubt it.

**LM: It's been almost a year since you got David Woodhouse and Paul Gortmaker signed up as embedded maintainers. How has having developers responsible for embedded changed the kernel development process?**

LT: Hmm... So I can't say that I personally have seen any major changes in the embedded area, but I also have to admit that if everything is working well then I wouldn't expect to see it much. It's more the other side of the equation (the embedded developers) who you should ask.

The problem with the embedded space was (is?) always that they'd go off and do their "own thing", and not try to feed back their work or even much talk about their needs and their changes. And then when they were ready—often several years later—the kernel they based their work on simply isn't relevant to mainstream kernel developers any more. And then the cycle starts all over again.

# Linus on Linux - The Linus Torvalds Interview

## With Don Marti

(Linux Magazine)

And there isn't so much we can do at our side of the development—David and Paul were never meant to help *me*. They are about trying to help the embedded people to learn how to interact with the development community.

And if that ever happens (happened?) then I hopefully would never notice, since by then the embedded developers look just like any other developer.

But if you want my honest opinion, then quite frankly, I don't think having "embedded maintainers" really ever solves the issue, and I'm actually hopeful that the whole dynamics in the embedded world will change. I think, for example, that projects like Android might be instrumental in bringing the embedded people more into the open, simply because it makes them more used to a "big picture" development model where you don't just look at your own sandbox.

And by the way, I would like to point out that we do try to do better on "our side" of the equation too. The whole "stable" vs "development" kernels (2.4.x vs 2.5.x) was our fault, and I'll happily admit that we really made it much harder than it *should* be for people who weren't core kernel developers to get stuck on an irrelevant development branch.

So I don't want to come off as just blaming the embedded people. They really have their reasons for going off on their own, and we historically made it very hard for them to be even *remotely* relevant to kernel development.

In other words, I am hoping that it's now easier for an embedded developer to try to stay more closely up-to-date with development kernels, and that we'll never have to see the "they are stuck at 2.2.18 and can't update to a modern kernel because everything has changed around their code since" kind of situation again.

**LM: A recent development tactic the Kernel has adopted is the drivers/staging subdirectory. These are for the so-called "crap" device drivers — which mostly seem to work — and have users, but which don't pass the mainstream kernel code quality standards. Is having drivers in the kernel tree, in staging, better for getting them up to mainstream quality than waiting to bring them in until they're cleaned up?**

LT: Well, the people involved (like Greg) do seem to feel it's a success, in that it does help get drivers into better shape.

And I have to say, I've personally hit a few machines where they had devices in them that didn't have good drivers, and the staging tree had an ugly one that worked, so I was happy.

So it saves people from at least a few of the incredibly annoying out-of-tree development efforts. When a driver is out-of-tree, it's not just that you have to fetch it separately, you have to *find* it first, and then it's likely a patch against some three-month-old kernel and hasn't been updated for the trivial interface changes in the meantime, yadda-yadda-yadda.

It's been working from what I can tell. Do I wish we just had better drivers to begin with? Yes, along with a mountain of gold. It's not an optimal situation, but it's better than the alternatives.

# Linus on Linux - The Linus Torvalds Interview

## With Don Marti

(Linux Magazine)

**LM: The Linux approach to fixing security-related bugs seems to be just fix them in the mainstream kernel, and if a distributor needs to put out an advisory for their vendor kernel, they do. Are users getting a more or less secure kernel that way than if the upstream kernel participated in what you called the “security circus?”**

LT: Hey, I’m biased. I think it’s much better to be open and get the advantages of that (which very much includes “faster reaction times” both because it makes people more aware of things, and because that way the information can much more easily reach the *right* people).

And it seems to be working. The kernel is doing pretty well security-wise.

That said, anybody who really wants more security should simply try to depend on multiple layers. I think one of the biggest advantages of various virtual machine environments (be they Java or Dalvik or JavaScript or whatever) is the security layering aspects of them - they may or may not be “secure”, but they add a layer of indirection that requires more effort to overcome.

So I think we’re doing pretty well, and I obviously personally think that the Linux kernel disdain (at least from some of us ;) for that “security theater” with all the drama is a good thing and is working. But I would always suggest that regardless of how secure you think your platform is, you should always aim to have multiple layers of security there. Anybody who believes in “absolute security” at any level is just silly and stupid.

## Part 2

**Your project was one of the early adopters of distributed revision control, starting with BitKeeper in 2002, and now Git. What advice do you have for other projects, either in-house or out in the open, that are considering moving to a distributed system?**

The SCM [source control management] choice ends up being a pretty personal thing, and much more important than the technical details of the SCM you end up using is the *workflow* you use.

Now, the reason a distributed SCM is important is not the distribution itself as much as the flow of information it allows. By avoiding the whole single point of concentration, you end up with the possibility for a much more dynamic development model.

And I say “the possibility”, because you can certainly continue with all the old rules, and the old development model, and use a distributed SCM the same way you use any random central one, but you won’t see the huge advantages if you do.

So the big piece of advice for anybody looking at switching SCM’s (or if you just want to really *think* about the SCM you use now, and what implicit assumptions you have in your current workflow) is to think about the SCM as just a tool, but also realize that it’s a tool that quite heavily constrains— or liberates—just how you work.

And a lot of the “how you work” things are not necessarily seen as limitations of the SCM itself. They end up being these subtle mental rules for what works and what doesn’t work when doing development, and quite often those rules are not consciously tied together with the SCM. They are

# Linus on Linux - The Linus Torvalds Interview

## With Don Marti

(Linux Magazine)

seen as somehow independent from the SCM tool, and sometimes they will be, but quite often they are actually all about finding a process that works well with the tools you have available.

One such common area of rules are all the rules about “commit access” that a lot of projects tend to get. It’s become almost universally believed that a project either has to have a “benevolent dictator” (often, but not always, the original author as in Linux), or the alternative ends up being some kind of inner committer cabal with voting.

And then there spring up all these “governance” issues, and workflows for how voting takes place, how new committers are accepted, how patches get committed, et cetera, et cetera. And a lot of people seem to think that these issues are very fundamental and central, and really important.

And I believe that the kinds of structures we build up depend very directly on the tools we use. With a centralized source control management system (or even some de-centralized ones that started out with “governance” and commit access as a primary issue!) the whole notion of who has the right “token” to commit becomes a huge political issue, and a big deal for the project.

Then on the other hand, you have de-centralized tools like git, and they have very *different* workflow issues. I’m not saying that problems go away, but they aren’t the same problems. Governance is a much looser issue, since everybody can maintain their own trees and merging between them is pretty easy.

But partly that very flexibility has then brought on a whole set of new guidelines for how to manage it—without it getting too messy and too disorganized for anybody to make sense of it. So anybody can well do their own tree, but then to counteract that we’ve grown rules for how to keep the end results clean.

In other words, it all boils down to getting a good development model. The SCM can stand in the way for it, or it can allow it, but in neither case does the SCM stand alone.

And never forget that you can build up reasonable development models around totally horribly bad SCMs (and people have had years of experience with them, and seem to be sometimes very *comfortable* with the absolute crap that passes for SCMs).

But moving to a distributed SCM just allows for much better models. And I think that what’s important in the kernel is how we’ve been able to scale our development and have a good model that allows literally thousands of people to co-develop things. And git has been part of it, but so is the much more pedestrian “send patches around with sign-off chains” thing too.

If you’re not willing to think about how your current development model really works, you probably shouldn’t be thinking about switching SCMs.

### **There are 132 git commands as of 1.5.6.5. How many do you use?**

Oh, the “tons of commands” thing is a total red herring. For a git tutorial I did last year at the Linux Plumbers Conference, I went through a totally ridiculous example of a few people working together, and noting every time we used a new git command.

# Linus on Linux - The Linus Torvalds Interview

## With Don Marti

(Linux Magazine)

I think we ended up with something like fourteen commands being used. And even that's more than most end developers even will need. That list of fourteen commands was for the whole "multiple people working together, including the person integrating things" workflow.

The reason so many commands *exist* is that Git was designed to be scripted, and in fact almost all the original git commands were really just shell-script wrappers around a few *really* core commands that were written in C.

And that whole scriptability, and the fact that we've encouraged lots of different workflows means that there is a combination of those core scripting commands that really nobody is expected to use directly on the command line (we call them "plumbing"—they're there, they form the base, but they're generally hidden from view) and then there are tons of those helper commands that are built for some specific purpose ("porcelain": the part you actually see, and that often comes with gilded knobs to make it all fancy and pretty).

We scared away a few people by making all the commands very visible (they all got installed in your \$PATH, for example, so "git-" would give all the newcomers a list of them *all*), but the point is, that's totally irrelevant in the end. You'd be expected to start with just a couple of commands, and it's actually much more important to understand about the notion of history (and real merges) than to know more than a couple of commands.

I just checked my own command line history with a simple sed-script and some sorting and counting:

history

```
| grep git
| sed 's/[ 0-9]*(git[ ]*[a-z-]*).*/1/'
| sort
| uniq -c
| sort -n
```

and I get

```
1 git archive
1 git clean
1 git dif
1 git ls-files
1 git shortlog
1 git tag
2 git rebase
3 git checkout
3 git fetch
3 git status
4 git add
5 git reset
6 git gc
9 git commit
12 git show
16 git push-all
```

# Linus on Linux - The Linus Torvalds Interview With Don Marti (Linux Magazine)

```
21 git am
34 gitk
42 git pull
51 git grep
82 git diff
94 git log
```

There are 22 commands I've used recently, and of those one is a typo ("git dif") and three are related to me doing a release ("archive", "shortlog" and "tag"). One isn't even a real git command, it's an alias I've set up for my workflow ("push-all").

Is that exhaustive? No, I use other commands too. The above is just a random collection from the last 1000 commands in my history buffer. But I'm a git *power user*, and I don't use that many commands. And they aren't that complicated.

**One of the git commands that's not on your list is git bisect. A user who finds a bug in one kernel version but not the previous one can build a series of intermediate versions to find the one change that's responsible. Do you see many users going through the bisect process, or are bisectors mostly people who are already kernel developers?**

We're not seeing a *lot* of people use "git bisect," but when people do, it's a very high-value debug tool. And while I suspect that people who are already using git for development are more likely to then use "git bisect" to find where problems occur, it does get used a fair amount by non-kernel developers too. (of course, they may use git for other projects, and I simply wouldn't know).

But bisection definitely was meant to be simple enough for a non-developer to use—the whole point, in fact, is that the only thing you need to have is a reliable symptom, and git will do all the rest of the work. So the process is designed to not really need any knowledge at all about the development details, and anybody can use it.

Of course, the downside is that especially during the merge window, when we've merged thousands of commits, the bisection process easily ends up involving ten to fifteen kernel compiles and reboots. It's all fairly mindless, but it's still a fair amount of effort.

**Kernel developers are also giving "Reported-by" credits to people who find bugs, several of the distributions are packaging upstream kernels to make it easier for regular users to test sooner, and there's even a kerneloops.org. How are the users doing at reporting problems? Are you getting enough information, or does the world need more testing motivation or tools?**

Some users are great problem reporters, others not so much. I'd love to have more motivation for people to test and report things, but it really has gotten so much better that I'm not really willing to complain. Could it be better still, and do we need even more testing? Oh, absolutely. But I also think that we're doing reasonably well.

One thing to note is that the way we handle the reports has been a big improvement. Big kudos to Arjan (and others) for setting up the automated kerneloops site and oops scraping. That, together with timely updates by distributions make us much more aware of new issues that pop up. But what I

# Linus on Linux - The Linus Torvalds Interview

## With Don Marti

(Linux Magazine)

personally find even more important is the regression tracking efforts first by Adrian Bunk and now for the last several releases by Rafael Wysocki.

The regression tracking not only helps developers, but it's one of the things that *does* add a lot of motivation on both sides. Problem reports still fall through the cracks, but they do so less now - especially if the reporter follows up on things. And that obviously motivates testers. But the way the regression tracking is structured (with aging of old and unconfirmed reports) also makes developers much more motivated to look at them, when they aren't just random collections of dubious reports.

My point here being that "process" is good to have (when it doesn't become a straightjacket that just kills everybody's motivation entirely), and that the people who do those kinds of process things are often really important. They make the work of others (whether bug reporters or developers) more relevant or more focused.

Final note: the reason we put in "reported-by" tags in commits is not just to give credit where credit is due, and to motivate people to report problems and show that they are first-class members of the community. It's also very useful when problems crop up. If a particular solution causes problems for other people, we most definitely want to then involve the person who reported the original problem if we have to modify the fix, just to test that the original problem doesn't re-appear.

That is, in the end, the biggest advantage of a lot of the tags in fact. We started doing them due to the whole "track where things came from" issue over the bogus copyright claims by SCO, but realistically, the biggest *practical* advantage is simply that when problems occur, we want to report the problems back through the whole chain, and it's not just an issue of some simplistic "author" or "committer".

**Just looking at "git log" it looks like you're actively committing kernel changes almost every day (and I hope your family is getting some of the no-git days). Do you ever get a chance to put your feet up and think about the overall design of the project?**

Oh, I spend just about *all* my time trying to look at the "big picture" and talking to people about maintenance issues (on a small scale, that means cleaning up their patches to be more maintainable, but on a large scale it's about things like keeping the git history clean so that you can see what is going on, exactly so that I, and others, can get the big picture more easily).

Yes, I commit most every day, but my commit statistics are very skewed: 95+% of what I do is merges, and just a couple of percent is actual "code" commits, and quite frankly, even those pitiful few ones tend to be about pretty trivial stuff like reverting somebody else's code that caused problems.

Oh, and my commit statistics are pretty lumpy. I may average 18+ commits a day, but that doesn't mean "just over one commit every waking hour". No, I go on "merge binges", when I apply big series of patches (especially from people like Andrew) or when I do five or six "git pull" requests within minutes of each other. Other days I *just* read email.

**How do you get these statistics?**

The way I did those statistics was to compare:

```
git rev-list --committer="Linus Torvalds" --since=6.months.ago HEAD | wc
```

# Linus on Linux - The Linus Torvalds Interview With Don Marti (Linux Magazine)

which shows how many commits I've done in the last six months. Just now, that number happened to be 3354 - so I average something like 18+ commits every day, day in and day out.

Ok, that's a fairly big number. BUT..

But then you can look closer, and look at how many of those commits was for something that I count as an *author* (change the "--committer" to "--author" in the above git command), and that number falls to less than a third: 1042 - two thirds of my commits are commits of other peoples code.

"But that's still almost six commits per day that you author!"

But.. Of the just over a thousand commits that are really *mine*, 90%+ are merges. Add a "--no-merges" to that git command line, and you end up with just 88 commits that I authored in the last six months. And most of those were really very trivial.

And how many commits did we have in total during those six months?

Right now that command line (without the "--committer" or the "--author" or "--no-merges" limiters) is 27,143.

So out of the almost thirty thousand commits, I was directly *involved* with a bit more than 10% (that is, almost 90% came in through git merges - I merged them, but I never needed to look at the individual commits), and I personally wrote a vanishingly small fraction.

In other words, I do basically no code of my own any more, and effectively *all* of what I do is merge patches that get emailed to me, or do git-to-git merges.

Of course, that is exactly how it *should* be, so I'd argue that those numbers are all good, but I'm trying to explain that all my time gets spent on other things than worrying about the code itself - I worry about development model details, about regressions, and about keeping the code/flow maintainable so that I can continue to work well.