

# There can be only one - The unified x86 architecture

Glauber Costa  
Red Hat Inc.  
Brazil  
glommer@redhat.com

## 1 Abstract

Once upon a time during the 2007 Kernel Summit, it was decided, not without an extensive discussion, that the kernel architecture trees for i386 and x86\_64 would be better as one.

One year later, the work has started and moved fast. But it is still far from being finished, with some core areas still needing to be addressed. What we see now is a hybrid world, in which two close brothers march together until traces of their differences completely vanish.

This paper looks into the past in a retrospective, attempting to answer main questions about this work. Was it worthy? Are we now in better shape than before? And more importantly, what are the main lessons that were learned, and can be used to increase the future quality of the Linux Kernel as a whole.

Last, but not least, we'll discuss what there is still to be done, which areas still need attention, and how you can help .

## 2 The history

It was the year of 2007, and as usual, a lot of work was happening on the Linux Kernel, done by happy programmers from all around the globe. Some of it touched the core architecture code, especially the x86 variants.

At that time, the Linux Kernel had support for both i386, the 32-bit variant of x86, and x86\_64, the 64-bit one. Both processors being very similar and equally braindamaged, one could expect that the kernel code itself didn't deviate so much.

In fact, as we shall see in section ??, a large part of the code was exactly the same. The build system then used Makefile rules to build files contained in `arch/i386` for x86\_64, and vice-versa.

Part of the code differed in implementation, but not in essence. For example, since i386 is a bit older<sup>1</sup> than x86\_64, it accumulated a lot of quirks and errata throughout the time. Those corner cases are just present in older machines, so were dropped in x86\_64 for the sake of code simplicity.

Also, some features were present in just one of the architectures. Some, like the IOMMU support (see section ??) were only meaningful for one of them anyway, since it was hardware support that were exclusive to that particular variant.

But of more importance, is the case of new features. Those features were sometimes just a better way of doing things, in the usual Linux way, or support for the hot new hype, but needed an architecture port anyway. As time passed, new code began to find its way through the kernel, making the architecture ports deviate more and more from each other.

Besides the obvious code duplication, people writing that code had a hard time realizing whether or not code was shared just by looking at it. Changing a file under the `arch/i386` directory could very well break x86\_64 due to a shared files issue. Because this sharing was so non-obvious, few people would test it, giving birth to bugs that could easily have been avoided.

<sup>1</sup>actually, it's more than a bit older. It's precisely 32 bits older

## 2.1 The `paravirt_ops` framework

Back in 2007, a major work was ongoing in the virtualization arena: the `paravirt_ops` framework. In a nutshell, it is a piece of infrastructure that would allow a Linux Kernel to run unmodified on top of a variety of existing hypervisors, using a vfs-like structure with pointers and guest-specific data. The work started in the `i386` field [?], by Rusty Russell and others.

A year later, Steven Rostedt and myself started porting it to `x86_64` [?]. One of the first things we noticed, is that it would lead to an incredible amount of code duplication. Some parts of the code would be just equal, and others could easily be made equal, given enough effort in rethinking some of the interfaces.

One example of this was the `pv-function` calling convention. Because `i386` registers are 32-bit, 64-bit arguments <sup>2</sup> have to be passed in two registers (usually `eax` and `edx`). For `x86_64`, it is enough to use one. One can easily think of a common implementation that abstracts the calling convention based on the type of data used.

Other than the Makefiles hack, however, we lacked good infrastructure to reuse pieces of the already existing code in a non-obscure way. We saw ourselves in a lot of situations in which the non-obviousness of the current shared infrastructure played tricks on us. We introduced bugs that would have been obvious if we knew where to look, but the Makefile schema was hiding it from our mental cache lines.

Then Steven Rostedt started the first attempt to unify the `x86` architecture. It was a little bit simpler than what we actually ended up doing [?]. Rostedt's proposal was to create a shared `arch/x86` directory that would hold files common to both architectures. That incipient proposal still had the notion of two different architectures, and would just account for the common code between them. The `arch/i386` and `arch/x86_64` would still exist, but the question of whether or not a piece of code was shared would have a quick answer.

## 2.2 High Resolution Timers

Meanwhile, Thomas Gleixner was finishing the work on High Resolution Timers (`hrtimers`)[?]. Again, it was a large and pretty central piece of software, that needed to be ported to any new architecture willing to use it. The initial port was done to the `i386` architecture. Besides the need to write a different port to `x86_64`, Thomas also ran into trouble with bugs that arose from the fact that shared files were there under the hood.

To quote Thomas himself:

I was in a boring meeting and hacked myself through the proxy server, so I got connection to the outside world. I connected to my devel box and applied the `hrtimer` queue to the latest Linus rcX, which fixed a reject in `x86_64` and compiled for both 32 and 64 bit. I boot tested 64 bit, and released the new `hrtimer` version. Half an hour later, a guy from the `powertop` project told me that I broke 32bit. It compiled but exploded on boot. The reason was a change in the `x86_64` code which was non obviously included into 32 bit as well :(

So I logged into a 32bit textbox and figured out what was exploding. The solution was simple but I was mighty grumpy. I had another 4 hours boring meeting and a 3 hour train ride home, where I started a first cut of the merger script (brute force version).

When I was home it compiled at least in a minimal config I showed the patches to Ingo and he helped me to get them working with all configs and on lots of test machines, and I sent out the RFC patch to LKML. So, the main impulse was just grumpiness

Even though it was the proposal that was eventually merged, the RFC Thomas Grumpy Gleixner talks about came with much dispute. It was well received by people working on major subsystems that touched a lot of code, such as virtualization, and by Linus himself. But it found fierce opposition from Andi Kleen, who was the maintainer at the time [?].

<sup>2</sup>functions that deal with MSRs, for example, all take a 64-bit argument

The work was done in a way to keep the in-transition tree fully bisectable. At that time, the concern was only having the code to dance inside the kernel tree to a new location. The files that were already the same were moved to their new home, and the ones that were obviously<sup>3</sup> equal were merged.

In the `include/asm-x86` directory, which replaced the old header-holder ones, header files were adjusted so they could keep their names. A common `foo.h` was created and itself, included `foo_32.h` and `foo_64.h`.

All the other files, were just copied to the new `arch/x86`, and got their names suffixed with either `_32` or `_64`. So for example, `arch/i386/smp.c` became `arch/x86/smp_32.c` while the `x86_64` version became `arch/x86/smp_64.c`. The bisectability state of the tree assured that even if something went wrong by mistake, one would be able to identify where and why.

From that point on, the work would have to be done by human beings (with a somewhat broad definition of “human beings”). One of the concerns that was raised at the time the merger was proposed was the availability of human resources to do it.

### 3 Merging x86 files

Since the initial integration work, some rules of thumb have been drawn up to help guide the integration work. Some of them are just the old rules all over again, and can safely be used elsewhere. Integration is really only a big stress test for best practices.

As usual, they are not exactly rules carved in stone, and work more like suggestion of best practices.

#### 3.1 Moving code around

The first suggestion regards changes that are purely a code move, with no behavioral changes. For that, people integrating code should verify that the patches indeed do what they claim to do, namely, change how the code looks, without changing the underlying object code.

The `size` utility can be used for that matter, like in the following example:

```
[glauber@poweredge linux-2.6-x86]$ size vmlinux.{old,new}
  text    data    bss     dec     hex filename
4318765  569156  618348  5506269  5404dd vmlinux.old
4318765  569156  618348  5506269  5404dd vmlinux.new
```

A simple change in a structure size, that may come from surrounding some of its fields with an `ifdef`, removing an `ifdef`, etc, is enough to cause a big impact in the output. The following example comes from `processor.h` integration. A slight change in `struct x86_info` that just moves fields inside the struct is responsible for

```
  text    data    bss     dec     hex filename
4318765  569156  618348  5506269  5404dd vmlinux.old
4318797  569156  618348  5506301  5404fd vmlinux.new2
^^^^^^^                ^^^^^^^  ^^^^^^^
```

Of course, it is impossible to have all the code merged without no binary diff at all. That is the low hanging fruit, and after they’re done, we still have a long road ahead.

Merging the code does not mean putting all the code in the same file. So the first attempt of many, to do

```
#ifdef CONFIG_X86_64
  <64-bit code>

```

<sup>3</sup>“obviously” here, mean obvious to a machine, in a diff output. For humans, some obvious things obviously are much less obvious than they look like.

```

#else
    <32-bit code>
#endif

```

is wrong, and actually not unifying anything. As an example, the page handling code does exactly the contrary: it unifies everything into a `page.h` file, but for the sake of clarity, leaves the crucial differences, that exist and are inevitable, into `page_32.h` and `page_64.h` [?].

Of course, there are valid cases of using architecture-specific conditionals like this. One of them is for works in progress, so a first pass can integrate large amounts of very similar code leaving small parts that deviate to a later integration step. An example of this is given in section ??.

Another case happens in header files, when it is used to hide the difference from the `.c` files. In `processor.h`, we read:

```

static inline int hlt_works(int cpu)
{
#ifdef CONFIG_X86_32
    return cpu_data(cpu).hlt_works_ok;
#else
    return 1;
#endif
}

```

There are also cases in which a piece of code is exclusive to one of the architectures. Not necessarily due to an architectural difference between `i386` and `x86_64`, but rather the presence/absence of a certain feature. In such cases, it's better to conditionalize the code on the feature, instead of `CONFIG_X86_32` or its 64-bit counterpart. This is done to avoid seeing the architectures as two different entities.

An example comes from the `smpboot.c` file. One of the `i386` sub-arches does not use APIC IPI messages to wake up new cpus; instead, an NMI is used. While it will happen only on (a subset of) `i386` machines for the foreseeable future, we test for the feature itself. The code looks like this:

```

#ifdef WAKE_SECONDARY_VIA_NMI

static int __devinit
wakeup_secondary_cpu(int logical_apicid, unsigned long start_eip)
{
    ...
}
#endif /* WAKE_SECONDARY_VIA_NMI */

#ifdef WAKE_SECONDARY_VIA_INIT
static int __devinit
wakeup_secondary_cpu(int phys_apicid, unsigned long start_eip)
{
    ...
}
#endif

```

In general, the way to go is to rethink the infrastructure to have a common version that accounts for both architectures wherever needed and necessary.

## 3.2 Bisectability

There is a big concern about bisectability to avoid regressions. Since the integration work is mainly a rework, every bug is considered a regression, with all the severity it implies.

In order to achieve it, the most basic requirement for any patch series that targets at integration is to compile in all different configurations. One particularly tricky thing is the sub-architecture support i386 has <sup>4</sup>, since it is highly dependent on configuration knobs, and few people would spontaneously test.

A useful tool in this step is `make randconfig`. But it's also useful to have configurations handy that test the presence or absence of any knob that might be affected by your series. For example, while doing smp integration, it's useful to have configs that have `CONFIG_SMP` both on and off, together with the random ones.

Of course, bisectability implies granularity. Being able to compile all revisions between a good and bad revision does not really help, unless you're able to look at the change and easily figure out why it breaks.

With newly written code, it is not always possible, and sometimes does not help at all. But with the integration work it is always possible, and should be done. The golden tip is to have each patch do a single thing, as small as it can get (but not smaller). Any thing that can possibly cause a change in behaviour of the code should be separate. An example comes from the delay functions integration. "Make both architectures use the same delay loop functions" is too coarse of a change. Rather, the preferred way is to do change one thing at a time, until both files are equal. The shortlog for the aforementioned integration says:

```
x86: don't use size specifiers.
x86: provide delay loop for x86_64.
x86: use rdtsc11 in read_current_timer for i386.
x86: explicitly use edx in const delay function.
x86: integrate delay functions.
```

As one can see, it first changes the size specifiers of the inline assembly constructs. Even though this is a simple change, it could trigger problems with instructions being wrongly translated, quantities not fitting registers, etc. Then it implements a function for `x86_64` that is already present for `i386` (usually a mere copy), and so on, until we have the code exactly equal. The merge is then done, and bisectable.

## 4 IOMMU: An example of free lunch

For a very long time, `x86_64` had support for hardware pieces called IOMMU. This is a very common setup on modern hardware, especially to overcome the limitation of the 32-bit address space of PCI buses. For the `i386`, this setup is not exactly common, and the Linux Kernel lacked support for it. The lack of hardware may lead one to think that this feature is not really needed. But it fails to account for the abuse-the-abstraction motto of Computer Science, as well of future developments.

One of the use cases for IOMMU is virtualization. It should allow a device to be seen directly from the guest address space, while still maintaining protection against the device writing in arbitrary memory areas.

Newer Intel hardware has support for VT-d hardware, which is a kind of IOMMU. Not to mention the possibility of building a paravirtualization solution on top of the IOMMU abstraction, usually generically referred to as `pv-iommu`.

The later work by Amit Shah, aiming to provide infrastructure for KVM pci-passthrough, made use of `dma_ops`. It is (was) a `x86_64`-only infrastructure, that allowed multiple kinds of IOMMU units to be registered in the Linux Kernel and to be chosen accordingly. This work is by no means `x86_64` specific, since newer `i386` ship with the VT-d capability, and even with older hardware the paravirtual solution

<sup>4</sup>which is being reworked right now to avoid this

could be used with it. The PCI-passthrough work, however, was started only in the x86\_64 front, since the infrastructure was lacking in i386.

With the merge efforts ongoing, the natural step was to just inherit the code base from x86\_64 [?]. Although a newer piece of infrastructure could be written (not likely to be accepted, in this context), this approach has the effect that all the usual QA & testing work usually done in a new infrastructure is severely diminished: x86\_64 was trusted, and we only had to account for small differences.

In fact, only one bug (which in this case, was unfortunately a regression) was found after the first release that had it, before i386 could enjoy the infrastructure needed for IOMMU.

## 5 What is still to be done?

As of today, some important subsystems merge are underway. The apic (and io\_apic) pieces are started, but not yet finished, and the time parts are passing through difficult times, due to corner cases that keep showing up. There are still some regressions that are not yet solved.

Also, not all merges are complete. Although we actively try to avoid it, not all unification stuff has the possibility of completely getting rid of all differences between architectures. In those cases, there are still places where ifdefs are used to differentiate between the variants.

A big representative of it, are subsystems that have dependencies on each other. As an example, while i386 had all of its per-cpu variables in a percpu area, x86\_64 had the concept of a pda. Before the percpu variables themselves were integrated, much of the code that touched it, looked like this:

```
#ifdef CONFIG_X86_64
    add_pda(apic_timer_irqs, 1);
#else
    per_cpu(irq_stat, cpu).apic_timer_irqs++;
#endif
```

The belief in this case is that a temporary macro or inline function would just hide the problem instead of solving it. Furthermore, this is likely a temporary condition. This makes a valid case for the conditionals. As soon as the percpu merge is finished, that code can be easily identified and changed to reflect that.

A simple script like the one below,

```
egrep -Erc "CONFIG_X86_(32|64)" arch/x86/kernel/* | grep -vE ":0$" \
| sort -t : -k2 -nr
```

can show the offending conditionals, so that the programmer interested in pushing this work forward has something to begin with.

According to the previous script, the top offenders are:

There are also places in which there are still two files holding the 32 and 64 version of the code in the tree. The script

```
find arch/x86/ -iregex ".*_(64|32)\.c"
find include/asm-x86/ -iregex ".*_(64|32)\.h"
```

can show them.

As of today, it sums up to 97 files in the architecture code, and 49 include files. The mere existence of the files does not say much, in some cases. An example is NUMA code: There are a lot of files that came from the x86\_64 port that do not have a \_32 counterpart. This reflects the fact that the NUMA implementations are largely different between them. One attempting to integrate it should spend a good amount of time thinking about how to bring them to a common ground.

file	occurrences
kernel/apic.c	31
kernel/io_apic.c	26
kernel/setup.c	20
kernel/cpu/common.c	20
kernel/ptrace.c	18
kernel/smpboot.c	16
kernel/cpu/amd.c	10
kernel/kprobes.c	9
kernel/i387.c	9
kernel/acpi/boot.c	9

Table 1: Top offenders for to-be-done unification. Number of occurrences of conditional code

## 6 Conclusion

After around a year after the x86 integration has started, the job is going smoothly and well, but some large pieces still need to be addressed, and contributors are welcome. The easy parts lie far behind in the past, and some of the remaining pieces would be enough to give Jack Bauer himself the creeps. But not the kernel hackers.

The x86 integration raise awareness from people involved in kernel development about the possibility of indirect breakages due to implicit shared code, which is something we should avoid in the future. The complaints about subtle havoc due to the under the hood code sharing seem to be over.

The development of new features has been made much faster. For the cases in which one of the architectures already had the necessary infrastructure, it was made “superfast”, while for the other cases, the cost of porting the code to a new architecture disappears.

Regressions were found during the still ongoing process, but the way the process is happening could assure they do not last long (well, most of them).

## 7 Acknowledgements

I would like to thank Ingo Molnar, Thomas Gleixner, and H. Peter Anvin, as the maintainers of the shared x86 architecture for keeping the bar high, ensuring a good quality work, and allowing me to prove that I’m able to jump over it. Andi Kleen, for raising so passionately the possible drawbacks of this work, Steven Rostedt and Chris Wright for their friendship and support, Chris Lalancete for reviewing this paper. Red Hat as my employer, and all of my countless friends, for being, you know, my friends.

My thanks also go to my wife for understanding how important x86 integration was, even though she was lying when she said that. Last but definitely not least, I’d like to thank the people of the city of Hamburg, for inventing the Hamburguer.

## References

- [1] Thomas Gleixner, Douglas Niehaus: *Hrtimers and Beyond: Transforming the Linux Time Subsystems* Ottawa Linux Symposium, 2006, Volume 1, pages 333-343
- [2] <http://lwn.net/Articles/194339/>

- [3] <http://lkml.org/lkml/2007/3/14/10>
- [4] <http://lkml.org/lkml/2008/4/8/208>
- [5] <http://lwn.net/Articles/216768/>
- [6] <http://lwn.net/Articles/243704/>
- [7] <http://lkml.org/lkml/2007/12/14/41>
- [8] Documentation/x86/pat.txt