Ugly Ducklings

Resurrecting unmaintained code

Dave Jones
SuSE Labs
davej@suse.de

Abstract

Throughout the development of the 2.5 kernel, a number of drivers and pieces of infrastructure that had been left to stagnate finally got a long overdue cleanup. In some cases, code that hadn't been touched for several years got overhauled. Each time another area got the cleanup treatment, patterns started to emerge.

This paper attempts to document some of these patterns so that hopefully by keeping them in mind, future driver authors don't fall into the pitfalls that some of these have fixed up such as over-abstracting, and massive duplication. By way of examples, it covers several areas that got cleaned up in the 2.5 series, but focuses on the bulk of the work the paper author did on the agpgart driver.

1 Introduction

It has been a long-standing philosophy that bending an existing driver to work on a new piece of hardware is much favoured over a new implementation which ends up with 99% the same code as the old. The typical life-cycle of a driver is as follows.

- Driver is written for hardware vendors A's new widget
- Vendor B makes a compatible widget.
- Id's for Vendor B's product get added to the driver
- (Repeat for several other vendors/other register compatible widgets)
- Slightly different widgets start appearing, which are still mostly compatible. Driver starts to take on new form where it needs to special case certain widgets in different code paths.
- Repeat for several more new widgets.
- Driver is now 100K+ of spaghetti.
- Original driver maintainer moves on to new project, leaving driver in current state.
- New widget Id's get added.
- Most people too scared to change too much of the code in fear of subtly breaking support for other widgets.

2 Cleanups overview

Throughout the development of the 2.5 kernel, a number of drivers and pieces of infrastructure that had been left to stagnate finally got a long overdue cleanup. Each time another area got the cleanup treatment, patterns started to emerge.

2.1 The splitting up of multiple instances

The "support all hardware all in the same driver `.c` file" approach is flawed. If you want to change how vendor B's products work, you shouldn't be touching any code for other vendors devices. With hundreds or thousands of lines of irrelevant code, it's also a pain to navigate your way around the source. By splitting the driver into multiple vendor `.c` files, you also start to notice patterns such as "this function is duplicated in all vendor files, so belongs in a generic `.c` file." Sometimes, however, things go the other way. In 2.4, we have several separate RNG (Random Number Generator) drivers. Jeff Garzik found that by merging all of these to the same file, lots of code duplication got removed. Whilst the number of RNG drivers is quite low, if there comes a day when the driver supports many more, it may make sense to abstract them back out into separate files again.

2.2 Reorganising directory structures

The whole idea of directories is to keep similar things together. One simple cleanup that happened in 2.5 was the introduction of the `drivers/char/watchdog` directory. Previously, `drivers/char` contained over 200 `.c` and `.h` files. By introducing the watchdog subdirectory, you can instantly find all relevant drivers. Useful when you have to make changes that affect all watchdog drivers (as was the case in 2.4 when a security bug had been copied through all drivers).

2.3 Simplification of abstraction layers

Sometimes, after introducing support for multiple widgets to a driver, people over-abstract. A prime example of this was the agpgart cache flush routine, which ended up calling through 4–5 function pointers before it actually got to do anything useful.

2.4 Moving to new APIs to decrease LOC

With code lying dormant and unmaintained for several years, it tends to miss the opportunity to take advantage of easier or faster ways of calling kernel-supplied functionality. As helper functions get continually added, the number of lines of code needed to be duplicated in drivers goes down.

3 Case study #1: IA32 CPU setup routines

This started out life as `arch/i386/kernel/setup.c`, and in 2.4, currently stands at 84KB of code which handles setting up of the CPU in terms of working around errata, enabling CPU specific features, and doing some detective work such as finding out the cache sizes. Initially supporting Intel CPUs, the clones started to follow. Today it supports dozens of different types of x86 CPU, from 10 different vendors. In 2.5, Patrick Mochel split this file up into per-vendor support files, and a few generic files. `arch/i386/kernel/cpu/` is a much simpler place to navigate, and is a lot nicer to hack on than its predecessor as a result.

4 Case study #2: IA32 MTRR driver

This monster has been around a few years, and it shows. 71KB of monolithic code, with multiple implementations built on top of each other. Each time, with the abstraction layer bent and twisted into something new. Initially supporting generic Intel MTRRs, it was bent into shape to deal with AMD K6's variants. Then Cyrix's ARR's. Then a myriad

of other clones which did things slightly different. Again, chopping this into per-vendor pieces makes things a lot simpler, and reduces the chance of breaking one vendor when fixing something for another (which has been the case in the past on more than one occasion).

5 Case study #3: Bluesmoke

Bluesmoke is the IA32 machine check exception handling support. As usual, it first only supported Intel P5 and P6 CPUs. Over time, things were changed to support AMD processors, Intel Pentium 4, IDT Winchips, and some additional features such as background checking. This all started to blow up the file size, and it became a pain to find your way around a file with a half dozen similarly named functions. Time for the split-up treatment. 2.5 now has 7 separate C files for the implementations, with a central 'generic' file which calls the specific per-vendor/model implementations.

Whilst it's theoretically possible that you could hack the Makefiles now to only build in (for example) the Intel code if you don't own any non-Intel parts, the added complexity and reduction in functionality for the net-gain of just a few KB of object wasn't deemed worth it. A bigger challenge which would benefit from this change came in the form of the final case study.

6 Case study #4: AGPGART

6.1 History of AGPGART

AGP support was added to Linux back in 1999. Subsequent updates were somewhat infrequent. The bulk of the code never really changed much. Each update just added PCI ID's of new devices, or occasionally a new agpgart implementation when things were just too different to the existing agpgarts.

6.2 How I got involved

During 2002, I was asked by SuSE to implement AGP support for the AMD x86-64. Thinking this would be easy basing assumptions on what I'd previously seen happen to agpgart (thinking it would be just adding some new PCI idents or the likes), I (foolishly?) agreed to do it. Shortly afterwards, I discovered the GART I was writing support for was unlike anything Linux currently supported. Firstly, the north-bridge was on-CPU, which meant on an SMP box, there would be more than one of them, and they would have to be kept coherent with each other. Secondly, it was the first GART to support version 3.0 of the AGP standard. Whilst this is backwards compatible for the most part, there are some additional features that need to be taken care of (such as the transfer speed selector working in completely different ways to how it did in previous versions of the standard). This was quite a lot to take on board, so I started staring at the 134KB of agpgart back-end code (there's also 25KB of front-end code).

6.3 More problems...

Getting up to speed on a driver of this size, which supports over 50 different AGP chipsets, is not a task that happens overnight. Lots of those implementations are either the same, or very similar, but it still leaves around a dozen or so separate code paths. Now to find out which one is most similar to the GART I'm writing for. I eventually gave up trying to find one similar enough, and just started from scratch. My mails for "help" to the original maintainer of agpgart went to `/dev/null`, which meant I had to figure out how a lot of it worked, the hard way. After finally getting things working, I had decided that enough was enough, and for 2.5, I was going to give this code a major overhaul.

6.4 How things were cleaned up

- As usual, first things first, split `drivers/char/agp/agpgart_be.c` (134KB) into lots of smaller source files. One per chipset vendor. This was an instant cleanup, which had no problems being merged. Shortly afterwards, Greg Kroah-Hartman converted the chipset probing routines over partially to some of the 'new' PCI API, killing off a bunch more useless, ugly code.

- With everything in per-vendor files now, things were a lot cleaner, but there was still some real bad mess that needed cleaning. The `agpgart_be.c` file still existed, which acted as a generic part which had all the bits to call the routines in the per-vendor files. One particular ugly that stuck out was the 350-line struct that matched known PCI IDs to init routines. The redundancy in this struct was really bad.

```
static struct {
  unsigned short device_id;
  unsigned short vendor_id;
  enum chipset_type chipset;
  const char *vendor_name;
  const char *chipset_name;
  int (*chipset_setup) (struct
      pci_dev *pdev);
} agp_bridge_info[]
  __initdata = {
\#ifdef CONFIG_AGP_ALI
  { PCI_DEVICE_ID_AL_M1541_0,
    PCI_VENDOR_ID_AL,
    ALI_M1541,
    "ALi",
    "M1541",
    ali_generic_setup },
... (Continue for dozens
     more entries) ...
```

  With this wasteful struct, if 20 out of those 50 entries are for Intel GARTs, we duplicate the vendor ID, vendor name string, and in a lot of cases, the setup routine too. This was cleaned up in several steps.

  - Split the structs out from `agpgart_be.c` to `$vendor.h` (I.e., move all the ALi entries to `ali.h`, AMD entries to `amd.h`, etc.)
  - Remove all duplication from each of these structs.
  - Replace the duplication with a 'header struct' containing the vendor ID, vendor name string, and a pointer to the remaining data.
  - Replace the struct in `agpgart_be.c` with a struct that points to the various split out structures in the `$vendor.h` files.

6.5 The "new" PCI API

Somewhat pleased with myself, I mailed off the changes to Linus, who told me to start again, this time using the pci_driver functionality. As GregKH had done part of the work here already, it wasn't actually that much work to bend what I had already into shape. This did, however, bring about a big change over 2.4's agpgart. With each of the per-chipset drivers now containing a pci_driver struct, they worked independently of the agpgart core as stand-alone modules. I wasn't initially happy with this, but Linus liked it, so it stayed that way. It did, however, mean a rewrite in module locking was needed, which nicely coincided with Rusty Russell rewriting how module locking worked.

6.6 Maintainership

By this point, I had completely gutted the way the agpgart backends worked. I felt I had made significant enough change to adopt the code,

and make an entry for myself in the MAINTAINERS file. Which was probably my second biggest mistake so far. Within just a few days of doing so, my mailbox was flooded with bug reports, stagnant patches, thank-you's, and insults. One thing that I hadn't anticipated was just how far-reaching this code was. Not only did I now have to follow and understand what was going on in the agpgart code, but also found myself digging further into DRI to follow its interaction with AGPGART. Subsequently, even parts of XFree86 came under scrutiny, and even FreeBSD (which interestingly did the 'separate-file-per-vendor' thing from Day One) to see just how much I could or couldn't change without breaking things too much from a userspace point of view.

6.7 Taking AGPGART forward: AGP 3.0 support

After getting on top of the various patches, and fixing the various problems the new code brought about, AGPGART had been dragged kicking and screaming into something that resembled a modern driver. Well, almost. I then moved on to start tackling the next big thing for agpgart: generic AGP 3.0 support. Matthew Tolentino from Intel had come up with a patch for Intel's AGP3.0 chipset (the I7505), and had re-implemented a bunch of code that I had written for the x86-64. After factoring out the common parts, this got to a state where things looked just fine.

6.8 Return of the previous maintainer

Just when things were beginning to go quiet (apart from additional AGP3 GARTs turning up needing implementing), Jeff Hartmann, the original maintainer of the 2.4 AGPGART, reappeared with a 130KB patch against the original 2.4 code. It offered various functionality, supporting AGP3, and cleaning up a lot of code in the process. In a lot of other ways, however, it was a huge step backwards. Splitting Jeff's huge patch into smaller pieces was a massive job. Bits of it went in, and Linus rejected a bunch of them, but there was worse to come (more diffs). At the time of writing, Jeff's outstanding diffs vs. 2.5.59 is around 380KB. A lot of this is unlikely to be merged before 2.6 without considerable rewriting.

6.9 Useless abstractions

Furthering the cleanup mantra, agpgart code has been described in many ways by many people (including *shit* by Linus himself). Pre-cleanup, however, my pet-name for this monster was "abstraction hell." As an excellent HOWNOTTO in abstraction, here's how agpgart used to flush the cache.

- At strategic parts of the code there are `CACHE_FLUSH();` calls.
- `CACHE_FLUSH` turns out to be a macro which expands to `agp_bridge.cache_flush`
- `agp_bridge.cache_flush` in 99% of cases, points to `global_cache_flush`. The remaining case could have been special-cased in `global_cache_flush`.
- On SMP, `global_cache_flush` is a define for `smp_flush_cache`. On UP, it's a define for `flush_cache`.
- `smp_flush_cache` just does an `smp_call_function` on `flush_cache`.
- Finally, `flush_cache` does a "wbinvd" on IA32/X86_64, "mb" on IA64, or `#error`s on anything else.

7 Future directions

7.1 AGPGART

There is still a lot of work to be done on AGPGART. All the work so far concentrated on the back end (which is where all the chipset magic happens).

- The front-end of the driver (ioctl interface, etc.) is almost as crufty, and needs a lot of work to rid it of silly things like open-coded list handling routines instead of using the generic `list.h` routines. (Yet more proof that duplicating functionality is a bad thing: it gets its double-linked list implementation horribly wrong.)
- More work on making the AGP3.0 support transparent.
- Inevitably more support for additional chipsets.
- Multiple AGP bridge support.
- `sysfs` migration to get away from the horrible ioctl interface. This will unfortunately make the Linux AGPGART completely incompatible with the FreeBSD implementation. The only people this causes concern for are XFree86 developers, who have to support an additional interface.

7.2 Other kernel work

- APIC drivers. The IA32 APIC code is quite horrible, and quite fragile. It supports a lot of different types of setup, from lots of different generic PCs, to the weird and wonderful bigger machines like NUMA-Q, Summit, and more. The x86 sub-architecture support cleaned up some of this by introducing the possibility for each sub-arch to implement their own APIC code, but it hasn't really improved readability or maintainability of the APIC code to any great length.
- Watchdogs. Small scale cleanup occurred already in 2.5, which was to just group all the watchdog drivers from `drivers/char` into a new subdirectory called imaginatively `watchdog/`. A lot of these drivers are duplicating lots of code, sometimes subtly differently, when they should be using the same code. For 2.7 a nice cleanup would be to abstract out the generic parts of this to a layer above the watchdog drivers in a similar way to what happened with AGPGART. In 2.4 there was a security hole which meant every single watchdog driver needed to be audited and fixed. By moving all this functionality out of the drivers, this could have been fixed in a single place.

8 Summary

- Split out multiple implementations to their own files unless they are small and/or similar enough to the existing implementation.
- Don't re-implement code unnecessarily, even if you think you may need something extra that the generic code doesn't give you. Build on top of the generic code rather than re-implementing.
- Use modern interfaces where possible. This isn't always easy if you want your driver to compile on earlier kernel versions as well (especially true for out-of-tree drivers).
- Before abstracting something out, think about why you actually need it abstracted. What will the callers of the abstraction do in the common case?

- Directories are there to keep similar things together. Use them. (Obviously, only when they make sense; a directory for 2–3 drivers is perhaps going too far).

- Don't disappear for four years and reappear with a 380KB patch against the last code you maintained. You may find that a lot has changed whilst you were gone, and merging will be a *nightmare*—especially if you didn't keep individual per-change changesets.

Proceedings of the Linux Symposium

July 23th–26th, 2003
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Alan Cox, *Red Hat, Inc.*
Andi Kleen, *SuSE, GmbH*
Matthew Wilcox, *Hewlett-Packard*
Gerrit Huizenga, *IBM*
Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Martin K. Petersen, *Wild Open Source, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*