Analyzing the Android Kernel

April 2014

Introduction

System complexity results from dependencies that can violate an intended architecture. Dependencies prevent modularity and the lack of a well-defined and well-understood architecture makes change difficult to manage. We decided to analyze the Android kernel to illustrate how understanding software architecture and identifying dependencies can help any development organization manage the impact of change, improve productivity, simplify code maintenance, and reduce risk. This is the first time an in-depth analysis of this kind has been done on the Android kernel.

The Android kernel comes in a variety of configurations. These have been given various names based on different hardware configurations. For the purposes of this paper, we focused on the *omap* project, which is used on PandaBoard and Galaxy Nexus.

The same approach can be used for an analysis of *msm*, *samsung*, *tegra* or other Android configurations. Furthermore, since the Android kernel is derived from the Linux kernel, the same technique can be applied to any Linux kernel.

Our goal is to look at the architecture of the Android kernel from a developer perspective. Using Lattix Architect, we will identify all code elements such as methods, structs, classes, and global variables and discover the dependencies between them. We will then aggregate them into the source files and directories where they are defined to create a big picture view of the Android kernel. This is a difficult task given the complexity of Android and its build environment.

Why do the Analysis?

Once we create a dependency map, it will enable us to answer a number of interesting questions.

- What does the big picture view look like? What are the lower level modules and what are the higher level modules within the kernel?
- How do the various parts of the kernel depend on each other? For instance, why does “kernel” depend on “ipc” or which “drivers” depend on “net?”
- How coupled is the entire code base?
- If we are planning to make one or more changes, what are all the things that could be affected either directly or indirectly by those changes?
- What are the differences between one version of the kernel and another?
- How can we re-organize the kernel to make it more modular so it becomes easier to understand, maintain, and test?

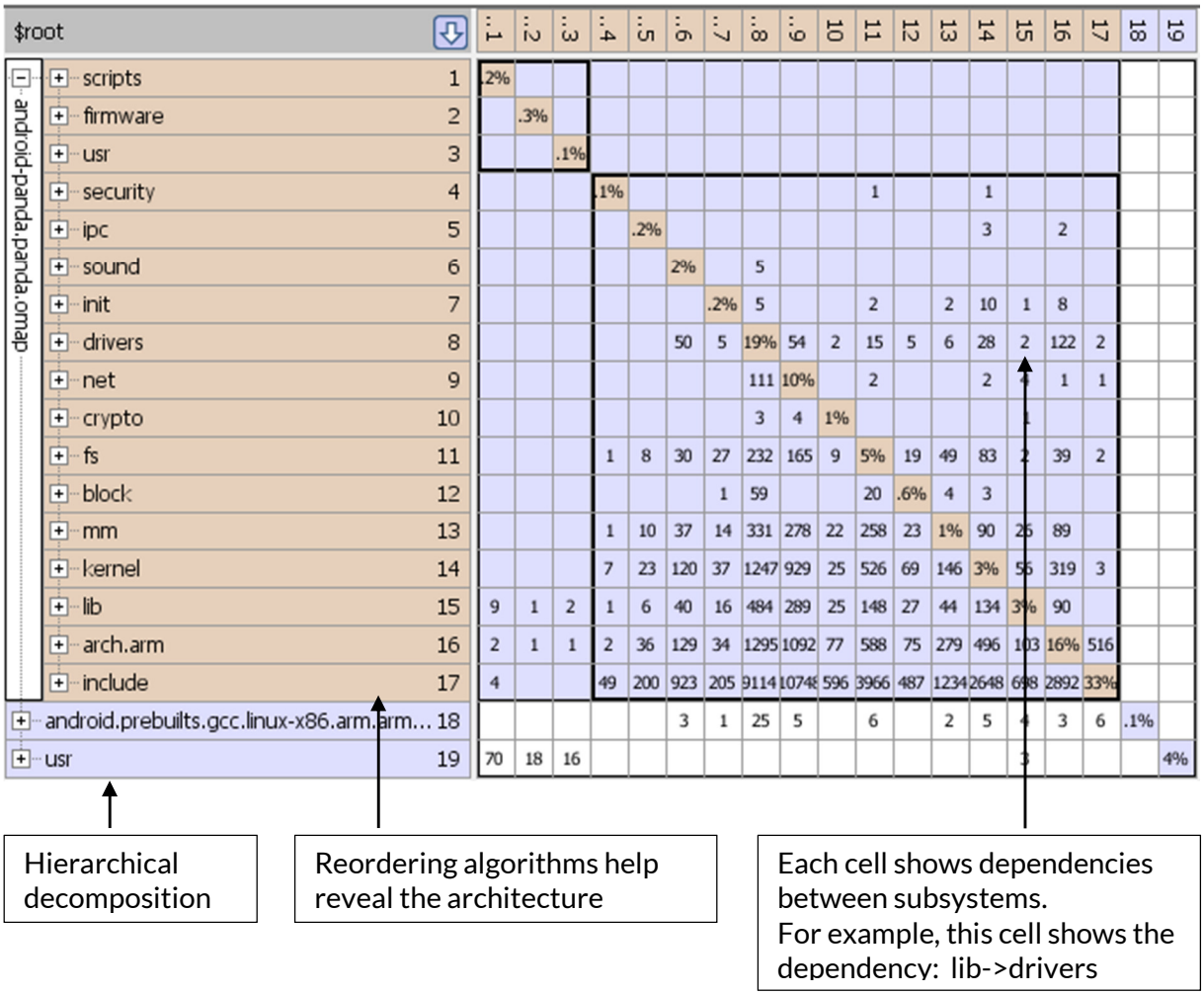
LATTIX

The Big Picture View

We analyzed 1.6 million lines of code consisting of:

Source Files:	1,531
Header Files:	1,827
Functions:	37,732
Data (static/global):	21,184
Structs:	9,601

The total number of model elements was 178,724. With Lattix Architect, we mapped the dependencies between all of these elements, while classifying them based on dependency types.



Dependency Map for Android Panda

- This is a Dependency Structure Matrix (DSM) view of the Android kernel. It is also possible to create a box and line diagram within Lattix Architect. The highly coupled nature of the Android kernel makes a DSM view particularly useful for the analysis.
- Each cell of the DSM represents the dependency of one subsystem on another. For instance, we see that 'lib' depends on 'drivers' with a strength of 2. In Lattix Architect, just click on the cell to see what the dependency is. You can also easily navigate to the source code in your favorite editor directly from Architect.
- The hierarchical nature of the map allows you to aggregate the dependencies and see how high level subsystems relate to each other. The initial hierarchy is the file structure of the code. The hierarchy allows you to drill down to the lowest level element (method, data, struct, etc.).
- A *partitioning algorithm* was applied to order the subsystems in such a way as to reveal the underlying design intent. The algorithm split it into two layers. Furthermore, elements in each layer were ordered to minimize the strength above the diagonal. For instance, the automatic ordering reveals that 'kernel' and 'lib' are at a lower level compared to 'scripts' or even 'ipc.' Partitioning also exposes how coupled the software is.
- It is possible for users of Lattix Architect to re-organize the hierarchy to reflect the intended design. For instance, it would be easy to restructure the system to split up the drivers into two groups. The first group would be the driver code that the lower level subsystems depend on. The second group would be drivers that depend on the lower level subsystems but are not depended on by them. Another example is to identify the global variables that could be easily moved from one file to another. It is easy to do this kind of "what-if" analysis. Furthermore, as you make changes, Lattix Architect will remember those changes and automatically generate a work list for them.

Conclusion

It is immediately apparent that the Android panda kernel is highly coupled. This makes it hard to understand. It is also expensive to maintain because even a small change can have unexpected consequences.

Some of the unnecessary coupling could be reduced by moving variables and methods from one subsystem to another. It is also possible to invert dependencies and to be explicit about the interfaces.

By analyzing successive versions of the code, it is easy to see what has changed from one version to another and to see whether those changes are contributing to further erosion of the intended design.

Try Lattix Architect on your Project

Is your software complex, buggy and hard to maintain?

Do you want to see what your code looks like and what you can do to modularize it?

If so, contact Lattix at sales@lattix.com.

Lattix, Inc.
www.lattix.com

LATTIX

