Electronic Communications of the EASST
Volume 8 (2008)

Proceedings of the Third International ERCIM Symposium on Software Evolution (Software Evolution 2007)

The Evolution of the Linux Build System

Bram Adams, Kris De Schutter, Herman Tromp and Wolfgang De Meuter

16 pages

Guest Editors: Tom Mens, Ellen Van Paesschen, Kim Mens, Maja D'Hondt
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

The Evolution of the Linux Build System

Bram Adams[1], Kris De Schutter[3], Herman Tromp[2] and Wolfgang De Meuter[4]

[1] bram.adams@ugent.be
[2] herman.tromp@ugent.be
GH-SEL, INTEC, Ghent University
Sint-Pietersnieuwstraat 41, 9000 Ghent, Belgium

[3] kdeschut@vub.ac.be
[4] wdmeuter@vub.ac.be
PROG, Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussels, Belgium

Abstract: Software evolution entails more than just redesigning and reimplementing functionality of, fixing bugs in, or adding new features to source code. These evolutionary forces induce similar changes on the software's build system too, with far-reaching consequences on both overall developer productivity as well as software configurability. In this paper we take a look at this phenomenon in the Linux kernel from its inception up until present day. We do this by analysing the kernel's build traces with MAKAO, our re(verse)-engineering framework for build systems. This helps us in detecting interesting idioms and patterns in the dynamic build behaviour. Finding a good balance between obtaining a fast, correct build system and migrating in a stepwise fashion turns out to be the general theme throughout the evolution of the Linux build system.

Keywords: build system evolution, case study, Linux kernel, MAKAO

1 Introduction

The majority of software evolution research is targeted at direct participants in the evolution process like source code and design artifacts. While these play —of course!— a major role, much can be learnt from indirect evolution partners. The build system is one of them.

A build system takes care of two things:

- it decides which components should be built and establishes any platform-dependent information needed to do so;
- it incrementally builds the system taking dependencies into account.

Before 1977, most people wrote their own ad hoc build and install scripts for this. Then, Feldman introduced a dedicated build tool named "make" [Fel79]. It was based on explicit declarative specifications of the dependencies between targets (executables, object files, etc.) in textual "Makefiles", combined with imperative "recipes" (list of shell commands) for building a target. A time stamp-based updating algorithm considerably improved incremental compilation

of software projects and enhanced the quality of builds. The underlying philosophy of "make" has influenced lots of other build tools, but traditional "make" systems are still in wide use today.

Configuration systems like e.g. GBS (GNU Build System)[1] complement build tools to constitute a full build system. They let build scripts abstract from platform-specific information like include directories or compiler versions, in most cases by means of parameters which are resolved right before build-time. At the same time, they help in composing a system from the various available modules and their different versions.

One example of a widely used build system can be found in the Linux kernel. Linux is a —by now famous— operating system, used both by enthusiasts as well as in industrial settings. It started out in 1991, when Linus Torvalds sent an email to the Minix newsgroup, stating that he had developed a free operating system he wanted to share with everyone. It was a monolithic kernel, in which device drivers were hardcoded (e.g. Finnish keyboard), with user space programs ported over from Minix. In a little over fifteen years, Linux has grown from this one-man hobby project into what is probably the largest open-source project on the planet, praised for its portability across computer configurations.

Linux has been the target of many studies. The one which is of most relevance to this paper was done by Godfrey and Tu in [GT00], where the kernel source code was studied from the perspective of software evolution. Contrary to Lehman's laws [LB85], the kernel exhibited a superlinear growth in size. This means that strangely enough the growing complexity did not temper the kernel's evolution.

Source code, however, can not evolve in isolation. We claim that the build system co-evolves with the source code. Every time new source code gets added, or existing modules are moved around, one is forced to deal with the build system in order to keep the software system compilable. An agile build system gives developers more freedom to restructure and refactor the source code. Hence, traditional evolution steps have an immediate impact on the build system, but the inverse also holds. Yet, no work to quantify this claim exists yet (the build system was explicitly ignored in [GT00]). In this paper, we will therefore examine this peculiar relation by means of the Linux kernel. Considering various major kernel releases from the very beginning to the recent versions (at the time of writing), we will investigate the various forces on its build system.

In Section 2 we will explain the setup of our case study on the Linux kernel and its build system. Our measurements will enable us to make three important observations. First (Section 3), we will investigate whether and how the build system evolves. Then (Section 4), we will look at the build system's complexity and discuss our observations in general. Finally (Section 5), we will go into more detail and zoom in on build system maintenance activities. As this is still preliminary work, we will point out future research directions (Section 6) before we summarise this paper's contributions in Section 7.

2 Setup of the case study

To study the evolution of the Linux kernel build system, we looked at most of the pre-1.0 releases of Linux, as well as the major stable post-1.0 releases (up to the 2.6 series). Table 1 gives an

[1] http://sources.redhat.com/autobook/

Date	Version	Date	Version
September 17,1991	0.01	March 13, 1994	1.0
December 8, 1991	0.11	March 7, 1995	1.2.0
May 25, 1992	0.96a	July 3, 1996	2.0.1
July 5, 1992	0.96c	January 26, 1999	2.2.0
July 18, 1993	0.99.11	January 4, 2001	2.4.0
September 19, 1993	0.99.13	December 18, 2003	2.6.0
February 3, 1994	0.99.15	June 11, 2007	2.6.21.5

Table 1: Chronological overview of the Linux versions we investigated.

overview of the processed versions, which span some 15 years of real-world development time. The distribution of systems seems to be skewed towards the earliest versions (1991–1995), but as Linux was still a young system back then, it was much easier to make drastic changes than later on. Indeed, samples in the later kernel series revealed no drastic build changes within a series, except for the 2.6 line. This is due to the changed development model, i.e. no unstable kernel anymore parallel to the stable one.

To each of the kernels of Table 1 we first applied David A. Wheeler's SLOCCount tool[2] in order to calculate the physical, uncommented SLOC (Source Lines Of Code) of source code (.c, .cpp, etc.), build scripts ("Makefile", "Kbuild", etc.), configuration data ("config.in", "Kconfig", etc.) and support build files (.sh, .awk, etc.).

Then, we compiled each of the kernels using an initial configuration we reused and enhanced with new kernel features as needed throughout the measurements. We opted for a run-off-the-mill configuration. Influence of specific configurations on our results is future work. The build traces were fed into MAKAO[3] [ADTD07], our re(verse)-engineering framework for build systems, in order to obtain the corresponding build dependency graphs, and some figures on the number of targets and dependencies.

Finally, each dependency graph was loaded into MAKAO to visualise, query and filter them. In this paper, we will report on the most striking maintenance activities like e.g. the transition from the 2.4 to the 2.6 kernel.

In our measurements, we exclusively focused on the Linux build's intrinsic complexity, not on how the end user or developer perceives the build system. Whereas the build system engine ("how") may significantly be optimised and tweaked, the Linux build users are largely shielded from this via domain-specific build and configuration languages ("what"). In personal communication, Linux 2.6 build maintainer Sam Ravnborg coined the phrase "simple syntax for simple things" for this. The quality of this build interface is hard to measure however, as it is more related to language design and evolution.

2 http://www.dwheeler.com/sloccount/
3 http://users.ugent.be/~badams/makao/

2.1 Build graph measurements

MAKAO is a re(verse)-engineering framework for build systems [ADTD07]. It is based on a directed acyclic graph (DAG) model of actual builds [Fel79] to which static build script data like command lists, line numbers, etc. has been attached. Nodes represent build targets and edges denote dependencies. The DAG is obtained by parsing a trace of the build, i.e. by running "make" with debugging flags and processing the output afterwards.

MAKAO is built on top of GUESS [Ada06], a graph manipulation tool in which nodes and edges are objects with their own state and behaviour. Besides visualising a build DAG, it can be controlled programmatically using the embedded Gython scripting language (derived from Python). This is ideal for performing detailed queries on a particular build graph, or writing build refactorings. We also wrote tool support to reify the graph as Prolog facts and apply logic rules to manipulate them. As such, dependency graphs can be inspected and studied in a more controllable way than by merely reading the build scripts. Validation is another useful application.

More details on MAKAO can be found in [ADTD07], which explains the rationale behind it as well as its applicability in the realms of build maintenance. The current paper focuses on one of the future work topics mentioned there, i.e. learning things about the source code by looking at the build system. One of the prerequisites for that is being able to measure their joint evolution behaviour from some important properties, i.e. the measurements presented above.

As MAKAO does not yet take configuration data into account, we only looked at the SLOC history for this. More detailed configuration analyses are future work.

3 Observation 1: the build system evolves

This section presents and discusses the measurements obtained with SLOCCount. Figure 1 shows the growth over time (in logarithmic scale) of the number of non-comment and non-blank lines which make up the source code, and the build system. The latter is split over three parts: (1) the actual Makefiles (`build`), (2) the configuration files which drive the selection process of what should get built (`config`), and (3) other tools and scripts which assist the actual build process (`rest`). This last category contains build-time scripts to extract symbol tables, install kernel components, etc.

Our measurements confirm the claim made in [GT00] about the source code's super-linear evolution in SLOC. For our purposes, the observation to be made from this graph is that the build system also evolves over time[4]. Over the course of fifteen years the SLOC of the Makefiles has grown by a factor of about 52. There are various reasons for this. Initially (until 0.96x kernels), build scripts flourished because of the rapid growth in source code, and also the addition of new subdirectories for driver code (block and character devices, networking, file systems, etc.), each with their own build scripts. In the 1.2.0 kernel, architecture-dependent code was separated from the architecture-independent code, and put under subdirectories of the "arch" directory. This resulted in a rise of SLOC of all build related artifacts.

What is also striking from Figure 1 is the amount of work which has been put into the config-

[4] Note that measurement of SLOC occurs statically, which means that it is independent of the chosen build configuration.

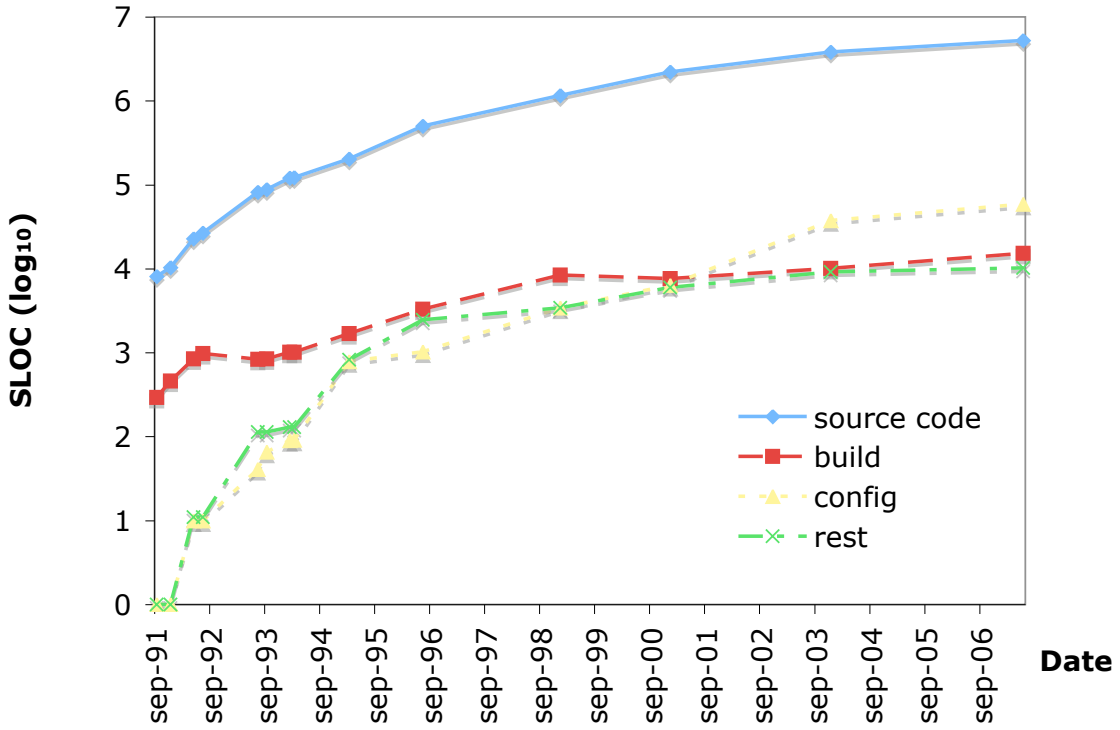

Figure 1: Linux kernel complexity (SLOC).

uration and support system. These have grown from nothing in the first version[5] to almost 60K and 10K lines resp., the former even getting ahead of the core build files. An important event for the configuration layer was the addition of a configuration Bash script in the 0.99.1x series. This took a file called "config.in" declaring the requested configuration and generated a ".config" file for usage during the build and an "autoconf.h" for usage inside the source code. Further milestones include support for the `oldconfig`[6] and `modules` phases in the 1.2.0 kernel, distribution of the central "config.in" across all directories in the 2.0.0 kernel together with the advent of graphical configuration frontends (partially explaining increase in SLOC for `rest`), and a configuration language overhaul in the 2.6 kernel (more on that later).

The graph also gives a strong indication of co-evolution between source code and build system, as both more or less follow the same growth pattern. More data, e.g. at the subsystem level [GT00], is needed to identify the exact relationship.

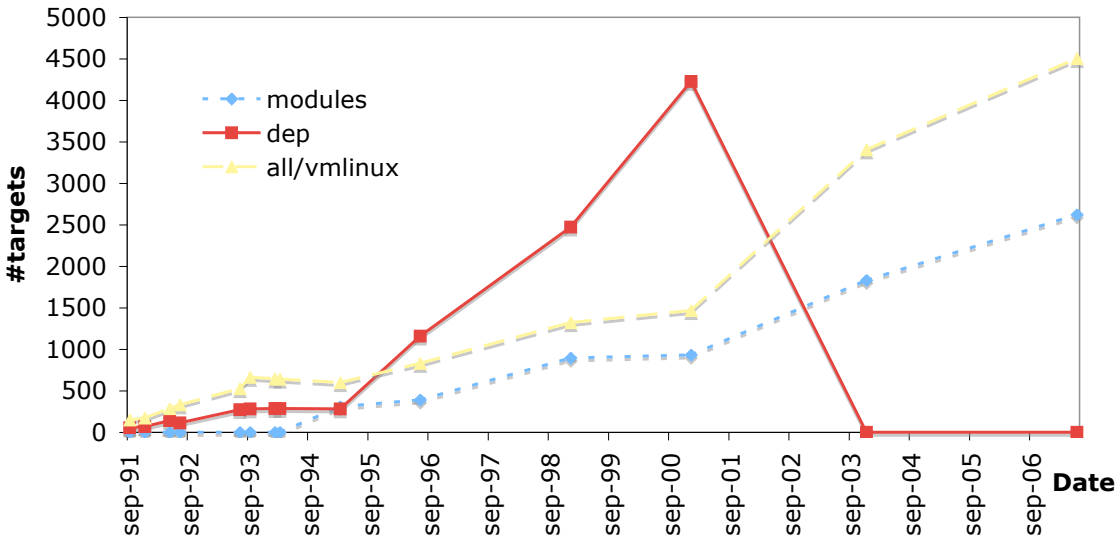

Figure 2: Linux build system targets.

4 Observation 2: complexity increases

In order to assess the evolution of the complexity of the build system we will take a look at some figures related to the actual Makefiles. As noted in the previous section, we will ignore configuration files and other supporting scripts in this section.

Figure 2 presents the growth over time of the number of targets which participate in a build. This point needs some elaboration. The Linux build process is divided into a number of phases, of which we will consider the most important ones. The kernel image is either built by a phase named `all` or `vmlinux` (starting from version 2.6.0 in 2003), while modules are built within the `modules` stage. Extraction of dependencies occurs during the `dep`-phase. As MAKAO works on dynamic traces of the build system [ADTD07], what Figure 2 shows is the number of targets checked or built by the build process during a concrete run of each of the three phases[7].

Overall, Figure 2 reflects the point made in the previous section: the build system grows, not only in lines of code, but also in the number of tasks it attempts to complete. Part of this should of course be blamed on the addition over time of new features in our build configuration, but there is much more to it than this. Another general remark we can make is that, starting from the 2.6 kernel in 2003, the `dep`-phase disappears and is subsumed by the other two phases.

Figure 3 and Figure 4 relate the growth over time of the number of explicit and implicit[8] build dependencies respectively, for each of the three different build runs. What we see here is

[5] Technically, the measurements for `config` and `rest` around '91–'92 have zero as value, which should be mapped to minus infinity on Figure 1. For practical reasons, we of course approximated this by using a logarithmic value of zero on the graph, corresponding in fact to 1 SLOC.

[6] Generates a new configuration from an older one.

[7] Every kernel was built from scratch, i.e. we did not measure incremental builds.

[8] Implicit dependencies are relationships which are not explicitly declared as Makefile rule dependencies, but occur as arguments or output of the rule's commands.

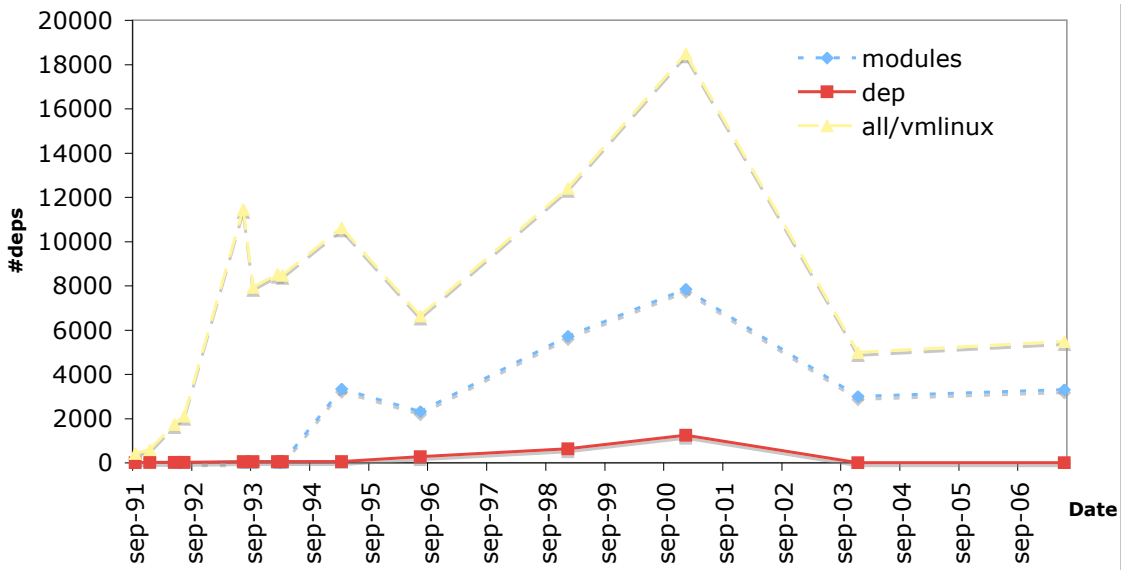

Figure 3: Linux build explicit dependencies.

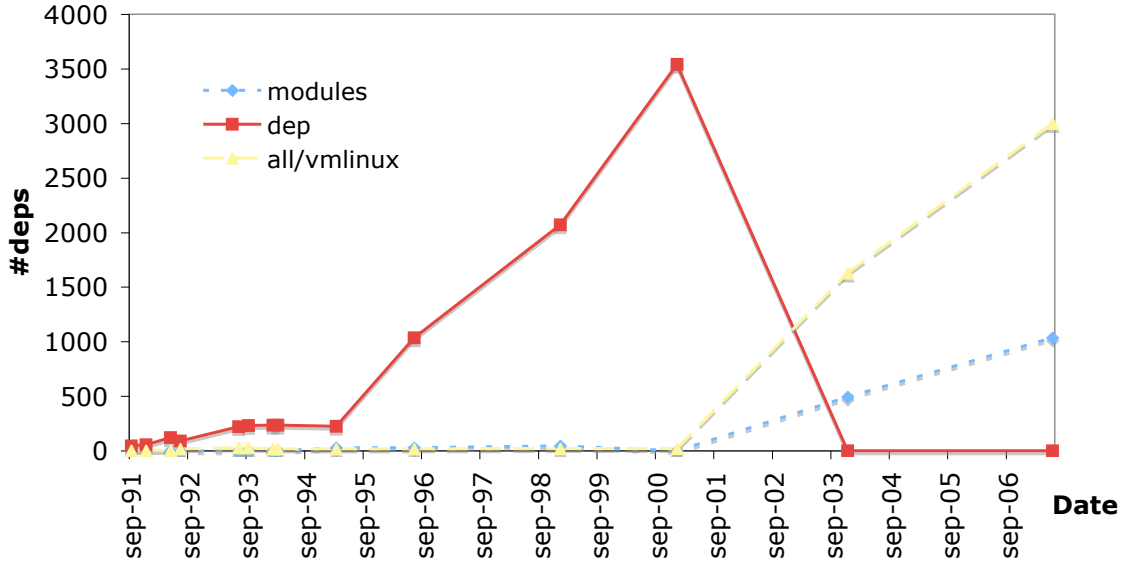

Figure 4: Linux build implicit dependencies.

a turbulent course culminating in a huge growth up to September 2000 (kernel 2.4), followed by a serious dip. We also notice that eventually the number of dependencies rises again, albeit at a slower pace.

The first point to note here is related to what dependencies stand for. Basically, dependencies capture the relationships between targets. As their number grows, so does the complexity of understanding the relationships between the targets, which in their turn relate back to physical

	all/vmlinux		modules		dep	
	explicit	implicit	explicit	implicit	explicit	implicit
0.01	147	2	N/A	N/A	8	46
0.11	170	2	N/A	N/A	12	55
0.96a	281	5	N/A	N/A	20	121
0.96c	314	16	N/A	N/A	22	91
0.99.11	503	26	N/A	N/A	50	222
0.99.13	634	28	N/A	N/A	50	232
0.99.15	631	13	N/A	N/A	52	233
1.0	628	13	N/A	N/A	52	233
1.2.0	583	17	282	26	57	226
2.0.1	817	16	376	19	171	986
2.2.0	1300	22	865	31	399	2072
2.4.0	1446	20	931	0	684	3542
2.6.0	1865	1538	1431	402	N/A	N/A
2.6.21.5	2158	2347	1877	746	N/A	N/A

Table 2: Analysis of the nature of the build targets in the investigated 2.4 and 2.6 kernels. For each phase, the sum of the number of explicit and implicit targets yields the data of Figure 2.

components of the software system and their interconnections. By "physical components", we mean the decomposition of the source code in a sufficiently succinct way such that the build system is able to create a working application from it. Hence, the growth of the number of dependencies shows at least that understanding the build system becomes harder and harder. More investigation is needed to verify whether this is also a symptom of growing complexity in the source code.

This problem becomes compounded when we consider the implications of Figure 4. What this shows is that there is also a steady growth in the number of implicit dependencies. This means that the number of relationships the build system knows nothing about is on the rise. This is not only problematic when trying to understand the build system, it also constitutes a potential source of build errors, and (at best) may lead to suboptimal builds. Luckily, most of the implicit dependencies originate from temporary files created during the `dep` phase, i.e. text files with dependency information extracted from the source code's #include relations. This can be deduced from the graphical representation of the build dependency graph[9], but also e.g. from the fact that the `modules` and `all/vmlinux` phases only start to exhibit implicit dependencies after the `dep` phase was merged with them.

This implies that between the 2.4 and the 2.6 kernel, the actual source code components did not change as dramatically as Figure 2 indicates at first sight. Table 2 shows e.g. that whereas the number of implicit targets[10] increased with more or less 1500 for the `all/vmlinux` phase, the number of explicit targets increased with approximately 400 between the 2.4.0 to 2.6.0 kernel

[9] The graph of the `dep` phase from in the beginning resembles a spanning tree, hence the resemblance between its corresponding charts in Figure 2 and Figure 4.

[10] An implicit target is a target which is the destination node of at least one implicit dependency. In many cases, the number of implicit targets equals the number of implicit dependencies. An explicit target is a target which is not implicit.

(which is less than between the 2.0.1 and 2.2.0 kernel). This observation does not entirely hold for the `modules` phase, which is probably due to the addition of extra drivers to the build configuration. Between the 2.6.0 and 2.6.21.5 kernel, extra (explicit) header file targets are added, while two new kinds of implicit targets account for the increase in implicit targets.

5 Observation 3: maintenance as driver of build evolution

Having observed that the build system evolves and that its complexity increases throughout, we will now examine efforts of the Linux build developers to reduce this growing complexity.

5.1 Maintenance until the 2.4 series

In the pre-1.0 era (1992-1993), effort has been spent to mitigate build complexity, as Figure 1 shows a decrease in SLOC of fourteen percent for this period. A new recursion scheme had been introduced along which each Makefile in a directory defines a variable with the subdirectories to traverse over together with a specific rule ("linuxsubdirs") to do the traversal. These rules depend on a phony target named "dummy" to make sure that each build executes the rule's command list.

```
obj-y :=
obj-m :=
obj- :=
obj-$(CONFIG_SOUND) += soundcore.o
```

Figure 5: List style build scripts.

Figure 3 also points to some reduction attempts. Between 1.2.0 and 2.0.0 (1995-1996), common build logic was extracted into a shared build script called "Rules.make", resulting in a 40% reduction of explicit dependencies (while the number of targets increased). Between 2.2.0 and 2.4.0 (1999-2000), the decrease of nine percent in SLOC can be attributed to a massive rewrite of the build scripts and "Rules.make" in a more concise "list-style" manner [linb]. The central "Rules.make" rule base of the 2.4.0 kernel contains rules expressed in terms of variables which denote lists of file names. These lists are initialised as shown in Figure 5 in the various Makefiles spread throughout the system. Depending on the fact that e.g. sound support (CONFIG_SOUND) is chosen as a built-in feature ("y"), module ("m") or left out ("") either the "obj-y", "obj-m" or "obj-" variable is assigned object file soundcore.o. The central rule logic is expressed in terms of these variables and just iterates through the list of names. In this way, build logic boils down to list processing. This approach was highly experimental in the 2.2.0 kernel such that the old "Rules.make" did not know about it and each new list-style Makefile had to convert its variables to the ones expected by "Rules.make". As more and more components switched to the list style during the 2.2.x series, many of the performance gains of the original "Rules.make" optimisations were becoming subdued and even introduced new problems like redundant work being done. This situation lasted until Linus Torvalds himself started to change this, when he

described his grievances about "Rules.make" in an email to the Linux kernel mailing list[11]:

> OK. That's it. I had enough. That whole [...] mess has to go. It's too much crap to carry around, and when trying to fix one kind of build we invariably break another.

As a result, the "Rules.make" was rewritten for 2.4.0, which forced all components in the system to migrate to the list style.

Another area of big build evolution steps is the `dep` phase. In the earliest releases, detection of source code (header file) dependencies within the build system was not that sophisticated. Releases 1.2.0, 2.0.0 and 2.2.0 each tried to make dependency management more correct without sacrificing (build) speed. New dependency extraction scripts, recursion schemes, decomposing the generated configuration header file for each configuration item, etc. tried to achieve this. Eventually, as people still encountered many nuisances, the separate `dep` phase completely disappeared in the 2.6 kernel and was merged into each build phase. The trade-off between accurate information versus build speed is very apparent in this area. This was also coined as "correctness trumps efficiency" by Michael Elizabeth Chastain, the Linux build maintainer [linb].

5.2 Towards the 2.6 kernel

This brings us to the huge maintenance step occurring between 2.4.0 and 2.6.0 (2001-2003):

- the separate dependency extraction step has vanished and is subsumed in the other phases (see previous section);
- the number of explicit dependencies has dropped drastically to one third of the 2.4.0 level;
- the number of targets in the `vmlinux` build has increased with a factor of 2.3.

There were a lot of political struggles involved with the 2.6 kernel overhaul. In 2000, people independently proposed substitutes for both the configuration and the build layer. CML2[12] tried to replace the informally defined configuration layer by a domain-specific language implemented on top of a custom rule-engine written in Python, and also to address the reliance on conditional compilation and building. Kbuild 2.5[13] was a rewrite of the Linux build layer as a "non-recursive build" (one "make" process builds everything instead of one process and Makefile per subdirectory [Mil97]), which fixed dependency generation problems, Makefile complexity, parallel builds, etc. The introduction of non-conventional kernel technologies like Python, and absence of incremental migration strategies (in contrast with the list-style Makefiles of Subsection 5.1) prevented both orthogonal initiatives from being accepted.

Instead, the existing build system was incrementally upgraded for the 2.6 kernel to squeeze every possible bit of performance out of it. Kconfig (initially called "LinuxKernelConf")[14] was designed by Roman Zippel and has been merged into the kernel in version 2.5.45. Basically, a standard configuration language specification was devised to which all backends should adhere. The existing "config.in" files were rewritten and renamed to "Kconfig", which explains

[11] http://people.redhat.com/zaitcev/notes/linus-makefiles.html
[12] http://lwn.net/2001/features/KernelSummit/
[13] http://sourceforge.net/projects/kbuild/
[14] http://www.xs4all.nl/zippel/lc/

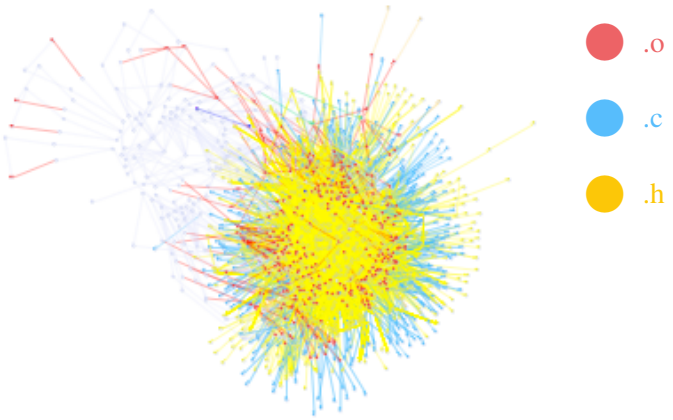

Figure 6: Linux 2.4.0 build process in phase `all` (with header file targets).

the slight increase in SLOC on Figure 1. The build scripts themselves were refactored by Kai Germaschewski. He managed to incorporate a number of Kbuild 2.5's proposed features on top of the existing build infrastructure, avoiding a non-recursive "make". For this, "Rules.make" had been made much more sophisticated (and renamed) and all "make" subprocesses were invoked with the top directory as working directory. Build speed was lower than in the Kbuild 2.5 case. We are now going to look in depth at the changes introduced by Germaschewski.

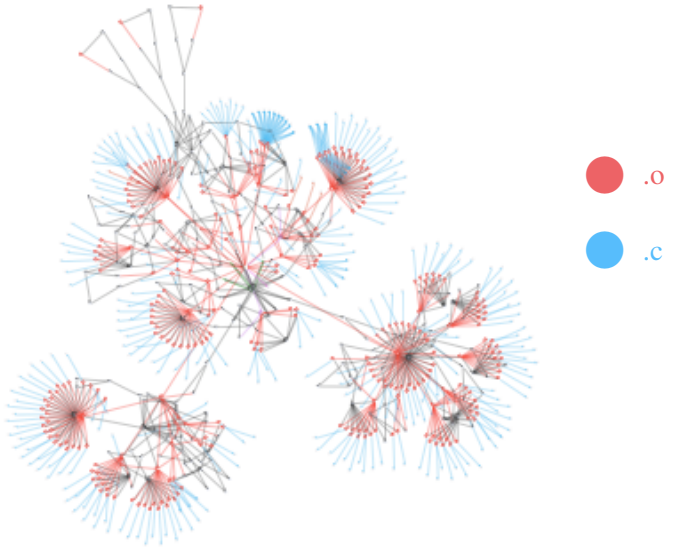

Figure 7: Linux 2.4.0 build process in phase `all` (without header file targets).

When looking at the dependency graph of the 2.4.0 kernel image build (`all`) on Figure 6, it is clear that there are a lot (16615) of dependencies on header files (yellow edges to yellow nodes). This becomes apparent on Figure 7, where all these edges have been elided. Only then, we clearly see the various object (red nodes) and source files (blue nodes) with dependencies between them. By querying of the static information attached to the nodes and edges, we find

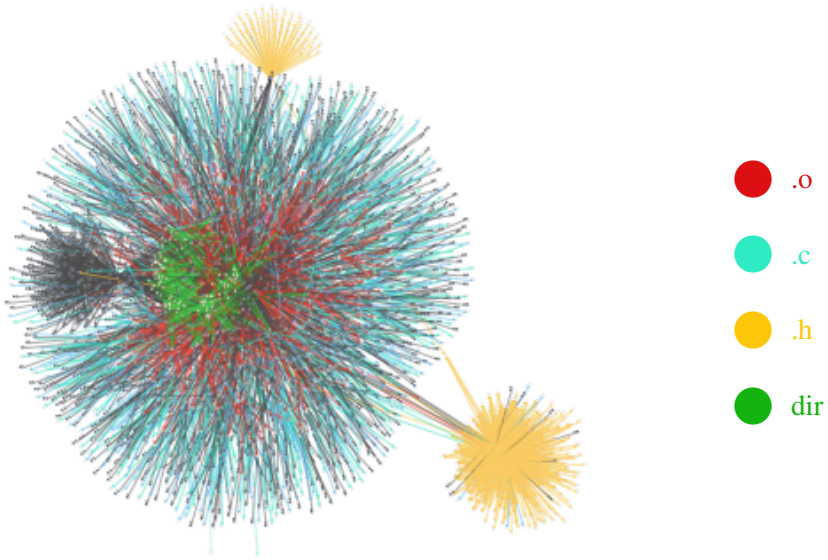

Figure 8: Linux 2.6.0 build process (phase `vmlinux`).

that each cluster of nodes corresponds to a particular directory. Kernel subcomponents seem to be delimited by directory boundaries.

Figure 8 shows the corresponding build dependency graph of release 2.6.0. Here, most of header file dependencies have vanished, and the remaining ones (843) are localised in two clusters. We see a very dense build, almost one huge cluster of nodes. Paradoxically, the dependency graph looks much more complex after the maintenance activities than before (Figure 7). Keep in mind that the end user is not directly confronted with this complexity.

There are various reasons for this. First, dependency generation now occurs as part of the build instead of as a separate phase (see section 5.1). To make this more efficient, a technique invented by Tom Tromey for automake[15] has been applied [Mec04]. Basically, if a source file has been changed one is certain that it should be recompiled. Hence, it is unnecessary to check its dependencies first. It suffices to generate the source code's new dependencies during its compilation, i.e. for use in future builds. As a consequence, during a clean build no source code file requires checking of its dependencies, which explains the disappearing of more than 15000 edges. Second, there are many new implicit dependencies which represent relations between object files and temporary files generated during the build. These hint at changed build commands.

Third, a couple of build idioms are in use, which tie together a lot of build targets. The most obvious one is the "FORCE"-idiom. Target `FORCE` is a so-called "phony" target, i.e. it should always be rebuilt as it is never up-to-date. Whereas the 2.4.0 kernel contains similar phony targets in every subdirectory, the 2.6.0's `FORCE` target is unique across the whole DAG. This indicates a change in working directory, one of the steps in shifting closer to a non-recursive build. The majority of build rule targets depends on the central `FORCE` target to make sure that their command list is always executed (as `FORCE` is always newer than a rule's build target). To avoid that everything is recompiled, the rules' recipes consistently call a custom "make"

[15] http://www.gnu.org/software/automake/

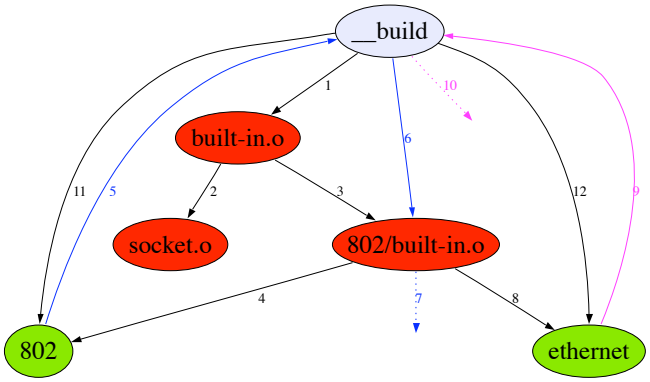

Figure 9: Circular dependency chain in the Linux 2.6.0 build system.

function which decides whether the build target is still up-to-date. Hence, "make" is solely used to express build dependencies. In practice, determining whether rebuilding is necessary still happens via time stamps, but is now controlled by the particular implementation of the custom decision function instead of by "make" itself. This backfires at MAKAO, especially in the case of incremental builds, as it will interpret all formal build dependencies as actual dependencies, although the custom function will have ruled out many of them as being up-to-date. To abstract away this idiom (and open up the graph), we can use MAKAO's Prolog component to write a logic rule [ADTD07] which removes `FORCE` and its associated edges.

The remaining graph reveals some other peculiarities. The kernel image consists of a number of modules which are big object files linked together from various smaller ones. It is easy to detect these "composite objects" [lina], as they are usually called `built-in.o` and their only dependencies are object files. Many of them have the center node of the graph, `__build`, as parent node, as illustrated by the blue and red nodes of Figure 9. Composite objects are in use since the very first Linux version to overcome "recursive make" ordering problems [linc]. Despite speed and correctness issues, they are still in use in the kernel build system.

In combination with composite objects, a phenomenon we named "circular dependency chain" occurs. We show a schematic overview in Figure 9, with green nodes representing directories. Edges with a common color occur within the same "make" process in the order given by the numbers, so three "make" processes are needed here to build the `built-in.o` node. A composite object within a given subdirectory D (e.g. "802") not only depends on D (arrow 4), but on all of D's sibling directories as well (arrow 8). This is strange, because these subdirectories contain driver code of different devices or subsystems. A second observation is that there are also dependencies from `__build` to all subdirectories (arrows 11 and 12).

The following note in the main Makefile gave us a hint: "We are using a recursive build, so we need to do a little thinking to get the ordering right.". Hence, just like composite objects the circular dependency chain (especially arrows 11 and 12) is a clever iteration strategy the Linux developers added to their recursive build to avoid problems with the evaluation order of targets [Mil97]. However, there is more. We tried to obtain a more natural build dependency graph by changing the Makefiles. Getting rid of arrow 8 was no problem, but required one extra "make" subprocess (following arrow 12). Additionally rerouting arrow 4 to `__build` could

compensate for this, but GNU Make is not expressive enough to express multi-target build rules with different dependencies per target. In other words, the idiom of Figure 9 is the actual engine of the 2.6 kernel build. Contrary to a normal "recursive make", it is a central, generic piece of build logic for which GNU Make's advanced features are exploited to their fullest. This does not suffice, so an equally fast, more complex compromise has been found (Figure 9).

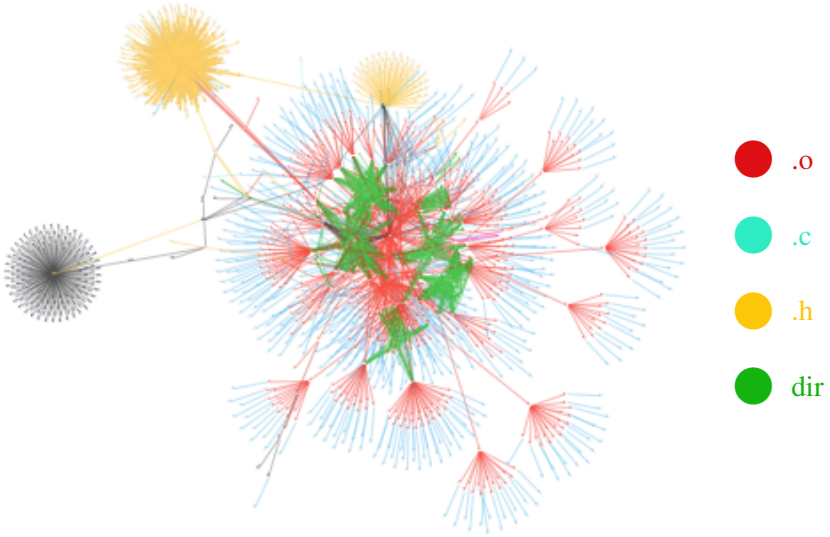

Figure 10: Phase `vmlinux` of the Linux 2.6.0 build process after filtering of complex idioms.

Figure 10 shows the resulting graph after filtering out the above idioms using logic rules. E.g. the circular dependency chain of Figure 9 has been hidden. The result resembles Figure 8's structure, which is plausible, as the basic directory structure has remained stable between 2.4.0 and 2.6.0, and each directory more or less corresponds to a composite object. Only the recursive process has changed in the meantime to produce a higher-quality build at the expense of extra complexity, both during the build as well as for the developer.

The 2.6.21.5 kernel continues down the path set out by 2.6.0. Some additions include 1835 extra dependencies on header files, of which 538 edges point to a common configuration header file, and 300 extra edges (compared to 2.6.0) have a header file as destination. Also, two extra types of intermediate (implicit) targets indicate small changes in build commands.

6 Future work

We consider future work of this topic to consist of three major tracks. For one, we should broaden the scope of the study by also looking at other real-world systems, both open- and closed-source. This is needed in order to check whether our observations hold for a larger class of software.

Also, it is important to stress that we limited ourselves to one representative kernel configuration which was continuously expanded if needed. As one of the reviewers noted, performing a configuration-aware study could learn us a lot too, i.e. determining configuration-(in)dependent parts of the build. The build could e.g. change more frequently in one of these parts than in the

other ones. In principle, MAKAO is armed to measure this kind of things. The same reviewer also suggested to relate our findings with changes registered inside the Linux source code repository, or e.g. with bugs submitted to the bug tracking system. Some general observations about the relationship between source and build commits have been made by Robles [Rob06], but more specific measurements for the Linux kernel could shed some new light.

Finally, there are two main questions which have been left open in this paper, and which we believe need answering: (1) what part do the configuration files and supporting scripts play in the evolution of this build system, and (2) to what degree can we show a causal link between the evolution of the source code and that of the build system? As a corollary: Can one claim that source code re-engineering approaches can only be effective if corresponding build system consequences have been taken into account?

7 Conclusion

This paper presented a case-study of the evolution of a real-world build system, namely that of the Linux kernel. We analysed its growth in number of source lines of code (SLOC), as well as the number of targets and dependencies (both explicit and implicit) which are part of the Makefiles. We also delved into some of the idioms which have shown up in recent kernel versions.

From this case study we can make the following observations: (1) the build system evolves, (2) as it evolves it grows in complexity, and (3) it is maintained in order to deal with this growing complexity. These observations are in line with Lehman's laws of software evolution. It also strengthens our conviction that in order to modify/re-engineer the source code one will have to modify/re-engineer the build system. That is, a part of the source code evolution bill implicitly is spent on evolving the build system.

Acknowledgements: The authors want to thank Kai Germaschewski, Sam Ravnborg and the anonymous reviewers for their suggestions. Bram Adams is supported by a BOF grant from Ghent University. Kris De Schutter received support from the Belgian research project AspectLab, sponsored by the IWT, Flanders. We would also like to thank Dieter Paeps for his help with some of the experiments.

Bibliography

[Ada06] E. Adar. GUESS: a language and interface for graph exploration. In *CHI '06: Proceedings of the 2006 Conference on Human Factors in Computing Systems*. Pp. 791–800. Montréal, Québec, Canada, April 2006.

[ADTD07] B. Adams, K. De Schutter, H. Tromp, W. De Meuter. Design recovery and maintenance of build systems. In *ICSM '07: Proceedings of the 23rd International Conference on Software Maintenance*. Paris, France, October 2007.

[Fel79] S. I. Feldman. Make - A Program for Maintaining Computer Programs. *Software - Practice and Experience*, 1979.

[GT00] M. W. Godfrey, Q. Tu. Evolution in Open Source Software: A Case Study. In *ICSM '00: Proceedings of the International Conference on Software Maintenance*. P. 131. IEEE Computer Society, Washington, DC, USA, 2000.

[LB85] M. M. Lehman, L. A. Belady (eds.). *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[lina] Linux kernel build documentation. Linux 2.6.0 edition.

[linb] Linux-kbuild mailing list. http://www.torque.net/kbuild/archive/.

[linc] Kbuild 2.5 history. http://kbuild.sourceforge.net/.

[Mec04] R. Mecklenburg. *Managing Projects with GNU Make*. O'Reilly Media, Inc., third edition edition, 2004.

[Mil97] P. Miller. Recursive make considered harmful. *Australian UNIX and Open Systems User Group Newsletter* 19(1):14–25, 1997.

[Rob06] G. Robles. *Software Engineering Research on Libre Software: Data Sources, Methodologies and Results*. PhD thesis, Universidad Rey Juan Carlos, February 2006.