A Study of Cloning in the Linux SCSI Drivers

Wei Wang and Michael W. Godfrey
David R. Cheriton School of Computer Science
University of Waterloo, Waterloo, ON, CANADA
Email: {`w65wang, migod`}`@uwaterloo.ca`

***Abstract*—To date, most research on software code cloning has concentrated on detection and analysis techniques and their evaluation, and most empirical studies of cloning have investigated cloning within single system versions. In this paper, we present the results of a longitudinal study of cloning among the SCSI drivers for the Linux operating system that spans 16 years of evolution. We have chosen the SCSI driver subsystem as a test subject as it is known that cloning has been embraced by these developers as a design practice: when a new SCSI card comes out that is similar to an old one, but different enough to warrant its own implementation, a new driver may be cloned from an existing one. We discuss the results of our qualitative and quantitative analyses, including how the layered architecture of the SCSI subsystem seems to have affected the use of cloning as a design tool, the likelihood of consistent and inconsistent change over time, and the predictive power of using cloning between two independent driver implementations to model the similarity between two target devices.**

I. Introduction

Code clones — that is, snippets of code that are similar to each other according to some definition of similarity[1] — are commonly found in both industrial and open source software. Clones are most often the result of copy-paste actions on the part of the developers. Cloning allows developers to reuse existing code-level designs within a new context rather than having to develop similar solutions from scratch. This kind of reuse is especially attractive when a complex design space has been conquered once, and a similar space beckons.

However, the benefits of cloning come at a cost. Careless use of cloning may negatively impact the code quality of the system, especially with respect to maintainability. Along with desired functionality, bugs may also be migrated into the new space. And if bugs are eventually found, the resulting fixes may not be applied to all of the clones. The more complex the code, the harder is it to maintain these clones in parallel.

Many approaches of automated clone detections have been proposed and evaluated, including sequence-based techniques, graph- and tree-based techniques, hashing- and metrics-based techniques, as well as hybrid methods. A typical clone detector usually generates an internal representation of the source code first and performs similarity comparison based on the internal representation. Previous case studies have shown that it is common for 5–20% of the source code base to have some cloning relationship with code elsewhere in the system.

[1]This somewhat facetious definition is due to Ira Baxter.

However, reducing the study of code cloning to what is effectively an algorithms problem misses much of the rich context in which cloning is performed in practice. To understand the motivation, risks, and potential benefits of cloning requires that we study the background of the systems being examined; it is our experience that the project history, its organizational structure, the software development practices used, and the problem domain all bear strongly on how and why cloning may be employed. Previous work along these lines has included examining the perceived motivation and cause of cloning [3] [11], the construction of taxonomies of clone patterns [10] [9], the relationship between software maintainability and cloning [16], and the impact of clones to software quality [19] [17].

This paper presents a longitudinal case study of cloning as a development practice within the SCSI (Small Computer Systems Interface) drivers subsystem of the Linux kernel. Architecturally, the SCSI subsystem is organized as a three level design, with infrastructure provided by the top two levels, and the lower level consisting of a large collection of hardware-specific low-level drivers that implement the same basic functionality. We investigate the relationship of clones within and across the different architectural layers of the subsystem, and we examine how clones between low-level drivers can be used to predict other kinds of observable similarity, such as hardware features.

Specifically, this paper addresses three research questions:

RQ1 *Do clones within the three architectural levels differ quantitatively?*

RQ2 *How do clones in the SCSI subsystem evolve between versions and over time?*

RQ3 *Can cloning be used as an effective predictor of hardware similarity?*

To measure cloning within the various architectural levels, we consider metrics such as absolute size of clones and clone coverage rates. To study the evolution of clones, we examine the ratio of clones over time, the lifespan of clones, and the kinds of changes encountered by clone classes over time. And to study the predictive power of cloning, we built three models — one based on cloning, one based on hardware vendor, and a random model — and evaluated how effective each was at predicting bus architecture compatibility.

There are three main contributions in this paper:

1) We study how the software architecture of a system can affect and be affected by the practice of cloning.
2) We present a longitudinal empirical study of cloning in a large open source system, where cloning is known to have been used as an intentional development strategy.
3) We present a study of how studying cloning can be used as a proxy for determining hardware feature similarity among files that implement a similar interface.

This paper is organized as follows: Section II presents related work on empirical cloning research. In Section III, we introduce the system of study, the Linux SCSI drivers subsystems, and we discuss our methodology and assumptions. Sections IV, V, and VI examine each of the research questions in turn. Finally, Section VII summarizes the work presented here.

II. Related work

Early research on source code cloning focused mainly on developing efficient and scalable clone detection algorithms, while empirical studies tended to focus on exploring cloning within single versions of systems. More recently, studies have started to explore how clones evolve over time within a system. For example, Kim et al. extracted clone genealogies by mapping text location and similarity between two versions and observed that most clones in their target systems are volatile [12]. Krinke studied the consistency of changes between clones as they evolve [13]. Bettenburg et al. examined change consistency of clones at the release level. [2]. Göde et al. designed a clone detector that incrementally identifies clones by using detection result of previous revision's analysis result and use it to model clone evolution in a more fine-grained fashion [4], [5].

There has also been work studying how and why cloning is used as a software development practice. Cordy's on-site study of a financial system led to the observation that cloning is often used as a development tool when a solution to a related problem already exists; the use of well-tested code as a starting point helps to mitigate risk of errors [3]. Kim et al. presented an ethnographic study observing developers' activity with an instrumented IDE and constructed a taxonomy of usage pattern of clones [11]. Kapser et al. proposed a taxonomy of likely intentions behind acts of cloning based on their empirical studies [10].

Antoniol et al. discussed some of the first results on clone evolution by examining the overall cloning ratio over time [1]. Li et al. also studied the clone coverage rate of the Linux kernel over 10 years and observed a steady growth of the clone coverage rate over time [14].

While it is widely believed that the practice of cloning can be detrimental to the quality of a system over time, empirical studies have not always supported this belief. Rahman et al. used the bug repositories of four large and well known open source projects, but found no association between error proneness and cloning [17]. Contrarily, a fine-grained analysis conducted by Selim et al. examined various features of cloned code to model the correlation relationship between cloning and software defects, and found that the error-proneness of clones is dependent on the subject systems [19].

Cloning within Linux device drivers was first examined by Godfrey et al. who presented a qualitative analysis of the Linux kernel and described several potential scenarios where cloning could be used as a design tool in developing Linux hardware drivers, and specifically among SCSI drivers [7]. More recently, Li et al. noticed a high clone ratio in Linux device drivers — especially within the SCSI drivers — when evaluating a clone detector targeting the Linux kernel [14]. They also suggested that the large number of similar drivers could be the reason for the high clone ratio.

III. Methodology

In this work, we have studied the subsystem of SCSI drivers of the Linux operating system. unlike some previous study, We have performed an architectural analysis, and measured the growth of the various components over time. We have also examined how cloning has been used as a development technique over time. We now provide some background information on the subject system, and discuss our methodology for performing these studies.

A. Subject system: Linux SCSI drivers

As mentioned above, the SCSI (Small Computer Systems Interface) subsystem is a major class of device drivers of the Linux operation system. It provides support for peripheral devices that support the SCSI interface, such as high-performance hard drives. The code for the SCSI subsystem can be found in the `drivers/scsi` directory of the source code distribution. SCSI support has been present in Linux since the first official release of the kernel in 1994. Like the rest of the kernel, the SCSI drivers are open source; however, most of the low-level drivers have been developed and then donated by the companies that developed the SCSI cards themselves. That is, unlike much of the core of the Linux kernel, the original development of most of the low-level driver code was performed within a company by full-time developers, and not by the open source community.

To perform a longitudinal study of clones in the SCSI subsystem, we analyzed the source code of the kernel starting with version 1.0 (which was released in March 1994) and picking one version every six months ending at version 2.6.32.15 (which was released in June 2010). In total, we analyzed 31 versions over the 16 year history of the Linux kernel.

One noteworthy feature of the SCSI subsystem is that it is designed and implemented as a strict three-level architecture: the top and middle levels provide a set of unified and consistent commands for the kernel to control various drivers,

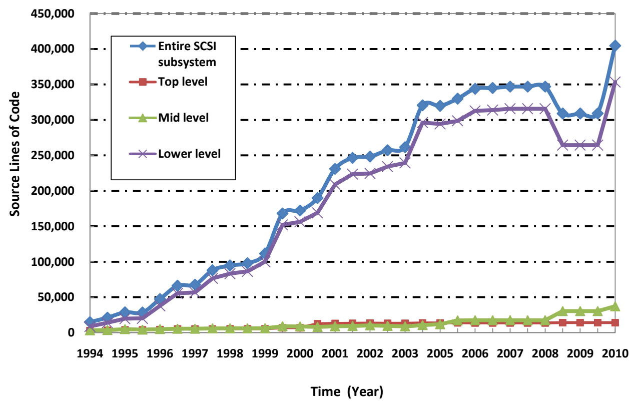

Figure 1. Growth in the number of Source Lines of Code (SLOC) for the SCSI subsystem as a whole as well as for each of the three architectural levels over all 31 snapshots, as measured by the open source tool *cloc*.

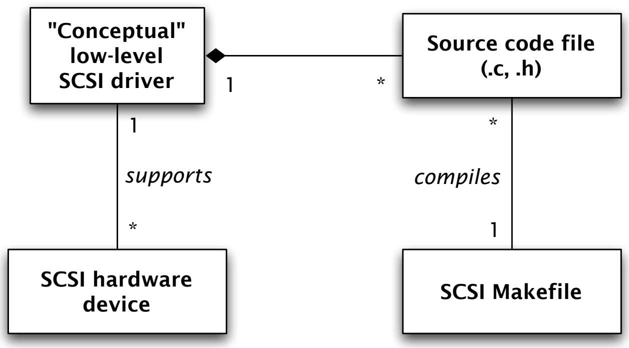

Figure 2. This UML class diagram illustrates the usual relationship between a SCSI device, its conceptual driver, and its source files. The composition relationship indicated by the solid diamond is slightly inaccurate as there are a few `.h` files that are shared among multiple conceptual drivers.

while the low-level drivers are concrete implementations of this functionality peculiar to the specifics of the particular hardware. Abstractly, the low-level drivers all implement the same functionality, such as hardware initialization, I/O handling, and error checking; they interact with the rest of the operating system only through the upper two levels.

We distinguish between "conceptual" low-level drivers and the files that implement them as follows: A conceptual low-level driver may provide support for a number of SCSI cards that share a similar hardware design. All drivers consist of at least two files: an implementation file (a "`.c`" file) and its associated header file (a "`.h`" file). However, sometimes a single conceptual driver will span multiple implementation files, depending on the practices of the development team. These relationships are illustrated in Figure 2. Additionally, there are some `.h` files that are used by multiple conceptual drivers, due to the strong similarity between the cards.

In the most recent version we studied, there are 96 conceptual drivers comprising over 319,000 SLOC[2] across 549 files. If we ignore shared `.h` files, we find that approximately 25% of the conceptual drivers are contained within a single source code file, about 50% are spread across two files, and the remaining 25% are spread across more than two files.

Since open source development allows for the reuse of existing code, it is not surprising that developers may choose to clone and then modify an existing low-level driver files when the underlying hardware of a new card is similar to an older one for which a driver already exists. Sometimes this cloning is explicitly mentioned in comments or documentation. For example, consider this extract from the header comment for the Always IN-2000 card driver (file `in2000.c`, version 1.3.97 or later):

I should also mention the driver written by Hamish Macdonald for the Amiga A2091 card [...] It gives me a good initial understanding of the proper way to run a WD33C93 chip, and I ended up stealing lots of code from it.

The driver for the Amiga A2091 card driver (`wd33c93.c`) was first included in the Linux kernel in the version 1.3.94 (released in April, 1996), while the driver for the Always IN-2000 card first appeared in version 1.1.67 (released in November, 1994) but was later completely rewritten by another developer in version 1.3.97. Looking at the clone analysis results, it seems clear that the new developer for IN-2000 driver made significant use of the code for the Amiga A2091 driver. We found 42 code snippets in `in2000.c` that appeared to be "stolen from" the file `wd33c93.c`; these snippets comprised over a third of the code in the new driver (838 SLOC of cloned code out of a total of 2375 SLOC in the driver).

For each file of each version, we categorized it as belonging to one of the three levels using available evidence such as comments, documentation, static analysis results, configuration information, and build recipes.[3] Then for each of the 31 versions in our sample, we measured the size of each level as well as the SCSI subsystem as a whole in SLOC. Figure 1 shows the growth of the three levels and the SCSI subsystem as a whole over time.

We can see that initially the three levels are of comparable size; in version 1.0 the upper two levels comprise about 6200 SLOC, while the lowest level comprises about 8700 SLOC. However, over time the top two levels have shown relatively little growth while the lowest level has grown almost 50-fold, making up about 95% of the SCSI subsystem's 400,000 SLOC in the most recent version. Prima facie, these trends suggest that the SCSI subsystem is well engineered: The

[2]Source Lines of Code (SLOC) counts all physical lines that are not entirely comprised of white space or comments. We used the open source tool *cloc* to perform our measurements.

[3]A detailed model of which file belongs to which layer can be found at `http://swag.uwaterloo.ca/~w65wang/LinuxSCSI`

infrastructure has remained very stable over time, while the number of distinct "users" (concrete drivers) has shown steady and strong growth [8] [18].

B. *Clone detection*

A clone detector automates the process of locating snippets of code with high similarity to each other. For this study, we have used the token-based incremental clone detection tool *iClones* from the University of Bremen [5]; its support for incremental detection made it particularly attractive to us for doing an evolutionary study.

Essentially, *iClones* takes multiple versions of a software system to be analyzed and computes the difference between each successive pair. Only the earliest version is processed with a conventional clone detection approach; clones in later versions are detected by textual differencing, combined with the detection results of the previous version.

Unsurprisingly, incremental detection using textual diffs is much faster than the naive approach of performing a clone detection across multiple versions of the full system's source code. Incremental detection has the additional significant benefit that it can track clone classes across versions automatically, since this is a simple side effect of the detection technique; to do so using the naive approach is likely to be significantly more effort and much less accurate.

In this paper, we used a threshold of 50 tokens as the minimum length for duplicated code to be considered as clones. The output of *iClones* detector reports a list of *clone classes* and each clone class contains a set of all *clone fragments* sharing the same duplicated code snippet.

C. *Results*

In the next three sections, we discuss the results of our study and examine each of the research questions in turn. We examine cloning within and across architectural levels over time, consistency of change between versions of clones, and cloning as a measure of feature similarity for low-level drivers.

IV. Cloning and Architecture

In this section, we address the first of our research questions:

RQ1 *Do clones within the three architectural levels differ quantitatively?*

We used the *iClones* tool to performed a clone analysis on all 31 versions of the SCSI subsystem in our data set; this data set consists of snapshots at six month intervals starting at version 1.0 in 1994 and continuing to the present. We also mapped each file and its constituent clones into its appropriate level within the three-level architecture of the SCSI subsystem, as described above.

We wanted to track the amount of cloned code versus non-cloned code within each of the architectural levels. However, simply summing the LOC in each of the clone classes would give a misleading view, as snippets from different clone classes may physically overlap, and we did not want to count any given LOC more than once. Consequently, for each line of code in each file, we categorized it as either "clone" or "not a clone", based on whether it belonged to a snippet in any clone class.

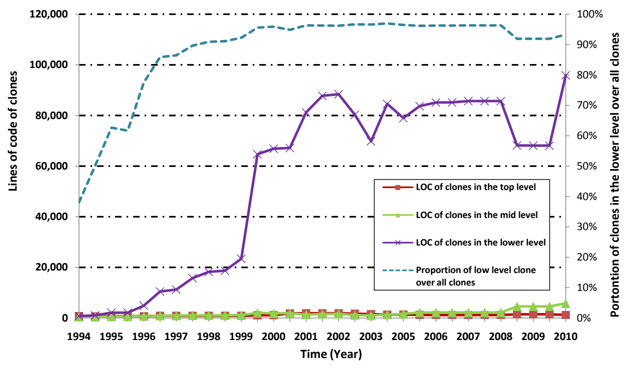

Figure 3. Absolute clone size in LOC for each level in the SCSI subsystem (solid lines, left y-axis), and the proportion of clones in the lower level over all clones (dashed line, right y-axis).

A. *Growth of clones in size of the SCSI subsystem*

The three solid lines in Fig. 3 shows (along the left y-axis scale) the absolute size of cloned code in each of the three architectural layers. It can be seen that the low-level drivers — which comprise most of the source code in the subsystem anyway — also comprise almost all of the cloned code in absolute terms. There are three dips in the absolute amount of clones in the lowest level; manual examination suggests that these dips were due to a number of deprecated drivers having been removed from the code base, rather than any large scale refactoring effort.

The dashed line in Fig. 3 shows (along the right y-axis scale) the percentage of clones in the system that reside in the lowest level. It can be seen that after the first four years, 90% or more of the clones in the system reside in the lowest level. There is a mild dip over the last few years when there was a decrease in the amount of cloned code in the lowest level, combined with a mild increase in cloning in the other two levels.

B. *Clone coverage rate of the SCSI subsystem*

To get a more nuanced view of cloning within each of the architectural levels, we also computed the *clone coverage* rate, which is simply the proportion of clone code versus all code. That is, if a file (or a set of files) has a clone coverage rate of 0.2, then 20% of the lines of code in that file belong to a snippet that is a member of one or more clone classes.

Figure 4 shows that in early snapshots of the SCSI subsystem the clone coverage rates for all but the top level are relatively low; furthermore, and somewhat surprisingly,

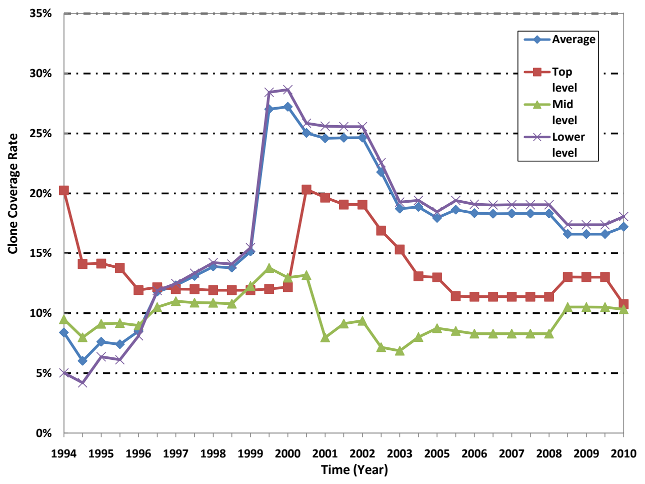

Figure 4. The clone coverage rates over time for each of the three architectural levels as well as the SCSI subsystem as a whole; this measures the proportion of code that is or has been cloned.

it is the low-level drivers that have the lowest clone coverage rate in this period. Over the long term, the top two levels have tended toward a rate of near 10%, although the top level has a large spike around 2000, which is immediately followed by a dip in the mid-level coverage rate.

The overall clone coverage rate has risen from just 6.0% to a maximum of 27.2% around 1999, and then fallen to just under 20% in recent years. It can be seen that the overall clone coverage rate closely matches that of the low-level drivers; this is because the low-level drivers come to overwhelmingly dominate the SCSI subsystem in terms of LOC, as Fig. 1 shows.

C. *Discussion*

We expected to see a significantly higher clone coverage rate among the low-level drivers compared to the rest of the system, since we had previously observed strong anecdotal evidence that cloning is used as a development strategy by SCSI driver developers [7]. We were surprised to find that, over the last several years, the clone coverage rate of the low-level drivers is almost double that of the two upper levels. A non-parametric Mann-Whitney U test revealed that the difference of the clone coverage rate results between the lower level and rest of the system is statistically significant with a confidence interval of over 99.99%.

Antoniol et al. studied clone ratios within and across subcomponents of the Linux kernel [1]. They found that clones were rarely scattered across subcomponents. In our study focusing on a single large subsystem of the Linux kernel, we also observed no clones spanning different architectural levels.

Li et al. analyzed clones in the Linux kernel using data mining techniques, and suggested that SCSI concrete drivers for similar devices could be one of the major sources of high clone coverage rate of the drivers top-level directory over the rest of the Linux kernel modules [14], to some degree echoing earlier conjectures of Godfrey et al. [7]. Our analysis here — using an entirely different clone detection technique from that of Li et al. — supports the hypothesis that low-level drivers contribute most of the clones, at least in the SCSI subsystem.

V. Clone Evolution

In this section, we address the second of our research questions:

RQ2 *How do clones in the SCSI subsystem evolve between versions and over time?*

After studying how the clone class evolves as a whole, we also examine the evolution of the fragments contained by the various clone classes.

A. *Change consistency of clones*

The *iClones* tool captures clone evolution over time by tracking code fragments over consecutive versions of the source code [4] [5]. We distinguish between four categories of clone classes:

Newly created — none of the fragments of a clone class has a predecessor in the previous system snapshot.

Deleted — none of the fragments of a clone class has a successor in the following system snapshot.

Consistently changed — all fragments of a clone class change consistently from the previous snapshot to the next.

Inconsistently changed — at least one fragment of a clone class was modified in a way that is inconsistent with some other fragments in that clone class.

Note that we ignore as uninteresting the case where a clone class is unchanged from one version to the next.

Two observations can immediately be drawn from Figure 5: First, inconsistently changed clone classes always outnumber consistently changed ones for all snapshots in our study where change was observed between versions. Second, new clones are being added to the SCSI subsystem continually over the time. The unusual rise in deleted clones observed in December 2008 (version 2.6.27.10) is due largely to the drop in the size of the subsystem, as can be observed in the dashed line plotted against the right y-axis.

B. *Clone lifespan*

To study the lifespan of code snippets of clones, we tracked all clone fragment traces over all 31 snapshots, and then calculated the absolute age of each fragment trace. Clone fragments traces are tracked across versions by using information such as content of source code, location and bourndary of each clone fragment; the tracing process is automated by the *iClones* detector. Implementation details can be found in Note that a significant proportion of clone fragments have survived over the entire period of all

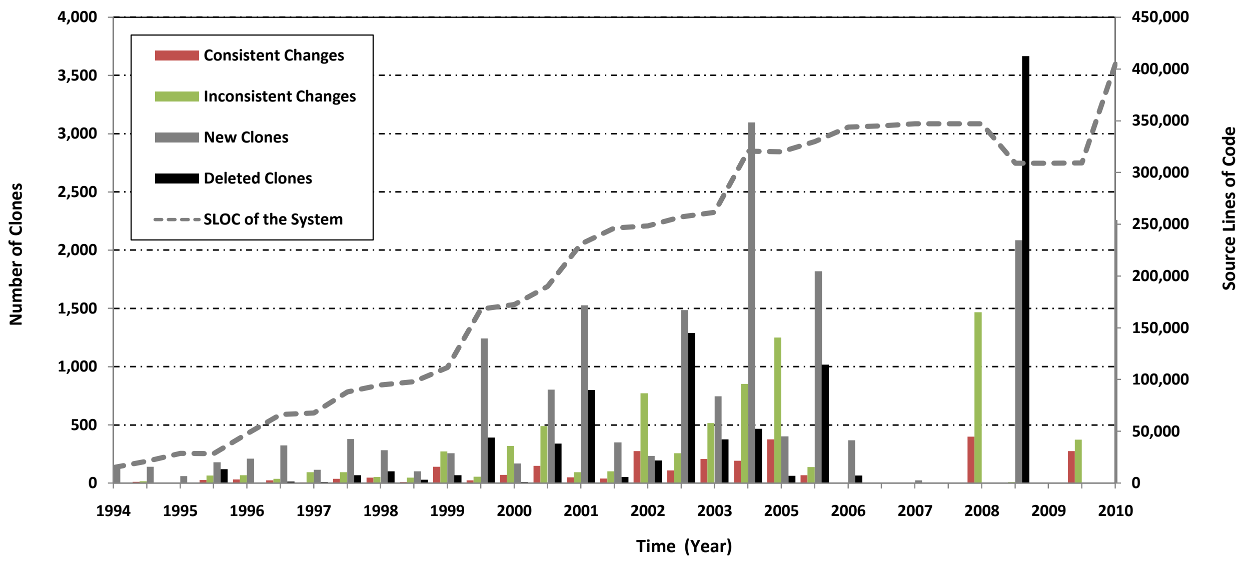

Figure 5. Clone change consistency and created/deleted clones for each version. The left y-axis represents the absolute number of clones and the right y-axis represents source code size in SLOC. Clones change during the entire evolution process we have studied. Inconsistent changes outnumber consistent changes for all versions of the SCSI subsystem we have studied.

snapshots we collected; in particular, recently created clone fragments are highly likely to persist in the most recent subsystem snapshot. For example, for a clone fragment first observed at version 2.6.27.44 (released in January 2010) and still alive at the latest snapshot we collected (version 2.6.32.15), it would be unrealistic to treat this fragment as if it had lifespan of only six months. It is also undesirable to exclude such survived clone fragments because, as shown in Table II, over 30% of all clone fragments belong to "surviving" clones. To address this problem, we use the Kaplan-Meier estimator, where these fragments are called "censored observations" in survival analysis theories. The Kaplan-Meier estimator takes into account both censored and uncensored data instances.

Given a data-set of lifespan of clone fragments, let $t_i, i = 1, \ldots, I$ denote the ordered failure or censoring times, let d_i denote the number of clone fragments that disappear at time i, and let n_i denote the number of clone fragments (including censored fragments) that are still alive at time i. The survival function *S(t)* for the Kaplan-Meier estimator at time t is:

$$S(t) = \prod_{t_i \leq t} \frac{n_i - d_i}{n_i} \tag{1}$$

We calculated $S(t)$ for every 6 months for all clone fragment traces; the survival rate for each level of the SCSI subsystem is shown in Figure 6. Overall, we can see that clone fragments in the top-level have a shorter lifespan than those in the mid- and lower-level. The lifespans of clones in the latter two levels are similar to each other for clones up to about the age of four years; after that, clones in the lower-level have a much stronger survival rate. We note that even after 180 months, there are still a non-trivial number of clones in the lower-level that have survived since the very first version.

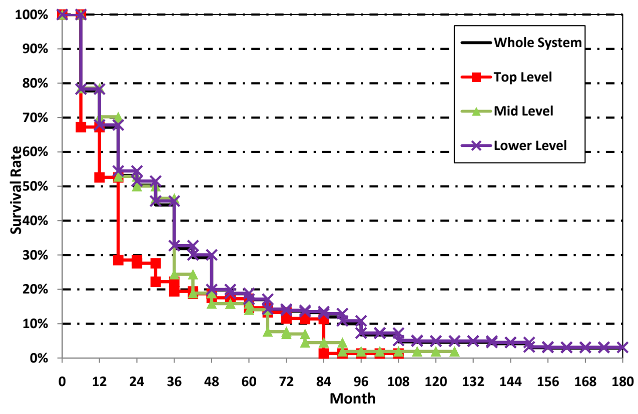

Figure 6. Kaplan-Meier survival curve for all clone fragments in the SCSI subsystem over all the snapshots we collected

Level	Censored	Died	Total
Top	344	1,951	2,295
Mid	682	1,104	1,786
Lower	12,898	34,278	44,121
Total	13,924	34,278	48,202

Table II
CLONE FRAGMENTS FOR THREE LEVELS OF THE SCSI SUBSYSTEM.

Level	Unchanged	Consistent Changes	Inconsistent Changes	Consistent and Inconsistent Changes	Total
Upper	518	161	632	65	1,376
Mid	641	87	493	19	1,240
Lower	16,121	2,933	10,490	503	30,047
Total	17,280	3,181	11,615	587	32,663

Table I
CLONE FRAGMENT MODIFICATIONS

The median lifespan of all clone fragments can be measured by solving the equation $S(t) = 0.5$ or its approximate value when no survival function is available. The median lifespan of all clone fragments is 30 months, and for the top-, mid-, and lower-level the median lifespans are 18, 30, and 30 months respectively.

As discussed above, naively calculating the actual arithmetic mean or median value of all clone fragments, including the right censored ones, will unfairly underestimate the lifespan of all clone fragments. In our study, the naive arithmetic mean lifespan of all clone fragments is 23 months while the median is 15 months, which is much small than the median average lifespan measured by the Kaplan-Meier estimator.

C. Discussion

To study evolution of clones in the SCSI subsystem, we examined the proportion of different kinds of changes for each clone class of each version, backtracked all clones in our latest version, and calculated the lifespan of clones using a statistical survival model.

One might expect that the upper- and mid-level would be less volatile than the lower-level, since changes in the higher levels might induce collateral changes to lower-level drivers. However, we have found no empirical evidence to support this notion. The proportion of unchanged clone fragments from the upper- and mid-level is not significantly lower than that in the lower levels. Furthermore, the median lifespan of clone fragments in the upper-level is even shorter than the overall median lifespan of all clone fragments.

Despite many differences in the systems being studied and other experiment configurations, we consider that it is useful to compare our results with findings from similar empirical studies on clone evolution. Kim et al. maps clone classes in different releases using location and textual comparisons [12]. They studied two systems; in the first, 37% of all genealogies lasted 41.7 days on average, and in the second 41% genealogies lasted 11.05 days on average. We note that the lifespan for both subject systems are shorter than observed in our system. We see two possible explanations for this. First, the systems studied may simply have different development patterns; second, we note that we study the lifespan of fragments while they studied the lifespan of clone classes.

Bettenburg reports a change consistency study on two open source projects — Apache Mina and JEdit — and found that the average lifespan for clone class genealogies are 4.59 releases (about 7.8 months) for Apache Mina and 9 releases (about 10.8 months) for JEdit [2]. This number is much smaller than the clone fragment lifespan observed in our study. Again, it could be caused by the nature of different subject software systems and different ways of estimating lifespan information.

Göde et al. also reported a study on the evolution of type-I clones of a large system, Bauhaus, mainly written in Ada [4][5][6]. Our findings about change consistency in the Linux SCSI subsystem supports their conclusion that inconsistent changes comprise the majority of all clone class changes. With respect to the lifespan of clone fragments, they found the median age for clone fragments in their subject system to be 55 weeks. However, since they consider right censoring cases as indicating end-of-life, the median lifespan calculated using the Kaplan-Meier estimator might be higher.

VI. CLONING AS A PREDICTOR OF HARDWARE SIMILARITY

In this section, we address the third and last of our research questions:

RQ3 *Can cloning be used as an effective predictor of hardware similarity?*

To evaluate this question, we built three models to predict bus type architecture compatibility: one based on cloning, one based on hardware vendors, and a random model. We now describe this work.

A. Background

In previous sections, we have shown that most clones in the SCSI subsystem are contributed by low-level drivers, and that the clone coverage rate within that architectural level is significantly higher than in the rest of the subsystem. While the Linux operating system supports a wide range of SCSI cards — there are over 90 "conceptual" drivers in the most recent version — each individual driver must still implement the same API within the three-level architecture specified by the Linux SCSI design documents. It is therefore not surprising that a developer creating a new driver might choose to copy and paste snippets of code from existing drivers, especially if there are commonalities in the underlying hardware, design philosophy, or if the concrete designs share cross-cutting concerns [11]. To find out how common this is — and specifically to find out how often

```
1  config SCSI_IN2000
2        tristate "Always IN2000 SCSI support"
3        depends on ISA && SCSI
4        help
5          This is support for an ISA bus SCSI host
6          adapter.  You'll find more
7          information in <file:Documentation
8         /scsi/in2000.txt>. If it doesn't work out of
9         the box, you may have to change the jumpers
10        for IRQ or address  selection.
11
12        To compile this driver as a module, choose M
13        here: the module will be called in2000.
```

Figure 7.   The compile configuration data for the Always IN-2000 device, as it appears in the file `drivers/scsi/Kconfig`

cloning seems to be driven by similar hardware features rather than, say, laziness — we built a cloning-based model within the low-level drivers to test its effectiveness as a predictor of hardware similarity

There are several dimensions along which one might organize a taxonomy of SCSI cards and their drivers, such as the generation of SCSI protocol being supported or the application domain of the device (e.g., storage device, CD-ROM drive, etc.). However, the availability and accuracy of information sources for many of these dimensions are problematic, especially for older devices. To overcome this, we use the computer bus type dependency extracted from the build configuration data (i.e., the "`Kconfig`" files) of the SCSI subsystem to measure the similarity between two drivers. Specifically, the `Kconfig` file shows the hard dependency of bus types for each driver (e.g., `ISA`, `PCI`, `EISA`) so that users can configure the build options to ensure the driver will run properly.[4]

The configuration data for all low-level drivers in the SCSI subsystem consists of at least three mandatory entries: *name*, *dependency*, and *help text*. Figure 7 shows the complete configuration for the Always IN-2000 card: Line 1 gives the name of the configuration option that corresponds to this driver as `SCSI_IN2000`, line 3 lists its required compilation dependencies, and lines 5–13 give the short, descriptive message intended to help the user understand what device support is being enabled by this option. The dependency entry lists the other configuration options that are required to be enabled to include support for the this card: `SCSI` and `ISA`.

There are several advantages of using the bus type dependency entry of SCSI card drivers as the variable to compare similarity:

1) Bus type dependency information is released with the `Kconfig` file of the Linux kernel to enforce correct configuration for a driver. That is, it is checked and maintained.
2) Bus type support is a fundamental feature of a hardware card, and driver developers must conform to a set of communication protocols determined by bus type support of the hardware. That is, it has essential semantic value to the driver.
3) Bus type dependency entries are represented as logical expressions which makes comparisons easier, automatable, and less error prone than natural language descriptions.

For these reasons, we consider using bus type dependency as the variable to compare similarity to be a reasonable choice.[5]

B. Methodology

To compare bus type dependency entries, we first convert each logical expression into a disjunctive normal form (DNF) formula. For a pair of DNF formulas, we define three levels of matching:

- Two entries are considered to be a *match* if there is at least one common conjunctive term that appears in both formulas.
- Two entries are considered to be a *partial match* if there is a conjunctive term in one formula can be expressed by concatenating extra variables in one conjunctive term in another formula.
- Two entries that are not a match or partial match are considered to be a *mismatch*.

For example, the DNF formula for bus type dependency for the IN-2000 card is "`ISA && SCSI`". This formula matches the formula "`(ISA && SCSI) ||(PCI && SCSI)`" because the formula for IN-2000 appears in one of the conjunctive term in the latter formula. It is also a partial match of DNF formula "`ISA && SCSI && PCI`" since the latter can be formed by concatenating extra variables onto the formula for the IN-2000. The formula "`SCSI && X86_32`" is a mismatch with IN-2000 because there is no way to make a match or a partial match. We built an automated tool to perform the matching evaluation.

After establishing a similarity metric for conceptual drivers based on bus type dependency, we decided to evaluate how strong an indicator cloning is for similarity in the underlying hardware. We built three models: one based on cloning information, one based on hardware vendor, and a random model. We then evaluated for each pair of entries in each model the strength of the bus type dependency match.

For the cloning-based model, we first mapped each low-level driver file to the conceptual driver that it implements, and then "lifted" the cloning information to the level of conceptual drivers. That is, suppose that we found a cloning relationship between two files, `alpha.c` and `bravo.c`, and that `alpha.c` and `bravo.c` are part of the Delta and Foxtrot conceptual drivers, respectively. Then we would

[4]`Kconfig` file dependency entries have previously been used to model features of the Linux kernel as a software product line [15] [20].

[5]We note that the `Kconfig` entries model other information also, but we have discarded all except the bus architecture data in our studies.

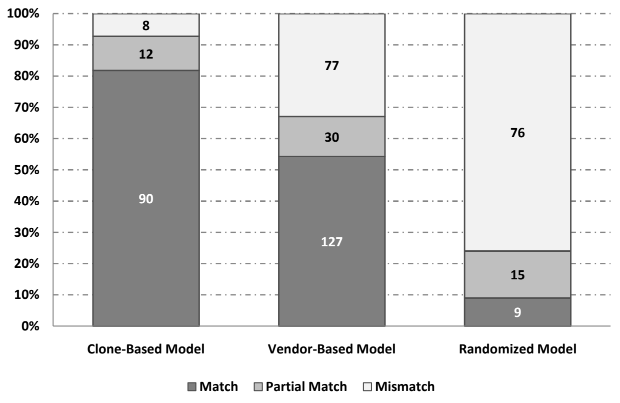

Figure 8. Accuracy of predicting bus type similarity using the cloning-based model, vendor name-based model, and a randomized model. The numbers on the bars show the absolute number of pairs in that category, while the y-axis shows the percentage.

infer a cloning relationship between Delta and Foxtrot. We then asked for each pair of conceptual drivers that have a cloning relationship: What is the strength of the match of the corresponding bus type dependency formula?

For the vendor-based model, we took pairs of cards from the same hardware vendor and evaluated the strength of the match of the corresponding bus type dependency formula. We took the vendor information from the `Kconfig` entry for each conceptual driver.

Finally, we created a randomized model to test if the results of the other models are meaningful. We generated this model by randomly selecting 100 pairs of drivers and comparing their bus type similarity. However, not all conceptual drivers specify their bus type information in the `Kconfig` file; consequently, to make the comparisons fairer we limited ourselves to those drivers that explicitly specified this information.

C. Discussion

As shown in Fig. 8, the cloning-based model has the highest accuracy rate of the three. Out of 110 pairs of conceptual drivers that share clone code snippets with each other, only 8 have a bus type mismatch while about 90 pairs match each other, and 12 are partial matches. We were surprised how much stronger these results were compared to the random model, where 76% of the pairs were a mismatch and only 9% were a match. This suggests that the existence of a cloning relationship between two conceptual drivers is a strong indicator of hardware similarity for bus architectures.

The cloning-based model also fared significantly better than the vendor-based model, which we found surprising. Although the vendor-based model performed much better than the random model, having the same vendor resulted in a bus architecture match only 54% of the time, with about a third being mismatches. This suggests that the vendor-based model is a good but not strong indicator of bus type compatibility.

Finally, we note that for the random model, 76% of the randomly chosen pairs were mismatches. This suggests that bus type compatibility is a non-trivial relationship, and that the results of the other two models are meaningful.

To summarize, we have presented strong evidence that cloning relationships are a good predictor of bus architectures in SCSI cards in the Linux kernel operating system. We feel that this opens up broader questions of the predictive power of cloning on other kinds of relationships, such as other kinds of similarity in the underlying hardware or feature sets of classes of software components that implement similar functionality; however, we do not address them here further.

VII. Conclusions

We now briefly discuss some threats to the validity of this work, and then summarize our results.

A. Threats to validity

Since the research community does not have an agreed-upon definition of just what a code clone is, any statistical results we present here will depend strongly on the choice of clone detection tool, the configuration and threshold settings we used, and our common sense. For example, when investigating RQ2 we let the *iClones* tool decide not only *"Are these two snippets clones?"* but also *"Have these two clones drifted too far apart?"* Consequently, direct quantitative comparisons to other empirical works is risky, while qualitative comparisons are less so. To allow other researchers to examine, replicate, and possibly extend our work, we have put our data files at a globally accessible URL: `http://swag.uwaterloo.ca/~w65wang/LinuxSCSI`

B. Summary

One of the main contributions of this paper is to demonstrate how we can combine cloning research with software architectural knowledge and evolutionary analyses to produce an informed history of a software system. Another key contribution is that we report on the accuracy of using cloning relationships between device drivers to predict the similarity of features in the underlying hardware.

More specifically, our study has addressed three research questions:

RQ1 *Do clones within the three architectural levels differ quantitatively?*

We found that both the absolute number and coverage rate of clones among low-level drivers are significantly higher than among files in the upper two architectural levels.

RQ2 *How do clones in the SCSI subsystem evolve between versions and over time?*

We observed many more inconsistent changes than consistent changes across clone classes over time. We also found that a spike in deleted clones was usually accompanied by a spike in new clones, suggesting that significant refactoring was occurring whenever old drivers were deprecated. And we found that new clones were continually being added to the system, even in the absence of observed refactoring.

We further used the Kaplan-Meier Estimator to study lifespan of all clones and found that the median lifespan of all clones is approximately 30 months.

RQ3 *Can cloning be used as an effective predictor of hardware similarity?*

We found that if two conceptual drivers had a cloning relationship, then there was an 82% chance that they had compatible bus architectures. This was a much better predictor of compatibility than having the same vendor or being chosen randomly; consequently, we conclude that cloning can sometimes be used as a predictor of hardware similarity. Of course, we have performed only one study measuring one possible dimension from one domain, so much work remains to be done to address the broader questions implied by RQ3.

Acknowledgements

We thank Rainer Koschke, Nils Göde, and Jan Harder for kindly providing the *iClones* incremental clone detector for our research. Nils Göde was particularly generous with his advice and time in helping us to configure *iClones*.

References

[1] G. Antoniol, U. Villano, E. Merlo, and M. D. Penta. Analyzing cloning evolution in the Linux kernel. *Information and Software Technology*, 44(13):755–765, 2002.

[2] N. Bettenburg, W. Shang, W. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan. An empirical study on inconsistent changes to code clones at release level. In *Proc. of the* 16th *Working Conference on Reverse Engineering*, WCRE '09, Antwerp, Belgium, Oct. 2009.

[3] J. R. Cordy. Comprehending reality—Practical barriers to industrial adoption of software maintenance automation. In *Proc. of the 2003 IEEE Intl. Workshop on Program Comprehension*, IWPC '03, Portland, OR, May 2003.

[4] N. Göde. Evolution of type-1 clones. In *Proc. of the 2009 Ninth IEEE Intl. Working Conference on Source Code Analysis and Manipulation*, SCAM '09, Edmonton, AB, Sept. 2009.

[5] N. Göde and R. Koschke. Incremental clone detection. In *Proc. of the 2009 European Conference on Software Maintenance and Reengineering*, CSMR '09, Kaiserslautern, Germany, February 2009.

[6] N. Göde and R. Koschke. Studying clone evolution using incremental clone detection. *Journal of Software Maintenance and Evolution: Research and Practice*, 23(3), April 2011.

[7] M. W. Godfrey, D. Svetinovic, and Q. Tu. Evolution, growth, and cloning in the Linux kernel: A case study. Presentation available at http://plg.uwaterloo.ca/migod/papers/2000/cascon00-linuxcloning.pdf, Oct. 2000.

[8] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Proc. of the 2000 IEEE Intl. Conference on Software Maintenance*, ICSM '00, San Jose, CA, Oct. 2000.

[9] C. Kapser and M. W. Godfrey. 'Cloning considered harmful' considered harmful. In *Proc. of the 13th Working Conference on Reverse Engineering*, WCRE '06, Benevento, Italy, Oct. 2006.

[10] C. J. Kapser and M. W. Godfrey. 'Cloning considered harmful' considered harmful: Patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692, December 2008.

[11] M. Kim, L. Bergman, T. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in OOPL. In *Proc. of the 2004 Intl. Symposium on Empirical Software Engineering*, ISESE '04, Redondo Beach, CA, Aug. 2004.

[12] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proc. of the* 10th *European Software Engineering Conference held jointly with* 13th *ACM SIGSOFT Intl. Symposium on Foundations of software engineering*, ESEC/FSE '13, Lisbon, Portugal, Sept. 2005.

[13] J. Krinke. A study of consistent and inconsistent changes to code clones. In *Proc. of the* 14th *Working Conference on Reverse Engineering*, WCRE '07, Vancouver, BC, Oct. 2007.

[14] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. on Software Engineering*, 32:176–192, 2006.

[15] R. Lotufo, S. She, T. Berger, A. Wasowski, and K. Czarnecki. Evolution of the Linux kernel variability model. In *Proc. of the* 14th *Software Product Line Conference*, SPLC '10, Jeju Island, South Korea, Sept. 2010.

[16] A. Monden, D. Nakae, T. Kamiya, S.-i. Sato, and K.-i. Matsumoto. Software quality analysis by code clones in industrial legacy software. In *Proc. of the 8th Intl. Symposium on Software Metrics*, METRICS '02, Ottawa, ON, June 2002.

[17] F. Rahman, C. Bird, and P. Devanbu. Clones: What *is* that smell? In *Proc. of the* 7th *IEEE Working Conference on Mining Software Repositories*, MSR '10, Cape Town, South Africa, May 2010.

[18] G. Robles, J. J. Amor, J. M. Gonzalez-Barahona, and I. Herraiz. Evolution and growth in large libre software projects. In *Proc. of the* 8th *Intl. Workshop on Principles of Software Evolution*, IWPSE '05, Lisbon, Portugal, Sept. 2005.

[19] G. M. Selim, L. Barbour, W. Shang, B. Adams, A. E. Hassan, and Y. Zou. Studying the impact of clones on software defects. In *Proc. of the* 17th *Working Conference on Reverse Engineering*, WCRE '10, Boston, MA, Oct. 2010.

[20] R. Tartler, J. Sincero, W. Schröder-Preikschat, and D. Lohmann. Dead or alive: Finding zombie features in the Linux kernel. In *Proc. of the* 1st *Intl. Workshop on Feature-Oriented Software Development*, FOSD '09, Denver, CO, Oct. 2009.