Analyzing Cloning Evolution in the Linux Kernel

G. Antoniol (antoniol@ieee.org)\textsuperscript{a}
U. Villano (villano@unisannio.it)\textsuperscript{a}
E. Merlo (ettore.merlo@polymtl.ca)\textsuperscript{b}
M. Di Penta (dipenta@unisannio.it)\textsuperscript{a}

\textsuperscript{a}University of Sannio, Department of Engineering - Benevento, Italy
\textsuperscript{b}École Polytechnique de Montréal - Montréal, Canada

Abstract

Identifying code duplication in large multi-platform software systems is a challenging problem. This is due to a variety of reasons including the presence of high-level programming languages and structures interleaved with hardware-dependent low-level resources and assembler code, the use of GUI-based configuration scripts generating commands to compile the system, and the extremely high number of possible different configurations.

This paper studies the extent and the evolution of code duplications in the Linux kernel. Linux is a large, multi-platform software system; it is based on the Open Source concept, and so there are no obstacles to discussing its implementation. In addition, it is decided by too large to be examined manually: the current Linux kernel release (2.4.18) is about three million LOCs.

Nineteen releases, from 2.4.0 to 2.4.18, were processed and analyzed, identifying code duplication among Linux subsystems by means of a metric-based approach. The obtained results support the hypothesis that the Linux system does not contain a relevant fraction of code duplication. Furthermore, code duplication tends to remain stable across releases, thus suggesting a fairly stable structure, evolving smoothly without any evidence of degradation.

Key words: clone detection, source code analysis, metric extraction

1 Introduction

Large multi-platform software systems are likely to encompass a variety of programming languages, coding styles, idioms and hardware-dependent code.
Analyzing multi-platform source code, however, is challenging. Assembler code is often mixed with high-level programming language. Furthermore, scripting languages, configuration files, and hardware specific resources are typically used.

Often, the system was originally conceived as a single platform application, with a limited number of functionalities and supported devices. Then, it evolved adding new functionalities and was ported on new product families: in other words, new devices and/or target platforms were added. When writing a device driver or porting an existing application to a new processor, developers may decide to copy an entire working subsystem and then modify the code to cope with the new hardware. This technique ensures that their work will not have any unplanned effect on the original piece of code they have just copy. However, this evolving practice promotes the appearing of duplicated code snippets, also said clones.

Indeed, only few papers have studied the evolution of similar code fragments among several versions of the same software system [18,19,22,23]. However, the information gathered accounts for local similarities and changes. As a result, the overall picture describing the macro system changes is difficult to obtain. Moreover, if chunks of code migrate via copy/remove or cut-and-paste among modules or subsystems, the duplicated code may not be easily distinguished from freshly developed one.

The goal of this paper is to study the evolution of the amount of cloned code in a large, multi-platform, multi-release software system. Intuitively, the larger the fraction of code fragments shared by two subsystems, the higher their similarity. Two completely different subsystems are likely to have a very low similarity and very little source code in common. Similarity between subsystems has been measured through the metric-based clone detection technique presented in [22] and evaluated by measurement of the common ratio (at function grain-level) between two subsystems proposed in [13].

Nineteen releases of a multi-million lines-of-code software, the Linux kernel (releases 2.4.0 through 2.4.18), have been used as case study. Linux is an open source UNIX-like operating system, created by Linus Torvalds with developers throughout the world. Originally, it was targeted to 32-bit x86-based PCs (386 or higher). Nowadays, the kernel 2.4.18 also runs on a variety of platforms including Compaq Alpha AXP, Sun SPARC and UltraSPARC, Motorola 68000,
PowerPC, ARM, Hitachi SuperH, IBM S/390, MIPS, HP PA-RISC, Intel IA-64 and DEC VAX. Port is currently in progress to the AMD x86-64 architecture.

The Linux kernel is almost entirely written in C language, with few assembler boot files (plus TCL/TK and Perl configuration scripts). The kernel configuration is controlled by macros and preprocessor switches (about 400). Macros allow to include/exclude kernel functionalities (e.g., math coprocessor emulation), specific device drivers (e.g., Adaptec AH 2940) and entire subsystems (e.g., ISDN), or to produce a module loadable at running time. We have parsed and analyzed the C source code of the Linux kernel, extracting a set of software metrics characterizing each function. Two or more code fragments (i.e., functions) were considered to be clones if the extracted metrics assume exactly the same values.

The evaluation of the cloning extent has been performed at different levels. Clones have been identified among top-level directories of the source tree, which essentially correspond to major subsystems. Furthermore, the same analysis has been performed between non top-level directories at the same nesting level of the source tree, i.e., within major subsystems.

In particular, the experimental activity we have carried out has addressed the following research questions:

- Which is the cloning extent within the Linux-2.4.x kernel major subsystems?
- Which is the cloning extent within the subsystems related to the different supported platforms?
- Is there a trend in cloning ratio when the system evolves?

This paper is organized as follows. First, the clone detection process is described in Section 2. Then, Section 3 presents the case study. The experimental results are reported and discussed in Section 4. Finally, related work is summarized in Section 5, while conclusions and work-in-progress are reported in Section 6.

2 The Clone Detection Process

The goal of this paper is to study the potential impact and the evolution of clones in terms of cloning ratio between different subsystems of a large multiprocessor software system. Clones are defined as code fragments indistinguishable under a given criterion. Different granularities may be considered when extracting clone information (e.g., compound statement or function body). In this paper, we focused our attention on function definitions. The process
defined to study clone evolution, outlined in Figure 1, relies on the concept of clone clusters. A cluster is a set of indistinguishable functions. The process consists of the following, subsequent phases:

(1) Handling of preprocessor directives;
(2) Function identification;
(3) Metrics extraction; and
(4) Cluster identification and computation of the cloning ratio.

![Diagram of the clone identification process.](image)

Fig. 1. The clone identification process.

Metrics extraction can be performed in a time linear in system size. However, since the metric extractor used was not optimized, the extraction of metrics for each Linux release required about one hour on a Pentium III (850Mhz 128 Mbytes RAM).

Once metrics were available, clone detection was performed. Clone detection (i.e., clustering) has $O(n^2)$ complexity, where $n$ is the number of functions. The entire process required about one day for all the nineteen Linux releases. The following subsections deal with the details of each phase.
2.1 Handling Preprocessor Directives

Parsing programming languages such as C or C++ poses several challenges. Besides the intrinsic programming language peculiarities (e.g., union, struct, classes, function pointers, etc.), preprocessor directives must be suitably handled. Preprocessing directives are usually managed by a dedicated compiler component, the *preprocessor* (e.g., the GNU *cpp*). Parsing multi-platform code where preprocessing directives are platform-dependent is equivalent to projecting the source code on a given hardware/software configuration.

To obtain information on several platforms, at least two approaches are feasible:

- Preprocess and parse the code sources with different configurations; or
- Adopt a fictitious reference configuration.

Unfortunately, for large size systems such as Linux, the first approach may not be realistic or feasible. For example, the Linux 2.4.0 kernel contains more than 7000 files, it runs on ten different processors. 400 preprocessor switches drive the actual kernel configuration. Each preprocessor switch can assume three values:

- **Y**: the code is included into the compiled kernel;
- **N** (or commented switch): the code is excluded; or
- **M**: a dynamically loadable module is produced.

Clearly, among the $10 \times 400^3$ possibilities there are many meaningless configurations (e.g., it is very unlikely that a machine has multiple different sound boards).

On the contrary, by defining a fictitious reference configuration, no specific architecture is identified. This approach is well suited for the identification of function clones among several platform-dependent sub-systems without re-compiling the kernel.

The heuristic adopted to handle preprocessor directives is based on the consideration that very often only the *then* part of an *#ifdef* is present; moreover, the *then* branch almost always contains more code than the *else* branch. Among the 3243 source (.c) files of the 2.4.0 kernel, 2172 contain at least one *#ifdef*, whereas only 1140 files have an *#else* preprocessor directive. The actual number of *#ifdef* is by far larger (22134) than the number of *#else* (8565), and, in terms of volume (measured in LOCs), the *then* branch is an order of magnitude larger (about 300 KLOCs versus 20 KLOCs).

Since preprocessor directives, i.e., *#ifdef*, must be balanced, a parsing of
the preprocessor statements can project the source on if branch; the \#ifdef
conditions were forced to be true, thus extracting the then branches. The
preprocessor elimination step generates sources with removed preprocessor
directives, regardless of the hardware/software architecture. Unfortunately,
there are few cases where this heuristic produces syntactically wrong C code.
Namely, a C scope (i.e., \{ ) may be opened in the then part of an \#ifdef
subject to condition EXP, and the scope end (}) be located in a different
preprocessor statement within the \#else part. This also means that the scope is
closed by a combination of expressions where EXP is negated. This is the
situation found, for example, in the Linux 2.4.0 ultrastor.c scsi driver. Due
to the very low number of such cases (in the 2.4.0 kernel, twenty on about
48000 functions), these were considered pathological situations, detected and
signaled for manual intervention.

2.2 Function Identification

Large C systems are likely to encompass a variety of mixed programming
styles, programming patterns, idioms, coding standard and naming conven-
tions. Most noticeably, both the ANSI-C and the old Kernighan & Ritchie
style may be present. A tool inspired by island-driven parsing has been imple-
mented to localize and extract function definitions. Once islands (e.g., function
bodies or signatures) were identified, the in-between code was scanned, and
the function definition extracted by means of a hand-coded parser.

2.3 Metrics Extraction

Following the approach proposed in [22], the functions extracted as illustrated
above were compared on the basis of software metrics accounting for layout,
size, control flow, function communication and coupling. In particular, each
function was modeled by 54 software metrics:

- The number of passed parameters;
- The number of LOCs;
- The number of statements;
- The cyclomatic complexity;
- The number of used/defined local variables;
- The number used/defined non-local variables;
- Software metrics accounting for the number of arithmetic and logic opera-
tors (++, --, >=, <, etc.);
- The numbers of function calls;
- The numbers of return/exit points;
- The numbers of structure/pointer access fields;
• The numbers of array accesses;
• Software metrics accounting for the number of language keywords (e.g.,
  while, if, do).

Different set of metrics could indeed be adopted (e.g., those used in [22]).
However, we experienced that, on sufficiently large systems, the use of different
sets of metrics does not significantly influence the results.

Differently from the procedure customarily followed in the past (e.g., in [22]),
function names and file/unit names were not used as metrics.

2.4 Clone Cluster Identification

Studying commonalities between software systems/sub-systems, function iden-
tity may be disregarded in favor of a different concept: clone clusters. A clone
cluster can be seen as a set of similar code fragments which contains identical
fragments or fragments exhibiting negligible differences from a given fragment
prototype. Each pair of functions was compared, and, exact metrics identity
was required to classify two functions as clones. This assumption corresponds
to the ExactCopy and DistinctName classes presented in [22].

Let \( M_f = \{m_1(f), \ldots, m_n(f)\} \) be the tuple of metrics characterizing a
function \( f \), where each \( m_i(f) \) \( (i = 1, \ldots, n) \) is the \( i \)-th software metric chosen to
describe \( f \) (e.g., number of passed parameters, number of LOCs, cyclomatic
complexity, number of used/defined local variables, and number used/defined
non local variables).

For any given function \( f \), let \( C_f \) be the \( f \)-th clone cluster. \( C_f \) is the subset of
function \( g \) belonging to the considered software system/sub-system \( S_k \), that
exhibits software metric values \( m_i(g) \) identical or similar to \( m_i(f) \):

\[
C_f \overset{\text{def}}{=} \{ g \mid g \in S_k \land m_i(f) \cong m_i(g), \; i = 1, \ldots, n, \; m_i(f) \in M_f \}
\]

This represents a necessary condition: the \( \cong \) operator was used to state that
\( g \) metric values, \( m_i(g) \), may be chosen to meet the specific goal. To identify
similar functions, a threshold may be adopted:

\[
m_i(f) \cong m_i(g) \Rightarrow (m_i(f) \leq m_i(g) \leq \theta_i) \land \theta_i \leq m_i(g) \leq m_i(f)), \; i = 1, \ldots, n \; m_i(f) \in M_f
\]
where $\theta_l(i)$ and $\theta_u(i)$ are the $i$-th lower/upper bounds. Inside this range of values, $g$ is considered to be a clone of $f$. Clearly, to collect exact, or nearly exact, function duplicates, $\bowtie$ is implemented by the equality operator.

2.5 Measurement of the Cloning Ratio

Given two different software systems, say $A$ and $B$, information about the cloning extent between such software systems can be measured in terms of common ratio. The Common Ratio (CR) between $A$ and $B$ is defined as the ratio of the number of functions belonging to $A$, having $|C_f| \neq 0$ when compared to functions in $B$, to the number of functions contained in $A$. In other words, it is the ratio of $A$ functions having clones in $B$ to $A$ size. It should be noted that, according to the definition above, and due to the possibly different number of functions in $A$ and $B$, the CR of $A$ to $B$ may be different from the CR of $B$ to $A$.

3 Case Study

Linux is a Unix-like operating system that was initially written as a hobby by a Finnish student, Linus Torvalds [25]. The first Linux version, 0.01, was released in 1991. Since then, the system has been developed by the cooperative effort of many people, collaborating over the Internet under the control of Torvalds. In 1994, version 1.0 of the Linux Kernel was released. The current version is 2.4, released in January 2001.

As far as code analysis, program understanding and reverse software engineering practices are concerned, the peculiar characteristics of the Linux kernel make it an ideal candidate as testbed for automated code examination and comprehension tools. It is based on the Open Source concept, and so there are no obstacles to discussing its implementation. It is not toy software, but one that is representative of real-world software systems. In addition, it is decidedly too large to be examined manually.

Unlike other Unixes (e.g., FreeBSD), Linux it is not directly related to the Unix family tree, in that its kernel was written from scratch, not by porting existing Unix source code. The very first version of Linux was targeted at the Intel 386 (i386) architecture. At the time the Linux project was started, the common belief of the research community was that high operating system portability could be achieved only by adopting a microkernel approach. The fact that now Linux, which relies on a traditional monolithic kernel, runs on a wide range of hardware platforms, including palmtops, Sparc, MIPS and
Alpha workstations, not to mention IBM mainframes, clearly points out that portability can also be obtained by the use of clever code structure.

<table>
<thead>
<tr>
<th>Release Series</th>
<th>Initial Initial</th>
<th>Number of Releases</th>
<th>Time to Start of Next Release Series</th>
<th>Duration of Series</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>9/17/91</td>
<td>2</td>
<td>2 months</td>
<td>2 months</td>
</tr>
<tr>
<td>0.1</td>
<td>12/3/91</td>
<td>85</td>
<td>27 months</td>
<td>27 months</td>
</tr>
<tr>
<td>1.0</td>
<td>3/13/94</td>
<td>9</td>
<td>1 month</td>
<td>12 months</td>
</tr>
<tr>
<td>1.1</td>
<td>4/6/94</td>
<td>96</td>
<td>11 months</td>
<td>11 months</td>
</tr>
<tr>
<td>1.2</td>
<td>3/7/95</td>
<td>13</td>
<td>6 months</td>
<td>14 months</td>
</tr>
<tr>
<td>1.3</td>
<td>6/12/95</td>
<td>115</td>
<td>12 months</td>
<td>12 months</td>
</tr>
<tr>
<td>2.0</td>
<td>6/9/96</td>
<td>34</td>
<td>24 months</td>
<td>32 months</td>
</tr>
<tr>
<td>2.1</td>
<td>9/30/96</td>
<td>141</td>
<td>29 months</td>
<td>29 months</td>
</tr>
<tr>
<td>2.2</td>
<td>1/26/99</td>
<td>19</td>
<td>6 months</td>
<td>still current</td>
</tr>
<tr>
<td>2.3</td>
<td>5/11/99</td>
<td>60</td>
<td>12 months</td>
<td>12 months</td>
</tr>
<tr>
<td>2.4</td>
<td>1/4/01</td>
<td>18</td>
<td>–</td>
<td>still current</td>
</tr>
<tr>
<td>2.5</td>
<td>22/11/01</td>
<td>8</td>
<td>–</td>
<td>still current</td>
</tr>
</tbody>
</table>

Table 1
Linux Kernels Most Important Events.

Linux is based on the Open Source concept: it is developed under the GNU General Public License and its source code is freely available to everyone. The most peculiar characteristic of Linux is that it is not an organizational project, in that it has been developed through the years thanks to the efforts of volunteers from all over the world, who contributed code, documentation and technical support. Linux has been produced through a software development effort consisting of more than 3000 developers distributed over 90 countries on five continents [24]. It should be noted that, due to the nature of the decentralized, voluntary basis development effort, no formalized development processes has been adopted, and thus it is worth investigating the quality characteristics of the resulting software.

A key point in Linux structure is modularity. Without modularity, it would be impossible to use the open-source development model, and to let lot of developers work in parallel. High modularity means that people can work co-operatively on the code without clashes. Possible code changes have an impact confined to the module into which they are contained, without affecting other modules. After the first successful portings of the initial i386 implementation, the Linux kernel architecture was redesigned, in order to have one common
code base that could simultaneously support a separate specific tree for any
number of different machine architectures.

The use of loadable kernel modules, which are dynamically loaded and linked
to the rest of the kernel at run-time, was introduced with the 2.0 kernel ver-
sion [14]. Kernel modules further enhanced modularity, providing an explicit
structure for writing hardware-specific code (e.g., device drivers). Besides mak-
ing the core kernel highly portable, the introduction of modules allowed a large
group of people to work simultaneously on the kernel without central control.
The kernel modules are a good way to let programmers work independently
on parts of the system that should be independent.

An important management decision was establishing, in 1994, a parallel re-
lease structure for the Linux kernel. Even-numbered releases were the develop-
ment versions on which people could experiment with new features. Once an
odd-numbered release series incorporated sufficient new features and became
sufficiently stable through bug fixes and patches, it would be renamed and re-
leased as the next higher even-numbered release series and the process would
begin again. The principal exception to this release policy has been the com-
plete replacement of the O.S. virtual memory system in the 2.4 version series
(i.e., within a stable release). The whole story, which has also led to the birth
of alternative kernel trees, is dealt with in [3,4]. At the time of writing (April,
2002), the latest kernel releases are 2.4.18 (stable) and 2.5.5 (experimental).

Linux kernel version 1.0, released in March 1994, had about 175,000 lines-of-
code. Linux version 2.0, released in June 1996, had about 780,000 lines-of-code.
Version 2.4, released in January 2001, has more than two millions lines-of-
code (MLOCs). The current 2.4.18 release is composed of about 14000 files;
it's size is about 3 MLOCs (.c and .h). Counting the LOCs contained in .c files
(i.e., excluding include files), its size is about 2.5 MLOCs (.c files only). The
architecture-specific code accounts for 422 KLOCs. In platform-independent
drivers (about 1800 files) there are about 1.6 MLOCs. The core kernel and file
systems contain 12 KLOCs and 235 KLOCs respectively.

Table 1, which is an updated version of the one published in [24], shows the
most important events in the Linux kernel development time table, along with
the number of releases produced for each development series.

4 The Linux Kernel Cloning Analysis

The results reported in this paper were computed using a slightly different
procedure as compared to the one followed in [13]. In [13], the clones were
identified considering all functions contained in the system regardless of func-
tion sizes (measured as the number of LOCs of the function body). Doing so, small functions (e.g., functions setting or getting the value of a structure) very often cluster together. However, it may be argued that these functions do not really represent clones, and thus that the resulting CR is biased by false positives.

To study the influence of short functions on CR, this index was computed for two different configurations. The first configuration corresponds to the assumptions made in [13]; namely, all functions, regardless of their size, were considered. In the second configuration, instead, all functions with a body shorter that five LOCs were discarded, detecting clone clusters and computing the CR only on the remainder.

Analyzing CRs on several Linux releases, we noticed that CRs among all possible combinations of Linux subsystems were often null or very low, thus leading to sparse cloning matrices. Furthermore, according to the definition of CR, high CR value does not necessarily imply high number of replicated code snippets. A 50% CR may correspond just to a couple of cloned functions, if small subsystems are considered. On the other hand, if the analyzed subsystems contain a high number of functions, say 1000, even a CR as low as 1% is worth to be considered. In the analysis that follows, we report results that were considered significant either in relative (e.g., high CR values) or in absolute terms (e.g., high number of cloned functions).

4.1 Kernel 2.4.18 Analysis

The experimental activity presented in this subsection was driven by the first two research questions specified in the Introduction, i.e., computing the cloning ratio among major Linux architectural components and the percentage of duplicate code among different supported platforms. However, in the authors’ knowledge, it does not exist any documentation of the Linux architecture, in that no document describes the system at a high level of abstraction. Bowman et al. derived both the conceptual architecture (the developers’ system view) and the concrete architecture (the implemented system structure) of the Linux kernel [6,10,8]. They started from a manual hierarchical decomposition of the system structure, consisting of the assignment of source files to subsystems, and of subsystems hierarchically to subsystems. As shown in [9], most of the times the extracted subsystems correspond to directories in the source code tree. For simplicity’s sake, in the analysis performed, it has been assumed that each directory of the source tree contains a subsystem (at a proper level of the system hierarchy). Thus the search for cloned code was performed by comparing the code contained in any two directories.
Figure 2 shows two different examples of function clones identified. The first clone pair (top of the figure), is an example of a function copied from mips to mips64 memory management subsystem. The second clone pair is instead a cross-system example: although the accessed data structure has different field names, the action actually performed is the same, i.e. the removal of an item from a concatenated list.

![Diagram showing two examples of clones found.](image)

Table 2 reports the CRs higher than 1% among Linux major subsystems (i.e., the twelve top-level directories, documentation and include directories excluded). CRs are reported along with the corresponding number of cloned functions, for both the considered configurations (i.e., functions longer than five LOCs, and all functions).

Observing Table 2, it can be easily recognized that:

- The table contains only seven rows, out of 144 possibilities; in other words, only very few subsystem comparisons gave raise to appreciable clone extents;
- The difference between the results obtained considering all functions and those obtained with a 5-LOCs threshold is relevant;
- Though CRs among major subsystems is not very high, even a small ratio (e.g. 1.43% between arch and drivers) corresponds to a non-negligible
<table>
<thead>
<tr>
<th>Subsystems Compared</th>
<th>Functions ≥ 5 LOCs</th>
<th>All Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Common Ratio</td>
<td>Functions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cloned</td>
</tr>
<tr>
<td>arch-drivers</td>
<td>1.43%</td>
<td>152</td>
</tr>
<tr>
<td>fs-drivers</td>
<td>2.06%</td>
<td>93</td>
</tr>
<tr>
<td>ipc-arch</td>
<td>1.45%</td>
<td>1</td>
</tr>
<tr>
<td>kernel-arch</td>
<td>2.11%</td>
<td>114</td>
</tr>
<tr>
<td>lib-arch</td>
<td>2.90%</td>
<td>9</td>
</tr>
<tr>
<td>lib-net</td>
<td>1.45%</td>
<td>4</td>
</tr>
<tr>
<td>mm-drivers</td>
<td>1.36%</td>
<td>18</td>
</tr>
</tbody>
</table>

Table 2
CRs ≥ 1% among major subsystems.

(152) number of cloned functions, as these subsystems are very large.

It is worth pointing out that in the two configurations CRs were computed considering the ratio to the total number of retained functions. This may lead to two counterintuitive phenomena: higher CR for functions ≥ 5 LOCs, and different CRs corresponding to the same number of cloned functions, because of the lower number of functions that are assumed to belong to the system.

A similar approach was followed to evaluate the cloning extents within the subsystems related to the different supported platforms. The arch directory contains fifteen sub-directories, each corresponding to a supported processor architecture (e.g., i386, s390, sparc). Each platform has, among others, its own kernel and memory management mm implementations. In particular, Table 3 shows the CRs among mm for the architectures supported by Linux 2.4.18. A different threshold (10%), higher than the 1% used for Table 2, was used to avoid reporting meaningless data. Only 10 rows out of 225 were retained and, as it can be readily seen in Table 3, the-mm subsystems contain only few cloned functions even if the CR values are non-negligible.

CRs were also computed for the core kernel subsystem (e.g., arch/i386/kernel versus arch/ppc/kernel). Those results were not presented here since, in a very similar way to mm, even if some architectures exhibit relevant CRs the number of cloned functions were very low (often one, sometimes two or three).

The data for Linux 2.4.18 confirmed the results obtained on different Linux releases [13,9]. In most cases, the implementation of similar functionalities was carried out by resorting to code reuse (function dependencies across different subsystems) rather than to cloning. This is clearly shown by the small number of subsystem comparisons exhibiting a non-negligible number of cloned
<table>
<thead>
<tr>
<th>Subsystems Compared</th>
<th>Functions ≥ 5 LOCs</th>
<th>All Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Common Ratio</td>
<td>Functions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cloned</td>
</tr>
<tr>
<td>i386-mips</td>
<td>11.11%</td>
<td>1</td>
</tr>
<tr>
<td>i386-s390</td>
<td>11.11%</td>
<td>1</td>
</tr>
<tr>
<td>i386-sh</td>
<td>14.81%</td>
<td>3</td>
</tr>
<tr>
<td>mips64-mips</td>
<td>22.61%</td>
<td>6</td>
</tr>
<tr>
<td>mips-mips64</td>
<td>11.59%</td>
<td>2</td>
</tr>
<tr>
<td>s390-arm</td>
<td>10.00%</td>
<td>1</td>
</tr>
<tr>
<td>s390-i386</td>
<td>15.00%</td>
<td>2</td>
</tr>
<tr>
<td>s390-mips</td>
<td>10.00%</td>
<td>1</td>
</tr>
<tr>
<td>sh-i386</td>
<td>15.00%</td>
<td>2</td>
</tr>
<tr>
<td>sparc64-sparc</td>
<td>12.77%</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 3
CRs ≥ 10% among mm architecture dependent code.

There are few exceptions, however. Among these, the CR between the mips64 and mips mm subsystems (22.61%, with six cloned functions). The ratio obtained without filtering out functions smaller than five LOCs was slightly higher (28.57%), but considerably smaller than the 38.4% computed on the Linux Kernel 2.4.0 and reported in [13]. However, even in this case, the absolute number of cloned function is low.

Table 4 reports data on CR and cloned functions among Linux drivers. Driver subsystems (e.g., the SCSI and IDE drivers, the char and USB or the PCI drivers) are the largest part of the kernel code and are subject to continuous evolution. CR among driver subsystems is fairly low, and in general only few functions are duplicated. An exception seems to be the number of duplicated functions between the char and subr systems, where 53 clone clusters were identified.

4.2 Cloning Evolution

This section aims to investigate how the cloning ratio varied in the Linux kernel from release 2.4.0 to release 2.4.18. The analysis has been performed at
different levels of granularity:

(1) The overall cloning on the entire Linux kernel;
(2) The cloning among major subsystems; and
(3) The cloning among architecture-dependent code of some subsystems.

Figure 3 reports the evolution of the overall CR, computed considering both all functions and only functions ≥ 5 LOCs. The figure shows that results are every different if small functions are filtered. In both cases, the cloning variation over releases is not relevant. Focusing our analysis on CR for functions ≥ 5 LOCs (as well as in all the further analyses presented in this subsection), the CR varies from 14.33% to 16.11% (i.e., a maximum difference of about 2%) and its standard deviation is 0.03. This supports the hypothesis that no considerable re-factoring was performed across 2.4.x releases.

The analysis of CR evolution among major subsystems confirms the previous impressions. Even in this case, no relevant change in the CR has been detected (variations are less than 2%). Figure 4 shows the evolution of cloning between fs and mm subsystems. It is worth noting that, from release 2.4.0 to release 2.4.4, the CR in mm decreased of about 1.6% (about 20 functions), indicating

<table>
<thead>
<tr>
<th>Subsystems Compared</th>
<th>Functions ≥ 5 LOCs</th>
<th>All Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Common Ratio</td>
<td>Functions Cloned</td>
</tr>
<tr>
<td>sbus-char</td>
<td>6.62%</td>
<td>53</td>
</tr>
<tr>
<td>sgi-char</td>
<td>6.80%</td>
<td>7</td>
</tr>
<tr>
<td>tc-char</td>
<td>9.38%</td>
<td>6</td>
</tr>
<tr>
<td>i2c-parport</td>
<td>5.44%</td>
<td>8</td>
</tr>
<tr>
<td>input-usb</td>
<td>5.88%</td>
<td>3</td>
</tr>
<tr>
<td>sgi-macintosh</td>
<td>5.83%</td>
<td>6</td>
</tr>
<tr>
<td>tc-macintosh</td>
<td>12.50%</td>
<td>8</td>
</tr>
<tr>
<td>zorro-pci</td>
<td>8.33%</td>
<td>1</td>
</tr>
<tr>
<td>sgi-sbus</td>
<td>10.68%</td>
<td>11</td>
</tr>
<tr>
<td>tc-sbus</td>
<td>10.94%</td>
<td>7</td>
</tr>
<tr>
<td>sgi-tc</td>
<td>7.77%</td>
<td>8</td>
</tr>
<tr>
<td>tc-sgi</td>
<td>12.50%</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 4

CRs ≥ 5% among drivers.
Fig. 3. Overall evolution of common ratio.

In a way similar to the results presented in the previous subsection, the most interesting behavior of CR evolution was found inside the mm subsystem, and in particular between the mips64 and mips architecture-dependent code. The values of CR are plotted in Figure 5. The figure shows that the CR ranged...
from 37.68% for release 2.4.0 (slightly different from the 38.4% reported in [13],
and computed considering all functions) to 22.60% for release 2.4.18.

![Graph showing the evolution of common ratio between mips and mips64 code inside the mm subsystem.](image)

Fig. 5. Evolution of common ratio between mips and mips64 code inside the mm subsystem.

One may argue that the programmers first ported the mm subsystem to the mips64 architecture by cloning portions of the mips code, and then performed a re-factoring. However, a more detailed analysis demonstrated the exact contrary. In fact, the number of functions (≥ 5 LOCs) composing the mips64 portion of mm varied from 69 in release 2.4.0 to 115 in release 2.4.18, in that the number of cloned functions remained constant to:

\[
37.68\% \text{ of } 69 = 22.60\% \text{ of } 115 = 26
\]

In other words CR, as much like any relative measure, should be used with great care, always resorting to the examination of absolute values.

5 Related work

Previous research studied both the detection and the use of clones for widely varying purposes, including program comprehension, documentation, quality evaluation, or system and process restructuring. Some of the techniques used for clone detection are based on a full text view of the source code [2,18]. Other approaches, such as those pursued by Mayrand and al. [22] and Kon-
togiannis and al. [19], focus on whole sequence of instructions (BEGIN-END blocks or functions) and allow the detection of similar blocks using metrics. Kontogiannis and al. [19] detect clones also using two further pattern matching techniques, namely dynamic programming matching and statistical matching between abstract code description patterns and source code. Finally, another clone detection tool, proposed by Baxter and al. in [5], relies on the comparison of subtrees from the Abstract Syntax Tree of a system.

Several applications of clone detection have also been investigated: Johnson [18] visualizes redundant substrings to ease the task of comprehending large legacy systems. Mayrand and al. [22], as well as Lagie and al. [21], document the cloning phenomenon for evaluating the quality of software systems. Lagie and al. [21] have also evaluated the benefits, in terms of maintainability of the system, of the detection of cloned methods. Finally, Baxter and al. [5] restructure systems by replacing clones with macros, in order to reduce the quantity of source code and to facilitate maintenance.

Several studies have been performed to analyze the Linux Kernel. As mentioned before, in [7] and [8] Bowman et al. recovered the actual Kernel architecture. Further analyses were executed by Tran et al. in [26] and [27]. The first experience in analyzing the evolution of the Linux Kernel in terms of metrics was done by Godfrey and Quiang-tr in [15]. Successively, the same authors performed a study of the evolution of one of the subsystems of the Linux kernel (the SCSI subsystem) also in terms of cloning ratio. They also developed a tool to aid software maintainers in understanding how large software systems have changed over time and, particularly, to help modeling long-term evolution of systems that have undergone architectural and structural changes. Results of these studies are summarized in [16].

Investigation performed by the Authors in predicting Linux kernel evolution using time series has been reported in [12]. Finally, an experience in applying time series to cloning ratio prediction was presented in [1].

6 Conclusions

The common ratio for several releases of the Linux kernel has been measured, discussing the process and the strategies that can be adopted to analyze a large multi-platform, multi-million lines-of-code real word software system. Software metrics at function level were extracted and duplicate code among kernel subsystems detected. Different thresholds were adopted to extract the common ratios, to avoid biased results due to false positives induced by small functions. In the present study, we considered two configurations, the first corresponding to the analysis of all functions belonging to the system, the
second discarding the functions with a body shorter than five LOCs.

Linux has not been developed through a well-defined software engineering process, but by the cooperative work of relatively uncoordinated programmers. Nevertheless, the overall common ratio, as well as the common ratios of its subsystems, are remarkably low, especially if small functions are not taken into account.

The answers to the research questions can therefore be summarized as follows:

- Cloning ratio among sub-systems can be considered at a physiological level;
- Recently-introduced architectures tend to exhibit a slightly higher cloning ratio. The reason for this is that a subsystem for a new architecture is often developed incrementally respect to a similar one (e.g. mips64 from mips); and
- The evolution of common ratio, at the overall level, tends to be fairly stable, thus suggesting that the software structure is not deteriorating due to copy-and-paste practice.

It is worth to point out that almost always even a relatively high common ratio value does not represent a remarkable number of duplicated functions. Code duplication may be considered relevant only among few major subsystems (e.g., arch versus drivers). But even in this case, due to the high number of functions in the subsystems, a common ratio value of about 1-2% ends up in just 100-150 duplicated functions.

7 Acknowledgements

Giuliano Antoniol and Massimiliano Di Penta were partially supported by the ASI grant I/R/091/00.

References


