Toward a Taxonomy of Clones in Source Code: A Case Study

Cory Kapser and Michael W. Godfrey
Software Architecture Group (SWAG)
School of Computer Science, University of Waterloo
{cjkapser, migod}@uwaterloo.ca

Abstract

Code cloning — that is, the gratuitous duplication of source code within a software system — is an endemic problem in large, industrial systems [9, 7]. While there has been much research into techniques for clone detection and analysis, there has been relatively little empirical study on characterizing how, where, and why clones occur in industrial software systems. In this paper, we present a preliminary categorization scheme for code clones, and we discuss how we have applied this taxonomy in a case study performed on the file system subsystem of the Linux operating system. Our case study yielded several interesting results, including that cloning is rampant both within particular file system implementations and across different ones, and that as many as 13% of the 4407 functions that are more than six lines long were involved in a clone-pair relationship.

1 Introduction

Code duplication, or code cloning, is generally believed to be common in large industrial systems [9, 17, 20, 18, 15, 2, 7]. Various problems are associated with code duplication, including increased code size and increased maintenance costs. While clone detection is an area of active research, and several tools exist to facilitate code clone detection, there has been relatively little empirical research on the types of clones that are found, or where they are found.

A code clone pair is a pair of source code segments that are structurally or syntactically similar. One of the segments is usually a copy of the other, perhaps with minor changes. Code cloning occurs when developers create two identical or similar code artifacts inside a software system. For example, developers may copy and paste code. Several methods exist for detecting code clones in software, such as simple string matching [9], using statistical fingerprints of code segments [12], function metrics matching [17, 20, 18], parameterized string matching [2, 15], and program graph comparison [7]. Problems related to clone cloning will be discussed in Section 2.

In the following case study, we begin to profile the code cloning activity within a large software system that is in widespread use in industry, the Linux operating system kernel. In doing so, we hope to gain more insight into how and why developers duplicate code, in an effort to aid the development of code clone detection techniques and code clone elimination strategies. We categorize different types of cloning activity using attributes such as location and size based on manual inspection of code clones found in the system. We then provide empirical analysis of these categories, and validation on our results using two different clone detection techniques. In this study we produce a taxonomy of code cloning which will help others examine code cloning, and we present a case study of a real software system.

The rest of the paper is structured as follows: in Section 2, we describe code cloning in more detail, as well as our study subject. In Section 3, we describe the tools we used and the methodology of our study. In Section 4, we describe the code clone categories we observed in the Linux file-system. In Section 5, we describe the empirical results we obtained. Section 6 describes related work, and Section 7 summarizes our work and indicate some future research.

2 Background

In this section, we provide background on code cloning as a problem in large software systems. We give examples of reasons why code cloning occurs, as well as several examples of problems caused by code cloning.

In addition, we give an overview of our candidate software system for this case study, the Linux kernel file-system subsystem. We will provide a brief description of the Linux file-system subsystem, as well as give reasons for choosing the file-system subsystem for our case study.

2.1 Code Cloning

Code cloning is considered a serious problem in industrial software [9, 12, 13, 8, 1, 17, 20, 18, 2, 15, 7]. It is suspected that 5 to 10% of many large systems is duplicated code [9, 3], and it has been documented to exist at rates of over 50% in a particular COBOL system [9]. Code cloning occurs for a variety of reasons[12, 20, 18, 15, 2, 7]: the short term cost of forming the proper abstractions may outweigh the cost of duplicating code; this occurs when the developer is aware of the existence of code that already performs functionality similar to, or the same as, the functionality required. Developers may duplicate code because they are under time constraints; these constraints may be imposed by deadlines, or by LOC performance evaluation. Another likely and reasonable circumstance where developers duplicate code is they do not fully understand the problem, or the solution, but they are aware of code that can do some or all of the required functionality.

Several problems can develop as a result of code copying. The size of the source code, and ultimately the size of the object code, may become significantly larger as a result of excessive code cloning[2, 12]. Cloning code can lead to unused, or "dead", code in the system, which can cause problems with code comprehensibility, readability, and maintainability [12]. Duplication of code may also introduce improperly initialized variables, which may lead to unpredictable behavior of a system, especially if a two clone segments share a common variable. Cloning may be an indication of poor design [12]. Code duplication may indicate design problems such as improper or missing inheritance, or insufficient procedural abstraction[7]. Copying code may also result in copying bugs within the code as well.These effects contribute to "software aging" [12]; over time the program becomes hard to change and possibly less reliable and more inefficient.

2.2 Case Study Subject: Linux File System

Linux is a Unix-like operating system, written by Linus Torvalds with assistance from a distributed team of programmers across the Internet. Linux aims towards POSIX and Single UNIX Specification compliance. The version of the Linux kernel we used for this study was 2.4.19, the most recent stable version at the time of the writing.

We chose the Linux File System as the study subject for our project because we hypothesized that many of the supported file systems would contain clones among them due to the similarity of their basic functionality. In addition, we know in advance that several components of the file subsystem that were created with heavy influence from existing file system types, namely `ext2`/`ext3` and `autofs`/`autofs4`.

The Linux file system subsystem is organized as a layered design, with the upper most layer being the Virtual File System (VFS). The VFS provides a standard interface for the operating system to use when interacting with various file systems types. The underlying file system types, such as `ext2` and `intermezzo`, provide function pointers for the VFS to use when interacting with the file system.

Because the various file systems must interact with, or provide service to, the same upper layer, and are providing similar functionality, we expected to see at least some cloning between file systems. After a preliminary inspection we expected to see a lot of cloning between `ext2` and `ext3`; `jffs` and `jffs2`; `fat` and `msdos` and `umsdos` and `vfat`; `autofs` and `autofs4`. These systems were either closely related in functionality or were known to have evolved directly from the same code base.

The Linux file system subsystem consists of the VFS infrastructure plus 42 file system implementations: `adfs, affs, autofs, autofs4, bfs, coda, cramfs, devfs, devpts, efs, ext2, ext3, fat, freevxfs, hfs, hpfs, inflate_fs, intermezzo, isofs, jbd, jffs, jffs2, lockd, minix, msdos, ncpfs, nfs, nfsd, nls, ntfs, openpromfs, partitions, proc, qnx4, ramfs, reiserfs, romfs, smbfs, sysv , udf, ufs, umsdos, vfat`. There are a total of 538 `.c` and `.h` files, and 279,118 lines of code (including comments and blank lines).

3 Study Methods

In this section, we describe the two methods we used to gather code clone information from the system. First, we describe parameterized string matching, as implemented by the tool CCFinder. Second, we describe our approach to metrics-based clone detection, for which we used Understand for C/C++ to obtain the raw metric information, as well as a set of Python scripts that we created to perform the clone analysis. Finally, we describe our methodology for performing categorization and analysis.

3.1 Clone Detection

In this study we have primarily used the tool CCFinder, developed by Toshihiro Kamiya et al [15]. The tool uses a parameterized matching algorithm to search for code clones within C/C++, Java, and COBOL files. This type of clone detection is good at finding clones with name substitution and line structure changes; the former can cause problems for line by line matching algorithms. Baker introduced a similar algorithm in [2].

The tool CCFinder begins by performing a lexical analysis of the source code, resulting in the creation of a list of tokens as part of the syntax of the given programming language. The tokens of all the files are concatenated into a single string. As part of the code transformation, all white space is removed from the string and comments as well.

Next, several language specific transformation rules are applied. Then type, variable, and constant identifiers are replaced by a special identifier (such as $P).

Once the source code has been transformed into this abstract token stream, an exact match algorithm is performed to find maximal matching strings within the transformed code. This is done by constructing a suffix tree and locating matching substrings within the tree, as proposed by Baker [2, 3]

After the exact matches have been found, parameter matching is performed. That is, starting from the beginning of a pair of exactly matched transformed strings, CCFinder begins parameter matching of the parameters on each line. As the parameters are matched, if a conflict is found but a sufficiently large number of lines have been matched, the clone is reported, and parameter matching begins again after the line creating the conflict.

Once the clone detection phase is complete, the detected clones are mapped back onto their source files. Then, this information is used as input in the GeminiE user interface, where clone classes are generated and the results of the clone detection are presented. These clone classes are generated based on the fact that the clone relation is an equivalence relation [15, 14]. The clone relation exists when two code segments match according to parametric matching. A clone class is the equivalence class of the clone pair relation, i.e., it is the maximal set of clones for which the clone relation holds [15].

The results of the clone detection process are presented in several ways in a graphical user interface [22]. The interface provides a scatter plot showing the user the matches between files, highlighted source code, and clone class metrics. Users can browse the detected clones pair by pair or by clone class. For a small number of files, the scatter plot can provide useful information, but when a large number of files is present with many lines, i.e., 200,000 or more, significant clones become difficult to detect through visual inspection of the scatter plot. For this study, we found that we made the most use of the tool by browsing the clone pairs individually, and by browsing the clones classes.

Before using the clone pairs extracted by this tool, we filtered out many of the clones we felt were meaningless, to improve the accuracy and relevancy of our results. Meaningless clones are segments or code that match but are not necessarily cloned code, or clones that were of no importance if they are duplicated code. For example, the inner block of structure definitions and lists of function declarations would we often considered meaningless clones. These clones were often unrelated and only appeared because of the detection process in parameterized string matching. After the initial extraction of clone pairs, we were presented with 5000 clone pairs, and 1809 clone classes. We deleted 1996 clones in an effort to remove at least a significant number of meaningless clone pairs. This left us with 1604 clone classes, a decrease of only 200 clone classes. We do not claim to have removed all of the meaningless clones, but we believe that we have removed a significant number of them.

3.2 Metrics-Based Clone Detection

Metrics-based clone detection methods use groups of metrics to generate “fingerprints” for each function in the source code. These metrics are often gathered using both the program source, as in the case of number of lines of comments, and from an Abstract Syntax Tree, as in the case of cyclomatic complexity. Metrics-based clone detection was introduced by Mayrand et al. in [20] and Kontogiannis et al. in [18]. Further studies using function metrics as a basis of clone detection include [4, 6, 5, 8, 1, 17, 21].

In our case study we used the following set of metrics:

1. Line counts: total number of lines, count lines blank, count of lines of code, count lines of declarations, count lines of executable code, count lines of comment.

2. Count number of parameters, number of global variables used.

3. Count number of parameters or global variables modified.

4. Cyclomatic complexity.

5. Maximum level of nesting.

These metrics are different than those used in other studies such as [20, 18] but as stated in [1], in large systems the choice of metrics does not significantly affect the results. We have used a subset of those metrics used in previous work [20, 18]. From this we would expect that our returned pairs be less precise, and more false positives to be present, but this is not the case. Upon visual inspection of several hundred of the clone pairs, false matches were very rare, confirming that the choice of metrics does not affect the results.

As in [1, 8] we searched for functions that had identical metric fingerprints. This corresponds to ExactCopy and DistrinctName classes which were defined in [20]. As in [1, 8], we did not use function name as a parameter.

To perform function matching based on metrics, we gathered our metrics using the tool Understand for C/C++. We then wrote a small program that performed the function matching grouping functions together one metric at a time. Function comparisons based on metric fingerprints can be done in $\mathcal{O}(n * m)$ time where n is the number of functions and m is the number of metrics.

3.3 Classifying and Evaluating Clones

To classify the clone pairs, we used the results from the clone detection using CCFinder. Because of the large volume of information presented to us, caused by the large wealth of information given to CCFinder, it was difficult to see any interesting trends that might occur amongst related files. This is because the clones were distributed among many files and many of the clone pairs appeared as blocks of code and it was difficult to get a feel for the cloning activity as a whole within the file system. To remedy this, we researched the file systems included with the Linux kernel to evaluate the relationships between them, and to pinpoint places where cloning activity is likely to have occurred. We also found some interesting relationships between several file systems, which we did not expect, by graphically displaying the amount of clone-pairs occurring between each file-system.

After narrowing down where we would begin to look, we manually viewed a large percentage of the clone pairs found in that area of the system. As we saw trends, we identified types of clones and began classifying many of the clone pairs that fell into these various categories. Once we had created many of the clone categories we have now, we browsed clones within the entire file system to find if there were clone categories we had not yet seen.

When we had a set of clone categories that we were satisfied with, we wrote scripts to place the clone pairs into the categories we had created. The criteria we based these scripts on were as follows: for functions to be classed as clones, 60% of their code must be common between the two. Initialization clones must start within the first 5 lines of a function and end within the first half of the code. Finalization clones must start in the last half of the function and end in the last 5 lines. Blocks of code not in the same function must not be in any of the above clone types. All clone types are exclusive, so a clone pair that is part of a cloned function relationship can not also be an initialization clone.

After categorization, and for any other empirical results we have presented, we performed manual inspection of a large percentage of the clone pairs in the given experiment to ensure that they were within the criteria that we specified and that they were accurately found as clones.

3.4 A Basis for Comparison

As a way to compare the results given to us from the two methods of clone detection, we manipulated the data extracted from one to be close in form to the other. Because full function matches were a smaller subset of CCFinder's returned clone pairs, we used only the function matches found by CCFinder to compare to the function matches found by metric based detection.

In doing so, we defined a criterion for which to decide when a function was matched with another based on the code segments matched between the two. For two functions to match, more than 60% of their individual code must be common between the two. This may be in the form of a single segment of code duplicated between the two, or several individual code segments.

Another issue to consider in comparing the two methods was the minimum size of code segments that could be classed as a clone pair. We found that five lines often found function matches that were too small for CCFinder to find, because we had set CCFinder to find code segments of a minimum size of 30 tokens. Several values of minimum line numbers were tried, five, six, and seven and the results of these are described in Section 5.3.

4 A Taxonomy of Clones

In the following subsections we present a taxonomy of the types of clones we found during this case study using the clone pairs from CCFinder; in the following section we analyze our findings.

The categories of clones are described using the following template: the first paragraph describes the structure of the clone; the second paragraph describes problems caused by this type of clone; the third paragraph describes reasons why these clones may be introduced into the software; and the fourth paragraph describes a possible solution to that form of cloning activity.

4.1 Duplicated blocks within same function

Characterized as repeated blocks of code within the same function, these blocks are of non-trivial size (such as 5 to 127 lines of code) and each copy expresses the same semantic idea, generally with very few variables changed (often only one). We found that this type of clone occurs often in the Linux file-system subsystem.

The major problem that this can cause is increased code size; in particular it can cause functions to grow long and unreadable. In addition, this type of cloning may lead to unintended diverging evolution of the code blocks if a developer changes one block, and not another. A bad initialization or 'value changed' type of error can very easily happen in this type of code, because it is likely variables are shared, intentionally or unintentionally, by the individual blocks.

Situations where this typically occurred was in control structures such as `switch` and `if/else` statements. The cause of this may be that some developers do not anticipate a condition that may require a similar block, so they do not think to make the block a function from the start. Also, making the function that encapsulates the functionality of this clone block may appear to be too much work, because of the number of local variables involved in the code block. Another reason may be time: it is very fast to just copy and paste the block just a few lines down, and the developer "knows" the code works, so it is a quick and dirty solution. Performance may also be an issue, if many local variables are required to be passed, stack creation and destruction may be time consuming.

A solution to this problem, as with many code clones, would be to create a new function or macro to represent the block, and call the function where these clones occur. Parameters to the function would be the few changed variables that occur in the code block. One would expect this change to be simple and straightforward to implement.

4.2 Similar functions, same file

This type of clone occurs when a programmer has two functions performing very similar tasks, with minor variations. These types of clones are often characterized by changing only a few function calls, variable initializations, constants, or other minor things. We consider any functions which both match 60% of their code to be cloned functions.

Consequences of this type of clone are increased code size. Also fixing bugs may be harder because same error may be spread across several functions, as well as the functions may evolve on separate paths as various maintainers update them.

Developers are likely to do this when the effort required to parameterize the code block and create a more general function appears to be too great when compared to simply copying the code. Also cloning the function may actually make the program conceptually simpler, because the function names can be specific and meaningful. This type of cloning we do not consider extremely harmful because clones are not physically far apart, but it is recommended that such cloning activity should be documented as it may not be apparent to future maintainers which functions are clones of each other.

Solutions for this can be very simple, or quite complex. Possible solutions would be to introduce function pointers to the parameter list, adding more parameters for initialization, etc.

4.3 Functions cloned between files within the same directory

This type of clone occurs when the same functionality is required among multiple files. The majority of code duplication that occurs within a directory (excluding duplication with the same file) is related to duplicated functions, more than 80% of clone pairs that occurred within the same directory (but not in the same file) were related to the duplication of functions. It often occurs with no changes at all to the cloned segment of code, or minor changes such as the function name and some variable or function calls. At times, several constants may be changed, global variables accessed and in these cases a solution is harder to find.

Consequences of this type of clone are code size increase, and increased difficulty in error finding and correcting. The copied code segments are no longer localized in the same file and easily identified, but may be scattered across as many as four or five files. At times, this type of code duplication may contribute to source code that is easier to read. Functions will be easier to understand because they will not include extra logic and flows of control which would be required to restructure a function to encompass the more general functionality required of it to eliminate duplicates. This case is less frequent however, and quite often the use of function pointers or some minor conditional operations would create a function which may perform the desired task.

A simple solution to this is to create a common file to use as a library, and migrate the function definitions and prototypes of the cloned functions to this file. This will work best in the case of exact copies, or clones with minor changes.

4.4 Functions cloned across directories

This type of clone may occur when the same functionality is common among several different components in the software. As with functions cloned within a subsystem, it may entail no changes at all to the cloned segment, or minor changes such as the function name and some variable or function calls. We often saw this type of clone for generic kinds of tasks such as parsing options or outputting errors.

Consequences of this type of clone are code size increase, and may increase labor for error fixing. Also, it may be the case that one developer created one component, and is unaware of the clones existing in the rest of the system. In this case, when an error is found, repairs may not even have a chance to be propagated to the rest of the clones.

This type of cloning may occur when a new subsystem is being created, and the design and implementation is based on previous work of another subsystem.

Creating a set of library function may be the easiest solution, but if the function is cloned only between several files, the effort put into creating a new library, and maintaining it, to be shared by all components may be more work than it is worth.

4.5 Cloned files (possibly with some changes)

This type of clone occurs when a new problem arises with requirements that are very similar to those of an existing software system, and the source code is readily available. For example, when new file system is introduced to the system, it may be possible to copy another's file, and make only minor changes. We saw a very good example of this when we compared `ext2` and `ext3`, in particular `buffer.c` in both systems. This is a very rare occurrence from what we have seen in the file-system subsystem, but in other systems such as this SCSI subsystem this type of cloning activity seems to be much more frequent [10].

Consequences of this type of cloning can be much more severe than function cloning, because the clone has now introduced a large number of lines of code that are common between the two files, and must be changed together, especially when bug fixing. Because it is likely that there will be some alterations to some of the code, it may not be clear where or how to change the cloned file when reflecting changes that have been made to the original code. Also, this is one of the worst-case scenarios for code size increase. In addition, it is possible that side effects (such as inefficient device usage and settings) can occur if the developer does not fully understand the code that he/she has copied. This may lead to inefficiencies in the code and instability. This type of cloning will occur when speed of development may be a factor, or a developer may not completely understand the problem at hand. We have also seen this when drivers are made for related hardware, although not part of this study.

Solutions to this problem may not be as simple as other cloning types. Because the two files are used on different products or include different features, they may need evolve separately from this point on. As well, changes that have been made to the duplicated code may make it difficult to re factor both subsystems completely just to remove to code duplicates. That said, a workable solution may be to try to take the common invariant code and place it into a common library file which both subsystems could use. This solution may lead to a slightly more complex architecture.

4.6 Blocks across files

This type of cloning is similar to the first one but it occurs in different files within the same directories or across file systems. Often, in the case of cloning blocks across directories, we see that the cloned block is in fact the remains of what appears to be a cloned function. The function is often changed to suit the developers own personal style and also to meet the specific needs of his/her own project. Based on our observations, we would argue that most clones that occur across files start out as whole function clones and then are manipulated to fit the current project goals until what remains are scattered blocks of code which can still be captured as code duplicates.

The main problem with this kind of clone is when the developer wants to modify or change these blocks of code or when they find bugs, it will be very difficult to fix and change these blocks everywhere else, and it is possible that the developer may be completely unaware of the other clones. If any logic on which this block depends changes, then all the blocks may be harmed, and it may be difficult to find all the blocks affected.

The solution for this problem is relative to the size and number of clones that occurs across files. In certain contexts it might be proper to leave the clones as it is, such as in the case of `if` or `case` statement, sometimes making function calls may break the understanding of the logic of the code. In other cases a common library should be made.

4.7 Initialization and finalization clones

This type of clone occurs within the same file or across file systems when initializing data parameters or cleaning up at end of function; we have found that the main portion of the function can perform quite different tasks. This usually occurs when using the same data types or when performing the same tasks such as memory allocation and de-allocation or variable initialization. Finalization clones often encompass exit conditions and logging.

Problems with this type of clone are much less severe than other clone types, and in many cases are unavoidable. Certainly increased code size may be an issue, but other problems related to code duplication do not seem as large of a concern.

Solutions to this sort of problem may be the use of macros or functions, but this seems too complex for something that is of such little issue.

5 Case Study Results

In this section, we discuss cloning activity in terms of clone pairs, not numbers of lines cloned. We consider that discussion about the number of lines that have been cloned can be misleading and confusing. In the case of clones within the same file, many clone pairs may overlap each other, in contrast to clone pairs outside of the directory, which in many cases do not intersect. The latter will seemingly have a larger number of cloned lines than the former, but in fact the degree of the cloning activity might actually be higher in the former.

In regards to the total number of lines cloned, allowing for lines to be counted more than once did not prove to be any more beneficial than discussing clone pairs, so we chose the former for simplicity.

The Linux file system contains 42 different file-system implementations in C. There are 538 .c and .h files, with a total of 279,118 lines of code. We detected 3116 clone-pairs after filtering, giving us 33,707 unique lines, or 12% of the source code, that were involved in code cloning activity. The average length of the clone pairs is 13.5 lines, with a median of 12 lines, an upper quartile of 15 lines and lower of 8 lines. The minimum length is 1 line and the maximum is 123 lines.

5.1 Families of Systems Based on Duplication

As illustrated in Figure 2, several families of file–systems, or groups where code is similar, become apparent. The most notable is the shared code between ext2 and ext3. Here we see 85 clones common between the two file systems. After investigating the code, the reasons are very obvious. Ext3 is based on ext2, and it appears the development of ext3 started by copying all of ext2 into a new folder.

Two unexpected results appeared when viewing this chart. The intermezzo file-system seems to be highly related to the main file-system code. By inspecting the code, we see that much of what was cloned involved getting and setting the path, and various navigation codes. The intermezzo was inspired by coda, although re–engineered and restructured, and we see some significant evidence of this by 11 clones appearing between the two. We also see that the JFFS file-system has cloned much from the inflate_fs. Here we see that most of the clones in this case are grouped into one file, although they are taken from many files within inflate_fs.

5.2 Frequency of Clone Types

As can be seen in by Figure 1, the major cloning trend is to duplicate code from within the same subsystem. 78% of code duplication occurs from within the same directory. Some notable exceptions to this trend are ext2/ext3 and AUTOFS/AUTOFS4. In these cases, ext3 was created based on ext2, and AUTOFS4 was created based on AUTOFS.

This result is significant. It suggests that problems associated with code duplication such as copying bugs in most cases will be restricted to within a single subsystem. This also gives developers good reason to focus their efforts on eliminating code duplication within subsystems first before doing system wide repair.

Reasons for this are probably the most obvious ones. The developer is most familiar with his/her own code, so is most likely to use code from within their own system. As well, because it is within the same system, it is more likely that relevant and similar code exists in this system.

Table 1 shows the number of clones that occur in the same file, in the same directory but not in the same file, and in different directories. From this table, we can see that the average size of a clone pair (the number of lines of code) is nearly the same, but the number of clones that occur in the same file is more than double the number of clones in the same directory but different files, or in different directories. We saw this again when analyzing the cloning activity on the 3D bar charts as described above.

Table 2 summarizes the frequency of the various types of clones. In this table, one should note that when we say that a function has been cloned, we mean that more than 60% of the code between two functions has been cloned. The number of duplicated functions in this table refers to the number of duplicated function pairs, or in other words pairs of functions that are in a clone relationship. In this table, count refers the number of occurrences of that type of clone, for example in the table we see 589 blocks of code were cloned in the same function, and 244 functions were cloned in the same file.

From Table 2, we see that over 30% of the clone pairs that occur within the same file are blocks of code duplicated within the same function. We also see that 244 function pairs occur within the same file. This number can be decomposed somewhat. From these pairs, there are 293 unique functions that take part in a code clone relationship. 341 clone pairs combined form this group of clones, of which 173 clone pairs encompass more that 60% of the functions which have been cloned.

Within the same directory, we see that there are 653 function pairs that are in a clone relationship. 658 clone pairs contribute to this, making more than 80% of the clone pairs occurring in the same directory but not in the same file part of a function clone relationship. 166 unique functions were cloned, meaning that many of these function pairs are part of larger clone classes.

Outside of a directory, there are 129 function pairs, with 156 unique functions. 175 pairs contribute to these full function matches. From this result, we see that function cloning decreases significantly, even though the actual amount of cloning activity does not drop so dramatically. We also see that functions are less likely to appear in clone classes when cloned across directories.

In Table 3 we see that the metrics–based clone detection validates our results that are based on parameterized string matching. Regardless of the constraint of minimum lines of functions, in all three cases, cloning of functions was most often found within the directory but not the same file, followed distantly by cloning of functions in the same file, and then cloning functions from outside the directory, just as

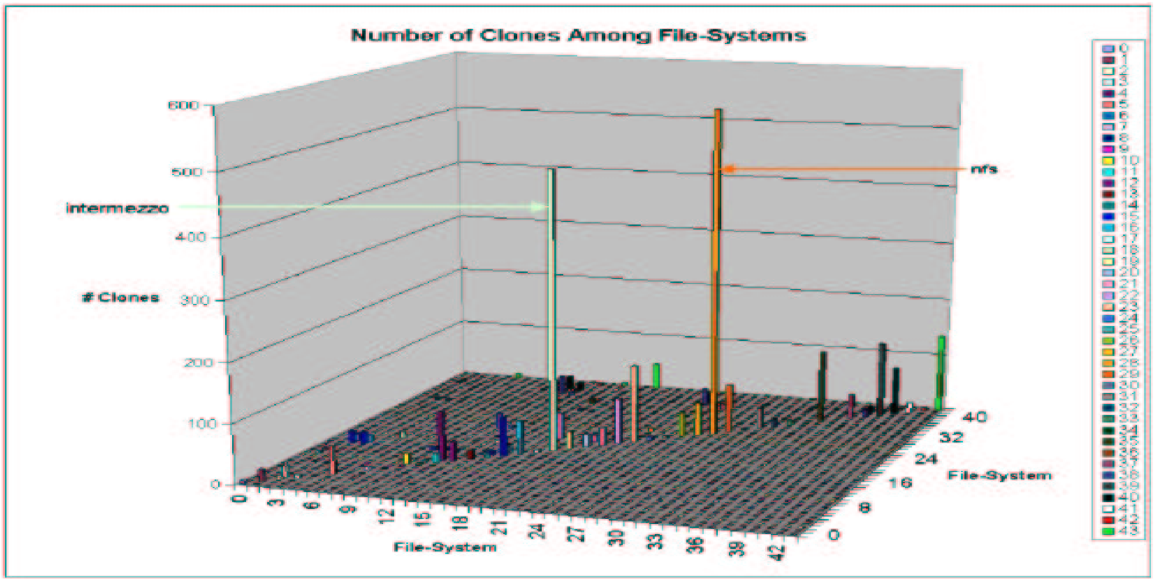

Figure 1: Number of Clone Pairs Between File Systems

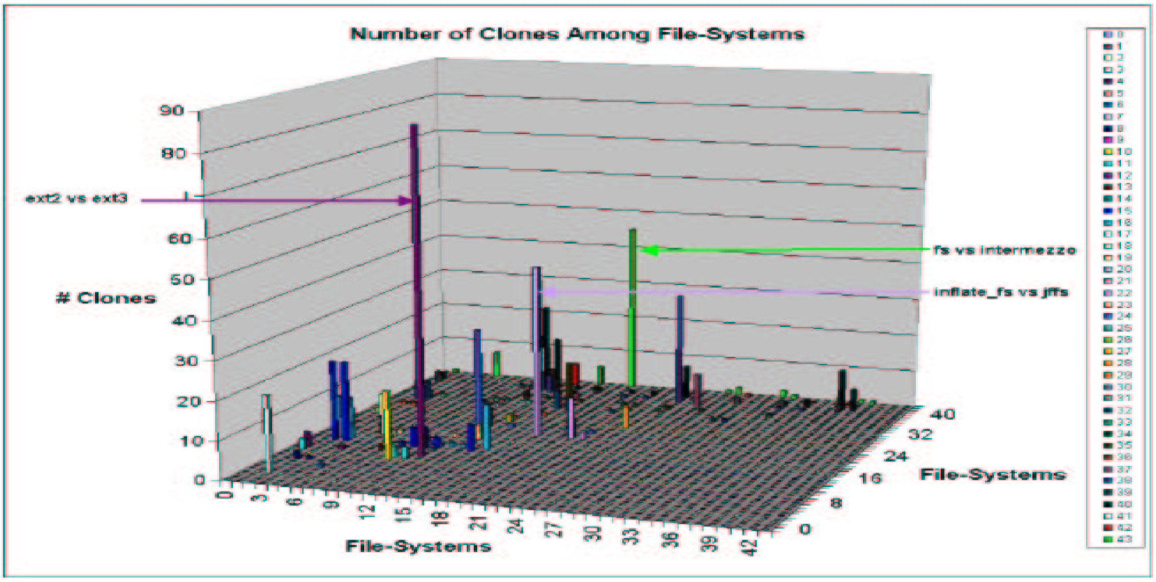

Figure 2: Number of Clone Pairs Between File Systems (excluding themselves)

	Clones in Same File	**Clones in Same Directory**	**Clones in Different Directories**
# of clone pairs	1628	806	682
Average LOC	12.7	14.5	14.3
Max LOC	63	71	123
Min LOC	2	4	1

Table 1: Profiles of cloning locality — All clones

Type	**Count**	**Average Length**
Same File		
Blocks in Same Function	589	13
Duplicated Functions	**244**	26
Initialization Clones	28	14
Finalization Clones	82	13
Cloned Blocks	588	13
Same Directory		
Duplicated Functions	**658**	16
Initialization Clones	2	14
Finalization Clones	11	10
Cloned Blocks	135	14
Different Directories		
Duplicated Functions	**129**	27
Initialization Clones	6	12
Finalization Clones	45	11
Cloned Blocks	456	14

Table 2: Frequency of various clone categories — Parametric String Match

what can be seen in the previous section. It is interesting to note how quickly the number of functions drops off as the minimum number of lines of a function is increased. A large portion of the functions that we lose are false matches, although some are not.

Initialization clones and finalization clones were not as frequent as were first expected they might be. The clones we did find, however were significant. We expect to find more of these clones in other parts of the Linux kernel, in particular driver source code. A surprising result is that initialization clones appear to occur much less often than finalization clones. After inspection of code block clones, we see that when code for initializing a function is copied, often local variables are added to the cloned list or removed from it. This makes it difficult to classify many of the initialization clones automatically. Therefore, we take the frequency of initialization clones as an underestimate. Better approaches to automatically find this class of clone need to be investigated further.

Cloned blocks of cloned code are difficult to characterize completely, as there are many circumstances leading to the cloning of these blocks. However, we have found that locality does correlate somewhat to the structure and reasons of this type of clone. In the cases of clones in the same file and same directory, these clones are often the product of copying blocks contained within a control structure, such as `if/else` statements. In some cases however, they are what remains of what was once a initialization clone or a finalization clone.

When we inspect clone blocks across directories, it is often the case that the blocks are the remains of copied functions, changed enough that the functions no longer can be classed as cloned functions technically, but by manual inspection these functions are still clearly in a form of clone relation. These blocks raise interesting questions about the evolution of clones.

In many cases, clone blocks represent function pairs where 60% or more of one function has been copied to another, and additional states have been added. These function pairs, which we call partial-match function pairs, represent an interesting form of code cloning. It would be difficult for this form of function cloning to be detected by metrics based clone detection algorithms, but they are certainly function clones. In many cases, one function is entirely copied, and a

	Metric Match			String Match
Minimum Function Length (LOC)	5	6	7	N/A
Same File	141	110	108	244
Same Directory	1157	1152	619	658
Different Directory	116	80	38	129

Table 3: Number of function clones found in metrics based clone detection and parameterized string match

significant number of statements have been appended to the end. In total, these partial function matches accounted for 72 same file clone blocks, 22 same directory clone blocks, and 109 different directory blocks.

This case study shows that the taxonomy is not complete. The presence of so many cloned blocks may be an indication that more categories of clones exist, and further investigation must be done.

5.3 Metrics vs. Parametric String Matching

In Table 4, we see the summary of results in comparing the function pairs found by the metrics method to those found by the parametric method. The first row presents the number of function clones found in both the metrics based clone detection and the string matching algorithm at the same time. Then second and third row show the number of function clones found by each detection method exclusively. In all cases, we see that between 708 and 716 function pairs were found by both methods. These function pairs are in most cases very clearly clones of one another. Also, in all cases, between 353 and 361 function pairs were only found by the parameterized string based approach. In these cases, the functions tend to be longer than average, their average length being 30 lines of code. Often, lines have been added and removed, or fan in or fan out metrics have changed. This shows us that using exact match criteria may not always be sufficient in searching for function pair clones.

In the cases of functions pairs found only by metrics we see two things. First, functions of sizes five LOC and six LOC are often too small to be detected when using a minimum criteria of 30 tokens in the parameterized string approach. Secondly, small functions of sizes five, six and seven LOC are often hard for parameterized string matching to detect clones in when there are enough tokens present. This is because if one token violates the parametric match, then there is little chance that enough tokens were already matched to make a clone that is large enough to report, and there is also little chance of enough tokens remaining in the function to find another clone. Often a function call that takes a different number of parameters or changed mathematical operators can cause the parameterized string matching to miss matches in small functions.

In general, we found when using our metrics-based clone detection tool, it was better at finding small function matches than CCFinder, but CCFinder was better at finding large function matches. When using CCFinder with the Gemini GUI, we found that it was difficult to grasp the total cloning activity in the system, but when clone pairs were grouped by the taxonomy we have presented, interesting cloning activity becomes more evident. We found that metrics-based clone detection finds very close matches, but CCFinder is able find function matches which exhibit more change. As a preliminary result, we found that parameterized string matching presented more interesting and useful clones to use than using ExactMatch metric-based clone detection. Future work will investigate the comparison of these two approaches but allowing more flexibility in the metrics-based matching.

6 Related Work

There are several types of clone detection techniques that have been developed. Metrics-based clone detection tools which detect clones of full blocks of code such as functions based on various metrics extracted from them have been developed by Mayrand et al. [20] and Kontogiannis et al. [18]. Parameterized string matching is discussed by Baker et al. [2, 3] and Kamiya et al. [15]. Baxter et al. [7] have developed a clone detection tool by performing subtree matching on abstract syntax trees. Program dependence graphs have been used by Krinke et al. [19] and Komondoor et al. [16] in detecting duplicated code. Johnson [13, 12] proposed using a fingerprinting algorithm on substrings of the source code. Kontogiannis et al. define two other methods to detect clones in [18]:dynamic pattern matching which finds the best alignment between two code fragments, and statistical matching between abstract code descriptions patterns and source code. Balazinska et al. [4, 6, 5] uses metrics based clone detection to quickly find candidate clones and uses an algorithm based on Kontaogiannis et al.'s dynamic pattern matching algorithm.

Clone detection case studies on the Linux kernel have been reported in [10, 8, 1]. In [8], Casazza et al. use met-

Minimum Number of Lines	**5**	**6**	**7**
Function pairs found by both	716	716	708
Found in Parametric Only	353	353	361
Found in Metrics Only	698	626	57

Table 4: Comparison of # of function clones found by the two clone detection algorithms

rics based clone detection to detect cloned functions within the Linux kernel. They performed analysis across the major subsystems, and then on the architecture dependent code of the memory management subsystem and the kernel core. To evaluate the degree to which cloning occurs, they define a common ratio between two files, which is the percentage of functions in one file which are cloned in another with respect to the number of functions in the first. As noted in [1], this common ratio must be used with great care and absolute values need to used as well. The conclusions of this study were that in general the addition of similar subsystems was done through code reuse rather than code cloning, and more recently introduced subsystems tended to have more cloning activity. Antoniol et al. [1] did a similar study, evaluating the evolution of code cloning in the Linux. They too used function metrics clone detection as their technique and their conclusions were the same, adding that the structure of the Linux kernel did not appear to be degrading due to code cloning activity. In [11] a preliminary investigation of cloning among Linux SCSI drivers was performed.

Kamiya et al. [15] performed tests on JDK to search for clones within the system, and they studied the cloning behavior between Linux, FreeBSD, and NetBSD. While [15] observes that clones in JDK seem to occur in near directories or files based on visual inspection of the scatter plot their tool presents, no quantitative data analysis is discussed concerning this point. None of the above studies have discussed the types of clones they have found, or discussed the locality of code cloning in detail other than comparing the level of cloning amongst subsystems.

A work similar to this also tries to categorize clones for the purpose of software maintenance. In [4], Balazinska et al. create a schema for classifying various cloned methods based on the differences between the two functions which are cloned. The results produced in [4] are used by Balazinska et al. in [6, 5] to produce software aided re engineering systems for code clone elimination. This differs from our work in that our classification scheme is based on locality as well as clone type, and copied functions are only one type in our case, although in [4] they break this down into 18 categories. One of our main research goals is to determine how much developers clone and from where. This question is not answered by the clone classification scheme in [4]. In addition, this work ignores code clones which are not function clones.

7 Summary and Conclusions

This preliminary study began as in-depth evaluation of cloning in a large software system. In this study we found that the Linux file-system subsystem has a significant amount of code duplication within it, the majority being localized within each individual file-system type, or sub-subsystem, similar to the activity in JDK noted in [15]. We also defined a preliminary taxonomy by which non-function and function clones can be categorized. This will be used in future research when characterizing cloning in all of the Linux kernel.

Our first goal, to begin to produce a finely grained analysis of code cloning in a large scale software system has begun, and future work will attempt to characterize more subsystems, in particular the driver subsystem where source code and functionality is vastly different from the Linux file-system subsystem. This work will provide support for generalizing these results, as well as more insight into the growth of the Linux kernel as documented in [11].

During our study, we found that 3D visualization provides much convenient information. From the 3D bar charts we were able to see very quickly related groups of subsystems, and also which subsystems were trouble spots for cloning activity. Further investigation on the scalability of the graph is needed, but at the current time we would suggest that including the ability to visualize clone detection results in such a way may be a very useful addition to maintenance environments involving clone detection.

8 Future Work

Research on this topic is ongoing. We intend to continue this study, to fully characterize the Linux kernel in terms of code clone activity. We will also investigate how other subsystems compare to these results. From preliminary testing, many may be quite similar.

We will also evaluate our taxonomy throughout the course of this study.

Additionally, we will study how code clones evolve over time, in particular, we would like to test the hypothesis that many code block clones start out as function clones, and as time goes on, the functions evolve away from one another. We will also investigate the possibility of using previous releases for detecting clones in current releases.

Acknowledgments

We would like to thank Dr. Ettore Merlo for his on going help and advice.

References

[1] G. Antoniol, U. Villano, E. Merlo, , and M. Di Penta. Analyzing cloning evolution in the linux kernel. In *Information and Software Technology 44(13)*, 2002.

[2] B.S. Baker. A program for identifying duplicated code. In *Proceedings of Computing Science and Statistics: 24th Symp. Interface*, pages 49–57, 1992.

[3] B.S. Baker. On finding duplication and near-duplication in large software system, 1995.

[4] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Measuring clone based reengineering opportunities. In *Proceedings of the Sixth International Software Metrics Symposium*, pages 292–303, 1999.

[5] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, and Kostas Kontogiannis. Partial redesign of java software systems based on clone analysis. In *The Proceedings of the 6th. Working Conference on Reverse Engineering*, pages 326–336, 1999.

[6] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, and Kostas Kontogiannis. Advanced clone analysis to support object-oriented system refactoring. In *Proceedings of the 7th. Working Conference on Reverse Engineering*, pages 98–107, 2000.

[7] Ira D. Baxter, Andrew Yahin, Leonardo M. De Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM*, pages 368–377, 1998.

[8] G. Casazza, G. Antoniol, U. Villano, E. Merlo, and M. Di Penta. Identifying clones in the linux kernel. In *First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 92–100. IEEE Computer Society Press, 2001.

[9] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In Hongji Yang and Lee White, editors, *Proceedings ICSM'99 (International Conference on Software Maintenance)*, pages 109–118. IEEE, 1999.

[10] Michael W. Godfrey, Davor Svetinovic, and Qiang Tu. Evolution, growth, and cloning in Linux: A case study. A presentation at the 2000 CASCON workshop on 'Detecting duplicated and near duplicated structures in largs software systems: Methods and applications', on November 16, 2000, chaired by Ettore Merlo; available at `http://plg.uwaterloo.ca/˜migod/ papers /cascon00-linuxcloning.pdf`.

[11] Michael W. Godfrey and Qiang Tu. Evolution in open source software: A case study. In *Proceedings of the 2000 International Conference on Software Maintenance*, 2000.

[12] J. H. Johnson. Substring matching for clone detection and change tracking. In *Proceedings of the International Conference on Software Maintanence*, pages 120–126, 1994.

[13] J.H. Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of CASCON 93*, pages 171–183, 1993.

[14] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. A token-based code clone detection tool - ccfinder and its empirical evaluation. Technical report, 2000.

[15] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. In *Transactions on Software Engineering 8(7)*, pages 654–670. IEEE Computer Society Press, 2002.

[16] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. *Lecture Notes in Computer Science*, 2126:40–??, 2001.

[17] K Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *Proceedings of Working Conference on Reverse Engineering*, pages 44–55. IEEE Computer Society Press, 1997.

[18] K. Kontogiannis, R. De Mori, R. Bernstein, M. Galler, and E. Merlo. Pattern matching for clone and concept detection, 1996.

[19] Jens Krinke. Identifying similar code with program dependence graphs. In *Proc. Eigth Working Conference on Reverse Engineering*, pages 301–309, 2001.

[20] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the International Conference on Software Maintenance*, pages 244–253. IEEE Computer Society Press, 1996.

[21] Qiang Tu and Michael Godfrey. An integrated approach for studying software architectural evolution. In *Proceedings of 2002 International Workshop on Program Comprehension (IWPC-02)*, 2002.

[22] Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Gemini: Maintenance support environment based on code clone analysis. In *Proceedings of the Eighth IEEE Symposium on Software Metrics*, pages 67–76. IEEE Computer Society Press, 2002.