How Slim Will My System Be? Estimating Refactored Code Size by Merging Clones

Norihiro Yoshida
Nagoya University
Japan
yoshida@ertl.jp

Takuya Ishizu
Osaka University
Japan
t-ishizu@ist.osaka-u.ac.jp

Bufurod Edwards III
Osaka University
Japan
BufordEdwards3@gmail.com

Katsuro Inoue
Osaka University
Japan
inoue@ist.osaka-u.ac.jp

ABSTRACT

We have been doing code clone analysis with industry collaborators for a long time, and have been always asked a question, "OK, I understand my system contains a lot of code clones, but how slim will it be after merging redundant code clones?" As a software system evolves for long period, it would increasingly contain many code clones due to quick bug fix and new feature addition. Industry collaborators would recognize decay of initial design simplicity, and try to evaluate current system from the view point of maintenance effort and cost. As one of resources for the evaluation, the estimated code size by merging code clone is very important for them. In this paper, we formulate this issue as "slimming" problem, and present three different slimming methods, Basic, Complete, and Heuristic Methods, each of which gives a lower bound, upper bound, and modest reduction rates, respectively. Application of these methods to OSS systems written in C/C++ showed that the reduction rate is at most 5.7% of the total size, and to a commercial COBOL system, it is at most 15.4%. For this approach, we have gotten initial but very positive feedback from industry collaborators.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**;

KEYWORDS

Code Clone, Refactoring, Size Estimation

ACM Reference Format:
Norihiro Yoshida, Takuya Ishizu, Bufurod Edwards III, and Katsuro Inoue. 2018. How Slim Will My System Be? Estimating Refactored Code Size by Merging Clones. In *ICPC '18: ICPC '18: 26th IEEE/ACM International Confernece on Program Comprehension , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3196321.3196353

1 INTRODUCTION

Once software systems are delivered to market, they are continuously used for a long time with various kinds of maintenance activities such as bug fixing and new feature addition, causing code decay and a serious increase in the maintenance cost [12, 28, 31, 32]. The maintenance cost of a software system is generally determined by the scale and complexity of the system, so the users of huge and decayed systems need to pay a huge amount of money every year [32]. At certain point of the continuous system's use, the users may evaluate the value of the current running system and determine if they will keep paying the maintenance cost, or abandon the current systems and rebuild a new system [34].

When we evaluate the current system value, knowing the size of the system is a fundamental and essential step. We would easily measure the system's sizes by the tools for counting LOC (Lines Of Code), but it would not be sufficient. The system may contain much unused code [11] or it may implement very complex architecture and algorithm. Or it may include a lot of code clones created over its evolution [14, 27].

We are interested in the effect of existence of code clone to the maintenance activities. In industry, so-called "software maintenance business" is very popular[20, 24], where a software maintenance company takes over the maintenance of the system which was not developed by itself. The maintenance activities might include system operation, bug fixes, refactoring of current system and these activities could lead to future restructuring or re-building (migration) of the system. In this business, the maintenance company analyzes the current system, and it should know various characteristics, such as system size and complexity, in order to properly estimate the maintenance cost and to construct a good maintenance plan. Among those various characteristics, code clone information is essential and important.

Along the evolution of the system, a lot of code clones might have been created, and many hidden dependencies among them would be generated, so that the maintenance becomes difficult and the maintenance cost will be increased. Those code clones might be easily refactored by a simple merge of those clones, so that the total system size and also the maintenance cost could be reduced. In this manner, the existence of code clones heavily affects the maintenance activities and their costs.

This paper proposes a novel approach to estimating the system size after refactoring by merging code clones. We call this clone merge refactoring *slimming* in this paper. The obtained estimation size strongly helps to make the maintenance plan and to predict the maintenance cost.

Let S denote the whole source program of a system, and $|S|$ represent the total size of S. S would contain code clones and we could remove all or part of those clones by the merge refactoring, producing a new slimmed system S'. To know the effectiveness of slimming, we define metrics *reduction* $r = |S| - |S'|$ and *reduction rate* $r/|S|$ by slimming. If the reduction rate is close to 0, this means that the system has little room to reduce its size by merging clones. On the other hand, if it is relatively large, say, 40%, then the system contains a lot of clones and by merging those clones, the system size could be reduced to 70%.

We can consider various approaches to slimming, and based on those approaches, the reduction rate would change largely. In this paper, we will propose three slimming methods, *Basic*, *Complete*, and *Heuristic* Methods. Basic Method is a simple merge only applied to function clones. Complete Method is a greedy method to eliminate all the code clones including overlapping and embedded clones. Heuristic Method employs heuristics for selecting merge candidates.

These methods do not guarantee the actual existence of S'. Some of the clone merging might be infeasible or impractical due to the programming language constraints or readability of program. However, those methods and their reduction rates will give the maintainer very good indicator for future maintenance overhead. If the reduction rate is high, there would be a chance to reduce the system size and its maintenance cost by the slimming. If it is low, we would need to continue current maintenance efforts.

Basic Method will show a lower bound of the reduction rate, which could be relatively easily accomplished by the simple function-level clone merging. Complete Method will indicate an upper bound of the reduction by the clone merging. Heuristic Method will provide a practical rate of other two methods.

We have applied these methods to various Open Source Software (OSS) systems are written in C/C++ and investigated the effects of the slimming methods. From the application results, we have known that the reduction rate is from 0.02% to 5.7% depending on the targets and the slimming methods.

Also, we have applied these methods to analyze an actually running business system written in COBOL, and found that reduction rates are from 8.1% to 15.4%.

The main contributions of this paper are as follows:

- We have formulated a novel problem, slimming, for software maintenance, which estimates the code size after the refactoring of code clones.
- We have devised three different methods for the slimming, Basic, Complete, and Heuristic, which provide a lower bound, an upper bound, and the modest rates of the reduction respectively.
- We have applied those methods to many OSS systems in C and found the tendency of the code clone rate and the reduction rate. Also those are applied to a commercial system in COBOL, which showed different tendency from OSS application.

The organization of this paper is as follows. Sec. 2 will introduce overviews of code clone and refactoring. In Sec. 3, Basic, Complete, and Heuristic Methods will be described respectively. Sec. 4 will show applications to OSS systems in C/C++, and Sec. 5 will present an application to a commercial system in COBOL. In Sec. 6, we will discuss our approaches and result, and also in Sec.7, we will show the relation of our work to previous works. Sec. 8 will discuss threats to validity, and Sec. 10 will conclude our discussions.

2 BACKGROUND

2.1 Code Clone Detection

A code clone is a code fragment that has identical or similar code fragments to it in the source code [22]. So far, a lot of code clone detection techniques have been developed to capture various aspects of source code similarity [18, 22, 29].

For the detection of syntactically identical or similar code fragments, token-based and tree-based approaches detect identical token sequence and similar syntax tree in source code, respectively [18, 22]. These approaches are able to detect useful clones (i.e., code fragments to be merged [15], inconsistent code clones that are suspected to include a bug [19]).

In this research, we use a token-based detection tool *CCFinderX* [21], as a clone detection tool for type 1 and 2 clones[1]. It reports clones in the form of *clone pair* (a pair of code clones, i.e., code fragments in clone relation) or *clone set* (i.e., a set of code clones identical or similar to each other).

2.2 Refactoring Code Clone

Clone refactoring is a series of the code transformations to merge similar parts of source code into a single program unit (e.g., Java method, C++ function). It is aimed at improving the maintainability of the source code [35].

Several approaches have been proposed for the identification and the categorization of clone refactoring opportunities in source code. Balazinska et al. [1] proposed an approach for supporting clone refactoring by categorizing code clones based on the differences between them. Baxter et al. [4] have developed a clone detection tool CloneDR based on AST similarity. CloneDR derives only syntactically-complete clones that can be easily refactored. Hotta et al. [17] focused on Form Template Method refactoring pattern [13] and proposed a specialized approach to identifying its opportunities. For the prioritization of clone refactoring opportunities, Higo et al. [15] and Choi et al. [7] proposed metric-based approaches respectively.

3 ANALYSIS OF CODE CLONE RATE

In this section, we introduce the definition of code clone rate in this study.

A cloned line is defined as a line that included in at least one code clone. $CLines(f)$ and $Lines(f)$ denote sets of cloned and all

[1]Type 1 clones are syntactically equivalent code fragments and type 2 are the same except for identifier names. type 3 allows changes of line insertions and deletions, and type 4 are semantically equivalent fragments[5, 33]

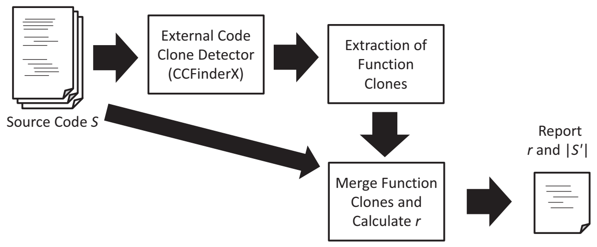
Source Code S
External Code Clone Detector (CCFinderX)
Extraction of Function Clones
Merge Function Clones and Calculate r
Report r and $|S'|$

Figure 1: Basic Slimming Method

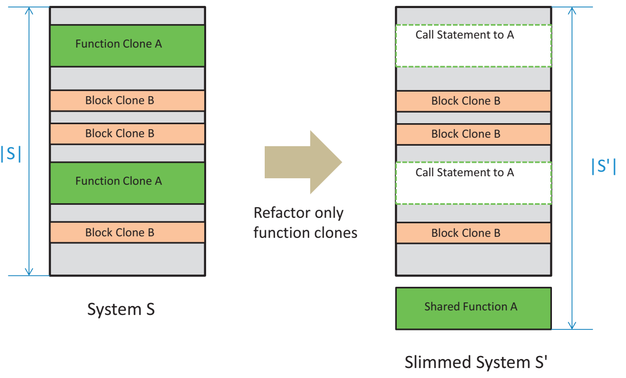

Figure 2: A Simple Example of Basic Method

lines in a file f, respectively. We define clone ratio $ROC(F)$ of a file set $F = (f_0, f_1, \ldots, f_n)$ as follows.

$$ROC(F) = \frac{\sum_{i=1}^{n} |CLines(f_i)|}{\sum_{i=1}^{n} |Lines(f_i)|}$$

The detection of type 1 and 2 clones in source code is a longest common substring problem where a string is a list of tokens in a file [5]. This means that a cloned line is often included in multiple code clones that belong to different clone sets, or we can say that the clones are *overlapped*. Here, let $Lines(C)$ denote a set of lines of all code clones in a clone set C. Then the following formula is established,

$$\sum_{i=1}^{n} |CLines(f_i)| \leq \sum_{j=1}^{m} |Lines(C_j)|$$

where $C_1, \ldots C_{m-1}, C_m$ are detected clone sets. This is because the right-hand side makes multiple count of cloned lines in source code. Developers should not regard the right-hand side value as the total amount of cloned lines because it is overestimated. Here in this paper, the size of code clones always means the left-hand side definition, without counting the duplication of the overlapped clone lines.

Kapser and Godfrey reported that developers make code clones due to the inexpressiveness of a programming language [23]. It is difficult to merge those code clones. A metric RNR(C) (Ratio of Non-Repeated tokens) is developed for clone set C, and Higo et al. found out that RNR(C) metric is effective to filter out code clones that are caused by language inexpressiveness [15]. A clone set whose RNR(C) is low means that code fragments in C mostly consist of repeated token sequences, such as repeated *if then else* statements or repeated simple assignment statements.

4 PROPOSED APPROACHES

4.1 Basic Method

In this section, we will describe a simple and fundamental method to slim the system by merging only function clones. The objective of presenting Basic Method is to show a lower bound of the code reduction with a simple strategy easily performed by developers.

In Fig.1, the overview of Basic Method is presented. It takes a set of source code S for a system as an input, and then it analyzes the code clones inside S, by using an external code clone detector. In this research, we will use *CCFinderX*[21], but we can replace it with any other clone detectors which report code clone pairs of type 1 and 2[33]. The reason for using only type 1 and 2 information is that merging type 3 and 4 clones are not so straightforward and generally very difficult.

CCFinderX is used with the proper configuration such that the detectable minimum token length is 50 and RNR(C) is 0.5. By these settings, we can eliminate the report of unnecessary code clone pairs for the following estimation process.

The output of the detector is the list of code clone pairs. These code clones include various kinds of code fragments such as a sequence of multiple statements, a part of a code block, a complete function, or a complete file. From the set of these code clones, we extract only ones of complete functions. This can be easily performed by a simple syntax analysis of the detected clones.

The extracted functions are the target of slimming, so the sizes of those functions are counted. Also, the number of the instances of the merged functions are counted. Fig.2 shows a simple example of Basic Method. In this example, there are two groups of code clones in a system S. Clone A is a complete function structure, and clone B is a code fragment making a code block (not complete function). Here, we consider that we can merge all instances of A into a single shared function, by creating a new shared function for A and replacing each clone instance for A with a calling statement and its appropriate parameter setting. Other clones which are not complete functions such as B are not merged and kept as they are. Finally, we compute the total reduction size r and the total size of the slimmed system $|S'|$, by generalizing A to any function clone set C_i, as follows.

$$\begin{aligned} r &= \sum_{C_i \in CC} \sum_{c_j \in C_i} (|c_j| - Call) + \sum_{C_i \in CC} (|C_*| + Init + End) \\ |S'| &= |S| - r \end{aligned}$$

Here C_i is a function clone set in all of clone sets CC in system S, and c_j is an instance of a clone set C_i. Also $Call$ is the size of the calling statement to the shared function. $|C_*|$ is the average size of C_i, and $Init$ and End are the sizes of the initialization and termination statements of the shared function, respectively. $Call$, $Init$, and End could be changed based on different clone set C_i, but we could simplify our discussion by assuming all of those as one line. They can be changed by setting the parameters of our estimation tool.

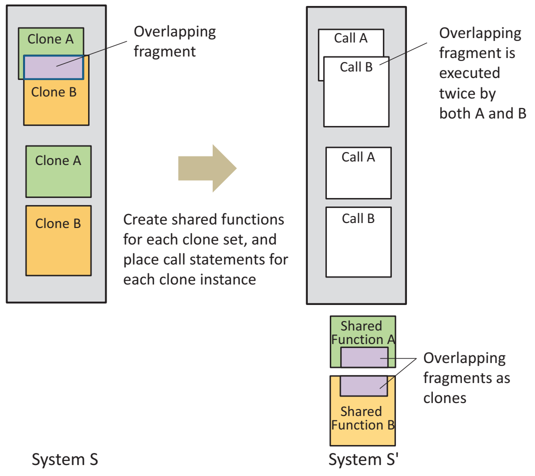
Clone A
Clone B
Overlapping fragment
Clone A
Clone B
Create shared functions for each clone set, and place call statements for each clone instance
Call A
Call B
Overlapping fragment is executed twice by both A and B
Call A
Call B
Shared Function A
Shared Function B
Overlapping fragments as clones
System S
System S'

Figure 3: Merge Clones without Care of Overlapping

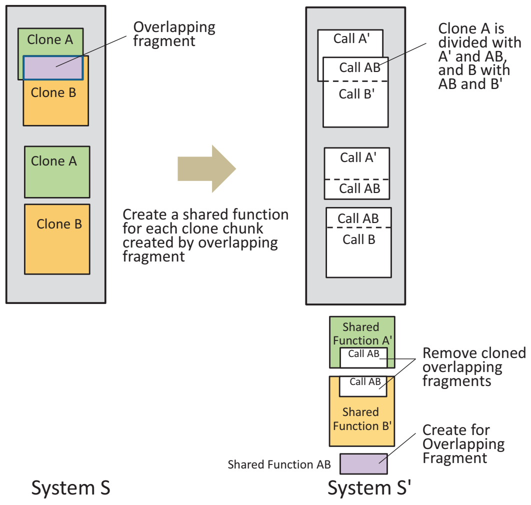
Clone A
Clone B
Overlapping fragment
Clone A
Clone B
Create a shared function for each clone chunk created by overlapping fragment
Call A'
Call AB
Call B'
Clone A is divided with A' and AB, and B with AB and B'
Call A'
Call AB
Call AB
Call B
Shared Function A'
Call AB
Call AB
Shared Function B'
Remove cloned overlapping fragments
Shared Function AB
Create for Overlapping Fragment
System S
System S'

Figure 4: Merge Clones with Cares of Overlapping

The final output of this method is the resulting r and $|S'|$ in LOC associated with the information of the mergeable function clones.

For example, libcurl is an OSS library written in C for URL transfer. The total system size $|S|$ in LOC (Lines of Code) excluding comment lines, white spaces, and new lines is 107.5K. It contains 11.0K code clone lines in total. After applying Basic Method, we got reduction r of 1.8K and $|S'|$ became 105.7K, and the reduction rate was 1.6%. These will be explained in detail later in Sec.5.

4.2 Complete Method

Complete Method is designed to aim at merging and removing greedily all code clone instances reported from the code clone detector. This method is complete in the sense that the refactored system S' contains no code clones reported from the clone detector, and it gives an upper bound of the reduction rate of slimming. Note that the other methods, Basic and Heuristic, may leave some of the code clones in S'.

The strategy of merging here is that any code clones in the list should be merged and no duplicated code fragments should remain in the output. Thus, any kinds of clone fragments, such as a complete function, partial and complete code block, and sequence of statements, are the target of merging. Since we are interested in only the maximum reduction of the system size in this research, we do not care about the feasibility and usefulness of this merging in practice.

Fig. 3 presents a straightforward strategy for merging the input clone list. In this figure, we assume that there are two sets of clones, A, and B. Then we prepare new shared functions for A and B, and the clone instances for A and B are deleted and the call statements to those shared functions are placed.

This strategy has two drawbacks if A and B overlap each other, i.e, they share a code fragment[2]. The simple placement of the call statements to the shared functions causes the execution of the overlapping fragment twice by the execution of A and B. Also, the shared functions of A and B make a code clone pair same as the overlapping fragment, which violates Complete Method policy of removing all clones.

Therefore, we have devices a method of dealing with such overlapping of code clones. Fig. 4 illustrates this method with the same example clones. This method detects an overlapping code fragment of clones A and B, and we employ a new shared function AB for the overlapping fragment.

The overview of Complete Method is presented as follows. The detail is described in our technical report[3].

(1) Clone Detection. Execute an external clone detector. We use CCFinderX as before, and get the list of clone pairs.
(2) Overlapping Detection. Each code clone instance is located onto the source code, and the overlapping fragments are identified.
(3) Clone Chunk Construction. Each code clone instance is divided into code fragments called *clone chunks* which internally contain no partial overlapping fragment with any other clones.
(4) Construction of Shared Functions. For each clone set, a set of shared functions are created based on the finest partition of each clone chunk.
(5) Removing Clones in Shared Functions. An overlapping fragment forms a clone pair in the shared functions, as mentioned above. Those clones are identified and removed.
(6) Size Calculation. The reduction size r and the slimmed system size $|S'|$ are computed by the following equations.

$$\begin{aligned} r &= \sum_{\beta \in CH} \sum_{\beta_i \in \beta} (|\beta_i| - Call) + \sum_{\beta \in CH} (|\beta_*| + Init + End) \\ |S'| &= |S| - r \end{aligned}$$

Here, CH is a set of all clone chunks, β is the set of the same clone chunks, $|\beta_i|$ is the size of an instance of clone chunk,

[2] For simplicity of discussion, here we do not consider the code clones of this overlapping fragment possibly contained in other instance of A and B. Even with such clones, the following discussions are not affected

[3] https://goo.gl/BS4gih

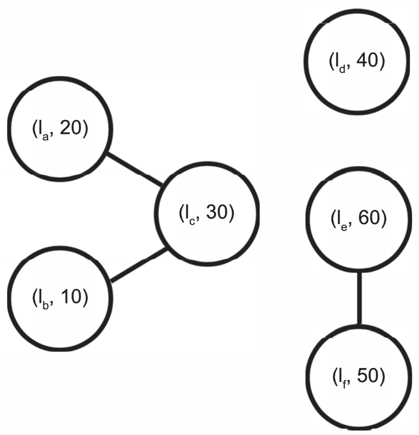

Figure 5: Graph for overlap relationship between clone sets

and $|\beta_*|$ is its average size. *Call*, *Init*, and *End* are the calling, initialization, and termination statements for clone chunks respectively.

For the same example target libcurl, we get reduction r of 6.1K LOC and $|S'|$ is 101.4K LOC. The reduction rate is 5.7%, which is 3.7 times higher than Basic Method.

Table 1: Analysis Target OSS Systems

System	Use	Ver.	Total KLOC	Clone KLOC (%)
git	Distributed versioning	2.9.2	157.4	1.9(1.2)
libcurl	Client-side URL transfer	7.50.0	107.5	11.0(10.2)
Skynet	Game framework*	1.0.0	31.4	0.6(2.0)
Linux	Operating system	4.5.3	11,852.6	864.2(7.3)

Note that Total and Clone are sizes in K lines of source code excluding comments, spaces, or new lines.
∗ https://github.com/cloudwu/skynet

Table 2: Execution Time for Analysis

System	Clone Detection(s)	Basic(s)	Complete(s)	Heuristic(s)
git	87.1	5.7	0.7	4.5
libcurl	52.8	5.6	0.6	3.9
Skynet	18.3	1.1	0.2	0.9
Linux	1.5 hours	484.1	103.0	407.5

The execution time for each method does not include the clone detection time.

Table 3: Reduction by 3 Slimming Methods

System	Basic(% against total)	Complete(%)	Heuristic(%)
git	63(0.04%)	1,209(0.8%)	681(0.4%)
libcurl	1,766(1.6%)	6,116(5.7%)	4,876(4.5%)
Skynet	6(0.02%)	246(0.8%)	212(0.7%)
Linux	117,025(1.0%)	531,285(4.5%)	372,453(3.1%)

Source Lines of Code

4.3 Heuristic Method

In this section, we regard the problem of estimating code size by merging code clones as a combinatorial optimization problem with constraints. The basic idea here is to select and merge only one of the overlapped clones. The overlapped clones not selected are left as they are without any merging. For the case of Fig.3, we may select clone A for merging, and so we will abandon B. By the selection, the reduction rate would decrease but the actual refactoring will be much easier than Complete Method. The selection strategy is to maximize the objective function, i.e., to maximize the reduction size in our case.

Let $D \equiv \{D_1, D_2, \ldots, D_{n-1}, D_n\}$ denote the set of all detected clone sets and $D' \equiv \{D'_1, D'_2, \ldots, D'_{n-1}, D'_n\}$ denote a sublist of D. When all of clone sets in D' are merged, total reduction size $reduction(D')$ can be estimated as follows[4].

$$reduction(D') = \sum_{i=0}^{|D'|} (CLines(D'_i) - average(D'_i))$$

where $CLines(D'_i)$ denotes a set of lines that are involved in clone set D'_i and $average(D)$ denotes the average number of lines of code clones in clone set D'_i.

Then the heuristic method in this paper is approximately maximizing the objective function $f = reduction(D')$ by choosing D' from D subject to the following constraint condition $cond$,

$$\forall D_i, D_j \in D' : CLines(D_i) \cap CLines(D_j) = \phi$$

where $CLines(D_x)$ denotes a set of lines that are involved in a code set D_x.

Since maximizing f is an NP-hard problem, we use a greedy algorithm as meta-heuristic to reduce the computational complexity of this combinatorial optimization.

For the greedy algorithm, we define the following function:

$$overlap(D_i, D_j) = \begin{cases} true \\ (CLines(D_i) \cap DLines(D_j) \neq \phi) \\ false \\ (otherwise) \end{cases}$$

$$reduction(D_x) = CLines(D_x) - average(D_x)$$

The heuristic method comprises the following steps:

Step1: Make a sorted list $L = [l_0, l_1, \ldots, l_{n-1}, l_n]$ of all detected clone sets D based on the descending order of $reduction(D_x)$
Step2: $k \leftarrow 0$; $H \leftarrow [l_0]$
Step3: $k \leftarrow k + 1$
Step4: Add l_k into list H if $\forall i (i < |H| \Rightarrow \neg overlap(h_i, l_k))$
Step5: If $k+1 = |L|$ then calculate reduction size $\sum_{j=0}^{|H|} reduction(h_j)$ from list H otherwise goto Step 3

The graph in Figure 5 illustrates overlap relationship between 6 clone sets. Each vertex has a label $(ID, number)$. ID and $number$ denote the ID and the $reduction(l_x)$ of a clone set. For example, 20 cloned lines are included in clone set l_a. Each edge means that at least one overlap exists between the two clone sets that are connected by the edge.

In the heuristic method, sorted list $L = [l_e, l_f, l_d, l_c, l_a, l_b]$ from clone sets in Figure 5 is made based on the descending order of

[4]For simplicity of discussion, we do not include call, initial, and termination statements here. In actual implementation of the tool, these are counted.

$reduction(l_x)$ at Step 1. And then clone set l_e is added to list H at Step 2. At Step 3, clone set l_f is omitted because $overlap(l_e, l_f)$ is $true$. After that, clone sets l_d and l_c are added to list H because any of those clone sets do NOT overlap with clone set l_e in list H, and clone sets l_a and l_b are omitted because both of those clone sets are overlapped with clone set l_c. Finally, total reduction size $\sum[reduction(l_e), reduction(l_d), reduction(l_c)] = 130$ is calculated.

Using this method, for the case of libcurl, we get 4.9 KLOC reduction and it is 4.5% of the total size.

5 APPLYING TO OSS SYSTEMS

To know various characteristics of the proposed methods, we have following RQ1.

RQ1: What are the reduction rates of Basic, Complete, and Heuristic Methods for popular OSS systems in written C/C++?

We have chosen four OSS systems written in C/C++ from various domains, such as OS kernel, game, network library, and development support tool, as presented in Table 1. It also shows the size of the systems and code clone size in LOC excluding the comments, white spaces, or new lines.

Clone rates are generally small between 1.2% to 10.2%, which are consistent with an earlier report on the clone rate [27]. Git shows very small clone rate, which means that it would have been developed very consistently.

Three tools have been implemented to compute r and $|S'|$ for the target system S. We have executed these tools on a 2.7GHz dual Xeon machine with 24GB memory and the execution time is as shown in Tab.2. As you can see this table, dominant of the total analysis time is the execution time of the code clone detector. Complete Method is faster than others due to its different implementation technique.

Tab.3 and Fig.6 show the result of the reduction rates for three methods associated with the code clone rates. Complete Method shows the highest reduction in three methods, and Heuristic Method follows. Basic Method does not reduce the size significantly, i.e., there are little opportunities for applying function level refactoring of clones.

Fig.7 shows the reduction rate relative to the code clone size. From this figure, we see that Complete Method reduction is about 40% to 65% of the clone size, and Heuristic Method is about 35% to 45%.

The difference of the reduction rates in three methods can be seen more clearly in Fig.8. Basic Method performed only 2.4% to 28.9% reduction of Complete Method, but Heuristic Method could do much better, i.e., 56.3% to 86.2% of the Complete reduction.

Answer to RQ1: Complete Method shows the highest reduction, and Basic Method shows the lowest. Heuristic Method is between them. Basic Method does not reduce effectively.

6 APPLYING TO COMMERCIAL SYSTEM

In the previous section, we have applied our methods to OSS systems written in C/C++. In this section, we will apply to a huge and actual commercial system written in COBOL and will see the result of a different environment based on the following research question.

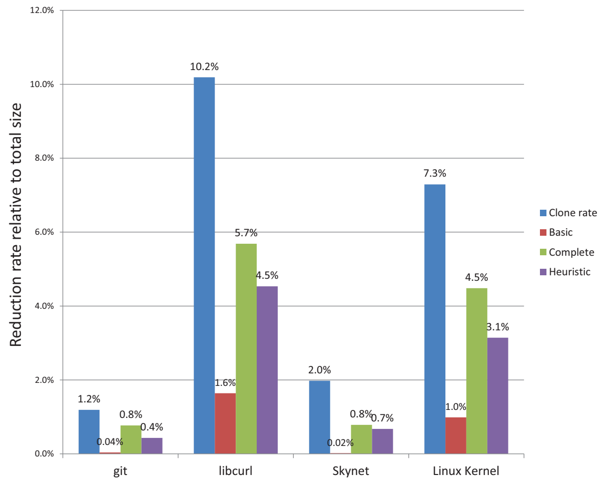

Figure 6: Reduction Rates of 3 Slimming Methods Associated with Clone Rate

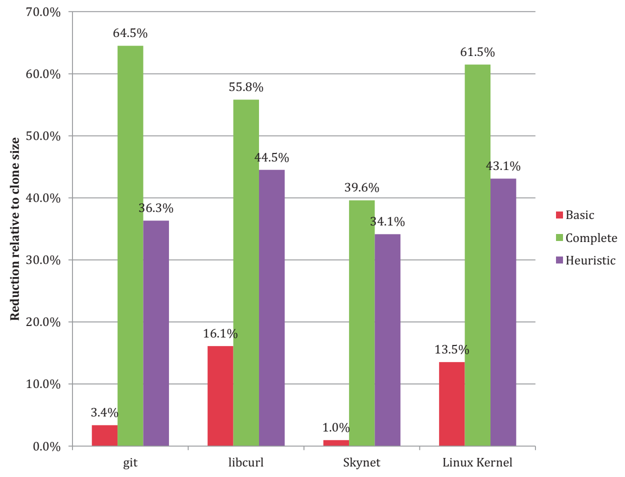

Figure 7: Reduction Relative to Clone Size

RQ2: Is the reduction rate is stable in a different environment?

The target program of this analysis is an account managing system for a chain store, and the total size is about 1.3M LOC in COBOL. Table 4 shows the total size, clone size, and the result of the application of Complete and Heuristic Methods. Basic Method was not applied due to our limited analysis resource. The execution time for this analysis was about 1.5 hours for clone detection and a half minute both for Complete and Heuristic Method applications. These execution times are sufficiently fast and acceptable as a practical tool.

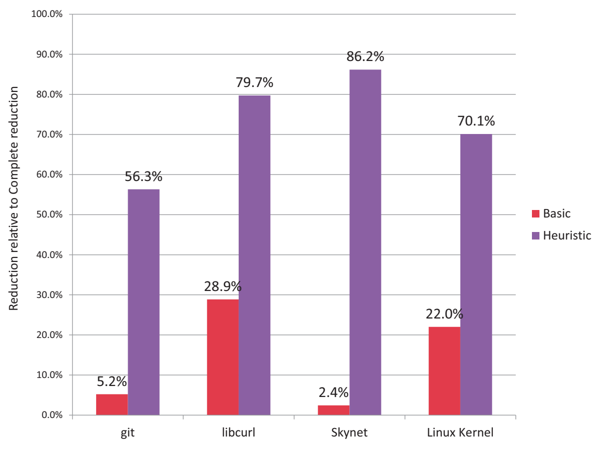

Figure 8: Relative Reduction to Complete Method

Table 4: Reduction Sizes for COBOL System

Total	Clone (% of Total)	Complete (%)	Heuristic (%)
1,259,184	853,271(68.0%)	194,358(15.4%)	102,113(8.1%)

Size in source lines of code and percentage relative to the total.

It is interesting to know that the clone rate is 68% which is fairly higher than the cases of C/C++ where those are around 1.2% to 10.2%. This is consistent with the report such that commercial software systems tend to have very high code clone rate [8].

For this target, the slimming methods were expected to reduce a large number of clones, but the result showed that they reduced only 8.1% and 15.4% of the total size. This suggests that there are a lot of overlapping fragments in cloned codes. They increase the overhead for the chunked procedures in Complete Method, and also it could reduce the merging opportunities in Heuristic Method.

Answer to RQ2: No, the reduction is quite different from the target's domain.

7 DISCUSSIONS

7.1 Slimming Methods

As we have shown in previous sections, each slimming method shows different reduction performance, due to its inherent nature of the algorithm. However, the following relation holds among the total and clone sizes and the reductions sizes.

$$\begin{aligned} &\textit{Total Size} > \textit{Clone Size} > \textit{Complete Reduction} \\ &\quad > \textit{Heuristic Reduction} > \textit{Basic Reduction} \end{aligned}$$

This relation is consistent with our expectation of each method.

Complete Method presents a theoretical upper bound of the reduction size, but performing actual refactoring based on this method would be almost impractical because it creates many small code chunks which will decrease the readability and maintainability of the system.

Basic Method presents a lower bound of the reduction, and it would be relatively easy to perform this refactoring. However, as shown in Sec.5, there is little opportunity to do this in practice.

The Heuristic method proposed here is a good compromise between the reduction and feasibility. We were able to get sufficiently high reduction sizes for OSS in C/C++. However, in the case of the commercial application written in COBOL, it did not provide a sufficient reduction under the very high clone rate. The reason is that the code clones are heavily overlapped each other, which does not allow simple merging of clones.

When we talk about the refactoring code clones, these three methods will be all important tools and they will complement each other. We can present a maximum reduction with a lot of efforts, a minimum reduction with light-weight effort, and an intermediate reduction with effort and performance compromise.

7.2 Feedback from Industry Collaborators

We had discussed the impact of estimating the slimmed system size by code clone merge refactoring, with industry collaborators from two different companies for program maintenance and migration services. The following are their comments on our approach.

- The approach is very important and practical because their customers always ask them (1) the maintenance cost for the current system without any refactoring or migration, and (2) the migration cost and the maintenance cost for the migrated system. This approach gives them a solid theoretical basis for the cost estimation.
- For the case of COBOL system application shown in Sec.6, the result is a little disappointing, because the clone rate is fairly high (68%) and so they had expected a higher reduction rate too. If so, the owner of this system would want to refactor the system or re-build a new system, that will make a big business profit. However, the analysis result showed the limited reduction possibility, so the owner would not be interested in a new investment.
- The approach is limited in the sense that it gives them the reduction sizes of the slimming refactoring, but it does not perform actual refactoring. They would like to have a tool performing actual refactoring for the clones which are safely refactored automatically.

For the last comment, we might be able to implement automatic refactoring tool only for very safe clone fragments. However, even for Basic Method, we have to take care of the difference of variable names in the clones, and so the automatic refactoring might not be straightforward. Also the opportunity of the function clone refactoring would be very limited.

8 RELATED WORKS

8.1 Mining Code Clones and Aspects

As presented in Sec.2.1, there are much research on code clone detection and its application [26, 33]. Higo et al. proposed a proper maintenance process with their tool for measuring and visualizing clone metrics [16]. This tool helps to understand the overall clone rate and individual clone fragment, but it does not provide any hint to actual reduction by removing clones.

High-level clone detection and understanding was proposed by Basit et al. [3]. They have shown a method of detecting architectural clones, and a way of understanding the system organization, although they did not show any approach to slim and simplify the target system.

Aspect Mining is one of the related areas of our work in the sense that it detects common processes placed in different places and merge them as a shared aspect [25]. The main focus of it is to extract and share semantically common processes to give better understandability and maintainability, which is different from our concern on the share of textual common fragments.

8.2 Refactoring

Some of general ideas and efforts for refactoring code clones have been presented already in Sec. 2.2. A sophisticated approach to finding refactoring opportunities using three metrics was proposed by Choi et al. [7]. The approach employs LEN, POP, and RNR where LEN is the length of a clone, POP is the number of instances in a clone set, and RNR is the degree of simply repeated subsequences in a clone. Combining these metrics values, effective refactoring opportunities are explored over all clone detector's output. The objective of this approach is to find effective clone merge opportunities and it is not to estimate and minimize the refactored system size as we do in this research.

Balazinska et al. proposed automatic detection of refactoring opportunity by analyzing code clone differences and their interpretation in terms of programming language entities [2]. Since they intend to perform automatic refactoring, the applicable opportunities are generally limited.

Krishnan et al. formulated the refactoring clones as an optimization problem to minimize and parameterize the difference of clones [30]. The idea could be used for our manual refactoring suggested by three slimming methods.

8.3 Code Compaction

Code compaction is very related topics to our work. It focuses on the generation of smaller programs with the same execution result [9]. Debray et al. proposed a binary rewriting method and tool using compiler optimization technique, especially with interprocedural code transformation and factoring [10], and they get averagely 30% reduction of the executable size. This is an interesting approach to the type 3 code clones in executable binary code, but it requires non-trivial interprocedural control and dataflow analyses on the basic blocks of the binary code and those cannot be applicable to source program targets.

Chen et al. devised a method of transforming cloned fragments with a single entry and multiple exits based on the control flow analysis [6]. They have shown an algorithm to detect mergeable code fragments with multiple exits, and presented an empirical result with about 25% reduction in the average. This assumes the control flow analysis of basic blocks and cannot be directly applicable to the source program targets.

Zastre developed a procedural level abstraction of clone fragments [36], but it is at binary level compaction, and is different from our objective to estimate the slimmed code at the source code level.

9 THREATS TO VALIDITY

Code clones are not always good candidates for refactoring. Some code clones are intentionally created and useful for overall quality of program, and merging those clones might give negative impact to the program maintenance [23]. Our three methods present possible candidates of refactoring for the size estimation, but all of those candidates might not be feasible in such sense, so the resulting size estimation might be overestimated. We might be able to improve this issue to identify clone candidates whose refactoring would be harmful.

For the proposed three methods, we can consider various variants. For example of Complete Method, we have taken a strategy to make many short code chunks, but we could do it differently by treating the overlap fragments with a more complex algorithm, and we might get a more reduction and a better upper bound.

We have used CCFinderX as an external tool of code clone detection for type 1 and type 2 clones, with minimum token length 50 and RNR 0.5. Based on our analysis of the output of CCFinderX, we think that these configurations are effective to remove useless refactoring candidates. We could use different clone detectors with different configurations, and this might affect our application results. Bellon et al. reported the distinction of tool's output for the same target [5]. In the sense of our objective of searching for refactoring opportunities, AST-based clone detector might easily handle the opportunities without specific parameter tuning.

The application to the OSS systems was performed to the limited target set of C/C++ programs. We can extend targets to various domains in the same languages, and also to different language programs. For the commercial program application, we had only one COBOL target due to our limited resources. We understand it is important to extend the application targets and domains and get the customer feedback for the result of our analysis.

In order to verify our approach, the tools for the three methods and the analysis result for the OSS systems are available online[5]. The source code for those systems can be found in those project's repositories. The analyzed data for the COBOL system is shown in Tab.4 in Sec.6, but the source code is proprietary so it cannot be opened.

10 CONCLUSION

In this paper, we have presented the slimming problem for estimation of refactored system size, and have proposed three methods for slimming. Tools implementing these methods have been implemented and applied to OSS systems in C/C++ and also a commercial system in COBOL. The reduction rate for OSS systems varies on the target program, but it is about 60% of clone rate of the target by Complete Method, and 43% by Heuristic Method. For COBOL system, the reduction rate is quite different from the OSS cases. In this case, the system contains 68% of code clones, but the reduction rate even with Complete Method is only 15.4%. This suggests that a high code clone rate does not always guarantee high reduction rate. Further investigation of the target domain and programming language would be needed to clarify the high clone rate and their overlapping situation.

[5]https://goo.gl/Ny5g24

REFERENCES

[1] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, and Kostas Kontogiannis. 1999. Measuring clone based reengineering opportunities. In *Proc. of METRICS 1999*. 292–303. https://doi.org/10.1109/METRIC.1999.809750

[2] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, and Kostas Kontogiannis. 2000. Advanced clone-analysis to support object-oriented system refactoring. In *Proc. of WCRE 2000*. 98–107. https://doi.org/10.1109/WCRE.2000.891457

[3] Hamid Abdul Basit and Stan Jarzabek. 2009. A data mining approach for detecting higher-level clones in software. *IEEE Transactions on Software engineering* 35, 4 (2009), 497–514. https://doi.org/10.1109/TSE.2009.16

[4] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone detection using abstract syntax trees. In *Proc. of ICSM 1998*. 368–377. https://doi.org/10.1109/ICSM.1998.738528

[5] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. 2007. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering* 33, 9 (2007), 577–591. https://doi.org/10.1109/TSE.2007.70725

[6] Wen-Ke Chen, Bengu Li, and Rajiv Gupta. 2003. Code compaction of matching single-entry multiple-exit regions. In *Proc. of SAS 2003*. Springer, 401–417. https://doi.org/10.1007/3-540-44898-5_23

[7] Eunjong Choi, Norihiro Yoshida, Takashi Ishio, Katsuro Inoue, and Tateki Sano. 2011. Extracting code clones for refactoring using combinations of clone metrics. In *Proc. of IWSC 2011*. 7–13. https://doi.org/10.1145/1985404.1985407

[8] James R Cordy. 2003. Comprehending reality-practical barriers to industrial adoption of software maintenance automation. In *Proc. of IWPC 2003*. 196–205. https://doi.org/10.1109/WPC.2003.1199203

[9] Bjorn De Sutter and Koen De Bosschere. 2003. Software Techniques for Program Compaction. *Commun. ACM* 46, 8 (2003). https://doi.org/10.1145/859670.859694

[10] Saumya K Debray, William Evans, Robert Muth, and Bjorn De Sutter. 2000. Compiler techniques for code compaction. *ACM Transactions on Programming languages and Systems* 22, 2 (2000), 378–415. https://doi.org/10.1145/349214.349233

[11] Sebastian Eder, Maximilian Junker, Elmar Jürgens, Benedikt Hauptmann, Rudolf Vaas, and Karl-Heinz Prommer. 2012. How much does unused code matter for maintenance?. In *Proc. of ICSE 2012*. 1102–1111. https://doi.org/10.1109/ICSE.2012.6227109

[12] Stephen G Eick, Todd L Graves, Alan F Karr, J Steve Marron, and Audris Mockus. 2001. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering* 27, 1 (2001), 1–12. https://doi.org/10.1109/32.895984

[13] Martin Fowler. 1999. *Refactoring: improving the design of existing code.* Addison Wesley.

[14] Anfernee Goon, Yuhao Wu, Makoto Matsushita, and Katsuro Inoue. 2016. Evolution of Code Clone Ratios throughout Development History of Open-Source C and C++ programs. *Technical Report of Software Engineering Lab, Dept. of Computer Science, Osaka University* (2016). http://sel.ist.osaka-u.ac.jp/lab-db/betuzuri/contents.ja/1046.html

[15] Yoshiki Higo, Shinji Kusumoto, and Katsuro Inoue. 2008. A metric-based approach to identifying refactoring opportunities for merging code clones in a Java software system. *Journal of Software Maintenance and Evolution* 20, 6 (2008), 435–461. https://doi.org/10.1002/smr.394

[16] Yoshiki Higo, Yasushi Ueda, Toshihro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. On software maintenance process improvement based on code clone analysis. In *Proc. of PROFES 2002*. Springer, 185–197. https://doi.org/10.1007/3-540-36209-6_17

[17] Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto. 2012. Identifying, Tailoring, and Suggesting Form Template Method Refactoring Opportunities with Program Dependence Graph. In *Proc. of CSMR 2012*. 53–62. https://doi.org/10.1109/CSMR.2012.16

[18] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. DECKARD: scalable and accurate tree-based detection of code clones. In *Proc. of ICSE '07*. 96–105. https://doi.org/10.1109/ICSE.2007.30

[19] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. 2007. Context-based detection of clone-related bugs. In *Proc. of ESEC-FSE '07*. 55–64. https://doi.org/10.1145/1287624.1287634

[20] Yasuo Kadono. 2015. *Management of Software Engineering Innovation in Japan.* Springer.

[21] Toshihiro Kamiya. 2010. the archive of CCFinder Official Site. http://www.ccfinder.net/ccfinderxos.html. (2010).

[22] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (2002), 654–670. https://doi.org/10.1109/TSE.2002.1019480

[23] Cory Kapser and Michael W Godfrey. 2006. "Cloning considered harmful" considered harmful. In *Proc. of WCRE 2006*. 19–28. https://doi.org/10.1109/WCRE.2006.1

[24] Koki Kato, Tsuyoshi Kanai, and Sanya Uehara. 2011. Source code partitioning using process mining. In *Proc. of BPM 2011*. 38–49. https://doi.org/10.1007/978-3-642-23059-2_6

[25] Andy Kellens, Kim Mens, and Paolo Tonella. 2007. A survey of automated code-level aspect mining techniques. In *Transactions on aspect-oriented software development IV*. Springer, 143–162. http://dl.acm.org/citation.cfm?id=1793854.1793862

[26] Rainer Koschke. 2007. Survey of Research on Software Clones. In *Duplication, Redundancy, and Similarity in Software (Dagstuhl Seminar Proceedings)*, Rainer Koschke, Ettore Merlo, and Andrew Walenstein (Eds.). Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany. http://drops.dagstuhl.de/opus/volltexte/2007/962

[27] Rainer Koschke and Saman Bazrafshan. 2016. Software-Clone Rates in Open-Source Programs Written in C or C++. In *Proc. of IWSC 2016*. 1–7. https://doi.org/10.1109/SANER.2016.28

[28] Jussi Koskinen. 2015. Software maintenance costs. (2015). https://wiki.uef.fi/download/attachments/38669960/SMCOSTS.pdf

[29] Jens Krinke. 2001. Identifying Similar Code with Program Dependence Graphs. In *Proc. of WCRE '01*. 301–307. https://doi.org/10.1109/WCRE.2001.957835

[30] Giri Panamoottil Krishnan and Nikolaos Tsantalis. 2013. Refactoring Clones: An Optimization Problem. In *Proc. of ICSM 2013*. 360–363. https://doi.org/10.1109/ICSM.2013.47

[31] Bennet P Lientz and E Burton Swanson. 1980. *Software maintenance management.* Addison-Wesley.

[32] Thomas M Pigoski. 1996. *Practical software maintenance: best practices for managing your software investment.* Wiley Publishing.

[33] Chanchal K Roy, James R Cordy, and Rainer Koschke. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming* 74, 7 (2009), 470–495. https://doi.org/10.1016/j.scico.2009.02.007

[34] Harry M Sneed. 1995. Planning the reengineering of legacy systems. *IEEE software* 12, 1 (1995), 24. https://doi.org/10.1109/52.363168

[35] Norihiro Yoshida, Eunjong Choi, and Katsuro Inoue. 2013. Active support for clone refactoring: A perspective. In *Proc. of WRT 2013*. 13–16. https://doi.org/10.1145/2541348.2541352

[36] Michael Joseph Zastre. 1995. *Compacting Object Code via Parameterized Procedural Abstraction.* Master's thesis. Department of Computer Science, University of Victoria. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.51.4235&rep=rep1&type=pdf