On the Effectiveness of Simhash for Detecting Near-Miss Clones in Large Scale Software Systems

Md. Sharif Uddin Chanchal K. Roy Kevin A. Schneider
University of Saskatchewan, Canada
s.uddin@usask.ca, {croy, kas}@cs.usask.ca

Abram Hindle
University of Alberta, Canada
ah@softwareprocess.es

Abstract— **Clone detection techniques essentially cluster textually, syntactically and/or semantically similar code fragments in or across software systems. For large datasets, similarity identification is costly both in terms of time and memory, and especially so when detecting near-miss clones where lines could be modified, added and/or deleted in the copied fragments. The capability and effectiveness of a clone detection tool mostly depends on the code similarity measurement technique it uses. A variety of similarity measurement approaches have been used for clone detection, including fingerprint based approaches, which have had varying degrees of success notwithstanding some limitations. In this paper, we investigate the effectiveness of simhash, a state of the art fingerprint based data similarity measurement technique for detecting both exact and near-miss clones in large scale software systems. Our experimental data show that simhash is indeed effective in identifying various types of clones in a software system despite wide variations in experimental circumstances. The approach is also suitable as a core capability for building other tools, such as tools for: incremental clone detection, code searching, and clone management.**

Keywords: software clones, clone detection, similarity hashing, fingerprinting, simhash

I. INTRODUCTION

Cloning is a common phenomenon found in almost all kinds of software systems. The larger the system, the more people involved in its development and the more parts developed by different teams result in an increased possibility of having evolving clone code. Several studies suggest that as much as 7-23% of code of a software system is cloned code [1, 17]. The presence of clones may lead to unresolved bug and/or maintenance related problems by increasing the risk of update anomalies [5].

Existing popular techniques [2, 12, 14] have several deficiencies, such as not supporting the detection of Type-3 near-miss clones where lines could be modified, added and/or deleted in the copied fragments, and not scaling adequately to handle clone detection in large systems [10]. One major problem for clone detection on large corpora is the performance of querying and retrieving possible clones. One way to improve this performance is to use near constant time techniques of querying a dataset for similar entities. For exact-duplicate matching, a simple match of fingerprints suffices and a common hash function is suitable for that, but it does not work that well for inexact matches. If one wants partial matches and yet wants a constant time operation, the hash function has to map similar documents close together. Manku et al. [15] showed that Charikar's *simhash* [3] is a very effective hashing technique for developing a near-duplicate detection system for a multi-billion page repository. *Simhash* has been used successfully in different areas of research, such as text retrieval, web mining and so on [7, 8, 18]. However, to the best of our knowledge, no clone detection study has been conducted using *simhash*. The objective of this study is to see how effective *simhash* is for software clone detection, especially in detecting Type-3 near-miss clones from large scale software systems. Thus, the research questions we would like to answer in our study are:

1) Is *simhash* feasible to be used in clone detection, especially in detecting Type-3 clones from a large codebase?
2) Does a *simhash* based technique yield faster detection of clones in large codebases?
3) What are the possible problems/limitations of using *simhash* in clone detection and how can we overcome those limitations?

We designed a *simhash* based clone detection approach, the performance of which will in turn be a measure of the effectiveness of *simhash* in this area. To evaluate that effectiveness, we chose an existing clone detection tool called NiCad [21] that uses the UNIX *diff*[1] algorithm to measure code similarity. NiCad is a state of the art clone detection tool that returns structurally significant clone fragments (e.g., functions or blocks). It has powerful features for source code pre-processing and normalization along with context sensitive transformations. Diff-based comparison with these features enables NiCad to detect both exact (Type-1) and near-miss (Type-2 and Type-3) clones with high precision and recall [19]. We replaced NiCad's *diff* based detection engine with our *simhash* based approach (marked as a dashed rectangle in Figure 1) that includes an indexing strategy and a data clustering algorithm. We call this new variant *simCad*. Using NiCad as the benchmark, we evaluate the effectiveness of simCad for large scale clone detection both in terms of detection time and the number of cloned fragments detected. This provides us with some insight into the effectiveness of the *simhash* based technique for clone detection and helps answer the research questions above. Third party clone detectors were not used intentionally since our primary objective is to evaluate the feasibility of simhashing for clone detection and we focused on structural (such as functions or blocks) clones. We chose to compare simCad with NiCad, not only because NiCad gives high precision and recall but also because both NiCad

[1] http://en.wikipedia.org/wiki/Diff (accessed on June 30, 2011)

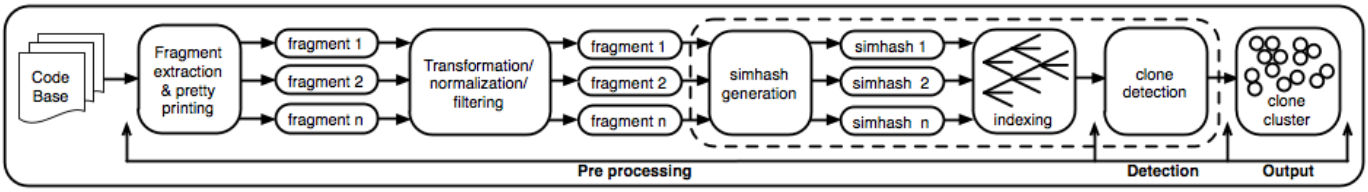

Figure 1. Clone detection process in simCad

and simCad use the same pre-processed structural code fragments in the comparison phase and thus comparing the results of using the *simhash* technique is straightforward, which is one of our major objectives.

***Contributions**:* This paper presents a novel approach for structural clone detection based on fingerprinting of source code using a hash technique called *simhash*. A multi-level indexing scheme is also proposed to organize the pre-processed codebase on which the clone detection algorithm is applied. The indexing scheme speeds up the potential clone search and allows the approach to be scalable by maintaining the indices in persistent storage (e.g., a database). This makes the approach capable of being used by an incremental clone detection tool or by a real-time code search engine. Our experimental analysis shows the effectiveness of *simhash* in clone detection and how it enables faster near-miss clone detection in large corpora.

***Overview**:* The rest of this paper is organized as follows. Section II covers some general terms and definitions of clones. Section III describes our proposed technique. Section IV provides the complexity analysis of the detection process. Section V presents the implementation and experimental results analysis. Section VI covers some related work on the application of fingerprint based similarity and finally, Section VII concludes the paper.

II. General Terms & Definitions

A. Software Clone and Clone Detection

There is as yet no universal definition for a *software clone*. It is usually described as portions of source code or code fragments at different locations in a software project/program that are identical or very similar. Being 'similar' may also be defined in various ways, and can refer to textual, structural or semantic aspects of the source code. Selim et al. [22] defined code clones as sets of syntactically or semantically similar code segments residing at different locations in the source code. In the words of Baxter et al. [2]: "Clones are segments of code that are similar according to some definition of similarity."

There is also no precise minimum size for a code clone. Clone studies define this size differently in terms of either number of lines, tokens or AST/PDG nodes with respect to their experimental context [20]. Thus, given the various definitions of a software clone, defining and measuring code similarity is an important aspect of any clone detection technique. That is, a key issue is the manifestation of source code similarity and the success of such a tool mainly involves how efficiency and accurately it can perform the task of identifying similar code fragments based on its self-defined similarity measurement.

B. Clone Types and Clone Groups

In recent literature, clones are broadly classified as one of the following four types.

Type-1 (Exact Clones): Code fragments, which are identical without considering the variations in white space and comments.

Type-2 (Renamed/Parameterized Clone): Code fragments, which are structurally/syntactically similar but may contain variations in identifiers, literals, types, layouts and comments.

Type-3 (Gapped Clones): Code fragments with modifications in addition to those defined for Type-2 clones, such as: insertion, deletion or the modification of a statement. Note that for Type-3 clones, the detection tool might requires setting up a boundary of acceptable differences (between two fragments of a Type-3 clone pair) that occurs for such modifications in terms of line numbers, token numbers, amount of text and so on.

Type-4 (Semantic Clone): Code fragments with the same functionality with or without being textually similar.

In this paper, by *near-miss clones* we mean both Type-2 and Type-3 clones with an emphasis on Type-3. Detection of Type-4 clones is not within the scope of this paper.

The output of a clone detection tool is usually in terms of code fragment groupings, which is either by clone pair or by clone cluster along with their location information. A *Clone Pair (CP)* is a pair of code portions or fragments that are similar to each other under a defined similarity measure. A *Clone Cluster (CC)* is a group of code portions or fragments which are pair-wise similar (inside that group) under a defined similarity measure.

III. Proposed Technique

As noted in the introduction, we will evaluate the effectiveness of *simhash* by evaluating the performance of simCad compared to NiCad. Figure 1 gives an overview of end-to-end clone detection process in simCad which uses a distance based similarity detection mechanism developed by Manku et al. [15] based on the fingerprinting technique called *simhash*. This mechanism has been proven effective in various domains such as data mining and web engineering, where it usually requires dealing with billions of dataset records [15]. In our proposed approach, we tried to deal with the clone detection process from a data mining perspective. We employ a data clustering algorithm with multi-level index based searching which enables fast detection of clones. From the experimental results presented in Section V, we see that simCad is very effective at fast clone detection in a large dataset. Our experimental implementation running on a standard desktop computer shows that the detection part of the process takes a fraction of a second (cf. Table 4) to detect

```
Algorithm simhash(doc, n)
doc: document for which simhash is computed
n: length of the desired hash size in bits

1.  doc is split into tokens (words for example) or super-tokens (word tuples)
2.  weights are associated with tokens (for example: frequency count)
3.  v = vector of size n, initialized to 0
4.  for each token in doc
5.    token_hash = make_n-bit_simple_hash (token)
6.    for i = 1 to n do
7.      if( i^th bit of token_hash = = 1)
8.        v[i] = v[i] + weight(token)
9.      else
10.       v[i] = v[i] - weight(token)
11. bit_vector = vector of size n, initialized to 0
12. for i = 1 to n do
13.   if(v[i] > 0)
14.     bit_vector [i] = 1
```

Figure 2. *simhash* algorithm [2]

all the Type-1/Type-2 function/block clones in the Linux kernel v-2.6.38, which has 12.5 million lines of code.

A clone detection tool may follow several phases in its detection process [20]. Similar to NiCad, simCad has three phases: *pre-processing, detection and output generation* (cf. Figure 1), which are discussed as follows.

A. Pre-processing

The pre-processing step sets up the environment and organizes the data over which the detection algorithm is applied. There are four sub-steps in pre-processing as shown in Figure 1 by solid rectangular boxes. The first two sub-steps: fragment extraction with pretty printing and source transformation/normalization (renaming the data-types and identifiers) are similar to those in NiCad, details of which are available in [21]. The remaining two sub-steps: *simhash* generation and indexing are new to our approach and discussed in detail below.

(i) Simhash generation

The core idea of our proposed approach is to determine code similarity using fingerprinting. Fingerprinting is a well-known approach in data processing that maps an arbitrarily large data item to a much shorter bit sequence (the fingerprint) that uniquely identifies the original data. A typical use is to avoid the comparison or transmission of bulky data. We have similar reasons for using this approach in our case. For a large dataset, the approach should reduce the data comparison time and effectively decrease the overall running time of a clone detection process. In our approach each code block will be transformed into an *n*-bit fingerprint, the *simhash* value of that block.

Charikar's *simhash* [3] is a dimensionality reduction technique that maps high-dimensional vectors to small-sized fingerprints [15]. Apart from being an identifier of source data, this technique has the property that fingerprints of near-duplicate data differ only in a small number of bit positions. As for a *simhash* fingerprint *f*, Manku et al. [15] developed a technique for identifying whether an existing fingerprint *f'* differs from *f* in at most *k* bits. Their experiment shows that

```
int sum (int num[], int len)
{
 int total = 0;
 for(int i=0; i < len; i++)
  total += num[i];
 return total;
}
(a)

int sum (int num[], int len)
{
 int total = 0;
 for(int i=0; i < len; i++)
  total += num[i];
 return total;
}
(b)

int sum (int num[], int size)
{
 int sum = 0;
 for(int i=0; i < size; i++)
  sum += num[i];
 return sum;
}
(c)

int sum (int num[])
{
 int total = 0;
 int len = sizeof(num) / sizeof(int);
 for (int i = 0; i < len; i++)
  total += num[i];
 return total;
}
(d)

void swap ( int & a, int & b )
{
 int temp = a;
 a = b;
 b = temp;
}
(e)
```

	32-bit simhash for the code fragments (a) to (e)		
(a)	11111101001111011110101011101000	Hamming Distance	Clone Type
(b)	11111101001111011110101011101000	sim (a, b) = 0	Type-1
(c)	11011101001111011010101111010 10	sim (a, c) = 3	Type-2
(d)	11110101001111001010101011101100	sim (a, d) = 4	Type-3
(e)	10001011110111000101100101111011	sim (a, e) = 18	Not a clone

Figure 3. Distance based similarity detection

for a repository of eight billion pages, 64-bit *simhash* fingerprints and $k = 3$ are reasonable parameters. In our proposed technique, we will refer this notion of bit difference (value of *k*) using a term called *SimThreshold* as a measurement of similarity, i.e., the lower the value of *SimThreshold,* the more similar the code blocks are. Figure 2 shows the pseudocode of the *simhash* algorithm using which a hash fingerprint is generated for each of the extracted code fragments. We have considered words (separated by whitespace) in pretty-printed source as tokens. For generating an n-bit simple hash (Figure 2, line 5), we have used a 64-bit Jenkin hash function[3] (from a choice of several cryptographic and non-cryptographic hash functions) based on our finding that it yields better simhash values than others with respect to similarity preserving behaviour (a small change in the source results in a small change in the simhash bits). This hash value is then used in the detection algorithm to detect code similarity that saves significant time by avoiding raw source string comparison.

(ii) Multi-level indexing

In our proposed approach, a two level indexing scheme has been introduced to organize the *simhash* data generated in the previous step. The goal of the data organization is to speed up the neighbour search query in the clone detection phase presented in the following section. The indexing is done first by the size of the code fragment in terms of lines of code, and then by the number of 1-bits in the binary representation of the simhash fingerprint. Thus, each of the first level indices points to a list of second level indices. Each of the second level index in turn points to a list of simhash values having the same number of 1-bits in its binary form (note, the position of the 1-bits might be different, which is not considered for this count). These *simhash* values in the final list are corresponding to the code fragments which are similar in size (lines of code). Figure 4 illustrates how our two level indexing scheme works.

[2] http://d3s.mff.cuni.cz/~holub/sw/shash (accessed on June 30, 2011)

[3] http://www.burtleburtle.net/bob/hash/doobs.html

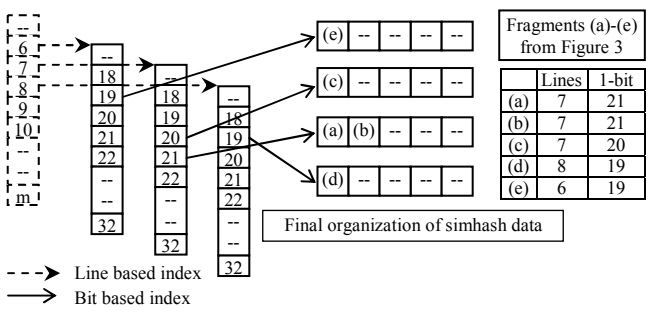

Figure 4. Indexing strategy used in simCad.

At the first level (line based index), the size of a code fragment in lines of code is used as a key to map the corresponding simhash data item. As mentioned earlier, the code has been pretty-printed in the pre-processing phase before computation of *simhash*. This allows us to use a data indexing strategy to store the *simhash* values and helps improve performance by avoiding computations that end up being unsuccessful. Since the *simhash* values are indexed according to their actual line sizes, a search for Type-1 and Type-2 clones does not require to compare the *simhash* of two fragments that have different line sizes. That is, we only need to consider *simhash* data for potential candidates that are computed from the code fragments having a line size that is the same as the new/unclassified *simhash* data item for which a search is being made. However, for Type-3 clones, the algorithm neither uses one particular size nor all sizes, instead it searches within a range of sizes. To limit the search, we put a cap on the differences in number of lines between two code fragments of a Type-3 clone pair. Thus the neighbour search is limited to a window of indices having a range, for example, ± 30% of the line size of the corresponding item for which a search is being made.

At the second level (bit based index), the count of 1-bits in the code fingerprint (*simhash* value) is used as an index for ordering those fingerprints. Since we are using *Hamming Distance* [15] between *simhash* values (equal to the number of bit positions at which the corresponding bits are different in their binary form, i.e., the value of *k* as discussed in the previous section) as the distance measure, the total count of the number of '1' or '0' bits in the *simhash* value can be a useful index. For example, if a neighbour search is made for a *simhash* value having 10 1-bits in its binary representation, all the Type-1 clones should have the same number of 1-bits in their fingerprint, and so is the case for Type-2 clones since when a transformation is applied in the pre-processing step, changes in identifier and function names are eliminated. However, the fingerprint of some other dissimilar code fragments might have the same number of 1-bits, but they would not be considered Type-1 clones because they might have those bits in different bit positions, yielding a *Hamming Distance* greater than zero. For Type-3 clones, this indexing scheme will also accelerate the searching by allowing a search window range (1-bit count in search item ± *SimThreshold*). For example, when *SimThreshold = 5* and the number of 1-bits in the search item is 10, then the algorithm will consider comparing fingerprints having 1-bits within a range from 5 to 15 and ignore others. In Section V we have shown how this indexing strategy improved the runtime performance of the clone detection process.

B. Clone Detection

This phase is responsible for identifying and grouping similar code fragments into clusters, which are called *Clone Clusters*. The grouping is done based on the *Hamming Distance* among the *simhash* values of code fragments extracted from the codebase and the value of *SimThreshold* controls the boundary of a *Clone Cluster*. The process is similar to a typical data clustering algorithm that partitions the data based on the similarity of individual records; the more similar the data, the more likely that they belong to the same cluster. The main goal is to identify clusters that maximize the inter-cluster distance and minimize the intra-cluster distance, so that we obtain clearly distinct groups of similar entities. Therefore, in our case, once we get all the *simhash* values of the code fragments, the problem of detecting code clones is essentially clustering the *simhash* values so that, in a cluster, the pair-wise similarity distance remains below to a pre-defined threshold value, *SimThreshold,* while restricting the cluster size to be no less than another pre-defined value, *MinClusterSize*. Figure 3 represents an example scenario showing how the similarity detection mechanism works in our proposed approach. Note that, the example scenario uses 32-bit hash fingerprint. However, in the actual experiment we have used 64-bit hash. In summary, a *simhash* dataset S of size N is defined as:

$$S = \{s_i\}_{i=1}^{N}$$

where s_i is the *simhash* of a code fragment.
For any pair of *simhash* values in *S*, the pair-wise distance is

$$D_{ij} = \text{hamming\_distance}\ (s_i, s_j)$$

The output of the algorithm is a cluster set C of size R:

$$C = \{c_i\}_{i=1}^{R}$$

where cluster c is a subset of *simhash* data of size L ($MinClusterSize \leq L \leq N$):

$$c = \{s_j\}_{j=1}^{L}$$

such that, for any pair of *simhash* values in cluster c,

$$D_{ij} \leq SimThreshold\ (i = 1 \ldots L,\ j = 1 \ldots L).$$

Thus, the results of the clustering algorithm over *simhash* values imply that, similar code fragments are grouped into the same cluster (or form their own cluster), which can be considered as a *Clone Cluster*/*Clone Group*.

The value of *MinClusterSize* is set to 2 and the value of *SimThreshold* is varied as required. For example, a *SimThreshold = 0* identifies exact clones and a *SimThreshold > 0* identifies renamed and near-miss clones. Note that the higher the value of *SimThreshold* the higher the possibility of false positive results (i.e., not an actual clone pair/group). We can empirically determine a cap for *SimThreshold* by evaluating the results at different increasing values until the presence of false positives in the detection result is acceptable. Section V describes the details of this process.

```
Algorithm DBSCAN(S, eps, minPts)
S: set of data item
eps: distance threshold
minPts: minimum cluster size
1. for each unvisited data s in dataset S
2. mark s visited
3. N = getNeighbours (s, eps)
4. if(sizeOf(N)) >= minPts
5.   C = a new cluster
6.   expandCluster(s,N,C,dt,minPts)
```

(a)

```
expandCluster(s, N, C, eps, minPts)
1. add s to C
2. for each data s' in N
3.   if s' is not visited
4.     mark s' visited
5.     N' = getNeighbours (s', eps);
6.     if(sizeOf(N')) >= minPts
7.       N = N joined with N'
8.   if s' is not yet member of any cluster
9.     add s' to cluster C
```

(b)

Figure 5. DBSCAN [6] algorithm

In our experiment, we have used a popular density based clustering algorithm named DBSCAN proposed by Ester et al. [6]. Here the formation of a cluster is based on the notion of density reachability. Basically, a point q is directly density-reachable from a point p if their pair-wise distance is no greater than a given distance *ε-neighborhood*, and if p is surrounded by sufficiently many points such that one may consider p and q to be part of a cluster. Figure 5 shows the pseudo-code of the algorithm. It requires two parameters: *ε-neighborhood* (mentioned as *eps* in Figure 5a, which we call *SimThreshold* in our proposed approach) and a minimum cluster size in number of points (mentioned as *minPts* in Figure 5a, which we call *MinClusterSize*). The algorithm starts with an arbitrary starting point that has not been visited. This point's *ε-neighborhood* is searched, and if it contains sufficiently many points, a cluster is started. Otherwise, the point is labeled as noise. This point might later be found in a sufficiently sized cluster of a different point and hence is made part of that cluster. In summary, the output of DBSCAN is basically some grouping of similar *simhash* values based on their *Hamming Distance*. Therefore, using this grouping we will get all the groupings of similar code fragments, i.e., all the *Clone Clusters*.

C. Output

From the previous phase we saw that the detection algorithm delivers clone detection output as lists of *Clone Clusters* each containing more than one code fragment (in its original form from the source code and its start and end line location in the file it is from along with the name of the file) which are pair-wise similar to each other. Additional post-processing steps might be required in some cases to filter out specific type of clones from the output. The output is in XML format which is convenient for displaying as a webpage using XSLT [4] or for importing into a clone visualization tool.

IV. Runtime Complexity of the Process

In this section the runtime complexity of the overall process is discussed. As mentioned in the previous section the process includes three phases: pre-processing, detection and output generation. The source code pre-processing and output generation time is linear to the size of the codebase. For the additional two pre-processing operations (mentioned in Section III), generation of *simhash* requires $O(q)$ time where q is the number of tokens in the code fragment for which *simhash* is being calculated and thus for the complete codebase overall complexity would be $O(n.q)$, where n is the number of code fragments extracted from the codebase. The indexing setup complexity is $O(n)$ since this step is also linear with respect to the input. The detection time of simCad depends on the DBSCAN algorithm that visits each of the n (to be more exact, the remaining non-clustered) data points of the dataset, possibly multiple times (e.g., as candidates to different clusters). However, the time complexity is mostly governed by the number of *getNeighbours()* queries (cf. Figure 5a-line 2, Figure 5b-line 5) that is executed for each data point. If a binary search scheme can be used that executes such a neighbourhood query in $O(\log n)$, an overall runtime complexity of $O(\mathrm{n}.\log n)$ is obtained [6]. In our proposed approach, the *simhash* values of the code blocks cannot be organized so that a binary search can be applied; instead a two level indexing scheme has been introduced to speed up the neighbour search query. First, by the size in lines of code and then by the number of 1-bits in the fingerprint; details of which was discussed in the previous section. For Type-1 and Type-2 clones, the search query requires $O(1)$ for the 1st level index, $O(1)$ for the 2nd level index and $O(m_{max})$ for a linear search where m is the number of elements/fragments that share the same number of 1-bits in their hashes. So, the overall runtime complexity is $O(n * m_{max})$. Here, the maximum value of m could be n but only when all the code fragments have the same number of lines and all the corresponding simhash values contain the same number of 1-bits, which is extremely rare. On average, m can take a value of $(n/(\text{Average values of } l + 64))$, where l is the size of a code fragment in lines. For Type-3 clones, the search query requires $O(l)$ for the 1st level index where l is the size of a code fragment in lines (note: the value of $l \ll n$ for a large system and typically has a maximum value not exceeding a couple of hundreds), $O(k)$ for the 2nd level index where k $(0 \leq k \leq 64)$ is the number of 1-bits in the hash and $O(m_{max})$ for a linear search where m is the number of elements that share the same number of 1-bits in the hash and thus the overall runtime complexity is the same as $O(n * m_{max})$.

V. Implementation, Analysis & Evaluation

This section summarizes the implementation and execution of simCad, the correctness measure of the detection, the evaluation of the detection results over four open source projects (cf. Table 1), the performance comparison with the original tool NiCad and finally some additional enhancements are suggested. The case studies were performed on a Linux OS (Ubuntu 10.10), which is running on a desktop PC with an Intel Core i7 3 GHz processor and 4 GB of RAM. We have implemented the proposed algorithm in Java. The size of *simhash* we used was 64 bit. This is good enough to be represented by the Java long data type, which is also a 64-bit, signed two's complement integer. Using a 32-bit hash improves the pre-processing and detection time to some extent. However, precision was found very low because of increased false positive detection even with lower *SimThreshold* values. Granularity was chosen both at the function and block levels.

[4] http://www.w3.org/TR/xslt

TABLE 1. SUBJECT SYSTEMS USED IN OUR EXPERIMENT

Subject Systems	Version	Language	Physical-LOC	URL
Eclipse-jdt	3.6.2	Java	289678	www.eclipse.org/jdt/core/
Jboss-AS	5.1.0	Java	563585	www.jboss.org/jbossas/
Firefox	2.0.0.4	C	2711444	ftp.mozilla.org/pub/mozilla.org/firefox
Linux	2.6.38	C	15720527	www.kernel.org

A. Detecting different types of clones using simCad

The simCad tool can be configured to detect different types of clones, individually or all at once. To detect only Type-1 clones, the value for *SimThreshold* is set to 0, i.e., the *Hamming Distance* between two *simhash* values is 0. Thus, *simhash* values of a potential Type-1 clone pair must be equal and only exact copies of code fragments will have equal *simhash* values. No source normalization (identifier/variable renaming) is required during the pre-processing phase for the detection of Type-1 clones except pretty-printing of the code. To detect Type-2 clones, the *SimThreshold* is also set to 0, but additional source normalization [21] is required over the pretty-printed source in the pre-processing phase in order to avoid the changes due to renaming of identifiers and/or function names. Note that, in this case the output will also contain Type-1 clones. The original, non-normalized source code is used to identify and filter out Type-1 clones and find the accurate Type-2 clone count.

To detect Type-3 clones, *SimThreshold* is set to an optimum value called *MaxSimThreshold*, which has been determined through an empirical process that will be described in the next subsection. Source normalization is optional during the pre-processing phase, but applying normalization will yield better detection results. Clone clusters containing only Type-1 and Type-2 clones are filtered from the output to get the exact count of Type-3 clones. Table 2 summarizes the complete setup for detecting different types of clones.

B. Finding the optimal value for SimThreshold

In the clustering algorithm, *SimThreshold* is used as a limit of acceptance for the neighbour search query (cf. *eps* in Figure 5a-line 3, Figure 5b-line 5) that looks for similar code fragments with respect to a candidate fragment (*s* in Figure 5a-line 3, *s'* in Figure 5b-line 5). As noted earlier, the greater the value of *SimThreshold*, the more Type-3 clones will be detected. But there is a limit to this value over which the algorithm will start including false positive clones in the detection results (cf. Figure 6). So the optimal value is a value that maximizes the detection of Type-3 clones and minimizes the presence of false positives in the detection result. Another important finding was that the optimum value for *SimThreshold* is not the same for all candidate code fragments (the fragment for which a neighbour search query is performed in the detection algorithm) of all sizes, i.e., it is size sensitive and choosing a larger value for smaller candidate fragments will allow the algorithm to pick some totally dissimilar code fragments as a cluster member.

TABLE 2. SUMMARY OF CLONE DETECTION SETUP FOR SIMCAD

Runtime Parameter	Value	Found Empirically
MinClusterSize	2	No
MinSimThreshold	0	No
MaxSimThreshold	12	Yes
Simhash size	64-bit	Yes
Granularity	fixed (function/block)	No

Clone Type	SimThreshold	Source Normalization	Post processing
Type-1	0	No	No
Type-2	0	Yes	Yes, removal of Type-1
Type-3	*MaxSimThreshold*	Optional, yes will yield better results	Yes, removal of clone class of Type-1 & Type-2
All at once	*MaxSimThreshold*	Optional, yes will yield better results	No

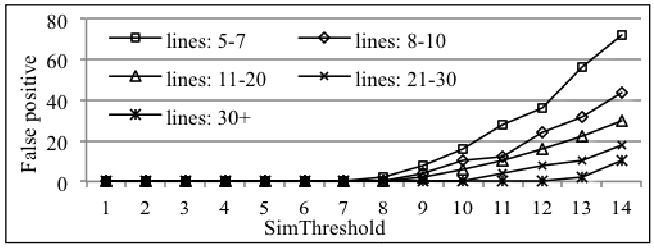

Figure 6. Relation between fragment sizes vs. *SimThreshold*

Figure 6 shows the distribution of the optimal threshold values for different size groups of fragments based on the two smaller (to make manual verification easier) projects among the four test projects used in our experiment. For each of the line size groups, we performed manual analysis to identify the presence and count of false positives in the detection results. In our implementation, the maximum allowed value for *SimThreshold* was found to be 12. However, it is clear from the graph that for smaller size code fragments the optimal threshold value is also smaller and for larger sizes it can have a larger value. This dynamic nature of *SimThreshold* has been incorporated into the implementation of simCad, i.e., the threshold value for candidate code fragment is adjusted with respect to its size at runtime when it searches for its neighbour code fragments. For example, when the algorithm executes the neighbour search query using the instruction "*getNeighbours* (*s, eps*)" (cf. Figure 5), if the size of *s,* $|s|$ is between 5-7 lines, the value of *eps* is set to 7; if $|s|$ is between 8-10 lines, *eps* is set to 8 and so on.

C. Measurement of correctness in clone detection

We have measured the correctness of simCad's detection result, which is an important step for the reliability of any clone detection approach. Our quantitative evaluation is based on three criteria: precision, recall and f-measure. Precision is the measure of actual clones in the detection result, recall is the measure of clones detected among the available clones and finally f-measure is the measure of a test's accuracy interpreted as a weighted average of the precision and recall; for all the cases the score reaches its best value at 1 and worst at 0. Following are the three equations for the measurement of correctness criteria.

$$precision = (reference \cap detected)/detected \quad (1)$$

$$recall = (reference \cap detected)/(reference) \quad (2)$$

$$f-measure = (2 * precision * recall)/(precision + recall) \quad (3)$$

To evaluate the correctness of simCad's detection result, we used a variant of the mutation/injection framework for evaluating clone detection tools as proposed by Roy and Cordy [19]. Instead of completely automating the framework, we followed a semi-automated approach and a test was carried out with function clones only. We chose a

small sized open source system (Apache-Ant) for this test. To create a mutated codebase, some functions were arbitrarily selected from the original codebase and from those a total of 100 different types of reference clones (cf. Table 3) were created manually by copying and/or modifying the code using diverse change patterns. These reference clones were then injected into original codebase at random locations that yielded the mutated codebase for our test. The location information of the injected reference clones was recorded for the future use for a validator program. After building the mutated codebase, we applied simCad on it and the clone detection results were analyzed by a validator program that looks for the reference clones in the result and then measures the values of the correctness criteria defined by equations 1, 2 and 3. Table 3 shows the result of the output analysis and the measured values of the three criteria. From the data we can see that, simCad correctly detected all the Type-1 and Type-2 reference clones but failed to detect one Type-3 clone among the 50 Type-3 reference clones. An investigation revealed that the code modification done while building this reference clone was so much that it went beyond the chosen maximum *SimThreshold* value of 12. Now that we know simCad is working correctly according to our mutation/injection based test, we next see how simCad does in comparison with another tool. As we said earlier, we will compare the performance of simCad considering NiCad as a benchmark over the same subject systems in order to find the answers to the research questions defined in Section I. But before doing that we need to come up with equivalent settings for both the tools to make the performance comparison fair.

D. Finding the equivalency between NiCad's UPI-Threshold and simCad's SimThreshold

Our goal is to compare the performance of NiCad and simCad with respect to time and the number of cloned fragments detected. Each of the tools has its own definition of threshold for setting the boundary of similarity. Similar to *SimThreshold* in simCad, detection in NiCad is governed by a threshold value called *UPI-Threshold* that puts a limit to the acceptable dissimilarity between two code fragments in a clone cluster. The value ranges from 0 to 1 where 0 means exactly similar and 1 means totally dissimilar. For simCad it is *SimThreshold,* the detail of which was given in Section III. A value of 0 for both the thresholds means exact similarity whereas a value greater than zero means some dissimilarity is acceptable; the larger the value the more dissimilarity is acceptable for being a member of a clone cluster.

From Table 2, we see that simCad uses a threshold value 0 for both Type-1 and Type-2 clones, which is the same for NiCad. Thus comparing the results of NiCad and simCad is easier in these two cases. However, for Type-3 clones, both the tools use a threshold value greater than 0 and hence to conduct a fair comparison in this case an equivalency between these two thresholds needs to be defined. To do that, we first detect all the three types of function clones using simCad with *SimThreshold* = 12, which we found to be the highest optimum threshold value (cf. Figure 6). The output has been analyzed by a separate program, which measures the coverage of the clones under the definition of NiCad's *UPI-Threshold* with increasing threshold values starting from 0. It has been found that, at *UPI-Threshold* = 0.4 almost all the clones (detected by simCad at SimThreshold = 12) are covered and hence in our experimental setup we are considering these two thresholds as equivalent to each other.

TABLE 3. MUTATION-BASED EFFECTIVENESS OF SIMCAD

		Type-1	Type-2	Type-3	Overall
clone	Reference	20	30	50	100
	Detected	20	30	49	99
	Missed	0	0	1	1
	False positive	0	0	0	0
effectiveness	precision	1	1	1	1
	recall	1	1	0.98	0.99
	f-measure	1	1	0.989	0.99

However, NiCad considers 0.3 as a standard value for UPI-threshold in Type-3 clone detection and here we came up with an equivalent threshold value more than that standard value. This is an important decision to make since this might be a threat to validity of our approach. For the tool comparison, we have decided to go with the value 0.4 for obvious reasons. We could have gone for finding equivalency in other direction, i.e., consider finding an equivalent SimThreshold value corresponding to the standard UPI-Threshold value which is 0.3; but in that case a lower or higher value than 12 for SimThreshold would make simCad miss potential Type-3 clone or reduce clone quality by including more false positives in detection result and none of those are expected. With simCad, our target is to get as many near-miss clones as possible while minimizing the presence of false positives in the detection results since we are considering this approach to be effective for large scale Type-3 clone detection. Thus, we have decided to continue the comparison with the value 0.4 for the UPI-Threshold. Below we will present the comparative performance analysis of both the tools running on these two equivalent thresholds.

E. simCad vs. NiCad: head-to-head comparison

We plan to examine simCad's detection capability so that we can examine the performance of *simhash*. The performance of simCad in terms of detection time and number of clones has been compared with NiCad based on four medium to large scale open source applications. The test data was taken with an equivalent similarity measurement setting for both the tools defined in Section V-D. Let us look at the detection time first. Table 4 and Table 5 show the comparison on detection time between NiCad and simCad for three types of function and block clones respectively. Since we are trying to compare the efficiency of the detection engine of both the tools, the time data presented here is only for the detection process and does not include any pre or post processing time. The experimental outcome clearly shows that our proposed approach significantly outperforms NiCad in detection time for all the three types of clones.

Note that, for a large scale system such as the Linux Kernel, simCad is 19 times faster than NiCad for Type-1/Type-2 clones and 12 times faster for Type-3 clones in case of functional clone detection. Similarly, for block clone detection on Linux, simCad is 31 times faster than NiCad for Type-1/Type-2 clones and 9 times faster for Type-3 clones.

Table 6 and Table 7 show the comparison in number of detected function and block clone fragments respectively. We see that both the tools are equal for Type-1 and Type-2 clones but differ for Type-3 clones, and again simCad performs slightly better than NiCad in this case. Analyzing the outcome of Type-3 clones for both the tools we have identified that, although simCad detects more Type-3 clones than NiCad, it does not cover all the clones detected by NiCad. That means under the equivalent threshold setting, each of the tools is detecting some unique clones that are undetected by the other tool. Common and unique Type-3 clone fragment count for both tools is shown in Table 8. We analyze the unique clone fragments for both the tools and investigate the cause of why one tool detected some of the clones that the other tool failed.

F. Analyzing the unique clones

Table 9 presents the categorization of unique clone fragments detected by simCad with different UPI-Thresholds. The important observation from the data distribution presented here is that NiCad should have detected the clones falling under UPI-Thresholds from 0.1 to 0.4 since the UPI-Threshold for NiCad was set to 0.4 for Type-3 clone detection. The likely reasons are summarized as follows:

i) Single data reference for cluster: NiCad uses a single data reference for cluster membership. So, it might miss those potential clones which fall under the UPI-Threshold in comparison to other members of the cluster but not for the reference member.

TABLE 4. SIMCAD VS. NICAD FUNCTION CLONE DETECTION TIME (MS)

Subject Systems	#of functions of size >= 5 lines	Type-1		Type-2		Type-3	
		NiCad	*simCad*	*NiCad*	*simCa*	*NiCad*	*simCad*
Eclipse-jdt	9832	309	6	313	7	7592	655
Jboss-AS	17601	667	8	760	10	15102	2737
Firefox	15285	358	5	364	6	32324	2392
Linux	198146	13343	713	14319	720	7660091	625563

TABLE 5. SIMCAD VS. NICAD BLOCK CLONE DETECTION TIME (MS)

Subject Systems	#of blocks of size >= 5 lines	Type-1		Type-2		Type-3	
		NiCad	*simCad*	*NiCad*	*simCa*	*NiCad*	*simCad*
Eclipse-jdt	10903	1383	8	389	8	15022	1255
Jboss-AS	31086	2213	10	2238	12	76022	5433
Firefox	39990	1972	9	1985	10	195215	11382
Linux	418605	250050	7851	256117	4865	20702947	2194557

TABLE 6. SIMCAD VS. NICAD FUNCTION CLONE FRAGMENT COUNT

Subject Systems	Type-1		Type-2		Type-3		All at once	
	NiCad	*simCad*	*NiCad*	*simCad*	*NiCad*	*simCad*	*NiCad*	*simCad*
Eclipse-jdt	555	555	1719	1719	3190	3319	3613	3802
Firefox	1273	1273	1679	1679	3593	3741	4317	4614
Jboss-AS	1463	1463	2102	2102	6780	6925	7806	7991
linux	2860	2860	18230	18230	40150	41114	52970	54534

TABLE 7. SIMCAD VS. NICAD BLOCK CLONE FRAGMENT COUNT

Subject Systems	Type-1		Type-2		Type-3		All at once	
	NiCad	*simCad*	*NiCad*	*simCad*	*NiCad*	*simCad*	*NiCad*	*simCad*
Eclipse-jdt	2002	2002	2863	2863	4054	4211	5749	5913
Firefox	5962	5962	7250	7250	12726	12996	14019	14318
Jboss-AS	5674	5674	6847	6847	10148	10353	10511	10957
Linux	53967	53967	63458	63458	142638	143479	160988	162043

TABLE 8. COMMON AND UNIQUE CLONES IN NICAD VS. SIMCAD

Subject Systems	Function Clone			Block Clone		
	Common	Unique		Common	Unique	
	NiCad/simCad	*NiCad*	*simCad*	*NiCad/simCad*	*NiCad*	*simCad*
Eclipse-jdt	2804	386	515	3795	259	416
Firefox	3233	360	498	12297	429	699
Jboss-AS	6494	286	431	9788	360	565
Linux	38740	1410	2374	141086	1522	2393

ii) Coarse grain change detection: NiCad uses the UNIX Diff in the detection engine to measure the dissimilarity between two code fragments. Diff works on line level changes, i.e., it does not care whether the change in a line is small or big. Thus if there are small changes in multiple lines of a code fragment, the dissimilarity measure might go beyond the accepted threshold limit which will cause the fragment to be undetected by NiCad.

iii) Sensitive to ordering of line: Here the Diff command puts another limitation on NiCad since it is sensitive to line ordering. Thus, re-ordering among the statements of a code fragment might result in a high dissimilarity value that might cause the fragment to be undetected.

In contrast to NiCad, simCad considers each of the cluster members as cluster references, uses fine grained (token level) change detection and is insensitive to instruction reordering except for Type-1 clones. Thus, our proposed approach overcomes some of the limitations in NiCad while taking much less time to detect the clones.

Table 10 shows the number of clones that are not detected in simCad but detected in NiCad with equivalent settings and categorized by different *SimThresholds*. These clones were detected in NiCad with UPI-Threshold 0.4, but it is clear from the table that those were not detected in simCad because they were beyond the equivalent SimThreshold value 12. Note, for each of these clones, the minimum distance with a cluster member was considered for categorization of that clone fragment. For example, in a clone cluster, if the distance of a clone fragment x_1 from other cluster members x_2, x_3, x_4 and x_5 are computed as 14, 13, 17 and 19 respectively, then x_1 is categorized (in Table 10) under 13. From Table 10, it is clear that the majority of the undetected clones are just above the *SimThreshold* value of 12. Hence, by increasing the *SimThreshold* value those clones can be detected. However, according to our earlier analysis (cf. Figure **6**) a threshold value greater than 12 will increase the chances of false positive detection. As a solution to this problem, we have implemented a couple of additional techniques to assist the original detection process, which allow simCad to use a high *SimThreshold* value while minimizing the presence of false positives. Those are presented as follows.

TABLE 9. UNIQUE CLONE FRAGMENTS IN SIMCAD WITH DIFFERENT UPI-THRESHOLDS

Subject Systems	Function Clone								Block Clone							
	UPI-Threshold								UPI-Threshold							
	0.1	0.2	0.3	0.4	0.5	0.6	0.7	total	0.1	0.2	0.3	0.4	0.5	0.6	0.7	total
Eclipse-jdt	0	8	162	221	88	26	10	515	0	3	98	196	82	29	8	416
Firefox	0	6	113	250	86	29	14	498	0	4	185	351	97	48	14	699
Jboss-AS	0	2	125	208	58	26	12	431	0	3	134	248	124	39	17	565
Linux	0	16	576	1085	448	182	67	2374	0	22	671	992	587	72	49	2393

TABLE 10. UNIQUE CLONE FRAGMENTS IN NICAD WITH DIFFERENT SIMTHRESHOLDS

Subject Systems	Function Clone							Block Clone						
	SimThreshold							*SimThreshold*						
	13	*14*	*15*	*16*	*17*	*18*	*total*	*13*	*14*	*15*	*16*	*17*	*18*	*total*
Eclipse-jdt	164	96	68	42	14	2	386	108	65	46	28	10	2	259
Firefox	124	72	83	66	9	6	360	96	88	145	78	14	8	429
Jboss-AS	98	62	67	44	10	5	286	98	93	82	64	17	6	360
Linux	713	316	218	92	44	27	1410	713	348	266	114	49	32	1522

1) Multiple simhash: If we look at the *simhash* algorithm (Figure 2, line 5), it uses a simple hash function to compute a hash for each token, which is, in our case the Jenkin Hash Function. It is possible to use another simple hash function here to generate a second *simhash* value for each fragment. Note that the second *simhash* will also hold the basic principle of *simhashing* although it might be a completely different hash value with respect to the first one and this second *simhash* must be compared with the second *simhash* of another fragment and never with the first one. Thus a combination of these two *simhash* values will allow choosing a higher *SimThreshold* value while minimizing the presence of false positives.

We have used a variant of the Jenkin hash in order to generate the second simhash and then again gone through the procedure discussed in section V-B to determine the optimum value for *SimThreshold*. This time the highest value was found to be 16. Now, if we see the statistics in Table 10, it is clear that with this *SimThreshold* value, simCad does not cover 100% of the unique clones detected by NiCad (it is still missing those at SimThreshold 17 and 18) but it does cover most of them (95-98%). Detection with multiple-hash also costs additional processing time in detection. From our experiment it has been found that this double *simhash* version requires 25-30% more time than the original one. The detection time comparison among the single *simhash* based simCad, dual *simhash* based simCad and NiCad is shown in the first four columns of Table 11. We can see, despite additional processing, simCad with dual *simhash* still performs better than NiCad.

2) Additional text-based comparison in the detection process: In the detection algorithm, distance based hash similarity comparison at higher thresholds can be assisted by some additional text-based comparison (for example the *diff* algorithm) to detect some potential clones that are undetected by simCad with its current settings. This text-based comparison will be applied conditionally. For example, with respect to the categorization presented in Table 10, when the distance between two *simhash* falls between 13-18, only then the text-based comparison will be applied to the corresponding source text to check similarity.

We have integrated an implementation of Myer's [16] *diff* algorithm in the detection process of simCad and used it to measure text similarity during the condition mentioned above. With this approach simCad was successfully able to detect all the unique clones detected in NiCad (cf. Table 10). Table 11 shows the time comparison of simCad including *diff* (last column) with the single and dual *simhash* based simCad and NiCad. We see that the approach takes 60-65% more time than the original version of simCad. Although it requires a bit more time than the multiple *simhash* based approach, it is still much faster than NiCad.

These workarounds can be seen as trade-off between the time efficiency and detection accuracy of our proposed approach for Type-3 clone detection and can be applied as per the detection requirement.

TABLE 11. TIME (MS) COMPARISON OF SIMCAD INCLUDING *DIFF* WITH OTHERS IN TYPE-3 FUNCTION CLONE DETECTION

Subject Systems	NiCad	simCad with single-simhash	simCad with dual-simhash	simCad with single-simhash & *diff*
Eclipse-jdt	7592	655	845	1106
Firefox	32324	2392	3011	4075
Jboss-AS	15102	2737	3388	4642
Linux	7660091	625563	769432	1000846

TABLE 12. TIME (MS) COMPARISON FOR DIFFERENT INDEXING STRATEGIES

Subject Systems	Type 1 Function Clone				Type 3 Function Clone			
	Naive	*L-index*	*B-index*	*M-index*	*Naive*	*L-index*	*B-index*	*M-index*
Eclipse-jdt	2712	281	266	6	5564	4182	5041	655
Jboss-AS	4270	462	430	8	6419	5704	6235	2737
Firefox	6645	657	615	5	10153	7877	9532	2392
Linux	218761	28975	27537	713	2239618	1815249	2033079	625563

G. Time performance gain from multi-indexing

To measure the efficiency of our proposed multi-level indexing strategy we have implemented the following four versions of simCad with different indexing strategies:

- No indexing: Naive
- Line based index only: L-index
- Bit based index only: B-index
- Multi (both line and bit based) index: M-index

Table 12 summarizes the detection time for these variants of simCad based on the indexing strategies. It is clear from the data that the multi-level indexing used in our proposed approach clearly sped the detection proccess up to a notable extent.

H. Addressing the research questions

We see that our experimental results and analysis strongly supports the effectiveness of *simhash's* use in the detection of exact and near-miss clones in a software system. Our *simhash* based approach allowed simCad to work significantly faster than NiCad, which is a challenge for a large system such as the Linux Kernel. simCad also overcomes some of the shortcomings of NiCad that allowed simCad to be able detect some new potential Type-3 clones. On the other hand, simCad has its own shortcomings for Type-3 clone detection, which can be overcome with a couple of additional techniques. In summary, *simhash*: *1)* is feasible to be used for software clone detection, *2)* enables faster detection of clones in large software systems, and *3)* has some limitations in detecting Type-3 clones that can be overcome with a few additional techniques.

VI. RELATED WORK

Fingerprinting techniques have been used in different areas of computing research. In software clone detection research a number of approaches used fingerprinting with normalized source code or with Abstract Syntax Trees (ASTs). Johnson [11] presents a detection mechanism that uses fingerprints to identify exact repetitions, which is not applicable for near-miss clone detection. Chilowicz et al. [4] proposed an approach that uses hash fingerprints on ASTs. State of the art approaches proposed by Hummel et al. [9] and Li et el. [14] are also based on statement fingerprints and very good in detecting Type-1 and Type-2 clones. However, since these techniques do not use a similarity preserving hash, Type-3 clone detection would be hard to

support or not possible at all. Smith and Horwitz [23] present a similarity preserving fingerprinting technique and Baxter et al. presents CloneDR [2] which uses a hash based AST comparison, both having some success in detecting Type-3 clones. However, the measure of effectiveness of their approaches in detecting Type-3 clones was not evident from the experimental results they provided. Jiang et al. [10] presents an approach that uses matching of a characteristic vectors on ASTs for identifying clones including Type-3. However, the claim of the approach being scalable in the case of very large datasets was not clearly explained. In summary, existing fingerprinting techniques are either incapable or have not been proven effective in the detection of Type-3 clones and/or are not scalable to large datasets. Type-3 clone detection in large datasets is one of the main concerns of our proposed approach since for Type-1 and Type-2 clones, there are a number of fast and elegant solutions available [9, 12, 14], but for Type-3 clones success is limited [20, 21].

Outside the area of clone detection, *Google* is using *simhash* for finding near duplicate webpages [8]. Gong et al. [7] presents an approach for detecting near-duplicates within a huge repository of short messages. Similarly, SimFinder [18] is a fast algorithm proposed by Pi et al. to identify all near-duplicates in large-scale short text databases. Our experimental outcome shows that simhash based fingerprinting has great potential for use in the area of clone detection.

VII. Conclusion and Future Work

Detection of clones provides several benefits in terms of maintenance, program understanding, reengineering and reuse [13]. We took an existing code cloning system and improved the time performance by an order of magnitude using simhash, and demonstrated its feasibility for use with large systems such as the Linux Kernel. As well, we adapted simhash to a code cloning framework and demonstrated its viability for the clone detection of Type-1, Type-2 and Type-3 clones in large-scale systems. Our experience confirms that diff-based comparison (as used in NiCad) works well for finding Type-3 clones, which was also observed by Tiarks et al. [24]. However, this comes with both a cost in time complexity, and lower recall (for the possible optimizations used in the comparison). On the other hand, *simhash* has significant potential for the fast and large scale Type-3 clone detection but comes with the increased possibility of false positive clones. However, we have shown that it is possible to overcome the limitations of *simhash* using techniques like multiple hashing or a conditional diff-based comparison. We believe that our study supports a call for further research in using/adapting simhash for clone detection research or similar studies.

References

[1] B. S. Baker. “A Program for Identifying Duplicated Code”. Proc. *CSS Interface*, 1999, Vol. 24, pp. 49-57.

[2] I. D. Baxter, A. Yahin, L. Moura, M. Sant' Anna and L. Bier, “Clone detection using abstract syntax trees”, Proc. *ICSM*, 1998, pp. 368-378.

[3] M. S. Charikar, “Similarity estimation techniques from rounding algorithms”. Proc. *STOC*, 2002, pp. 380-388.

[4] M. Chilowicz, E. Duris and G. Roussel, “Syntax tree fingerprinting for source code similarity detection” Proc. *ICPC*, 2009, pp. 243-247.

[5] J. R. Cordy, “The TXL Source Transformation Language”. *Science of Computer Programming*, 61(3):190-210, 2006.

[6] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, “A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise”, Proc. *KDD*, 1996, pp. 226-231.

[7] C. Gong, Y. Huang, X. Cheng and S. Bai. “Detecting near-duplicates in large-scale short text databases”, Proc. *PAKDD*, 2008, pp. 877-883.

[8] M. Henzinger, “Finding near-duplicate web pages: a large-scale evaluation of algorithms”, Proc. *SIGIR*, 2006, pp. 284-291

[9] B. Hummel, E. Juergens, L. Heinemann and M. Conradt, “Index-based code clone detection: incremental, distributed, scalable”, Proc. *ICSM*, 2010, pp. 1-9.

[10] L. Jiang, G. Misherghi, Z. Su and S. Glondu. “DECKARD: Scalable and Accurate Tree-based Detection of Code Clones”. Proc. *ICSE*, 2007, pp. 96-105.

[11] J. H. Johnson, "Identifying redundancy in source code using fingerprints", Proc. *CASCON*, 1993, pp. 171-183.

[12] T. Kamiya, S. Kusumoto and K. Inoue, “CCFinder: a multilinguistic token-based code clone detection system for large scale source code”, IEEE *TSE*, 28(7):654- 670, 2002.

[13] B. Lague, D. Proulx, E. Merlo, J. Mayrand and J. Hudepohl, “Assessing the benefits of incorporating function clone detection in a development process,” Proc. *ICSM*, 1997, pp. 314-321.

[14] Z. Li, S. Lu, S. Myagmar and Y. Zhou. “CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code”. IEEE *TSE*, 32(3):176-192, 2006.

[15] G. S. Manku, A. Jain and A.D. Sarma, “Detecting Near-Duplicates for Web Crawling”, Proc. of *WWW*, 2007, pp. 141-150.

[16] E. W. Myers, “An O (ND) difference algorithm and its variations”, *Algorithmica*, 1(1):251-266, 1986.

[17] J. Mayrand, C. Leblanc, and E. Merlo, “Experiment on the automatic detection of function clones in a software system using metrics”, Proc. *ICSM*, 1996, pp. 244-254.

[18] B. Pi, S. Fu, W. Wang and S. Han, “SimHash-based Effective and Efficient Detecting of Near-Duplicate Short Messages”, Proc. *ISCSCT*, 2009, pp. 020-025.

[19] C. K. Roy and J. R. Cordy, “A mutation / injection-based automatic framework for evaluating code clone detection tools”, Proc. *ICST Mutation Workshop*, 2009, pp. 157–166.

[20] C.K. Roy, J.R. Cordy and R. Koschke, Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Science of Computer Programming*, 74 (2009): 470-495, 2009.

[21] C. K. Roy and J. R. Cordy. “NiCad: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization", Proc. *ICPC*, 2008, pp. 172-181

[22] G. M. K. Selim, K. C. Foo and Y. Zou. “Enhancing Source-Based Clone Detection Using Intermediate Representation”, Proc. *WCRE*, 2010, pp. 227-236.

[23] R. Smith and S. Horwitz, “Detecting and measuring similarity in code clones”, Proc. *IWSC*, 2009, pp. 28-34.

[24] R.Tiarks, R. Koschke, and R. Falke, "An extended assessment of type-3 clones as detected by state-of-the-art tools", *Software Quality Journal*, 19(2): 295-331, 2011.