2013 Third International Conference on Advanced Computing & Communication Technologies

A HYBRID CLONE DETECTION TECHNIQUE FOR ESTIMATION OF RESOURCE REQUIREMENTS OF A JOB

Madhulina Sarkar[2], Sameeta Chudamani [1], Sarbani Roy[1], Nandini Mukherjee[1]

[1]*Department of Computer Science and Engineering, Jadavpur University, Kolkata-32,*

[2]*Department of Computer Science and Engineering, Govt. College of Engineering and Leather Technology, Kolkata-98,*

madhulina.sarkar@gmail.com, sameemani@gmail.com, sarbani.roy@cse.jdvu.ac.in, nmukherjee@cse.jdvu.ac.in

ABSTRACT

Resource requirement estimation in large distributed systems is a difficult job because of the heterogeneity and dynamism of the environment involving modern distributed systems. A feedback-based job modeling scheme based on clone detection technique was proposed in [6]. This paper extends the taxonomy of clones proposed by other researchers [1] in order to make resource requirement prediction more effective. It also presents a hybrid clone-detection technique, consisting of metrics-based, PDG-based and AST-based clone detection, to make the clone detection process more reliable and robust.

KEY WORDS

Software clones, Taxonomy, Clone-detection, Metric-based, PDG-based, AST-based.

1. INTRODUCTION

Modern distributed systems, like Cloud and Grid, comprise dynamic pool of heterogeneous resources which are distributed over a large geographical area and administered by multiple administrative domains. In such a large distributed system, jobs are allocated to resources on the basis of the resource requirements of each job. Otherwise, either a job with higher resource requirements may be allocated to low-capacity resources or a job with low resource requirements may be allocated to resources with higher capability. Thus, resource requirements of a job must be estimated and resources should be dynamically characterized and allocated to the job on the basis of this estimation.

We propose a feedback-guided job modeling scheme [6] for resource requirement prediction. In our proposed technique, a newly submitted job is categorized as a clone job with respect to the jobs executed earlier in the system. For every job, relevant data are retrieved before executing (BE data) the job and after executing (AE data) the job. These data are stored in an execution history. Whenever a new job is submitted, its clone jobs are searched in the execution history using different clone detection techniques and the new job is categorized as a specific type of clone of the jobs stored in execution history. Later, some statistical methodologies are applied to predict the resource requirements of the new job. Thus, our proposed feedback-guided job modeling is strongly based on clone detection techniques. Job modeling scheme proposed by us is incorporated in the PRAGMA tool [24][25] that we use for maintaining performance guarantee in a large distributed environment.

In this paper, we first give an overview of different clone detection techniques. Next we propose a taxonomy of clones which is an extended and modified version of the existing taxonomy [1]. The taxonomy incorporates clone types which are relevant for resource requirement estimation. A hybrid clone detection process is discussed that is used for feedback-guided job modeling.

The paper is organized as follows - Section 2 presents a brief overview of the clone detection techniques we propose to use in our job modeling scheme. Section 3 presents the taxonomy of existing clones and also defines some new clone types based on the requirement. Section 4 describes the implementation of the clone detection technique that has been used in our work. Section 5 describes the experimental set up and the performance of the implementation measured in terms of the overhead incurred at phases of the clone detection process. Finally, Section 6 presents the related work and section 7 concludes with a direction for future work.

2. CLONE DETECTION TECHNIQUES

Clone detection techniques are heavily used in software engineering for the purpose of software maintenance [21, 22]. Software clones are codes or segments of code that are similar to some codes according to some definition of similarity [5]. The definitions of similarity can be based on several categories like text, syntax, semantics, and pattern. Depending on these categories some clone detection techniques have been proposed in the literature [1, 5]. We propose to use three of all these clone detection techniques in our job modeling scheme. These are metric based, PDG based and AST based clone detection techniques. Following is a brief overview of these techniques.

Metric based clone detection technique: In metric-based approaches, different metrics (such as, number of lines of source code, number of function calls contained) for code fragments are retrieved and these metrics are compared instead of comparing codes directly [1, 7]. An allowable distance (for instance, Euclidean distance) for these metrics can be used to detect similar code. In most cases, the source code is parsed to its AST/PDG representation for calculating such metrics. [7]

PDG based clone detection technique: Here, the control flow graph and the data dependency graph are used to generate the Program Dependency graph (PDG).Then isomorphic sub graph matching is done to detect clones from the PDGs. Komondoor and Horwitz proposed a PDG-based technique [8] which finds isomorphic PDG sub graphs using program slicing [9]. Their technique is one of the leading PDG based clone detection techniques.

978-0-7695-4941-5/13 $26.00 © 2013 IEEE
DOI 10.1109/ACCT.2013.43

CPS Conference Publishing Services

AST based clone detection technique: In this technique, the source file is parsed to generate its Abstract Syntax Tree using a parser of the target language. The AST matching is then done using some tree-matching technique. All information of the job is contained in the nodes of the parse tree. Baxter et al proposed a technique for clone detection based on Abstract Syntax Tree [5]. They partitioned sub trees of the Abstract Syntax Tree of a program based on a hash function and then compared sub trees in the same partition through tree matching (allowing some divergences) [5].

In the next section we provide a description of different types of clones which are generally found in programming languages (in our current work we focus on C language) and a categorization of clones is proposed that will be effective for resource requirement prediction.

3. A Taxonomy of Clones for Efficient Resource Requirement Prediction

In this section, we propose taxonomy of job clones for efficient resource requirement prediction. The taxonomy presented here extends the categories provided in [6] and we have renamed the category of exact clones in [6] as Alike clones. Clones are broadly classified into three categories: (a) alike clones, (b) Near-Miss clones and (c) Miss clones. While resource requirements are same for alike clones, statistical methodologies can be applied to predict the resource requirements for near-miss clones and miss clones [6]. The three categories are further subdivided. Each of the sub-types is discussed with an example.

A) Alike Clones

Alike clones are those clones where the clone job is very similar to the original job syntactically and the resource requirements are same. We categorize this type of clone jobs as follows:

i. Exact Clones

Two or more code segments are called exact clones if they are identical to each other with some differences in comments or white spaces or layout leading to same resource requirements for both. Editing activities like changing the comments, restructuring in layout, that is, changing the positions of begin-end brackets or other language elements through adding/removing tabs, blanks, and new lines can lead to such job clones. Figure 1 shows two example code fragments which are considered as exact clones.

<pre>i=0;
  while(n>0)
  {
      x[i]=n%2;
      n=n/2;
      i++;
  }</pre>
	<pre>i=0;
	    //convert to binary
	    while(n>0){
	        x[i]=n%2;
	        n=n/2;
	        i++;
	    }</pre>

Figure 1: Example of Exact Clones.

ii. Renamed Clones

The term renamed clones is used when identifier names, literals values change in the job clone. The identifiers can be renamed consistently or inconsistently in the job clone. Figure 2 shows two example code fragments which can be considered as renamed clones.

```
for(k=1;k<=n;k++){
 sum=b[k];
 for(l=1;l<=n;l++){
    if(k!=l){
      sum=sum-(a[k][l]*x0[l]);
    }
 }
 x[k]=sum/a[k][k];
}
```

```
for(i=1;i<=n;i++){
 sum=b[i];
 for(j=1;j<=n;j++){
    if(i!=j){
      sum=sum-(a[i][j]*x0[j]);
    }
 }
 x[i]=sum/a[i][i];
}
```

Figure 2: Example of Renamed Clones

iii. Parameterized Clones

A parameterized clone or p-match clone is a type of renamed clone with systematic renaming of identifiers. A parameterized clone is a renamed clone but not vice-versa. Both consistent and inconsistent renaming of identifiers is considered as renamed clones. Only those with consistent renaming are parameterized clones. Figure 3 shows two p-match code fragments.

```
if(a < b){
   a=a+b;
   b=a-b;
   a=a-b;
}
```

```
if(k < l){
   k=k+l;
   l=k-l;
   k=k-l;
}
```

Figure 3: Example of Parameterized Clones

iv. Non-parameterized Clones

It is a type of renamed clone which does not follow consistent renaming of identifiers. Figure 4 shows two non-parameterized clone code fragments.

```
if(a < b){
   b=a+b;
   a=b-a;
   b=b-a;
}
```

```
if(k < l){
   k=k+l;
   l=k-l;
   k=k-l;
}
```

Figure 4: Example of Non-parameterized Clones

v. Reordered Clones

The job clone has reordered statements with respect to the original but the control flow dependencies are not violated. Figure 5 shows two reordered clone code fragments.

```
while (a>b){
        a=a-c;
        b=c++;
        c=c-b;
}
```

```
while (a>c){
        b=c++;
        c=b-c;
        a=a-b;
}
```

Figure 5: Example of Reordered Clones

B) Near-Miss Clones

Near-miss clones are those clones where the clone job is very similar to the original job syntactically, but the resource

requirements may differ. We categorize this type of clone jobs as follows:

i. Transformed Clones

If the data types of the input data* are different in the jobs, then their resource requirements will be different. These are Transformed clones which can be categorized under Near-Miss Clones. Figure 6 shows two code fragments which can be considered as Transformed clones.

```
float a[INDEX][INDEX], b[INDEX][INDEX], c[INDEX][INDEX];

--
--

 for (i=0;i<INDEX;i++)
   for(j=0;j<INDEX;j++)
    for(k=0;k<INDEX;k++)
      c[i][j]=c[i][j] + a[i][k] * b[k][j];
```

```
int a[INDEX][INDEX], b[INDEX][INDEX], c[INDEX][INDEX];

--
--

 for (i=0;i<INDEX;i++)
   for(j=0;j<INDEX;j++)
    for(k=0;k<INDEX;k++)
      c[i][j]=c[i][j] + a[i][k] * b[k][j];
```

Figure 6: Example of Transformed Clones

ii. Scaled Clones

Two jobs are termed as Scaled clones to each other if the input data** sizes of the source codes are not equal, but syntactically and semantically they are similar. Figure 7 shows two Scaled clone fragments.

```
#define DATASIZE 1000
--
--
for(;;)
{
        OldNum = NewNum;
        NewNum = FibNum;
        FibNum = OldNum + NewNum;
        if(FibNum > DATASIZE)
        {
         printf("  ");
         break;
        }
}
```

```
#define DATASIZE 200
--
--
for(;;)
{
        OldNum = NewNum;
        NewNum = FibNum;
        FibNum = OldNum + NewNum;
        if(FibNum > DATASIZE)
        {
         printf("  ");
         break;
        }
}
```

Figure 7: Example of Scaled Clone

C) Miss-Clone:

Two jobs are termed as Miss Clone jobs if they are completely different according to their syntax, hence their resource requirements will differ.

---

* The data structures are used in the major operations in a code and therefore the running time is dependent on these data structures.

** Execution time of a job increases with the increase in data input size and decreases with the decrease in data input size.

In the next section we give an overview of the process of detection of different clones

4. IMPLEMENTATION OF CLONE DETECTION

The overall process of clone detection consists of several phases [1]. The clone detection phases are very general and one or more of the phases may be ignored in some clone detection process. Hence, in the proposed work clones are detected through three phases - 1) Preprocessing 2) Transformation and 3) Match Detection.

In the proposed clone detection technique, we serially use metric based, PDG based and AST based techniques. Metrics are used to detect similar jobs at the first level of clone detection. PDG-based technique is used in detecting clones at the next block-level stage and finally an AST-based technique is used to detect clones at the more detailed statement-level stage. Combining these methods, several clones can be detected effectively. Following are the phases of clone detection:

i. Preprocessing:

This is the first phase of any clone detection process. It involves removing non-useful parts from the source code like comments and determining the comparison units of the target source code. Filtering the parts which are not required during the detection process should be done before we proceed to the next phase.

ii. Transformation:

In this phase, the comparison units of the source codes are transformed to another intermediate representation for easy comparison or for retrieving comparable properties. Here, at first some metrics (the data related to source code like lines of code, input data size, number of function calls, number of loops, number of operators, number of variables) are generated through static analysis of the source code. Next, the program dependency graph (PDG) for the source code is generated. Sub graphs of PDGs are used as source units or comparison units. Then the entire source code is parsed to generate the parse tree or (annotated) abstract syntax tree (AST) and the sub trees of the parse tree or AST are compared in order to find clones. The three sub-phases of the Transformation phase are discussed below in detail.

Initial Phase: In the initial phase of Transformation, several metrics from each function of the source code are retrieved and these metric values are used for finding clones. As comparison metrics, the data related to lines of code, input data sizes, number of functions, number of loops, number of operators, number of calls, loop shape and nesting level of loops of a source code are retrieved. Only those metrics are required to be retrieved at this stage that may affect the job's execution performance and hence the resource requirement of the program. The source file of a job is scanned and preprocessed statically to retrieve metrics from it. String processing of the scanned source file of a job is done to retrieve the required metrics [6].

PDG-Generation Phase: In the next phase, Program Dependency Graphs of the jobs are generated. Figure 8 shows the activity diagram for generating PDG in XML format. The control flow graph for a sequential C job is generated by using

the fdump-tree-cfg option of the GCC compiler [2, 3]. The generated control flow graph is then scanned to generate a Program Dependency Graph (PDG) for the C job. The PDG has an adjacency list representation which is implemented using linked lists and the representation is converted into XML format.

Figure 9 shows the class diagram of the PDG Generator. Here, the PDGGenerator class uses the methods in JobTypeC class to process logic for jobs in C language. JobTypeC class uses methods in CFGProcessor class which contains instance of CodeBlock class. The CodeBlock class is used to store details for each logical block of the code which corresponds to functions of the target C language. The xmlSerialize method in PDGGenerator then takes the list representation of the PDG as an argument to generate the PDG in XML format.

AST-Generation Phase: There are many tools for generating the Abstract Syntax Tree (AST) of the jobs. In this work, ASTs are generated using the tool ANTLR [4] (ANother Tool for Language Recognition). ANTLR is a language tool that provides a framework for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions containing actions in a target language. The ANTLR uses the grammar description file of a target language to generate the corresponding tokens, lexer and parser files. The grammar file for C is given as input to generate tokens, lexer and parser files. Using these files, the AST of a job is generated and represented in XML.

Figure 10 shows the activity diagram of the procedure to generate AST. Figure 11 shows the class diagram of the AST Generator. The AST Generator class uses methods of CLexer and CParser classes and has one instance of the class CprogAST class, which is the customAST tree, generated from the target C grammar using ANTLR tool. The CLexer and CParser files are generated by the tool ANTLR.

Each job, whenever it is submitted to the job modeling tool and preprocessed, it undergoes the above three phases of transformation. First the required metrics are retrieved. Then the PDG is generated and finally the AST is generated. Metrics, PDGs and ASTs corresponding to all the earlier jobs executed in the environment are stored in an Execution History [6]. Therefore the metrics, PDG and the AST of the newly submitted job can be matched with the earlier entries in the execution history to detect similar jobs. Next phase is Match Detection which describes how the descriptors of a job are matched to find clones.

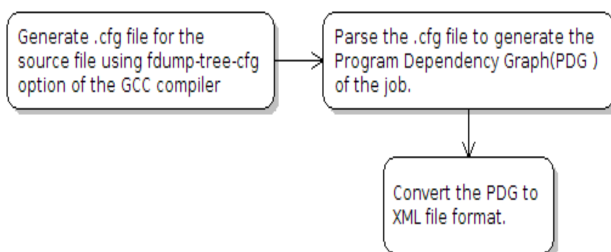


Figure 8: Activity diagram for PDG generation.

iii. Match Detection: In Match detection phase, the transformed code i.e. the metrics, PDGs and AST are used as input to a suitable comparison technique to find a match between code fragments. This comparison technique is done through the following stages – 1) metrics matching, 2) PDG matching and finally 3) AST matching.

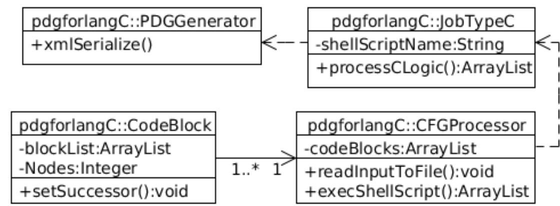


Figure 9: Class Diagram of PDG Generator

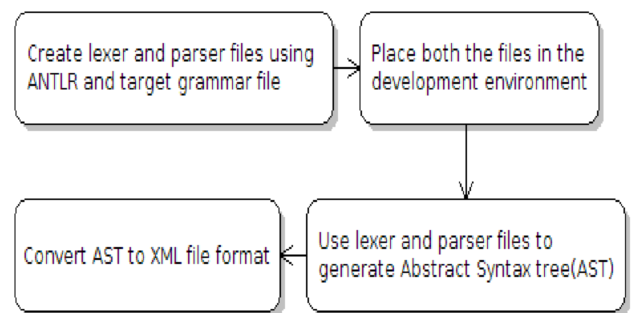


Figure 10: Activity Diagram for AST generation

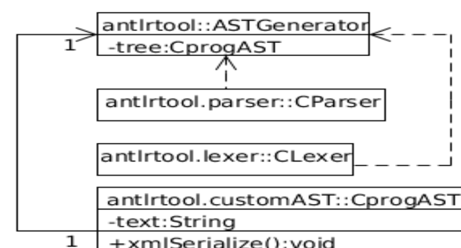


Figure 11: Class Diagram of AST Generator

For metric based comparison, the following condition must be checked.

Let, $M_x = <m_1(X), m_2(X),......,m_n(X),>$ be the tuple of metrics characterizing X; $m_i(X)$ (i=1...n) stands for i^{th} software metric used to describe X and n is the fixed number of metrics describing the entities. Here, entity represents a code. Any other entity Y described by the same set of metrics will be considered indistinguishable from X if and only if the following condition is satisfied: $m_i(X) = m_i(Y) \mid \forall i=1...n$

Metric-based matching is done to filter out the Miss-clones from probable clones. If the metrics of the newly submitted job do not match with a job in the execution history, then the job is surely a Miss-clone of the newly submitted job.

After the metrics matching phase, we proceed to the next stage, i.e. we match the PDGs of these jobs. PDG-based matching is done to match jobs at the block-level. The control

flow graph of the job is generated using the fdump-tree-cfg option of the GCC compiler [2, 3] and stored in a .cfg file. The control flow of the job in terms of basic blocks of the job is output in the .cfg file. The basic blocks of a job are determined by the GCC compiler. This .cfg file is used to derive the PDG of a job. PDGs of Miss-Clones do not match, though they match for other types of clones which have been discussed in section 3. Thus, PDG-based matching helps in differentiating between the Miss-Clones and other types of clones. PDGs of two jobs match when the following conditions are satisfied:

1) The nodes (basic blocks in the .cfg file) of the PDGs are same.
2) Both PDGs contain same sets of edges.
3) The PDGs are isomorphic.

Once the PDG-based matching phase is complete, we are able to determine the Miss-clones at the block-level matching. Next, we go for AST-based matching to determine the types of Near-Miss clones or the types of Exact clones. Thus, AST-based matching determines whether the jobs are Transformed or Scaled clones, or one of the Parameterized Renamed clones, Non-Parameterized Renamed clones and Reordered clones. It is clear that AST-based matching is done to detect clones at the statement-level. Therefore, during this phase, Miss-clones, which cannot be captured through metrics-based or PDG-based matching, can be identified at statement-level matching. The AST matching is done through the following steps.

- Number of sub-trees in the ASTs should be equal. If they are not equal, the ASTs do not match and the jobs are detected to be miss-clones at the statement-level matching through AST.
- On finding the number of sub-trees are equal, we proceed to categorize the job clones as follows:
- For Exact Clones, the ASTs are structurally and textually equal. The sub-trees are the same in both the ASTs. The ASTs are matched node-by-node to check whether they are exactly matching clones.
- For Renamed Clones, the ASTs are structurally the same and the sub-trees match. But the identifiers are different in both the jobs. If the identifiers are consistently renamed in the AST, then they are parameterized clones otherwise they are non-parameterized clones.
- For Reordered clones, the ASTs are checked for isomorphic sub-trees. Presence of isomorphic sub-trees in the ASTs indicates that the jobs are reordered clones.
- For Transformed clones, the data type of the variables declared is found to be different in the job clones. In the current implementation, the data type of the variables is only primitive data types. The ASTs have matching sub-trees that differ only in the data types of variables in both jobs.
- For Scaled clones, the data size of the input data is found to be different in the job clones but the ASTs are structurally same.
- If the sub-trees of the ASTs do not match, they are classified as Miss-clones.

After metrics-based matching, the probable clones and Miss-clones are differentiated. After PDG-based matching and AST-based matching, Near-Miss clones, i.e. Transformed and Scaled clones are determined. Other types of clones, like Exact clones, Renamed clones and Reordered clones are also determined. When PDGs and ASTs of two jobs do not match, they are filtered as Miss-clones.

5. RESULTS

Experimental Set-up

We have carried out experiments related to clone detection using the techniques proposed in the previous section. The experiments have been performed on a machine with Intel(R) Core(TM)2Duo CPU T6600 @ 2.20 GHz with 3.4GB RAM running Ubuntu Linux as the Operating System. MongoDB [23] has been used as a database for storing the Execution history.

Initially, the experiments have been carried out on the basis of an Execution History containing information about 15 different sequential compute-intensive jobs. A total number of 10 new jobs have been submitted one after another, their BE data have been extracted and our proposed hybrid clone detection technique is applied to determine the clone type of each new job in comparison with the 15 jobs stored in the Execution history.

Here, only the source codes implemented in C language are considered. The source codes include programs for molecular dynamics, matrix multiplication, gauss elimination, calculation of π, factorial of large numbers, computing Fibonacci series, different searching and sorting algorithms etc.

Table1 presents the BE (metrics) data and AE data of the 15 jobs stored in Execution History. The metrics are Lines of Code (LOC), Number of Loops, Number of Functions (NOF), Number of Operators (NOOP), Number of variables (NOV) and CPU usage in terms of Processor Time. The BE metrics are chosen with the consideration that they may affect the computation involved and thereby the resource requirements (e.g. CPU usage) of the program. Figure 12 shows sample AST and PDG of jobs in XML format which are stored in Execution History.

Experimental Results

Table 2 shows the clone types of the incoming new jobs in comparison with the stored jobs in Execution History. XX in the table denotes that the indicated phase of match detection need not be performed for a particular job.

Table1: BE and AE data of 15 sample jobs in Execution History.

Job#	BE data					AE data
-	Lines of Code (LOC)	Number of Loops (NOL)	Number of Functions (NOF)	Number of Operators (NOOP)	Number of Variables (NOV)	Processor Time (ms)
HJ1	11	0	1	8	3	0.12
HJ2	11	0	1	8	2	0.14
HJ3	13	3	1	8	2	0.16
HJ4	13	1	1	9	3	0.1
HJ5	18	1	2	8	3	0.14
HJ6	21	4	1	16	6	2230
HJ7	22	1	2	19	6	0.14
HJ8	23	1	1	16	3	0.1
HJ9	24	0	1	12	4	0.09
HJ10	26	1	2	17	4	0.45
HJ11	27	0	2	22	2	0.15
HJ12	61	8	2	72	13	0.16
HJ13	74	5	2	82	15	0.18
HJ14	121	26	1	205	13	157518
HJ15	162	24	7	384	48	87024

```
<HJ1>
  <PDG>

    <Function name="main">
      <BLOCK>
        <Number>1</Number>
        <Successor>2</Successor>
      <BLOCK>
      <BLOCK>
        <Number>2</Number>
        <Successor>4</Successor>
      <BLOCK>
      ---
      ---
    </Function name>
      ---
  </PDG>

  <AST>
    ----
    <clonedetection.antlrtool.customAST.CprogAST text="=" type="26">
      <clonedetection.antlrtool.customAST.CprogAST text="j" type="4" />
      <clonedetection.antlrtool.customAST.CprogAST text="0" type="7" />
    </clonedetection.antlrtool.customAST.CprogAST>
    <clonedetection.antlrtool.customAST.CprogAST text=";" type="24" />
    <clonedetection.antlrtool.customAST.CprogAST text="j" type="4" />
    ----
  </AST>
</HJ1>
<HJ2>
  <PDG>
    ----

  </PDG>
    ---

</HJ2>
```

Figure 12: Sample AST and PDG of jobs in Execution History.

Predicting Resource Requirements

Once the clone type of a new job is determined with respect to the jobs stored in the execution history, statistical methods are applied to estimate the resource requirements of the new job on the basis of the AE data of the clone job. Some of the statistical techniques used in our scheme have been discussed in [6]. Detailed discussions of those techniques are not within the scope of this paper. Also, here we are predicting only the CPU usage of the new job.

Table 2: Detection of clone level of New jobs in comparison with the jobs in Execution History.

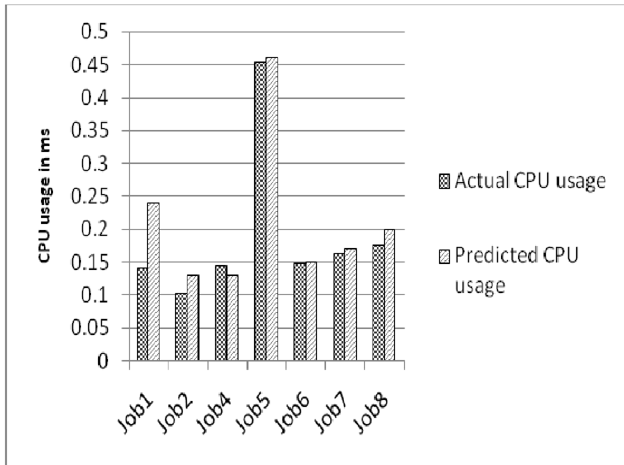
New Job	After metrics matching	After PDG-based matching	After AST-based matching	After AST/PDG processing	Clone Type
Job 1	not matched	XX	XX	-	Miss-Clone
Job 2	matched	matched	matched	Datasize mismatch	Scaled Near Miss Clone
Job 3	matched	matched	matched	Datatype mismatch	Transformed Near Miss Clone
Job 4	matched	matched	matched	Arithmatic operator mismatch	Arithmetic Operator Mismatch Near Miss Clone
Job 5	matched	matched	matched	Datatype mismatch	Transformed Near Miss Clone
Job 6	matched	matched	matched	Exact AST match	Exact Alike Clone
Job 7	matched	matched	matched	Identifier mismatch	Renamed Alike Clone)
Job 8	matched	not matched	matched	-	Loop Mismatch Alike Clone
Job 9	matched	matched	matched	Statement Order mismatch	Reordered Alike Clone
Job10	matched	matched	matched	Compound Statement mismatch	Compound Statement Alike Clone

Once the resource requirements are predicted, we execute the new job on a resource provider as suggested by our PRAGMA tool [24][25]. Figure 13 (a) and (b) compare between the actual CPU usage and predicted CPU usage of new jobs which have been used for our experiments.

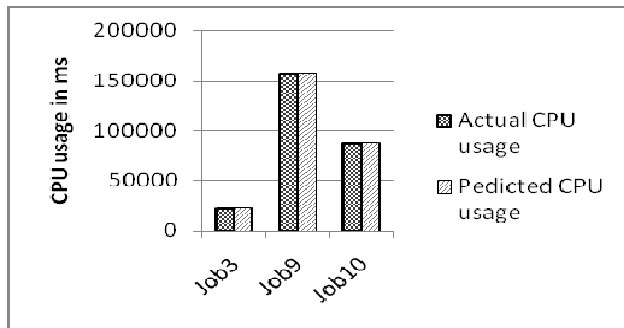
In the results shown in this paper, whenever a near-miss clone or an Alike clone is found for a new job, resource requirements are predicted on that basis. We have not considered the chances of finding multiple clone jobs (of different types) of a particular new job. In our future work we shall assign preferences to the clone types to handle the situation when multiple clone jobs are found.

Overheads

Figure 14 depicts the aggregate of BE, PDG and AST extraction times as the total overhead against the lines of code of the jobs. It can be seen that, as expected, the BE metric extraction time gradually increases with the increase in the LOC. Although the increase is not uniform, as depicted in the graph. The reason behind this non-uniform increase is that the BE extraction time also depends on other metrics like nesting levels of loops, shape of the loops etc. The PDG extraction time also gradually increases with the LOC of the jobs. However, as shown in the figure, this increase is not always uniform, because the time for PDG extraction also depends on the number of functions in a job. Figure 15 shows the total time taken for the matching phase, i.e. for metric-matching, PDG-matching and AST- matching. This time also varies with LOC of jobs. Hence, it can be said that clone detection overhead primarily depends on LOC of jobs.



(a)



(b)

Figure 13: Comparison between Actual CPU usage and Predicted CPU usage for (a)Low compute intensive jobs (b) High compute intensive jobs.

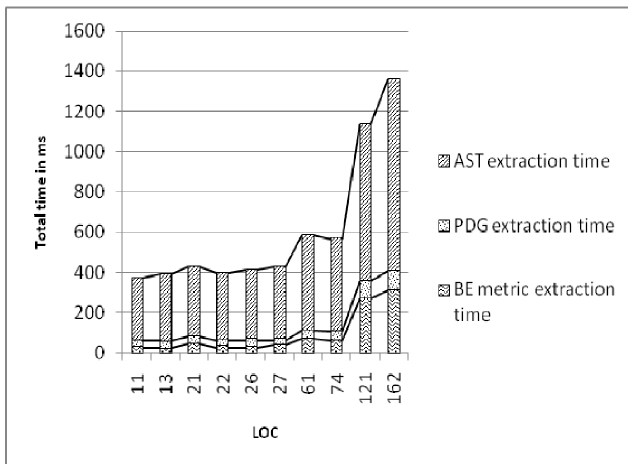


Figure14: Total overhead for transformation phase.

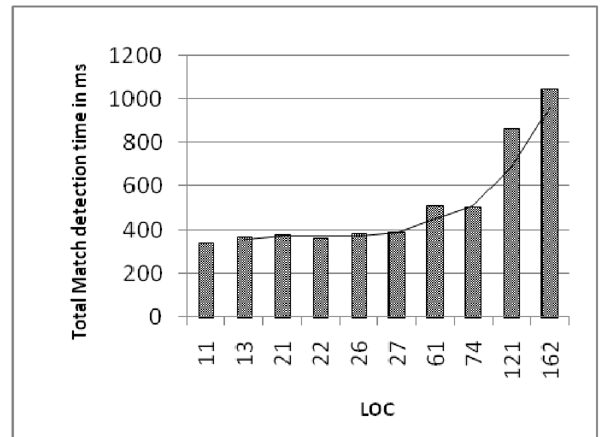


Figure 15: Total overhead for match detection phase.

6. RELATED WORK

Analysis of clones in software engineering has been discussed in [21] and [22]. The papers also discuss the usage of clone detection in software maintenance. A review of the existing clone taxonomies has been provided in [1]. We have extended the taxonomy to define some new categories of clones. The extension is required for job modeling purpose.

The clone detection techniques like Metrics-based, PDG-based, AST-based and hybrid clone detection techniques have been discussed in [1]. In [7], Mayrand et al. estimated several metrics, such as number of lines of source code, number of function calls contained, number of CFG edges, etc. for each function unit of a program. Metric-based clone detection technique has also been executed for finding duplicated web pages or finding clones in web documents. Di Lucca et al. [18] proposed metric based clone detection approach to find similar static HTML pages by estimating the distances between items in web pages and evaluating their degree of similarity. In our work, metric-based technique has been used as the first step for filtering out the miss-clone jobs. We have considered certain parameters of the source file as the metrics like number of operators in the source file, number of local and global variables in the source file, number of loops in the source file etc. as mentioned in the Table 1 of Section 5.

A brief overview of program dependence graphs is given in [2]. Detecting clones using program dependency graphs is discussed in [3, 11]. In [17], Komondoor and Horwitz's proposed a PDG-based clone detection approach PDG-DUP that finds isomorphic PDG sub-graphs using program slicing [12].

Baxter et al proposed a technique for clone detection based on abstract syntax tree [12]. They partitioned sub-trees of the abstract syntax tree of a program based on a hash function and then compared sub-trees in the same partition through tree matching (allowing some divergences) [12]. A similar approach was proposed earlier by Yang [19]. They recognized syntactic differences between two versions of the same program by generating alternative parse trees for both the versions and then they applied dynamic programming approach

[20] in searching similar sub-trees. An XML-based clone detection model has been discussed in [10].

Some other techniques [13, 14, 15, 16] are also proposed by researchers based on the combination of different above mentioned techniques. We have used a new hybrid clone detection technique which collectively uses metrics-based, PDG-based and AST-based clone detection techniques and uses XML based matching of PDGs and ASTs, to detect clones.

Overview of our job modeling technique has already been discussed in [6]. To make clone detection technique more reliable and robust, in this paper we concentrate on other clone detection methodologies and also propose some different clone jobs that can be used for resource requirement prediction in Grid.

7. CONCLUSION AND FUTURE WORK

The paper focuses on the use of clone detection techniques for resource requirement prediction of jobs running in a large and dynamic distributed system. Because estimation of resource requirement is the primary objective, a new taxonomy of clones is proposed in this paper. A hybrid clone detection technique is also proposed and its efficiency is demonstrated by carrying out experiments with some test codes. We shall extend our work by carrying out these experiments in a larger environment and with more real-life codes.

In future, the statistical methodologies will be further improved to predict the resource requirements for every job and also the suitable resource provider based on the dynamic status of the resource providers within the environment. Our future work will also focus on efficient storage of the execution history and data retrieval.

REFERENCES

[1] C. K. Roy and J. R. Cordy (2007), “A Survey on Software clone Detection Research Techniques”, 115(2007-541), 115. Citeseer. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.62.7869.

[2] Robert A. Balance and Arthur B. Maccabe, Department of Computer Science, The University of New Mexico: “Program Dependence Graphs for the Rest of Us”, October 1992.

[3] Kelton Augusto Pontara, Valentin , and J.L. Silva , University of Sao Paulo ,Brasil : “Chipcflow - A Dynamic Dataflow Machine Compiler to convert C into a dataflow graph.”, April 2009. Retrieved from http://www.chipcflow.eti.br/Kelton_journal.pdf

[4] ANTLR, web site http://www.antlr.org/

[5] Ira Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant Anna. “Clone Detection Using Abstract Syntax Trees”. In Proceedings of the 14th International Conference on Software Maintenance (ICSM’98), pp. 368-377, Bethesda, Maryland, November 1998.

[6] Madhulina Sarkar, Sarbani Roy, Nandini Mukherjee, “Feedback-guided Analysis for Resource Requirements in Large Distributed System”, published in 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid 2010).

[7] Jean Mayrand, Claude Leblanc, Ettore Merlo. “Experiment on the automatic Detection of Function clones in a Software System Using Metrics”. In Proceedings of the 12th International Conference on Software Maintenance (ICSM’96), pp. 244-253, Monterey, CA, USA, November 1996.

[8] Raghavan Komondoor and Susan Horwitz. “Using Slicing to Identify Duplication in Source Code”. In Proceedings of the 8th International Symposium on Static Analysis (SAS’01), Vol. LNCS 2126, pp. 40-56, Paris, France, July 2001.

[9] M. Weiser. “Program Slicing”. In IEEE Transactions on Software Engineering, Vol. 10(4):352-357, July 1984.

[10] ZhongMei and Liu Dongsheng. An XML Plagiarism Detection Model for C Program .In Proceedings of the 3rd International Conforence on Advanced Computer Theory and Engineering(ICACTE) , 2010.

[11] Jens Krinke. Identifying similar code with program dependence graphs. In Proceedings of the Eighth Working Conference on Reverse Engineering(WCRE) , 2001.

[12] M. Weiser. “Program Slicing”. In IEEE Transactions on Software Engineering, Vol. 10(4):352-357, July 1984.

[13] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, Kostas Kontogiannis.” Measuring clone Based Reengineering Opportunities”. In Proceedings of the 6th International Software Metrics Symposium (METRICS’99), pp. 292-303, Boca Raton, Florida, USA, November 1999.

[14] LingxiaoJiang, GhassanMisherghi, ZhendongSu, and Stephane Glondu.”DECKARD: Scalable and AccurateTree-based Detection of Code clones”. In Proceedings of the 29th International Conference on Software Engineering (ICSE’07), pp. 96-105, Minnesota, USA, May 2007.

[15] Rainer Koschke, Raimar Falke and Pierre Frenzel. “Clone Detection Using Abstract Syntax Suffix Trees”. In Proceedings of the 13th Working Conference on Reverse Engineering (WCRE’06), pp. 253-262, Benevento, Italy, October 2006.

[16] Robert Tairas, Jeff Gray. “Phoenix-Based clone Detection Using Suffix Trees”. In Proceedings of the 44th annual Southeast regional conference (ACM-SE’06), pp. 679684, Melbourne, Florida, USA, March 2006.

[17] Raghavan Komondoor and Susan Horwitz. “Using Slicing to Identify Duplication in Source Code”. In Proceedings of the 8th International Symposium on Static Analysis (SAS’01), Vol. LNCS 2126, pp. 40-56, Paris, France, July 2001.

[18] G.A. Di Lucca, M. Di Penta, and A.R. Fasolino and P. Granato. “Clone Analysis in the Web Era: an Approach to Identify cloned Web Pages”. In Proceedings of the 7th IEEE Workshop on Empirical Studies of Software Maintenance (WESS’99), pp. 107-113, Florence, Italy, November 2001.

[19] Wuu Yang. “Identifying syntactic differences between two programs”. In Software Practice and Experience, 21(7):739755, July 1991.

[20] D. S. Hirschberg. “A linear space algorithm for computing maximal common sub-sequences”. Communications ACM, 18(6):341-343, June 1975.

[21]Cory J. Kapser and Michael W. Godfrey, “Supporting the analysis of clones in software systems”. Journal of Software Maintenance and Evolution: Research and Practice 2006; 18(2):61–82.

[22]Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou, “CP-Miner: Finding copy-paste and related bugs in large-scale software code”. IEEE Transactions on software Engineering 2006; 32(3):176–192.

[23] Sushan Chakraborty, Madhulina Sarkar and Nandini Mukherjee, “Implementation of Execution History in Non-relational Databases for Feedback-guided JobModeling” published in CUBE 2012International IT Conference and Exibition..

[24] Roy S., N. Mukherjee: “Adaptive Execution of Jobs in Computational Grid Environment”, Published in Journal of Computer Science and Technology, Springer, vol.24, No.5, September 2009.

[25] De Sarkar A., S. Roy, D. Ghosh, R. Mukhopadhyay, N. Mukherjee : “An Adaptive Execution Scheme for Achieving Guaranteed Performance in Computational Grids”, Published in the Journal of Grid Computing, ISSN: 1570-7873 (Print) 1572-9814 (Online), vol. 8, pp. 109-131, 2010.