Efficient and Scalable Collection of Dynamic Metrics using MapReduce

Shallu Sarvari
National Institute of Technology
Jalandhar, INDIA
E-mail: sarvari.shallu@gmail.com

Paramvir Singh
National Institute of Technology
Jalandhar, INDIA
E-mail: singhpv@nitj.ac.in

Geeta Sikka
National Institute of Technology
Jalandhar, INDIA
E-mail: sikkag@nitj.ac.in

***Abstract*— Dynamic metrics are known to assess the actual behavior of software systems as they are extracted from runtime data obtained during program execution. However, recent literature indicates that dealing with dynamic information remains a formidable challenge due to the huge size of execution data at hand, resulting in long processing delays. We present an efficient and scalable technique to extract design level dynamic metrics from Calling Context Tree (CCT) using cloud based MapReduce paradigm. CCT profiles having node count up to 40 million are used to extract a number of dynamic coupling metrics. On an average, 73% increase in performance is observed as compared to sequential analysis. Also other performance characteristics like speed-up and scale-up are analyzed to strengthen the applicability of our parallel computation approach.**

Keywords—Dynamic Metrics; Hadoop MapReduce; Calling Context Tree; Program Profiling; Performance Analysis; Cloud Computing

I. Introduction

Software product metrics allow us to make quantified and meaningful estimates for a product, ensuring sound managerial and technical decisions that are needed to build quality software. There are two types of software metrics - static and dynamic. Due to the occurrence of runtime features such as polymorphism, dynamic binding, inheritance and unused code, the traditional static metrics have been found to be inaccurate while dealing with modern object-oriented software. Since dynamic metrics are computed using data collected during actual execution of the system, they are capable of capturing the impacts of above mentioned runtime features enabling an accurate assessment of software quality attributes.

For the collection of dynamic metrics, dynamic analysis is required which involves processing large amount of dynamic information. The navigation and exploration of such data for the collection of a particular dynamic metric turns out to be quite a challenging task that hinders the widespread adoption of dynamic metrics. In this respect, the metric collection performance in terms of computational time could be enhanced through parallelization. Hadoop MapReduce [27] is known to be an ideal technique to develop such highly scalable applications that swiftly process massive amount of data in parallel over clusters of computing nodes. This choice of technique was also stimulated by the fact that it has become the de-facto standard for parallel computation in industry, and is well supported to work on the cloud [3]. On the basis of these considerations, we propose and demonstrate a distributed and parallel dynamic metric evaluation approach using Hadoop MapReduce. The efficiency of the proposed parallel approach for dynamic metric evaluation as compared to the single machine processing is empirically evaluated.

The rest of this paper is organized as follows. Section II briefly discusses the background of the research problem. Section III overviews related concepts and research work. Study design is detailed in Section IV. Section V explains the customization of JP2 tool to assist the parallelization process. Section VI describes our Map and Reduce algorithms for dynamic metric collection. The results are presented in Section VII. Finally, the paper is concluded with some useful future recommendations in Section VIII.

II. Background

A number of dynamic metrics [2][5-10] have been proposed over the years for a variety of software engineering tasks such as program comprehension, re-engineering, etc [12-16]. These metrics have the potential to be good indicators of quality attributes of software such as maintainability, testability, understandability, error-proneness, etc [7, 33].

There exist different methods to collect dynamic metrics [1]. As the areas of dynamic metrics and dynamic analysis are naturally interconnected, the most frequent dynamic metric evaluation method is through execution trace mining performed using various dynamic analysis techniques [2,7], which is also the focus of our study. An alternate method is to use the data collected from simulation of runtime behavior based on executable modules and interaction diagrams for metric evaluation [6]. The latter is considered to be not as accurate and precise as its execution-based counterpart.

Despite many advantages, the number of experiences and industry reports using dynamic metrics is very less; reason being the scalability issues of the underneath dynamic analysis techniques [18]. One of the major issues is to extract relevant metrics from large amount of data generated during software execution [17]. The processing of this much data takes time and thus creates delay in getting the final metric values. The kind of resources required to process huge amount of execution data is also an obstacle in the acceptance of

1530-1362/15 $31.00 © 2015 IEEE
DOI 10.1109/APSEC.2015.21

dynamic metrics. Hence overall these scalability issues are hampering the validation work required to explore and unleash the real potential of such metrics. This work focuses on defining efficient and scalable approach for the collection of dynamic coupling metrics. Although scalability problem also relates to the memory overhead of the profiling task, our focus is towards the time overhead incurred while extracting dynamic metrics from large runtime data. A potential future direction is seen in large scale industrial studies to evaluate the usefulness of dynamic metrics in real life scenario [11].

III. Related Concepts and Previous Work

A. Overview of Related Concepts

1) Calling Context Tree (CCT)

A sequence of methods on the call stack at some point during the program execution is referred to as a calling context. A data structure that records all the unique calling contexts of an execution scenario is known as a CCT [24]. Fig. 2(a) represents a conceptual diagram of an example CCT. Each node in the tree symbolizes a method invocation and creates a child node for every distinct method it invokes. Hence, each path from root to a node of the tree represents a distinct calling context. When the same method is invoked in different calling contexts (or different callsites), it is represented by distinct nodes in the CCT (like *digitHandler* method in Fig. 2(a)). However, if a method is invoked in the same calling context several times (and also from the same callsite), the execution count is incremented for the same CCT node (like *action* method in Fig. 2(a) having an execution count of 4). Any number and type of platform-independent (execution count of methods, number of executed bytecodes) or platform-dependent (CPU time, number of cache misses) dynamic metrics can be kept in a CCT node. In our approach, we evaluate the dynamic metrics from CCTs generated using a calling context profiler, JP2 [25].

2) MapReduce

MapReduce is an elegant and flexible paradigm which enables the development of large-scale distributed applications [26]. It consists of two distinct functions - *Map* and *Reduce*, which are combined together in divide and conquer fashion. *Map* handles parallelization while *Reduce* collects and merges the results. A master node splits the initial input in several pieces; each one of which is identified by a unique key. Further, it distributes them via the *Map* function to the slave nodes (i.e. Mappers) which work both in parallel and independent to each other, while performing the same task on different pieces of input. The *Reducer* (machine running the reduce task) identifies and collects the output as soon as each *Mapper* finishes its job. Each *Mapper* produces a set of intermediate key/value pairs. All the intermediate values associated to the same key are grouped together which are then exploited by one or more *Reducer* to compute the list of output results. The model automatically invokes and allocates a number of distinct *Reducers* that correspond to the number of distinct intermediate keys. The MapReduce framework fragments the input data, schedules and executes *Map* and *Reduce* tasks on existing machines (in cluster), manages communication and handles the transfer of data among machines. In our approach, we use the most popular open-source implementation of MapReduce framework, Apache Hadoop [27].

B. Related Work

Although, the evaluation of dynamic metrics has been carried out several times in past, none of the previous works focused on the performance overheads of metric collection mechanisms [8-9][20]. In most of the cases, software under study are too small to cause any performance related issues [5][10][19][31]. Mitchell and Power [9] defined a new set of dynamic metrics, and mentioned the problems faced while extracting metrics from large volumes of data. They performed their experiments on the applications included in SPECjvm98 benchmark suite [35]. Later, Mitchell and Power [20] also quantified the run-time behavior of Java GUI applications to state that dynamic analysis of any program involves a huge amount of data processing. Hassoun *et al.* [19] in their empirical validation study of dynamic coupling metrics considered the scalability of models as a threat to validity. Sarimbekov *et al*. [28] measured the calling context sensitive Java bytecode metrics from XML file (containing CCT profile) created using JP2. They mentioned that the time and space consumption of such an analysis depends largely on the XQuery processor used, and hence it becomes crucial to use a processor that is proficient in streaming large input document (like the XML-based calling-context-tree profiles) since their size may exceed the main memory limits. Sewe *et al.* [29] characterized the workloads of DaCapo and Scala benchmarks on the basis of dynamic metrics' evaluations, carried out on a considerably powerful machine (i.e. an Opteron machine with 40GiB RAM).

MapReduce paradigm has been used in various areas of software engineering research where scalability is known to be an issue. Sajnani *et al.* [21] implemented a MapReduce based parallel algorithm for code clone detection that is capable of scaling to thousands of projects. Geronimo *et al.* [22] proposed a parallel genetic algorithm that uses MapReduce to automatically generate JUnit test suites. Bianculli *et al.* [23] exploited the MapReduce framework to check specifications expressed in a metric temporal logic with aggregating modalities (over large execution traces).

To the best of our knowledge, MapReduce paradigm has not been yet explored to tackle the scalability issue faced while processing large runtime data for collecting dynamic metrics. There is a body of research towards trace reduction techniques [32] that aims for fast dynamic analysis at the cost of accuracy; but our focus is rather towards the complete and accurate measurement of dynamic metrics.

IV. Study Design

This section presents the research questions and experimental study design that includes dynamic coupling metrics under consideration, sample applications, tool used for profiling the sample applications, methodology and experimental setup.

A. Research Questions

We aim to investigate factors which may support the fact that MapReduce paradigm is apt for efficient and scalable collection of dynamic metrics. For this purpose, we have defined four research questions:

a) RQ1: Could the dynamic information be processed in parallel by dividing it into smaller independent chunks?

b) RQ2: Is the MapReduce approach for dynamic metric evaluation on multiple machines faster as compared to single machine processing?

c) RQ3: How does the MapReduce approach for dynamic metric evaluation handle scale-up and speed-up?

d) RQ4: How does the MapReduce approach for dynamic metric evaluation perform as the number of base program units (classes in this case) submitted as a batch for evaluation increase?

Scale-up is defined as the ability to provide additional resources that do not allow the performance (in terms of time) to drop when there is an increase in load. Speed-up is defined as the gain in time when more number of resources is allotted to the same job.

B. Dynamic Coupling Metrics

Tahir et al. [1], in their systematic mapping study on dynamic metrics, found that coupling is the most widely studied design quality attribute at runtime. Thus, we chose a set of dynamic coupling metrics for the evaluation of our MapReduce approach. Dynamic coupling (class level) is intended to be measured in two forms – when a class is accessed by another class at runtime, and when a class accesses other classes at runtime (i.e. to account for both callers and callees). The following class level metrics selected for this study are designed to work in both directions i.e. both import and export, and are collected using method invocations captured during program execution at different levels of granularity namely object/class, method, message and callsite.

1) Total Dynamic Messages: Within a runtime session, the total number of messages sent from (or received by) a class to (or from) other class, is considered as Total Dynamic Messages (TDM).

2) Distinct Class Couples: The number of distinct classes that a particular class interacts with, is defined as Distinct Class Couples (DCC).

3) Distinct Method Couples: The number of distinct method couples drawn from the method calls to (or from) a particular class is defined as Distinct Method Couples (DMC).

4) Hot Callsite (HC): A callsite refers to the exact instruction inside a method of a class where the call is made. Hot Callsite inside a class is the coupling contributing callsite having maximum number of executions in a single run.

C. Inputs

For inputs selection, there was a clear criterion: the number of nodes generated in the resultant CCT after a single JP2 run for a specific application. We selected two applications (Eclipse-large and Jython-default) from the DaCapo-9.12-bach suite [30] which produced the desired largest CCT profiles

TABLE I. INPUT APPLICATIONS

	No. of Classes (traced)	No. of CCT Nodes
Eclipse (large)	1235	40544790
Jython (default)	578	8760832

These applications turned out to be appropriate for such a study that targets scalability and performance issues. The DaCapo benchmark applications can be divided into four categories on the basis of size namely small, default, large and huge (not available for all applications). We needed the profiles to have varying node counts for the same application in order to test scale-up. Hence, Eclipse profiles of different sizes from small (4316973) to large (40544790) were undertaken. Table I shows the attributes of selected applications. For more on selected applications, refer [30].

D. Tool Used – JP2

We used an open source tool JP2 [25] to produce CCT profiles for our evaluation. JP2 is a calling context profiler for the Java Virtual Machine, which collects complete and accurate profiles. It is also able to differentiate between multiple call sites of the same method. JP2 incurs small overhead which makes it possible to profile a wide range of real-world applications. JP2 profiles were all captured on Intel(R) Core(TM) i3-4010U CPU @ 1.70Ghz×2 with 8Gib RAM extended to 16 GiB via swap space running Ubuntu-14.04.1 operating system. We used OpenJDK Runtime Environment (build 1.7.0_65) with HotSpot 64-Bit Server VM (build 24.65-b04, mixed mode).

E. Methodology

Fig. 1 gives an overview of our experimental methodology. We decided to use the cloud service of MapReduce for our evaluation. To start with, a *custom dumper* is designed for the JP2 profiler so that we get the CCT in the form of multiple independent XML files, which can be processed in parallel (the approach used is explained in Section V).

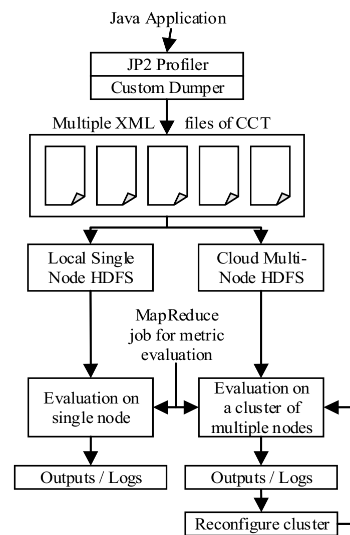

Fig. 1. Methodology followed in our work

A Hadoop MapReduce job is created for the evaluation of selected dynamic coupling metrics. The detail of this job is given in Section VI. We start the metric evaluation process by passing the Java application to the customized JP2 profiler. The resulted profiler output is a set of multiple XML files of the CCT generated using our custom dumper. These files are uploaded to the HDFS (Hadoop Distributed File System) on local machine as well as cloud. We study the performance, output and logs of the MapReduce jobs both on cloud and local machine. The steps to set up the cluster, reconfigure Hadoop and execute the MapReduce job are iterated on cloud several times in order to study the effects of scale-up, speed-up and increase in number of classes submitted as a batch for evaluation.

F. Hadoop Setup – Local and Cloud

1) Local: For the local evaluation, a single node Hadoop 2.6.0 cluster on a machine with 4Gib RAM and Intel Core 2 Duo CPU T6400 @ 2.00GHz × 2, running Ubuntu 14.04.1 operating system was set up. The same OpenJDK and the HotSpot Server build was used as specified in Subsection D. This machine specification was used because it is comparable to the specification of the nodes in the Hadoop cluster used on cloud. Enough heap space was given to the metric evaluating MapReduce job so that it could perform as efficiently as possible. The heap space matters when the job's mapper tries to parse the XML document.

The implemented Hadoop MapReduce job allows passing any number of class names as arguments to evaluate the dynamic metrics in class batches. We selected 20 most frequent and well distributed classes with respect to each input XML file.

2) Cloud: We used Amazon Web Service's Elastic MapReduce (AWS-EMR), a Hadoop distribution service, to set up our cluster for the parallel evaluation on cloud [34]. In order to make sure that the exact performances of the respective MapReduce jobs for a single node and the multiple nodes are compared, we used an extra initial step of transferring data to the HDFS of cloud cluster. The multiple files to be transferred were in a gunzip format, and hence the file sizes as large as 12GB were compressed to a mere 427MB whose transfer was not a problem. A single node in a cluster consisted of EC2 instances (machine type on AWS) having two EC2 *Compute Units* (two physical cores) of Amazon's previous generation processors with 7.5GiB RAM running on a linux terminal. We used Hadoop 2.4.0 distribution in the EMR cluster. EMR allows three groups of nodes - master, cores and slaves. A master node is responsible for task distribution and tracking. EMR does not allow the master node in a Hadoop cluster to be used for computation tasks. Hence, a single EC2 Compute unit machine with 3.5 GiB RAM was allotted for the master node in each cluster setup. After careful testing, 1792 MB memory was provided to each map task so that maximum number of map tasks could be launched.

Cost: One EC2 *Compute Unit* of AWS EMR costs $0.307. For a cluster having one EC2 instance of 1 *Compute Unit* and 3 instances of 2 *Compute Units*, the cost will be $2.152 per hour.

V. CUSTOM JP2 DUMPER

In this Section, we explain how the single CCT file generated by the calling context profiler JP2 is transformed into multiple files appropriate for parallel processing in MapReduce. When JP2 profiles an application, it dumps the resulting CCT from the memory into a single file in Text/XML format. We selected XML format as it is best suited to explore the nested structure, and also because quite efficient parsers are available for XML format files. The output XML file contains a highly nested structure which would result in semantic loss if division is just based on the length of the file. We came across various lossless techniques which could be deployed to divide an XML document but all such techniques would require an extra pass on this large single file. Hence, in order to avoid the information loss or an extra pass, a custom dumper class for JP2 was designed which dumps the CCT (in memory) directly into multiple independent files.

Since the coupling among only the classes belonging to the sample applications are of interest, any calls to (or from) the methods of Java library classes are omitted by blocking the instrumentation of such classes. Also basic block execution data is omitted on the basis of study scope.

To accomplish this task, default XML dumper in JP2 is customized. While dumping the complete CCT from main memory into the XML file, all open tags in the XML document are recorded separately for predefined time duration at the end of which open tags are closed and so is the file. In the new file, firstly the previously opened tags followed by a special tag <newFile> are written, so that the metric evaluation program can identify the duplicate tags*. Further, the CCT dumping carries on in the new file. The increase in file size due to extra tags incurs negligible overhead. For example, a single CCT file of *fop* (the smallest application in DaCapo benchmark) has 22935KB size; while the total size of multiple CCT files is 22974KB, i.e. a space overhead of only 39KB. When the file size reaches gigabytes, the difference is even more inconsequential.

This approach does not require an extra pass over CCT. Also the extra overhead with respect to time is very small. To dump the Eclipse profile containing 16 million CCT nodes, our multiple file dumper approach took 271019 msec (~ 4.52 minutes); while JP2's single file dumper took 241864 msec (~ 4.03 minutes) i.e. a negligible time overhead of 29155 msec.

VI. MAPREDUCE FOR DYNAMIC METRICS

As discussed in Section III, there are two main functions in a Hadoop job – Map and Reduce. There are specific input types that Hadoop MapReduce can handle like text, binary, etc. but unfortunately XML is not one of them. In order to handle the XML format files, two classes of Hadoop MapReduce framework are overridden i.e. *FileInputFormat* and *RecordReader*. Their aim is to instruct the MapReduce job for the division of data meant for parallel computation.

1) Mapper: As defined earlier, mapper will produce the intermediate key/value pairs from the split provided by *RecordReader*. Fig. 3 describes the pseudocode of our Map

*Refer http://pvsingh.com/s_sarvari/ for additional information.

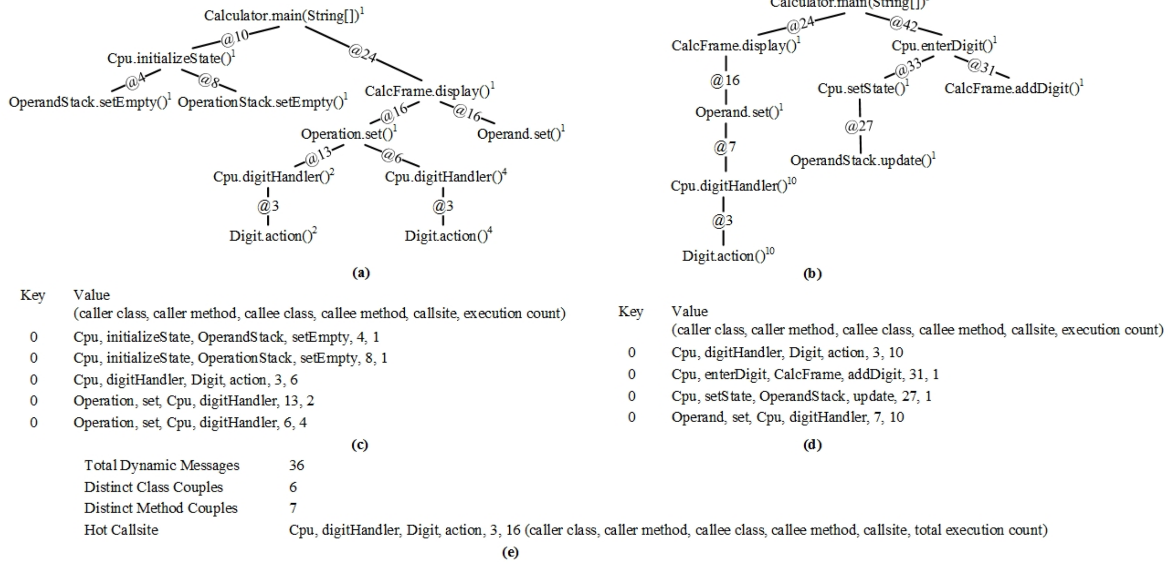

Fig. 2. (a) and (b) represent example CCT profiles generated by the custom dumper. *Superscript* of nodes depicts the execution count and *weight* on the branch shows the callsite. (c) and (d) show the respective outputs of the Map and Combine functions. Note that the execution count of *Action* method in the example is aggregated by the combiner. Also the *0* key value corresponds to the *Cpu* class. (e) shows the output of reduce function as metric values.

```
1  Map(key=null, value=xml file as Text)
2  begin
3    xmlDoc ← value
4    for each class c given for evaluation do
5        extract the <newFile> element tag from xmlDoc
6        beforeList ← All the elements from root till new file tag
7        for each element e in beforeList do
8          if(declaringClass(e)=c)
9          begin
10           childElement←child of method type(child of callsite type(e))
11           if(declaringClass(childElement)≠c) and childElement also exists in
               beforeList)
12             add the childElement to childDuplicateList
13           if(parent(e) ≠ root and declaringClass(parent(e)) ≠ c)
14             add the element to coupleDuplicateList
15         end
16       end
17       coupleList←Extract the list of method elements from xmlDoc containing
           declaringClass=c
18       for each element e in coupleList do
19         if(e≠child(root))
20         begin
21           if(declaringClass(parent(e))≠c)
22           begin
23             if element e also exists in coupleDuplicateList
24               delete element from coupleDuplicateList
25           end
26           else
27             arrayString←declaringClass(parent(e)),name(parent(e)),declaring-Class(e),
                 name(e), executionCount(parent(e)), callsite(parent(e))
28             output (c,arrayString)
29         end
30       end
31       childCallsiteList ← Extract the list of children of type callsites
32       execCount= execitionCount(e)
33       for each element cc in childCallsiteList do
34         childMethodList ← Extract the list of child methods
35         for each element ccl in childCallsiteList do
36           if(declaringClass(ccl)≠c)
37           begin
38             if the element ccl also exists in childDuplicateList
39               delete the element at no. flag from childDuplicateList
40           end
41           else
42             arrayString ← declaringClass(e), name(e), declaringClass(ccl), name(ccl),
                 execCount, callsite(e)
43             output (c,arrayString)
44         end
45       end
46   end
47 end
```

Fig. 3. Pseudocode for Map function.

function. It starts by building an XML document from the *Text* value (Fig. 3, line 3) which contains a single extracted XML file. It finds the tags listed before the special tag *<newFile>*, so that duplicate couples can be removed (Fig. 3, line 5-16). The elements containing the required class (under measurement) are extracted from the XML file into a list (Fig. 3, line 17). For each element in this list, there are two tasks to be performed. First, find the parent class and check if export coupling exists (Fig. 3, line 18-30). Second, find all the relevant children from immediate callsites and check if import coupling exists (Fig. 3, line 31-44). While doing this, duplicate tags inserted in the XML file before the *<newFile>* tag are also taken care of. The output of the mapper contains index of the class as *key* and the details of import/export couples found for that class as *value*. These metric collection steps are iterated for each class in the batch.

2) Combiner: Combiner function sums up the execution counts of output couples (key/value pairs resulting from map function) containing the same caller class, caller method, callee class, callee method and callsite for each key. This function also executes at mapper and limits the data transfer between map and reduce tasks.

3) Reducer: The reducer is fired once for every key after collecting key/value pairs from all the mappers. Its execution is deferred in the job configuration until all the map tasks are completed, because our Reduce function can perform logical aggregation only after all the files have been processed. Reducer calculates the selected metrics for each class by aggregating the outputs of each mapper. For each value belonging to a specific key, the reducer: a) increments the dynamic message count (Fig. 4, line 4); b) checks if the value

contains a newly found unique couple involving the given class (Fig. 4, line 5-11); c) checks if it forms a distinct import (export) method couple with the methods of key class (Fig. 4, line 12-18); d) aggregates the execution count for all the callsites in the given class that are responsible for coupling (Fig. 4, line 19-22). At the end, execution counts of the callsites are sorted to get the most frequently visited callsites (Fig. 4, line 24) and the final metric values are written to the output text files (Fig. 4, line 25-29). Memory leaks can severely hamper the performance of a MapReduce job, hence both the map and reduce algorithms are closely scrutinized for any leaks.

Once the reduce step is completed successfully, the MapReduce job ends. Fig. 2(a) and 2(b) present the two example CCT profiles generated by the custom dumper of JP2. Fig. 2(c) and 2(d) give the detail of key/value pairs generated by the Mapper for respective CCT profiles. Fig. 2(e) states the final dynamic coupling metric values which is the output of Reducer*.

```
1  Reduce(key,values containing array of couple details)
2  dynamicMessages←0, classCouples←0, methodCouple←0
3  for each val in values do
4    dynamicMessages←dynamicMessages+execCount of val
5    if class couple in val already exists in classList
6      do nothing
7    else
8    begin
9      add an item to classList containing calleeClass and callerClass
10     classCouples++
11   end
12   if method couple in val already exists in methodList
13     do nothing
14   else
15   begin
16     add an item to methodList containing calleeClass and callerClass
17     methodCouples++
18   end
19   if callsite couple in val already exists in callList
20     update the execution count of the callsite couple in callList
21   else
22     add an item to callList containing calleeClass and callerClass
23 end
24 sort callList as per callCount
25 output(key, dynamicMessages)
26 output(key, classCouples)
27 output(key, methodCouples)
28 for top item s in callList
29   output(key, s[0]+ s[1]+s[2]+s[3]+ s[4]+s.callCount)
```

Fig. 4. Pseudocode for Reduce function.

VII. Experimental Results and Analysis

Tables III shows the dynamic coupling metric values for 2 classes of Eclipse (large) and Jython (default) evaluated using the MapReduce approach. The same results are obtained using parallel computation on cloud as well as single node (local) computation. Once the Map function explorations for a specific class are completed in all CCT files, the required dynamic coupling metrics at every granularity level (dynamic messages, class, method, callsite) are rapidly aggregated out by the Reducer function.

Table II shows the percentage decrease in metric collection times with the use of parallel evaluation on 4-node Hadoop cluster for each input profile, as compared to single node evaluation. This results in an average increase in time performance of 73% at a very small cost. This shows that

*Refer http://pvsingh.com/s_sarvari/ for additional information.

TABLE II. The Performance (Time) comparison between MapReduce jobs for single node and 4-node parallel set up of Hadoop.

Input Systems	Number of CCT files generated	Number of CCT Nodes	Single Node Hadoop cluster (msec)	4-Node Hadoop cluster (msec)	Percentage decrease in time (%)
Eclipse	16	4316973	1319137	228414	82.7
Eclipse	112	10279202	2513796	1122289	55.4
Eclipse	55	16701372	5137499	1104423	78.5
Eclipse	307	25275546	6262757	2313663	63
Eclipse	186	34215774	8877195	2981805	66.4
Eclipse	214	40544790	10933955	2428596	77.8
Jython	24	8760832	2820024	282906	90

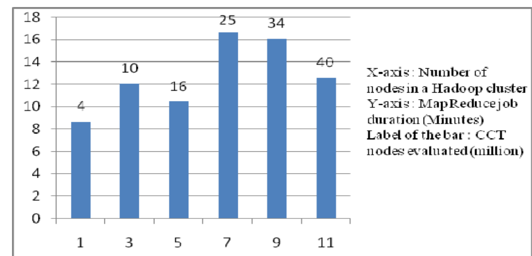

Fig. 5. Scale-Up for Eclipse CCT profiles with increasing number of nodes in a Hadoop cluster as well as increasing CCT node counts.

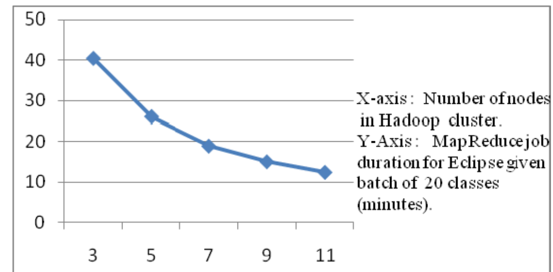

Fig. 6. Speed-Up for Eclipse profile with increasing number of nodes in a Hadoop cluster.

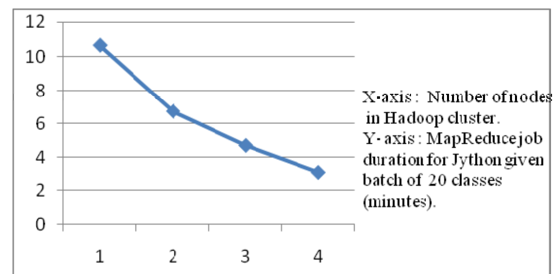

Fig. 7. Speed-Up for Jython profile with increasing number of nodes in a Hadoop cluster.

MapReduce provides a faster method for dynamic metric evaluation. As shown in Fig. 5, we get the expected scale-up i.e. approximately same performance is observed as the number of nodes in a Hadoop cluster is increased with an increase in CCT profile size. The long time durations observed at 25 million and 34 million CCT nodes is due to the large difference between the consecutive CCT nodes i.e. 9 million (instead of 6 million), but the increase in cluster size remains the same (2 in this case).

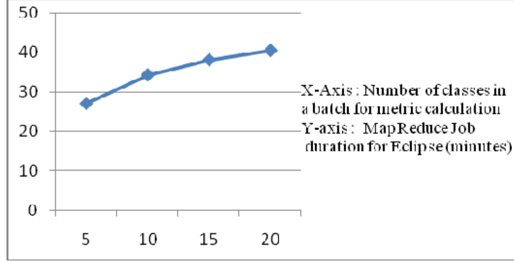

Fig. 8. Metric evaluation performance for Eclipse profile (containing 40 million nodes) with increase in batch size of classes.

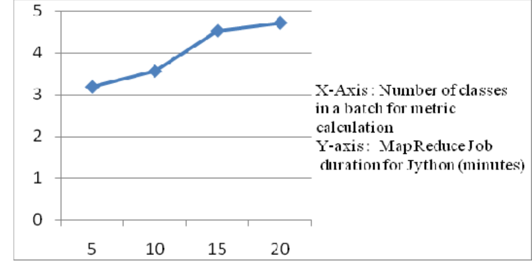

Fig. 9. Metric evaluation performance for Jython profile (containing 8 million nodes) with increase in batch size of classes.

TABLE III. DYNAMIC COUPLING METRICS VALUES EVALUATED USING MAPREDUCE APPROACH.

Class	TDM	DCC	DMC	Hot Callsite			
				Method	*CI*	*EC*	*%C*
Eclipse (large)							
Lorg/eclipse/jdt/internal/compiler/lookup/Scope	110470884	101	537	getBinding	57	1620714	1.4
Lorg/eclipse/jdt/internal/compiler/lookup/ReferenceBinding	61796685	62	394	isOrEnclosedByPrivateType	2	1975592	3.1
Jython (default)							
Lorg/python/core/PyType	9977583	210	651	Lookup	29	2381281	24
Lorg/antlr/runtime/BaseRecognizer	2305050	3	153	Match	7	220441	9.5

CI - Callsite Instruction, EC - execution count and %C - Percentage of coupling resulting from hot callsite out of the total TDM of class

Fig. 6-7 show the percentage decrease in time as the number of nodes in a Hadoop cluster is increased (given batch of 20 classes); for Eclipse (3 to 11 nodes per cluster) and Jython (1 to 4 nodes per cluster) profiles containing 40 million and 8.7 million CCT nodes respectively. Increasing the nodes in a Hadoop cluster from 3 to 11, results in approximately 70% speed up in case of Eclipse. It should be noted here that 3 and 11 are the worker nodes in a Hadoop cluster excluding the master node (with lesser configuration). In case of Jython, 71% speed-up is observed when the number of nodes in the Hadoop cluster is increased from 1 to 4. Hence, the evaluation time of metrics can be increased by pushing more number of nodes into a Hadoop cluster.

A point worth noting here is that the number of nodes in a Hadoop cluster for Jython profile is increased to a maximum of 4 nodes as opposed to a maximum of 11 nodes for Eclipse profile. Given a CCT node count of 8 million for the selected Jython execution instance and 24 resulting XML files, there is a negligible performance (time) gain and a substantial wastage of resources as the number of nodes per cluster are increased above 4.

Fig. 8-9 show the performance of MapReduce job on a 4-node cluster with an increase in class batch size (5 to 20) submitted for metric computation over Eclipse (large) and Jython (default) profiles respectively. An increase in metric collection time is observed as the number of classes per batch is increased. However, this increase is uneven which can be attributed to different number of method calls (pertaining to a particular class) that are unevenly distributed in CCT profile.

Getting back to our research questions defined in Section IV, *RQ1* is answered by developing a new approach to divide the CCT file with no loss of information enabling the parallel evaluation of metrics from CCT profiles. An average of 73% increase in performance is observed with our MapReduce approach for metric evaluation, which answers *RQ2*. Responding to *RQ3*, speed-up of about 70% is measured and scalability is assured. Further, RQ4 is answered by the fact that the time taken for metric evaluation increases with an increase in the number of classes per batch for reasons mentioned above.

VIII. CONCLUSIONS AND FUTURE WORK

Recent research has shown concerns over hindrance posed by the scalability issue in the widespread acceptance of dynamic metrics. These performance issues are faced while extracting dynamic metrics out of huge runtime information and hence act as a major hurdle in validation of such metrics. We set out to investigate whether the MapReduce paradigm, which is a popular approach for big data nowadays, can prove to be an efficient and scalable solution for the above mentioned problem. A custom dumper is implemented for JP2 profiler to produce multiple files instead of a single large file, so that the processing could be done in parallel. MapReduce functions are implemented to calculate various dynamic coupling metrics at multiple granularity levels. We conducted a number of experiments on AWS-EMR service and on local machine single node cluster for required comparisons. On an average, 73% increase in performance was observed with a 4-node Hadoop cluster as compared to the local set up. Our approach yielded 70% speed-up and scaled well with increasing cluster size. The dependency of the approach on number of classes per batch given for evaluation was also studied.

Several future directions can be derived from the outcome of this research. Firstly, our evaluation could be extended to real time large GUI applications. But to accomplish this, availability of highly scalable profilers is a prerequisite. JP2 is the state-of-the-art work but the size of CCT profile generated is limited by the memory available. There is also a possibility to work on a cloud solution for this limitation and make it

scalable by streaming the updates of a CCT to cloud without overloading the main memory. Another research direction is to validate various kinds of dynamic metrics on large real world applications and test their impact on quality attributes. Industrial studies could also be undertaken to analyze whether the developers find the use of dynamic metrics less cumbersome with the fast calculations supported by MapReduce. Complex dynamic metrics could be evaluated (like dynamic LCOM, DCC, DCM etc.), to get more benefits out of proposed parallel approach.

REFERENCES

[1] S.G. Tahir and A. Macdonell, "A Systematic Mapping Study on Dynamic Metrics and Software Quality," In *2012 28th IEEE International Conference on Software Maintenance*, Trento, 2012, pp. 326 - 335.

[2] E. Arisholm, L. C. Briand and A. Foyen, "Dynamic Coupling Measurement for Object-Oriented Software," *IEEE Transactions on Software Engineering*, vol. 30, pp. 491-506, 2004.

[3] Amazon Elastic Compute Cloud. "http://aws.amazon.com/ec2"

[4] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 467-493, 1994.

[5] A. Mitchell and J. F. Power, "Using object-level run-time metrics to study coupling between objects," In *ACM Symposium on Applied Computing*, 2005, pp. 1456-1462.

[6] S. M. Yacoub, H. H. Ammar and T. Robinson, "Dynamic metrics for object oriented designs," In *International Symposium on Software Metrics*, Boca Raton, FL, 1999, pp. 50 - 61.

[7] A. Mitchell and J. F. Power, "Run-time Coupling Metrics for the Analysis of Java Programs - preliminary results from the SPEC and Grande suites," Department of Computer Science, National University of Ireland, Technical Report 2003.

[8] Á. Mitchell and J. F. Power, "An empirical investigation into the dimensions of run-time coupling in Java programs," In *the 3rd international symposium on Principles and practice of programming in Java*, 2004, pp. 9-14.

[9] A. Mitchell and J. F. Power, "Toward a definition of run-time object-oriented metrics," In *Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, 2003.

[10] V. Gupta and J. K. Chhabra, "Dynamic cohesion measures for object-oriented software," *Journal of Systems Architecture*, vol. 57, no. 4, pp. 452-462, 2011.

[11] J. K. Chhabra and V. Gupta, "A survey of dynamic software metrics," *J. Comput. Sci. Technol.*, vol. 25, no. 5, pp. 1016-1029, 2010.

[12] H. Pirzadeh, A. Agarwal and A. Hamou-Lhadj, "An Approach for Detecting Execution Phases of a System for the Purpose of Program Comprehension," In *International Conference on Software Engineering Research, Management and Applications*, Montreal, QC, Canada, 2010, pp. 207-214.

[13] B. Cornelissen, A. Zaidman and A. v. Deursen, "A Controlled Experiment for Program Comprehension through Trace Visualization," *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 341-355, 2011.

[14] E. Stroulia and T. Systä, "Dynamic analysis for reverse engineering and program understanding," *ACM SIGAPP Applied Computing Review*, vol. 10, no. 1, pp. 8-17, 2002.

[15] A. Zaidman and S. Demeyer, "Automatic identification of key classes in a software system using webmining techniques," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 20, no. 6, pp. 387-417, 2008.

[16] W. Yuying, L. Qingshan, C. Ping and R. Chunde, "Dynamic Fan-in and Fan-out Metrics for Program Comprehension," In *International Workshop on Program Comprehension through Dynamic Analysis*, 2005.

[17] R. D. Venkatasubramanyam and G. R. Sowmya, "Why is dynamic analysis not used as extensively as static analysis: an industrial study," In *Proceedings of the 1st International Workshop on Software Engineering Research and Industrial Practices*, 2014, pp. 24-33.

[18] B. Cornelissen, A. Zaidman, A. v. Deursen,L.Moonen and R. Koschke, "A Systematic Survey of Program Comprehension through Dynamic Analysis," *IEEE Transactions on Software Engineering*, vol. 35, no. 5, pp. 684 - 702, 2009.

[19] Y. Hassoun, R. Johnson and S. Counsell, "Empirical validation of a dynamic coupling metric," School of Computer Science and Information Systems, Birkbeck College, University of London, UK, Technical Report 2004.

[20] Á. Mitchell and J. F. Power, "An approach to quantifying the runtime behaviour of Java GUI applications," In *Winter International Symposium on Information and Communication Technologies*, 2004, pp. 1-6.

[21] H. Sajnani and C. Lopes, "A parallel and efficient approach to large scale clone detection," In *7th International Workshop on Software Clones*, San Francisco, CA ,2013, pp. 46 - 52.

[22] L. D. Geronimo, F. Ferrucci, A. Murolo and F. Sarro, "A Parallel Genetic Algorithm Based on Hadoop MapReduce for the Automatic Generation of JUnit Test Suites," In *IEEE Fifth International Conference on Software Testing, Verification and Validation* , 2012, pp. 785 - 793.

[23] D. Bianculli, C. Ghezzi and S. Krstić, "Trace Checking of Metric Temporal Logic with Aggregating Modalities Using MapReduce," In *12th International Conference on Software Engineering and Formal Methods*, 2014, pp. 144-158.

[24] G. Ammons, T. Ball and J.R. Larus, , "Exploiting hardware performance counters with flow and context sensitive profiling," In *10th conference on Programming Language Design and Implementation*, 1997, pp. 85–96.

[25] A. Sarimbekov, A. Sewe, W. Binder, P. Moret and M. Mezini, "JP2: Call-site aware calling context profiling for the Java Virtual Machine," In *Workshop on Academic Software Development Tools and Techniques* , 2014, pp. 146–157.

[26] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107-113, 2008.

[27] Apache Hadoop Map Reduce. "http://hadoop.apache.org/mapreduce/"

[28] A. Sarimbekov, A. Sewe, W. Binder, P. Moret, M. Schoeberl and M. Mezini, "Portable and accurate collection of calling-context-sensitive bytecode metrics for the Java virtual machine," In *9th International Conference on Principles and Practice of Programming in Java*, 2011, pp. 11-20.

[29] A. Sewe, M. Mezini A. Sarimbekov and W. Binder, "Da capo con scala: design and analysis of a scala benchmark suite for the java virtual machine," In *ACM international conference on Object oriented programming systems languages and applications*, 2011, pp. 657-676.

[30] S. M. Blackburn, R. Garner, C. Ho_mann, A. M Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage and B. Wiedermann, "The DaCapo Benchmarks: Java Benchmarking Development and Analysis," In *21st annual ACM SIGPLAN Conf. on Object Oriented Programming Systems Languages and Applications*, 2006, pp. 169-190.

[31] Y. Hassoun, S. Counsell, and R. Johnson, "Dynamic coupling metric: proof of concept," *IEE Proceedings-Software*, vol. 152, no. 6, pp. 273-279, 2005.

[32] B. Cornelissen, L. Moonen and A. Zaidman, "An Assessment Methodology for Trace Reduction Techniques," In *IEEE International Conference on Software Maintenance*, Beijing, 2008, pp. 107-116.

[33] A. Tahir, S. G. Macdonell, & J. Buchan, "Understanding class-level testability through dynamic analysis," In 9th International Conference on Evaluation of Novel Approaches to Software Engineering. Lisbon, Portugal, 2014, pp. 1-10.

[34] Amazon Elastic MapReduce."https://aws.amazon.com/elasticmapreduce/"

[35] SPEC JVM98 Benchmark Suite. "https://www.spec.org/jvm98"