2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Accelerating Functional Verification Coverage Data Manipulation Using Map Reduce

Eman El Mandouh, Mentor Graphics Corporation, (eman_mandouh@mentor.com)

A. Gamal, A. Khaled, T. Ibrahim, Cairo University,(ahmed.mohamed94, ahmed.abdelraouf94, taha.soliman94@eng-st.cu.edu.eg)

Amr G. Wassal, Elsayed Hemayed , Cairo University, (wassal@eng.cu.edu.eg, hemayed@ieee.org)

***Abstract*— As the size and complexity of today's HW designs have been increased significantly, a huge amount of coverage information is generated during the design simulation and throughout the coverage closure cycle. Accordingly an efficient method for coverage data manipulation will be of great help to reduce the time and accuracy of the coverage data analysis process. This paper proposes the utilization of a distributed data processing technique, namely MapReduce, to accelerate the merging of coverage data sets from multiple simulation sessions. Inspired by the spirit of MapReduce, we formulate coverage data merging problem into tasks that are associated with keys and values and perform massively parallel map and reduce operations on distributed systems. Our coverage data manipulation framework is able to handle coverage data that is stored in coverage data bases or in multiple coverage reports. We demonstrate how the proposed approach can speed up the merging step by up to 1.8x factor with respect to other traditional merging techniques using real industrial designs.**

Keywords—Functional Verification ; Coverage Driven Verification ; Big-Data ; Map Reduce

I. INTRODUCTION

Functional Verification is the process of checking if the design under verification conforms to its specification of functionality, timing, testability and power dissipation [1]. Simulation-based verification continues to be the dominant verification methodology using random, constraint random or directed test generation [2]. The success of simulation-based verification depends heavily on the quality of the tests in use so in order to judge tests effectiveness, coverage is used as a metric to measure verification progress and completeness. Coverage metrics tell us what portion of the design has been activated during simulation and more importantly identify the portions of the design that were never activated during simulation, which allows us to adjust our input stimulus to improve verification [3]. The coverage achieved during verification is the most important parameter in determining the quality of verification results [4]. Figure 1 describes a verification cycle of HW designs. Traditionally the verification engineers start with the design functional specifications. This is followed by building up a complete verification plan that lists the verification goals and identifies the exit criteria for the verification effort.

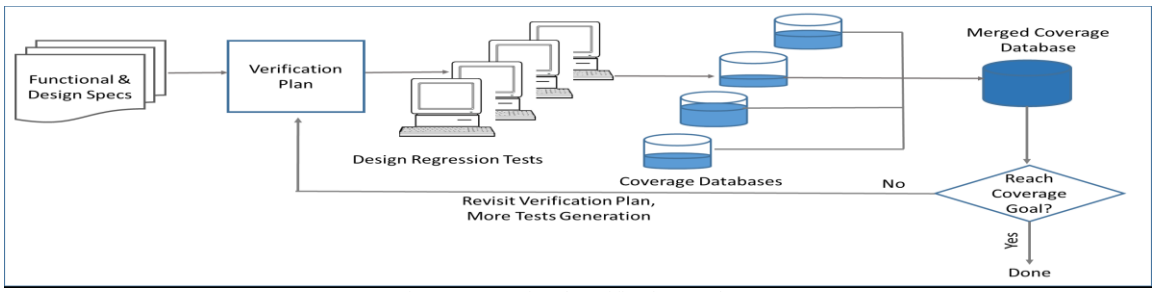

Figure 1: Functional Verification Cycle

During the verification process the same verification environment is executed repeatedly using different regression tests until the coverage goals are met. One test can differ from another by configuring or constraining the verification environment in a different way or merely by starting simulation with a different random seed. It is common to run multiple tests in parallel on compute server farms [5]. The coverage results from individual tests need to be merged together and annotated back onto the verification plan to form a cumulative record of progress. Finally the analysis of merged coverage data should be done to identify coverage holes. After analysis, features that still remain uncovered can be reached by adding further specific tests to verification plan. Merging of coverage data needs to be done efficiently in order to provide up-to-date information for verification management in a fast

accurate manner. Accordingly, coverage merging across tests is the fundamental operation with respect to coverage analysis phase. Recently merging of coverage data is getting harder with the increase of today's HW designs' size/complexity as well as the large number of cover targets that required to be captured in the final coverage model such as statement, toggle, branch, FSM, expression, assertion, cover-groups and cross-cover-bins. Additionally, coverage merging step is repeatability made during the verification life cycle until achieving the required coverage closure criteria. Apparently with this dramatic increase of the coverage data size any attempt to accelerate coverage data manipulation will be of great help to decrease the turnaround time of coverage closure and analysis cycle. This paper proposes the utilization of big-data analysis techniques and MapReduce distributed data processing framework to speed up coverage data merging step. We successfully investigated the applicability of MapReduce to accelerate coverage data merging. Our framework is very general in gaining massively-parallel computations for different cover items by extracting from each coverage data chunk the cover-items associated with their coverage results in "Key-Value" pairs of the map/reduce functions, while parallelization details are encapsulated in a MapReduce library [6]. Our framework reduces the coverage analysis time, by speeding up the time of consecutive coverage data merging steps during design regression run iterations. We have seen a substantial speedup from the experimental results.

The rest of the paper is organized as follows. Section II goes through related background about coverage in HW verification as well as an introduction to MapReduce methods. Additionally, it briefly reviews related work in coverage data manipulation techniques. Section III formulates the problem and explains our proposed framework. Section IV demonstrates the feasibility of our approach against a group of industrial designs. Finally, conclusion and future work directions are given in section V.

II. BACKGROUND AND RELATED WORK

A. *Coverage in Functional Verification*

Coverage is one of the most important metric to measure verification progress and completeness and in determining the quality of verification results. Broadly speaking, there are two types of coverage metrics used in the production of today's industrial designs: code coverage and functional coverage [3]. Code coverage is a measurement of the structures, such as statement, branch, conditional, toggle or expression, within the source code that have been activated during simulation. Functional coverage is a user-defined metric that measures how much of the design specifications have been exercised. System Verilog provides two language constructs for easy specification of the functional coverage models which are the Cover groups and the Cover Properties [7]. Covergroup construct encapsulates the specification of the coverage model, it consists of a set of coverage points as well as cross coverage between the coverage points. It records the number of occurrences of variables/expressions that are specified as its coverpoints as well as cross coverage between them when it contains cross coverage specification. Cover properties are the other major System Verilog constructs to specify functional coverage models, it has the same anatomy of the hardware design properties
Unified coverage databases, UCISs [8], have been developed to allow coverage metric interchange between different functional verification solutions such as simulation, formal, static checks or even emulators. The data model of UCIS allows the presentation of a wide range of coverage information models used in practice, this includes statements, FSMs, toggle, cover properties, cover-groups, and cross-cover bins to list a few. A standardized mapping, naming conventions and primary key management make the data objects universally recognizable. Besides, an API (Application-Programming Interface) was defined that standardizes the way data is written or queried from this data model. The API functions enable opening and closing a UCIS DB, navigating through scopes, extraction, and manipulation of data inside a UCIS DB. Coverage producers may use UCIS to generate coverage data to capture the coverage status during design execution. While coverage consumers can utilize UCIS APIs and the interchange format to perform analysis tasks such as combining coverage results from independent simulation runs or coverage report generation [9].

B. *MapReduce: An Overview*

Since being first introduced by Google in 2004 [10], the MapReduce programming paradigm has been widely applied to many domains such as data mining, database system, and high-performance computing [11]. MapReduce is a framework for processing parallelizable problems across large datasets using a large number of computing nodes. The processing can occur on unstructured-data that is stored in a file system or with structured data in databases. The main principle behind MapReduce program lies in (key-value) pairs which are generated and manipulated by user-defined mapper/reducer functions. MapReduce consists of three main steps:

- ***Map Step***: where each computing node applies the "map function" to its local data chunk, and writes the output to a temporary storage.
- ***Shuffle Step***: where the entire computing nodes distribute data based on the output keys such that all the data belonging to one key is located on same computing node.
- ***Reduce Step***: where computing nodes process each group of output data per key in parallel

MapReduce operations start with some large coverage data set. This data are divided into chunks that are distributed across multiple computing nodes. The mapper at each node performs parallel processing for its assigned data chunk and emits a (key-value) pairs. For example, every cover-item associated with its hierarchal scope and its RTL source information constitutes a unique key that is mapped to its achieved coverage scores captured in every coverage database. So same cov_item_key may appear multiple times in the same operating node as well as across multiple operating nodes with its associated coverage scores. During the shuffle phase, every (key, value) pair is assigned to a computing node such that all the occurrence of specific key is assigned to the same node so this phase ends up by having unique keys on every machine. During the Reduce phase an aggregation function operates on all coverage counter-values/Flags associated with a specific key followed by saving of aggregated results to disk. There are many state-of-the-art libraries to automatically schedule parallel map and reduce operations to handle the input data on a distributed system, our approach utilizes Apache Hadoop [6].

C. Related Work

Many EDA vendors for functional verification solutions deploy the merging capability as part of their coverage analysis engine [12] [13] [14]. The coverage data contained in the merged coverage DB are a union of the items in the coverage DBs being merged. The merge algorithm is a union merge. Cover-items of the same type which are associated with same RTL line numbers are merged together. For the cover-items that have no source information (FSM, Toggle), objects with the same name are always merged together [12]. Speeding up of functional verification coverage data merging has recently been improved by supporting parallel merge that allows merging subset of input coverage DBs which is then merged into higher level merged coverage DBs and so on until a final result is produced. It is a process that benefits from the ability to run merges in parallel, thereby improving the overall merge time.

III. PROPOSED FRAMEWORK

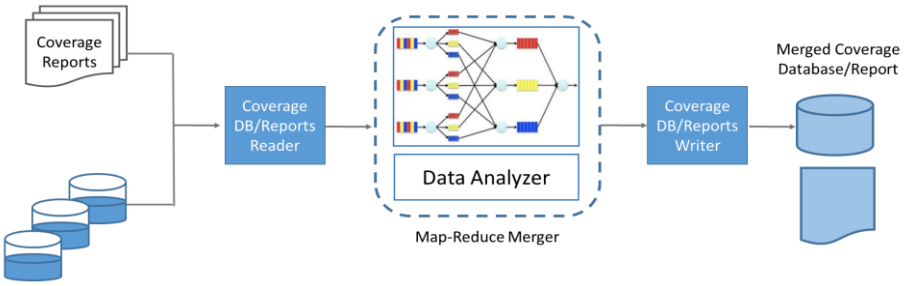

Figure 2. Proposed MapReduce Coverage Data Merger

Figure 2 demonstrates the proposed framework. The coverage Data Reader reads the input coverage DBs using read-streaming mode [8]. The coverage DBs are then converted into an intermediate text format which is appropriate for Hadoop Distributed File System (HDFS) operations [15] [16]. For MapReduce parallel operations, the design coverage-data must be presented in a fully independent format. Accordingly, each functional/code cover-item is presented independently from all other items to allow its further individual processing. All the required cover-item associated information must be attached to it, such as its hierarchical scope path or its RTL source information. Additional pre-processing of the input coverage files is performed to optimize their structure for better further MapReduce module operation as will be illustrated below. The coverage text files are then uploaded to Hadoop Distributed File System, HDFS.

The MapReduce starts operation by splitting these coverage text data into chunks, a chunk for each mapper task. Each mapper reads its chunk line-by-line, maps the bins with their path, name, line and values to generate a key-value pair for each design cover-item. Multiple mappers run in parallel and the output pairs are then sorted in order to group the pairs with same key together and to be sent to the reducers, i.e. shuffling the bins such that each group has cover-items with the same key. Each reducer handles a key and the group of values associated with it. It generates a single key-value pair, i.e. it reduces the same bins to one bin with final merged coverage value. Finally, in the last stage the output file from the MapReduce is written to the final coverage DB. This is done by

starting with a single coverage DB file with the most cover items as a base file and writing the correct merged values into that file. For any missing items or scopes it can be cloned directly from its source coverage DB file. The main building blocks of the proposed framework are described in details in the following sections:

A. Coverage Data Reader

The Coverage Data Reader operations start by traversing all the scopes in the coverage database in Read-Streaming Mode [8]. Recall that the coverage data can be retrieved from coverage DB by In-memory or Read-Streaming modes of operation. During In-memory mode, the entire DB image is loaded into the memory for fast read and write access. While Read-streaming mode allows limited-access to a narrow window of the coverage database as it is traversed. At any time the visibility is limited to the iterated coverage object and the ancestral objects of the current object only. Once the reading is moved forward, access to earlier parts of the DB is not doable. Some types of data are maintained globally, but the goal of this mode is to minimize the memory profile of the reading application by keeping only a small "window" of data in memory.

Read-Streaming DB access mode has been selected because it is aligned with the need to handle DBs chunks by Hadoop Distributed File System that splits the file into 128 MB blocks; hence in-memory mode cannot operate with chunks of the database instead of the complete file and it can't provide enough information for merging process. The merging process can't rely on statistical coverage data without detailed information about the cover items. The most appropriate mode for HDFS operation is the read-streaming mode because it doesn't have a big memory footprint and it can be used to stream the data to the mappers easily with some modifications on its source code to be compatible with HDFS. Additionally, all merging related data can be accessed in this mode.

The traversal of coverage DB is followed by the extraction of all required information for every single cover item, such as its scope and attributes. The extracted cover-items list associated with their attributes are directed to a text-format file that contains most of the information required for the merging operations, in our work we give this file a terminology of Unified Coverage Text Base UCTB. Additionally, this file contains extra information needed to write back the final merged data into the resulted coverage DB file. Cover-items' attribute data-fields are carefully chosen to build a composite key to guarantee collision-free merging for the coverage count values. Further optimizations are applied to the generated text coverage data to reduce the impact of redundant information that is stored per every cover-item to allow its independent processing during the map and reduce functions. The hierarchal path of the design cover points is replaced with numerical indices using hashing techniques. This is followed by further compression of the coverage input files in order to shrink their sizes for better utilization of the storage disk and faster processing time. LZO [17] is chosen for data compression at this step. Using LZO compression in Hadoop allows for reduced data size and shorter disk IO read times.

B. MapReduce Coverage Data Merger

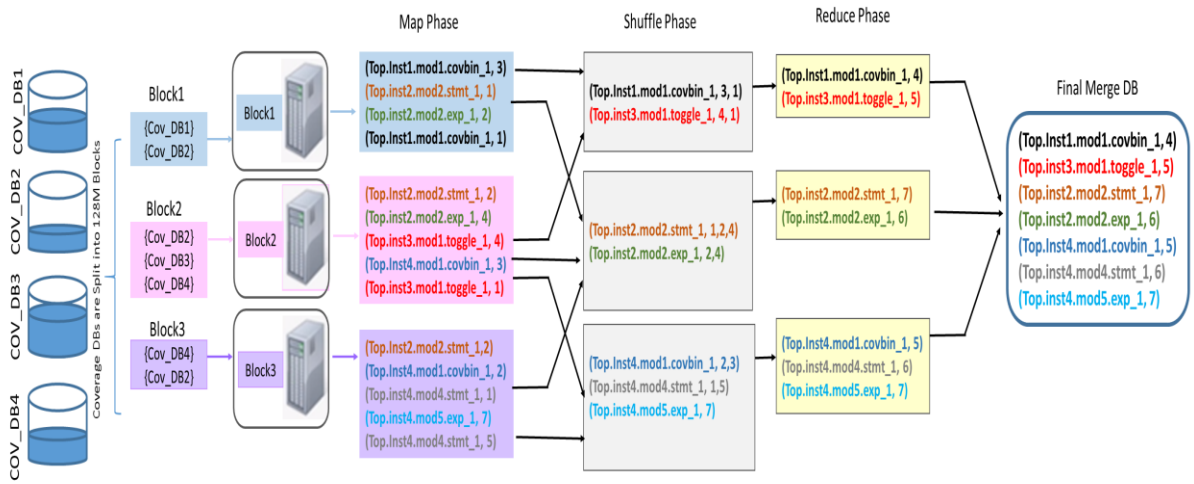

Figure 3. MapReduce Operations with Coverage DBs

In order to develop a MapReduce program, computations that can be issued to parallel map and reduce operations must be exploited from our problem. Merging cover-items from simulation coverage data is a parallel operation with nature. The MapReduce step consists of three main steps: Mapper Function, Shuffle Step and Reducer Functions as illustrated in Figure 3. MapReduce starts by dividing large data files into blocks of 128Mbs

each, every data block is assigned a computing node. Each map task typically operates on a single HDFS block. The map function processes each individual line in the input Unified Coverage Data Text Base (UCTB) trying to extract from every line (Key, Val) pair that represents for every cover-item a unique key associated with its coverage data. The cover-item unique key is decomposed from the concatenation of its scope hierarchal path in the coverage DB, type, name, source information (file, line number, and token number). The value associated with this key is built from the cover-item, counters (coverage results holder), flags and path of coverage database to which this cover-item originally belongs. The map step is followed by shuffle step that Hadoop does automatically to sort and consolidate intermediate data from all mappers and before reduce tasks start. During this step every (key, value) pair is assigned a computing node such that all the occurrence of specific key lands on the same node, For example, in Figure 3, all the occurrences of cover-item "*Top.inst2.mod2.exp_1*" has been assigned to node2. The final step is the Reduce step, where the aggregation operation is done to collect all the coverage data associated with the specific cover-item and to produce final coverage results. The actual merging is done in the Reduce function, our work uses a union merge algorithm [12] in which the merging of cover-item instances with numerical coverage count is done by summing them up. While the merging of cover-item instances with binary coverage data (Assertions and Cover Directives) is done by applying a logical-or operator. Another part of the merging algorithm is the flags merging, the exclusion flags are AND-ed together while the rest of the flags are OR-ed together. Algorithms 1 and 2 explain the map-reduce functions in our proposed framework.

Algorithm 1. Coverage-Data Merging Map-Function

Inputs: *Cover-Item Record. Cov_Rec*
Outputs: *(Key, Val) Pairs for every Cover-Item in Coverage DB*
//Input *Cov_Rec* consists of Cover Item :
// Hierarchal Scope Path, Name, Source Info, Token
//Coverage DB File Path, Coverage Count, Type and the Flags
1. *Key <- Concatenate { Hier Path, Type, Name, Source Info, Tag}*
2. *Val <- Concatenate {Count, Flags, DB File Path}*
3. *Emit (Key, Val)*

Algorithm 2. Coverage-Data Merging Reduce-Function

Inputs: *P (Key, Val) Cover-Items with Unique Key Value from Map Function*
Outputs: *(Key, Total_Cov_Val) Pairs for every Cover-Item in Coverage DB and the final aggregated merged Coverage Count/Flags*
1. *Cov_Count_Sum <- 0*
2. *Cov_Aggregated_Flags <- 0*
3. **foreach** *item ϵ P* **do**
4. *{Cov_Count, Cov_Flags, Cov_Path} <- Split (item.Cov_Val)*
5. *Cov_Type <- Split (item.Key)*
6. **If** *(Cov_Type in (Assertion, Cover_Directive))* **then**
7. *Cov_Count_Sum = Logical_OR(Cov_Count_Sum, Cov_Count)*
8. **else**
9. *Cov_Count_Sum += Cov_Count*
10. **endif**
11. *Cov_Aggregated_Flags <- MergFlags(Cov_Flags)*
12. **end**
13. *Total_Cov_Val <- Concatenate (Cov_Count_Sum, Cov_Aggregated_Flags , Cov_Path}*
14. *Emit (Key, Total_Cov_Val)*

The communication load is a non-negligible cost for a MapReduce program in particular during the collate operation. During the MapReduce operations, a huge amount of string data-pairs are transferred from the mappers to the reducers. This data is actually written to the disk and consumes a lot of time to be written by mappers and then read back by the reducers. In order to minimize the communication load, Hadoop "VIntWritable" data-type has been used to store directly the coverage <Key-Value> pairs from the mappers in the memory to be consumed by the reducers during the collate operation. "*VIntWritable*" data-type is essentially an "int" data type, but with variable length instead of the fixed 4 bytes of int decreasing the amount of data transferred from mappers to reducers achieved better processing performance.

C. Coverage Data Writer

Coverage Data Writer concerns with writing the merged coverage data from coverage text base UCTB Format to Coverage DB Format. Having final merged coverage results in DB format allows simulations, formal and other coverage consumers to access the merged coverage database. The final merged coverage DB is also used in our verification framework when our merged DB is compared with Traditional Simulation Total Merge results. The coverage DB writing starts by picking up a master DB, i.e. Largest DB from the input DBs that contains the largest number of cover-items. This class is the base class to which the merged values from UCTB is annotated. For any missing items or scopes, a clone is triggered for them from their source coverage DB file, as the path of the file is stored in the value emitted by the reducer operation, this is followed by the annotation of the merged coverage value from UCTB file.

IV. EXPERIMENTAL RESULTS

In this experiment, the MapReduce Coverage Merger has been exercised against a group of real industrial designs coverage DBs. Table 1 lists some information about the coverage DBs under study such as number of input coverage DBs for merging, Number of Branch, Expression, Statement, Toggle, Assertions Cover Items. The MapReduce merge results is compared against Totals Merge Algorithm Used in [12]. Totals Merge algorithm sums the coverage of the coverage scopes, design scopes, and test plan scopes. The counts are totaled (ORed together, in the case of vector bin counts) and by default the final merge is a union of objects from the input files. During totals merge the multiple test data records are retained from all merge input databases, however, this

merging technique loses the information about which test contributed what coverage into the merge is lost [12]. Our MapReduce experiments use Hadoop Framework, which is hosted on four computing nodes.

Table 1. Testcases Coverage DB Characteristics

Design Name	DB_Test1	DB_Test2	DB_Test3	DB_Test4
Coverage DB Size in (MBs)	2.5	17	20	44
Total Numbers of input Coverage DBs	1000	120	120	52
Total Size of Coverage DBs in (GBs)	2.5	2.04	2.4	2.3
No. of Assertions Bins	N/A	803	803	978
No. of Statement Bins	N/A	621,406	621,406	13,105
No. of Branches Bins	N/A	383,861	383,861	5,501
No. of Toggle Bins	N/A	0	3,097,204	13,646
No. of Focused Expression Coverage Condition Bins	N/A	33,831	33,831	1,815
No. of Focused Expression Coverage Expression Bins	N/A	858,673	858,673	10,917
No. of FSM States	N/A	5,013	5,013	0
No. of FSM Transitions	N/A	15,457	15,457	0
No. of CoverPoints Bins	524,288	0	0	754,154
Total No. of Cover Items	524,288	1,919,044	5,016,248	800,116

A. Experimental Merging Scenarios and Use Cases

Table 2: Different Experiments w.r.t No of Merged UCDBs, Size & Merging Time Comparisons

Testcase	Experiement_ID	No of Merged UCDBs	Size of Each UCDB (in MBs)	Total UCDBs Size (in MBs)	Merging Time Using Map-Reduce (in Secs)	Flattened UCTB Creation Time (in Secs)	Writing UCDB Time (In Secs)	Total Time in (Secs)	Ref Merging Time in (Secs) [12]
DB_Test1	Experiement_1	125	2.5	312.5	99	125	15	239	139
	Experiement_2	250	2.5	625	197	250	15	462	269
	Experiement_3	500	2.5	1250	367	500	15	882	548
	Experiement_4	1000	2.5	2500	757	1000	15	1772	1061
DB_Test2	Experiement_1	16	17	272	86	86	14	186	216
	Experiement_2	32	17	544	118	172	14	304	481
	Experiement_3	64	17	1088	236	344	14	594	1003
	Experiement_4	128	17	2176	477	688	14	1179	2019
DB_Test3	Experiement_1	16	20	320	121	107	23	251	294
	Experiement_2	32	20	640	223	214	23	460	579
	Experiement_3	64	20	1280	449	428	23	900	1176
	Experiement_4	128	20	2560	894	856	23	1773	2520
DB_Test4	Experiement_1	6	44	264	24	72	110	206	658
	Experiement_2	12	44	528	25	144	110	279	976
	Experiement_3	24	44	1056	34	288	110	432	1655
	Experiement_4	52	44	2288	52	624	110	786	3974

Table 2, sums up the conducted experiments across four different industrial designs DBs as explained in section IV. Each design has been exercised to merge different number of varying size coverage databases. Table 1 lists for each experiment, the number of used coverage DBs, the size of each DB and the total size of coverage data in use. Then for each experiment, we captured the time of reading coverage DB and converting it to flatten unified coverage text base UCTB that is appropriate for the MapReduce operations. We also capture the time of MapReduce merging and the time of writing the final merged coverage results in coverage DB. Our work is compared to the Totals coverage merge algorithm of industrial simulation [12] . The correctness and completeness of coverage merging results is checked by comparing the coverage data from MapReduce merger vs traditional resulted merged coverage DB from simulation. Table 2 demonstrates that on average our framework accelerates the coverage merging timing w.r.t traditional simulation merging technique by 1.8x factor, this includes our overhead of converting the coverage DB to flatten text base format and the writing back of final merged coverage results to coverage DB format.

B. Few Large sized Coverage DBs vs Many small sized Coverage DBs

In this section, we study per each test experiments the overhead time consumed in reading the coverage DBs, converting them to a flattened coverage text format and writing-back the merge results to final coverage DB w.r.t the time consumed in merging operation of Map-Reduce merger. Recall that data conversion to an intermediate presentation is a must to have the coverage data in a format appropriate to MapReduce independent proceeding for the cover-items across different DBs as explained in section III. Figure 4 demonstrates our experimental trials. In DB_Test1, we exercised a large number of small sized DBs. DB_Test2, and DB_Test3 have moderate number of medium sized DBs while DB_Test4 has a small number of large sized DBs. Figure 4 illustrates that the coverage DBs reading time overhead is high across the different test cases. The write-back time is of low impact because of the use of master-DB to annotate the merged results to it. Yet the writing time with DB_Test4 is relatively high because it has few huge-sized DBs. Figure 4 concludes that the MapReduce merging step is highly effective across all exercised testcases and especially with huge sized coverage data and hence emphasizes on the effectiveness of the proposed approach to accelerate big coverage data analysis/processing.

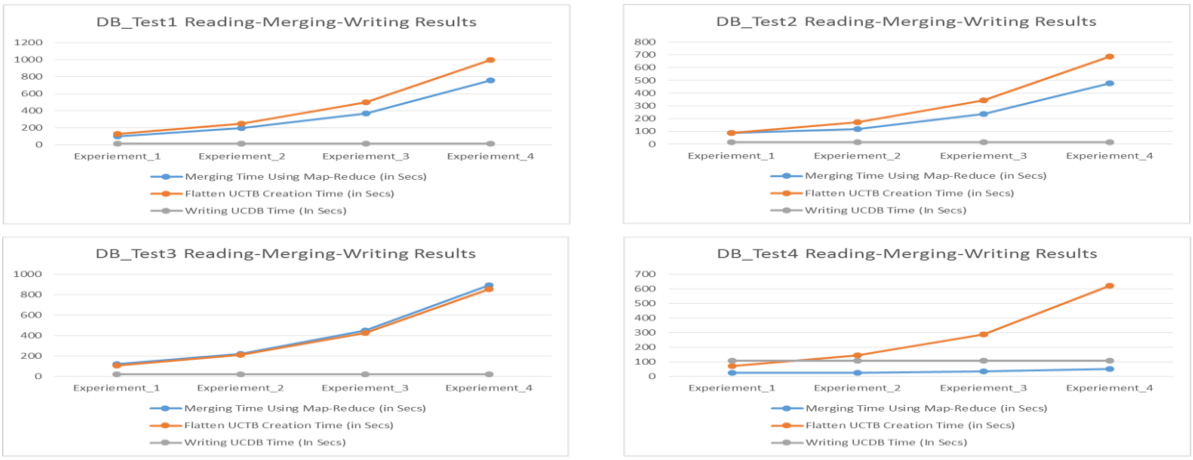

Figure 4. Coverage Data Conversion, Merging and Writing-back Times

C. *MapReduce Coverage Merger with Traditional Simulation Merging Approach*

In this section we compare our proposed framework with traditional simulation merging algorithm "Total Merge" [12]. Two comparisons have been conducted, the first one (Merging Time use Map-Reduce vs Ref Merging Time in Table 2) compares MapReduce merging time vs the simulation merging time. The second comparison considers the entire total time, including merging time plus the coverage DBs read/write-back time overhead with the simulation merge time (Total Time vs Ref Merging Time in Table 2). Figure 5 demonstrates that for all cases the MapReduce merging step beats the traditional simulation merging time. Additionally, for all tests the total_flow_run_time (reading, merging, and writing) is lower than simulation merge time, except for DB_Test1 where the total_flow_run_time is higher than simulation merge time. This is because in DB_Test1 the overhead of creating parallel mappers for many small-sized coverage DBs is larger than the speedup gain from the parallel operations of the map-reduce. This occurs due to the fact that Hadoop creates separate mapper for each DB block for its HDFS operations, so for many small-sized files the speed up gain will be less.

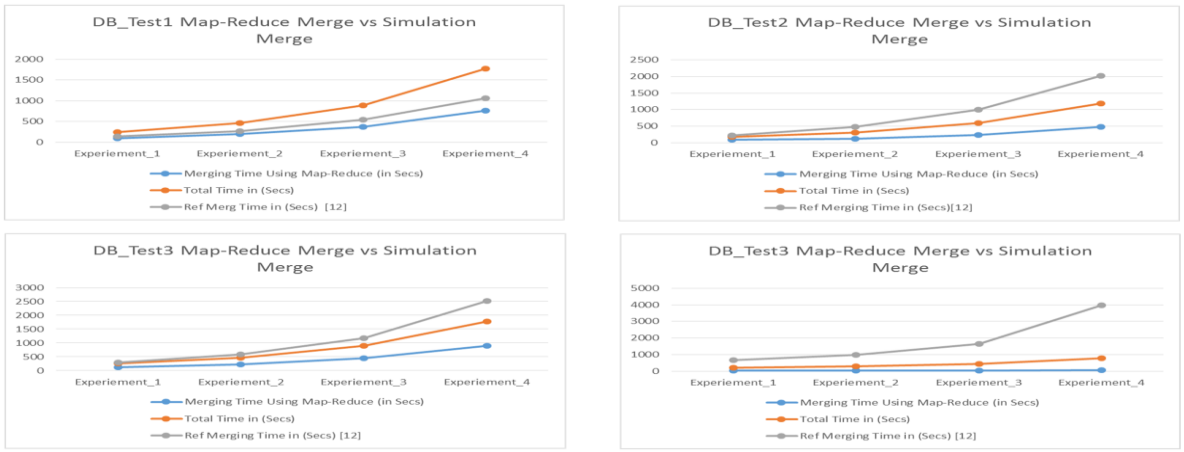

Figure 5. MapReduce Merge-Time w.r.t Simulation Merge-Time

Finally, Figure 6, summarizes the comparison of merging time across the different 4 test cases with all the experimental trials with and without including the reading/writing overhead. It is obvious that the proposed framework demonstrates an acceleration for the coverage merging time across the entire used test suite.

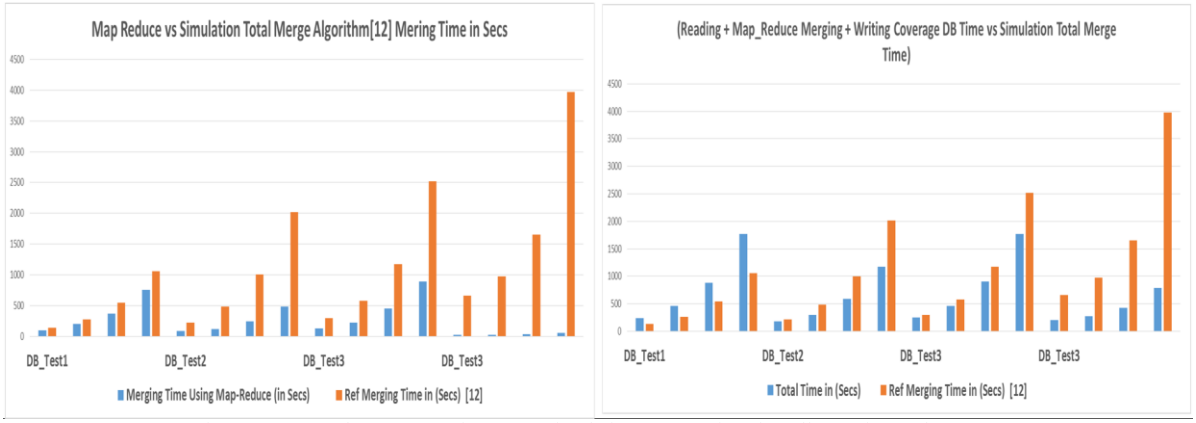

Figure 6. MapReduce Merge-Time w.r.t Simulation Merge-Time for All experimental Testcases

V. CONCLUSION AND FUTURE WORK

We propose how MapReduce can be used to accelerate coverage data merging step. Our approach demonstrates up to 1.8x speedup in the merging time of big coverage data. Yet the concept still valid to accelerate other coverage data processing steps like coverage report generation or mining coverage data for coverage trend construction. We believe that having a flattened unified coverage format, that inherits the storage optimizations of UCIS representation, but at the same time allows independent proceeding for the cover-items, should be the next step in coverage data presentation. Because this presentation will allow more explorations to the big-data analysis methods in the field of coverage data manipulation. Our future work aims to explore how the unified coverage DBs can be proceed directly as HDFS file. This change will allow direct processing of coverage databases and faster manipulation to coverage items. Another direction for our future work is to extend the system to include data analyzing and mining that can result useful conclusions about the coverage data and hence help further analysis of coverage information.

REFERENCES

[1] C. I. Castro , M. Strum and W. J. Chau, "Automatic Generation of a Parameter-Domain-Based Functional Input Coverage Model", In Latin American Test Workshop (LATW), IEEE 2010, pp. 1-6

[2] M. Chen and P. Mishra, "Functional Test Generation Using Efficient Property Clustering and Learning Techniques", In Computer-Aided Design of Integrated Circuits and Systems, 2010, pp. 396-404

[3] G. Allan, G. Chidolue, T. Ellis, H. Foster, M. Horn, P. James and M. Peryer, "Coverage Cookbook, Mentor Graphics", available on-line https://verificationacademy.com

[4] V. Athavale, S. Ma, S. Hertz and S. Vasudevan, "Code Coverage of Assertions Using RTL Source Code Analysis", In Design Automation Conference and Exhibition, DAC 2014, pp. 1-6

[5] Doulus,"Coverage-Driven Verification Methodology" , https://www.doulos.com/knowhow/sysverilog/uvm/easier_uvm_guidelines/coverage-driven

[6] Hadoop Apache, http://hadoop.apache.org/

[7] IEEE 1800-2012, System Verilog Unified Hardware Design, Specification and Verification Language, http://standards.ieee.org/getieee/1800/download/1800-2012.pdf

[8] Accellera Organization, Inc. Unified Coverage Interoperability Standard (UCIS). Accellera Organization, Inc. http://www.accellera.org/downloads/standards/ucis, June 2012

[9] C. Kuznik, M. F. S. Oliveira, B. G. Defo, W. Müller, "Systematic Application of UCIS to Improve the Automation on Verification Closure", In the Proceedings of Design Verification Conference (DVCon), 2013.

[10] J. Dean and S. Ghemawat "MapReduce: Simplified Data Processing on Large Clusters",OSDI'04: ,In the proceedings of Sixth Symposium on Operating System Design and Implementation(OSDI'04), 2004

[11] T. Huang, M. D. F. Wong, "Accelerated Path-Based Timing Analysis with MapReduce", In the Proceedings of The International Symposium on Physical Design (ISPD), 2015, pp. 103-110

[12] Questa Advanced Simulator, https://www.mentor.com/products/fv/questa/

[13] Cadence Incisive Enterprise Simulator, https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/incisive-enterprise-simulator.html

[14] Synopsis Functional Verification Solutions, https://www.synopsys.com/verification/simulation/vcs.html

[15] T. White, "Hadoop: The Definitive Guide", 2015, 4th Edition, ISBN: 978-1-491-90163-2

[16] D. B. Patel, M. Birla ; U. Nair, "Addressing big data problem using Hadoop and Map Reduce", In the proceedings of Nirma University International Conference on Engineering (NUiCONE),2012,pp. 1-5

[17] A guide to using LZO compression in Hadoop, 2012, https://nnc3.com/mags/LJ_1994-2014/LJ/220/11186.html