IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received September 10, 2020, accepted October 1, 2020, date of publication October 13, 2020, date of current version October 22, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3030562

Towards a Novel Framework for Automatic Big Data Detection

HAMEEZA AHMED AND MUHAMMAD ALI ISMAIL, (Member, IEEE)
Department of Computer and Information Systems Engineering, NED University of Engineering and Technology, Karachi 75270, Pakistan
Corresponding author: Hameeza Ahmed (hameeza@neduet.edu.pk)

This work was supported by the NED University of Engineering and Technology Research Grant, and National Centre of Big Data and Cloud Computing, NED University of Engineering and Technology, Karachi, Pakistan.

ABSTRACT Big data is a ''relative'' concept. It is the combination of *data*, *application*, and *platform* properties. Recently, big data specific technologies have emerged, including software frameworks, databases, hardware accelerators, storage technologies, etc. However, the automatic selection of these solutions for big data computations remains a non-trivial task. Presently, the big data tools are selected by analyzing the problem manually, or by using several performance prediction techniques. The manual identification is based on the data properties only, whereas the performance predictors only estimate basic execution metrics without linking them with big data (3Vs) thresholds. Hence, both ways of identification are mostly incorrect, which can lead to inefficient use of 3Vs optimizations, resulting into global inefficiency, reduced system performance, increasing power consumption, requiring greater effort on the part of the programming team, and misallocation of the hardware resources required for the task. In this regard, *a novel framework* has been proposed for automatic detection of 3Vs (Volume, Velocity, Variety) of big data, using machine learning. The detection is done through *static code features*, *data*, and *platform properties*, leading to relevant tool selection, and code generation, with minimal overheads, lesser programmer interventions, higher usability, and portability. Instead of handling each application with big data specialized solutions, or manually identifying the 3Vs, the framework can automatically detect and link the 3Vs to the relevant optimizations. Several standard applications have been tested using the proposed framework. In the case of volume, the average detection accuracy is up to 97.8% for ***seen*** and 95.9% for ***unseen*** applications. In the case of velocity, the average detection accuracy is up to 97.3% for ***seen*** and 92.6% for ***unseen*** applications. There is no margin of error in variety detection, as it has straightforward computations without any predictions. Furthermore, an airline recommendation system case study strengthens the effectiveness of the proposed approach.

INDEX TERMS Big data (3Vs), detection, LLVM, machine learning.

I. INTRODUCTION

Recently, there has been an explosion of data from daily sources such as credit cards, phones, social networks, astronomy, sensors, genomics, eCommerce, etc., [1], due to the Internet of Things (IoT), artificial intelligence, robotics, quantum computing, biotechnology, and fifth-generation wireless (5G) technologies. Such a huge amount of real-time heterogeneous data which is unable to get handled by conventional hardware or software solutions is termed as *big data* [2]–[7]. Big data is the combination of data, application, and platform properties. The application A operating with data D is considered as big data, if its processing gets difficult on platform P. The 3Vs (Volume, Velocity, Variety) appear to be the defining characteristics of big data for modeling data processing challenge [8], [9].

As per literature, APIs, programming models, domain-specific languages, and databases are invented to process big data applications [2], [6], [7], [10]–[12]. However, these solutions are adopted by manually testing the 3Vs data properties [8], or using performance predictors for estimating the computational requirements [13]–[18]. Although, the possibility of 3Vs can be roughly determined by observing use cases, still data characteristics are not enough to dictate

The associate editor coordinating the review of this manuscript and approving it for publication was Junaid Shuja.

the presence of big data, ignoring application and platform details [8]. Similarly, the automated techniques only predict the generic information like job's timing or resource requirements, without linking these metrics with big data (3Vs), thus selecting the big data tools due to higher execution time, or heavy resource consumption, not due to 3Vs presence.

This way, big data identification is mostly incorrect by both the methods, leading to global inefficiency, reducing system performance, increasing power consumption, requiring greater programming efforts, and misallocation of the hardware resources required for the task. For example, the adoption of a server is justified for processing a high *volume* data, which is unable to be executed on a small machine due to large memory consumption. Similarly, executing a job on a Spark cluster is justified, for faster processing of high *velocity* data having strict deadlines, otherwise, the same job can be run with fewer resources saving overall cost. Finally, adaptive schema techniques are justified for high *variety* data, whose format is unable to be handled by the routine applications. In this way, the prior cost-benefit analysis of big data (3Vs) is needed. Hence, an identification mechanism is necessary [19], which can trigger the solutions only after detecting big data problems at the right time. Up till now, to the best of our knowledge no such technique has been proposed which can automatically detect the big data workloads, before triggering the specialized solutions.

The detection phase is substantial for determining the presence of 3Vs. In case, if each incoming application is handled with specialized big data solutions, the resource wastage and overheads are expected to increase significantly, which ultimately leads to reduced overall performance. The detection stage can automatically trigger the specialized 3Vs optimizations, only if the application is big data, otherwise routine processing is continued. The 3Vs optimizations can be incorporated at various levels of big data stack such as hardware (*GPGPUs*, *FPGAs*) [20], [21], compiler (*garbage collection*, *parallelization*, *loop*, *type inferencing*, *data layout* optimizations) [22]–[24], third party libraries (*Hadoop*, *Spark*, *Flink*, *Storm*) [25]–[27], databases (*MongoDB*, *VoltDB*) [20], etc. Several benefits of automatic big data detection include optimal computational resource utilization, selection of appropriate tools, triggering of relevant codes either for General Purpose Processors (GPPs) or varying accelerator architectures, minimal overhead & user intervention.

This paper proposes a *novel framework* for detecting big data applications before the execution phase, which to the best of our knowledge is the first of its kind. The framework is comprised of *feature extractor*, *predictor*, and *detector*. It can identify the 3Vs of big data including Volume, Velocity, & Variety using *memory consumed ratio*, *execution time*, & *heterogeneity ratio* metrics respectively. Firstly, the application code is passed through the *feature extractor* for acquiring relevant code features using compiler Intermediate Representation (IR).[1] The IR emits source and target independent 3Vs features. The feature set includes *allocation size* for volume, *instructions count* for velocity, & *data format* for variety. These features can only be determined at run time, where the control flow information is available. Conventionally, the profiling technique has been used for determining code features, but it is discouraged for large scale applications due to substantial overhead [28]–[31]. Therefore, the proposed framework collects the feature set by first instrumenting each application, and then running it with smaller data sets. The computed features are then extrapolated for considered data sizes of test applications. The instrumentation approach is simpler and lightweight as compared to full system profiling because only the relevant information is emitted [32].

In the case of volume and velocity, the feature vector is passed to the *predictor* for estimating the application consumed memory and execution time using a suitable regression model. The predicted consumed memory and time are then passed to the *detector* for determining the existence of volume and velocity based on predefined thresholds. Similarly, the existence of variety is determined using heterogeneity ratio and predefined thresholds. With the large and complex datasets, one time profiling seems infeasible [30], [33] for computing the memory consumption and time. Similarly, static complexity analysis is not a viable and precise approach for real-world application codes due to nondeterministic execution paths [30], [34], [35]. In this manner, the proposed prediction based approach using *machine learning* and *instrumentation* appears to be reasonable for big data applications [33].

For testing the proposed detection framework, various application programs have been taken from standard benchmark suites [36]–[43]. The considered benchmark applications and datasets are diverse in nature, covering maximum possible use cases of big data. The feature extraction algorithms have been implemented as an analysis pass in Low Level Virtual Machine (LLVM) compiler for 3Vs. The predictor and detector have been implemented in statistical tool R [35]. For predicting memory consumption and time, SVM [44] regression has been used due to a lower error rate. The proposed framework serves as a milestone for identifying big data (3Vs) characteristics at an early stage with overall acceptable accuracy. Due to the generic 3Vs features, simplicity, and lightweightedness, the framework can be implemented in any third party big data processing tool (*script*) or commercial compiler (*flag*). Hence, there is no compulsion to handle each application with 3Vs optimizations, nor a need to manually identify the presence of 3Vs, because the *framework* can automatically detect and map the 3Vs to the respective optimizations without incurring substantial overheads. Following are the main contributions of this paper,

1) To the best of our knowledge, the first framework for automatically detecting characteristics of big data applications before job execution.
2) Feature engineering for 3Vs i.e selection of suitable feature set for estimating the memory consumption, time, and heterogeneity ratio.

[1] Code used internally by a compiler to represent source code.

3) Source and target independent 3Vs feature extraction via compiler IR which can work with any computational engine.
4) Cross platform compilation i.e target platform features are extracted by compiling code on any host platform without the need for target machine compilations like profiling approach.
5) Automatic selection of hardware & software solutions.

Rest of the paper is organized as follows: Section 2 discusses background & motivation. The framework is presented in section 3. Section 4 analyzes the results. The case study is presented in section 5. Section 6 discusses the related work followed by the final conclusion in section 7.

II. BACKGROUND & MOTIVATION

Recently, *big data* workloads have emerged, involving huge amount of real-time heterogeneous data, which is unable to get handled by conventional hardware or software solutions [2]–[7]. The 3Vs (Volume, Velocity, Variety) appear to be the defining characteristics of big data and are enough to model data processing challenge [8], [9]. Big data is a ''relative'' concept, hence the thresholds are determined via *data*, *application*, and *platform* characteristics [2], [5], [9], [45], [46]. Volume is indicated by higher memory or I/O consumed ratios, velocity emerges when the data generation rate gets greater than the data processing rate,[2] and due to dissimilarity between application and external data formats variety is true.

In the past decade, there has been an emergence of big data specific technologies, including software frameworks like *Hadoop*, *Spark*, NoSQL databases like *MongoDB*, *Cassandra*, hardware accelerators like *GPGPUs*, *FPGAs*, storage technologies like *NVRAMs*, *Cloud*, etc., [7], [20], [20], [47]. However, automatic selection of these solutions for big data computations remains a non-trivial task. Presently, the selection task is done by skilled personnel or performance predictors.

The personnel manually analyze the problem and then select the appropriate big data tool. This analysis is mostly incorrect as it is based on the data properties only, ignoring application and platform features, for example, if the scenario involves GB size or heterogeneous formats, big data technology is used. However, the larger data size does not necessarily involve heavy computations that are unable to get done via conventional resources. Nor the heterogeneous formats reflect the handling deficiency of application. This way, adopting big data tools by only considering the data size or format is a costly decision that can lead to excessive power & energy consumption and wasting of the team's programming, administration, & management efforts. Hence, the exact need for big data technology can only be known by executing the given data with an application on a specific platform.

In contrast to the manual approach, several automatic techniques have been developed [13]–[18]. However, these techniques select big data tools by only predicting generic metrics like job's timing and resource consumption, without relating these metrics with big data (3Vs) thresholds. This way, before using big data tools the 3Vs existence status is still *unknown*, resulting in inefficient resource exploitation. The big data tools are adopted because of longer execution time or higher resource consumption, not because the conventional resources are incapable to handle such jobs.[3]

The adoption of resource intensive big data tools for such workloads which can be handled with conventional resources is a costly choice resulting in heavy resource wastage. For example, using a high-end multi-core server is not wiser for a job that can be processed with a laptop. The system will consume high power & energy even with only one or two cores are being utilized. Similarly, hiring of Hadoop experts to perform data analytics is not worthy, if the same analytics can be performed by a simple Python script. This way, a mechanism is necessary which can automatically detect and offload big data workloads to appropriate tools as shown in Figure 1.

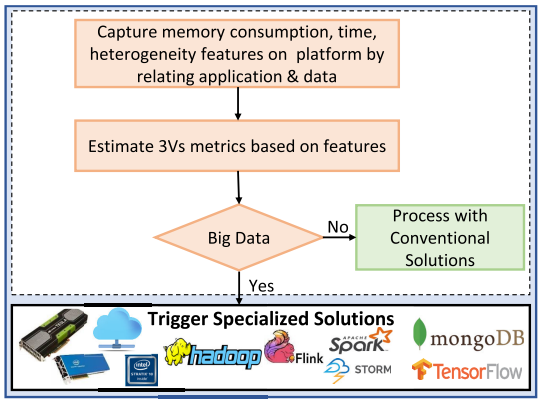

FIGURE 1. Big data detection & processing.

A. BIG DATA PROCESSING

In literature, numerous techniques are available at different layers of computation stack that can address big data processing challenge [2], [6], [7], [10]–[12]. It includes *domain-specific languages*, *software programming models*, *third party libraries*, *hardware accelerators*, *storage devices*, and so on. To better understand the role played by the existing techniques in solving big data problem, we have designed the big data computation stack in Figure 2. It consists of the common layers that are part of a computation stack, through which every high-level application is passed to get executed on hardware. In a similar manner, a big data application possessing Volume, Velocity, Variety is passed through these computation layers of the stack [25]. We have surveyed and mapped each big data solution to the relevant layer of the stack in Figure 2. Each level exhibits numerous opportunities

[2]It means processing time gets greater than generation time.

[3]The fundamental definition of big data workloads says these are unable to get handled by conventional resources.

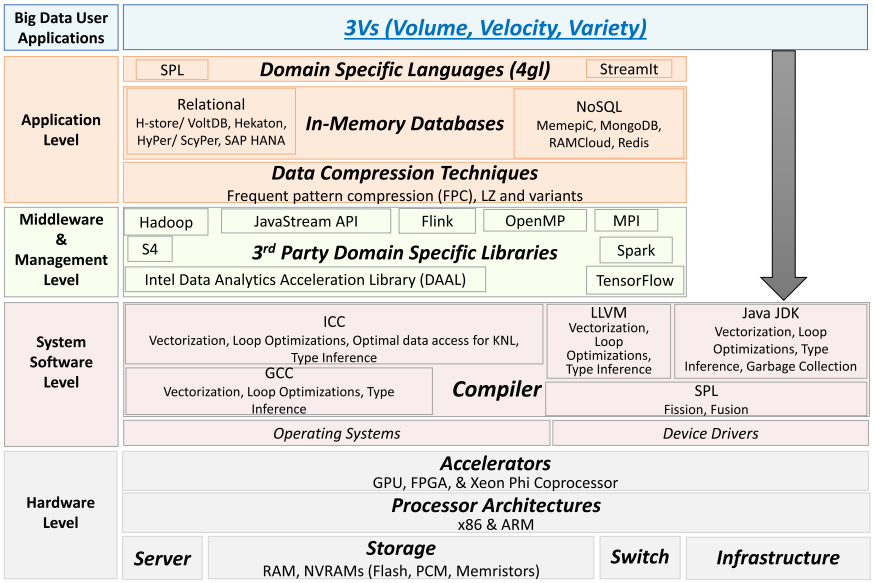

FIGURE 2. Big data computation stack.

for attaining the 3Vs performance objectives as discussed below.

1) APPLICATION LEVEL

The solutions are easier to design and manage from a programmer viewpoint at this level. It includes domain-specific fourth-generation languages, in-memory databases, and data compression techniques [20], [48]. The development in this phase is rapid and regular.

2) MIDDLEWARE AND MANAGEMENT LEVEL

It is less abstracted in comparison to the application level, the design and management tasks are more difficult for end programmers. The growth is significant by means of programming models and third party libraries [25]–[27].

3) SYSTEM SOFTWARE LEVEL

It is the middle stage that utilizes the resources to the maximum capability by mapping the above software to the hardware [24], [48], [49].

4) HARDWARE LEVEL

Considerable growth is observed at this level, in terms of dedicated accelerator architectures and storage devices [20], [21], resulting in enhanced overall performance.

B. BIG DATA DETECTION

The need for *big data detection* before *selecting specialized solutions* (section 2.1), can be understood in the light of a 4 node virtual Spark [50] cluster that has been configured on Hadoop Yarn [51]. A set of Spark jobs from *big-databench* [52] benchmark suite has been executed in *local* and *cluster* modes. As per Figure 3, the *volume* is indicated

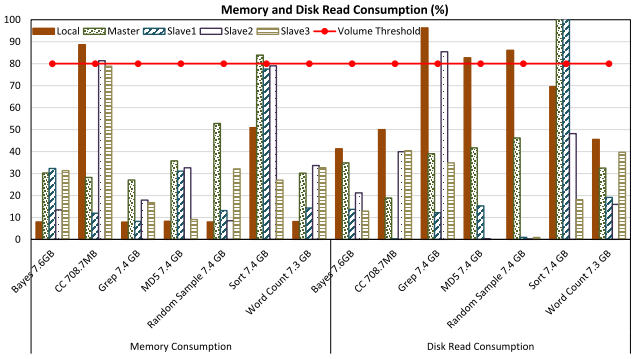

FIGURE 3. Volume test for spark jobs.

by memory or disk read consumption of above threshold (80%). For *bayes*, *sort*, and *word count*, memory & disk read consumptions are less than 80% in local mode, indicating the absence of volume problem. This way, volume specialized solution is not needed for these jobs. However, if these jobs are executed using a server or a cluster with large memory, the memory consumption fraction becomes very low, but the power and energy consumptions are likely to be greatly increased while operating these high machines, which is not worthy because the memory resource is under utilized.

Similarly, as per Figure 4, *velocity* is indicated, if execution time is greater than the threshold (deadline). In this manner, cluster adoption is justified only for *connected component (CC)* execution, due to shorter deadline. Hence, it is not justified to run such jobs on the cluster, which do not represent velocity problem, as these are increasing the overheads in terms of expansive nodes, unnecessary resource wastage by cluster daemons, programmers time, and cluster management. For instance, *grep* execution time is already below

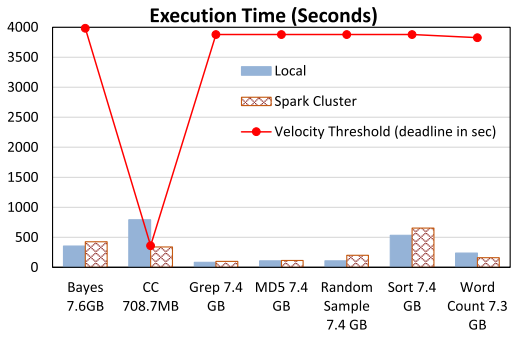

FIGURE 4. Velocity test for spark jobs.

the deadline, but when it is executed in a cluster, the communication cost between nodes is unnecessarily increased which leads to increase execution time. Also, the cluster resources are idle most of the time but the power & energy are still consumed as well as personnel is regularly involved in programming, execution, and management of the given job on cluster.

Despite the processing of GB size data by the jobs, 3Vs requirements vary greatly. For instance, by observing the same GB size, both *grep* and *sort* are categorized as high *volume*. However, this manual detection gets incorrect as both the jobs have different consumptions. Similarly, *CC* size is much lesser than *grep*, but *velocity* is present in *CC*, due to higher execution time. In this manner, *manual* testing is certainly not enough for offloading jobs to these rich resources, because the *application* and *platform* knowledge are also needed along with *data* for making the right decisions.

Moreover, the existing performance predictors [13]–[18] are not enough, as these can only determine the job's timing and resource requirements, without linking them to 3Vs thresholds. For example, these techniques will prescribe to run majority applications on the cluster to reduce execution time. However, with the execution of such jobs on the Spark cluster which does not represent *velocity*,[4] the cluster resources are inefficiently exploited, resulting in huge penalties in terms of power & energy consumption and wastage of personnel efforts. In this regard, our approach is beneficial as it makes prescriptions based on 3Vs metrics, hence cluster is prescribed for *CC* only, after being detected as *velocity* problem. Finally, the selection of adaptive schema techniques is justified if *variety* is observed, that is the application is not capable to handle incoming data. Otherwise, the adoption can lead to unnecessary hardware, software, and personnel resources spent in managing such complex techniques.

This way, an *automatic 3Vs detection* mechanism is inevitable for appropriate processing of jobs, instead of executing all the larger size, heterogeneous datasets, or slower jobs with these expansive big data tools. In the considered scenario, the *prior detection of big data* can save the slave node resources, including RAM 3*(6 GB), disk 3*(20 GB), and CPU 3*(1 core @ 3.60 GHz), as well as for single node up to 50% RAM & 31% disk reads are saved. Further, the jobs can be sequentially coded by ordinary programmers using conventional tools. Hence, the computation and human resources are reduced up to 75% (3/4) by utilizing only the single machine of a 4 node cluster.

[4]Such jobs can easily meet processing deadlines in local mode.

III. 3Vs DETECTION FRAMEWORK

The 3Vs (Volume, Velocity, Variety) detection framework is presented in Figure 5. Only Intermediate Representation (IR) code and test data properties are input to the framework comprising of *feature extractor*, *predictor*, and *detector*. The 3Vs features are extracted via IR, which ensures *genericness* due to source and target independence. The feature vector construction is followed by regression based prediction of volume and velocity metrics. The presence of Vs is detected through computed metrics.

A. VOLUME DETECTION

The presence of volume is indicated by higher memory or I/O consumed ratios for a given size and platform [22] as represented in Equation 1. Where θ_{v_1} is volume threshold, $P(\alpha)$ is a process memory consumed ratio (memory intensive), and $P(\beta)$ is a process disk I/O consumed ratio (I/O intensive). Memory consumed ratio is the proportion of maximum physical memory consumed by a process with respect to the total available RAM. It can be computed through the Resident Set Size (RSS)[5] of a process. Similarly, I/O disk consumed ratio is the proportion of maximum disk reads per second done by a process with respect to maximum disk reads per second allowed for a hard disk.[6]

$$Volume = true, \text{ if } P(\alpha) > \theta_{v_1} \vee P(\beta) > \theta_{v_1} \tag{1}$$

In this paper, memory consumption is considered for categorizing volume based applications. The detector can automatically determine the memory consumed ratio, by knowing the RSS and total system RAM. The RAM size can be measured one time per platform, but the estimation of RSS at compile time is not straightforward. For accurate measurement of RSS, an application is required to be profiled each time before passing through the detection framework, which will affect the major objective of requirement identification before the actual execution [28]–[31]. On several occasions, one-time profiling is justified for optimizing the same application repeatedly [28]. However, executing an unoptimized application with a large amount of data incurs substantial overhead by consuming time and resources [30]. In this way, full profiling is not reasonable, even for the purpose of accurate detection. Hence, our framework relies on machine learning for predicting per platform RSS [28], [31], [33], which may be less accurate at the benefit of minimal overhead. The training set builds the regression model offline,

[5]Resident Set Size (RSS) is defined as the portion of physical memory occupied by a process.

[6]Both RSS and kB_rd/s are measured per process using linux *pidstat -urd -h -G* command.

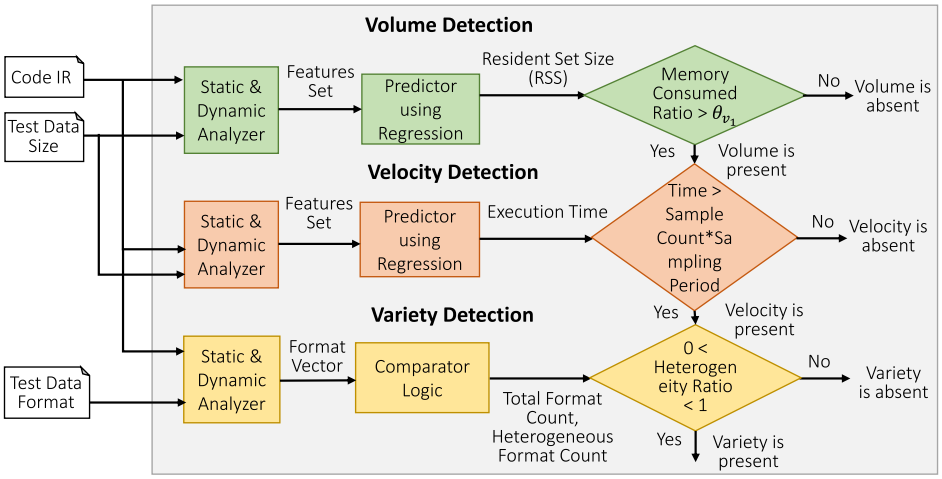

FIGURE 5. Big data detection framework.

which is used by the incoming applications for online RSS predictions [28], [31], [33]. The RSS is not the same for different applications involving the same data size. In this regard, along with data size, stack, heap, or other types of memory allocation operations are required for predicting the RSS.

Volume is detected via *feature extractor*, *predictor*, and *detector* modules. As the incoming application is passed through the compiler analysis pass, the RSS specific features are extracted by the feature extractor, which are then fed to the predictor for estimating the RSS using the trained regression model. Finally, the detector decides the existence of volume using the predicted RSS.

1) FEATURE EXTRACTOR

Memory consumption of an application is due to code binary, linked shared libraries, global data, stack, heap, or other allocations. The decision of allocating pages in memory is taken by the operating system, hence the absolute value of RSS can only be determined at run time. However, maximum RSS can be roughly estimated at compile time by considering the memory allocation features. The heap is allocated by *malloc*, *calloc*, *realloc*, and *new* operator, and stack allocation is done through the *function calls*. Similarly, the *mmap* function is used for allocation. The data and bss segment are allocated via *global variables*. For a single application, the variation in data size can be observed by dynamic allocation functions (*malloc*, *new*, *mmap*, etc), which allocate memory with respect to external data [53].

There exist *static* and *dynamic* memory allocation categories. Static allocation refers to cases where the allocation amount is determined at compile time. For example, at compile time it is known that the allocated memory is equivalent to data size. Whereas, in the case of dynamic, the amount of allocation is not fixed and can only be determined by reading the data file at run time [53]. In this regard, maximum static and dynamic allocation sizes are selected as features for estimating memory consumption. Static refers to the maximum fixed allocated bytes on the stack, heap, data, bss, or other region at one time that can be computed by static analysis, whereas dynamic refers to the maximum variable allocated bytes on the heap, or other region at one time [53]. Dynamic deals with external file contents which are only determined at run time.

The procedure for extracting the RSS feature set is depicted in Figure 6. The high-level code is converted into Intermediate Representation (IR) through compiler front end. Then, the IR is statically analyzed to record the functions, basic blocks, loops, and memory allocation instructions. The IR is instrumented with few necessary instructions for determining the reachable basic blocks and functions which can only be obtained at run time, hence instrumentation cannot be avoided. However, the instrumentation overhead is likely to be less, as few instructions are run with three smaller size datasets for recording the basic block and function names. Obviously, the execution flow remains the same for other data sizes as well. The three output sets are compared for categorizing the basic blocks. With the same frequency, the basic blocks are static, otherwise, they are dynamic. The frequency difference indicates that certain blocks are only executed for a larger data size, due to varying iteration count. Similarly, the basic blocks having loops with unknown trip count are categorized as dynamic. The executed basic block information is then mapped to static IR information obtained previously. In this manner, only the allocation instructions of executed basic blocks are considered.

The static count is computed via size allocated by stack or global allocation instructions. Besides, malloc, calloc, realloc, new or mmap functions can also contribute to static count, if their allocation size is constant. If the allocation size is available at run time, then the dynamic count is computed in terms of n, which is unknown at compile time, but for rough estimation, it is substituted as data size. In this manner, memory consumption cannot be estimated directly for

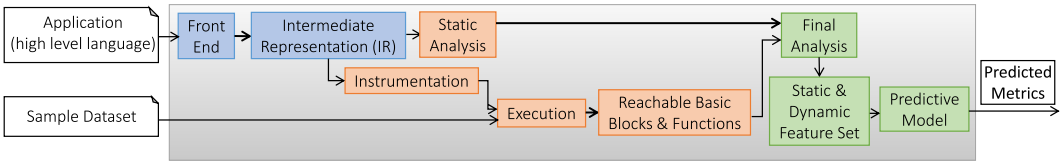

FIGURE 6. 3Vs metrics estimation.

TABLE 1. Experimental configuration.

Environment	Total Memory (RAM)=32862376 kB, Total Swap=0, Buff/cache=26566756 kB, Model Name=Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz, x86_64 Architecture, Page Size=4kB, Hard disk=1TB SATA Harddisk Max Read rate=219MB/sec, Operating System=Linux Ubuntu 16.04.4 LTS, L3 Cache=8192 KiB, 16-way Set-associative, L2 Cache=256 KiB, 8-way Set-associative, L1I, D cache=32 KiB, 32 KiB 8-way Set-associative, LLVM 4.0, R version 3.5.2
Volume Static Train+Test Set	Grep (1.1 GB, 5.4 GB, 10.7 GB, 16.1 GB, 21.5 GB, 26.8 GB, 32.2 GB, 37.6 GB, 42.9 GB, 48.3 GB, 53.7 GB), Histogram (104.5 MB, 418.1 MB, 1.4 GB, 2.1 GB), Linear Regression (108.5 MB, 542.5 MB, 1.1 GB, 2.2 GB, 4.3 GB, 8.7 GB, 13 GB, 17.4 GB, 21.7 GB, 26 GB, 30.4 GB), Nearest Neighbor (411 MB, 822.1 MB, 1.6 GB, 3.3 GB, 6.6 GB, 9.9 GB, 13.2 GB, 16.6 GB, 21.2 GB, 26.3 GB, 30.8 GB),Reverse Index (1.01 GB), String Match (108.5 MB, 542.5 MB, 1.1 GB, 2.2 GB, 4.3 GB, 8.7 GB, 13 GB, 17.4 GB, 21.7 GB, 26 GB, 30.4 GB), Word Count (52.5 MB, 105.3 MB, 1.1 GB, 2.1 GB), Bzip2d (10 MB)
Volume Dynamic Train+Test Set	Bfs (292 MB, 592.5 MB, 745.5 MB, 1.2 GB, 2.6 GB, 5.2 GB, 7.3 GB, 8.2 GB, 10.7 GB, 13.5 GB, 14.7 GB), Connected Component (6 MB, 18.1 MB, 60.7 MB, 185.5 MB, 607.8 MB, 1.8 GB, 6.1 GB), Page Rank (6 MB, 18.1 MB, 60.7 MB, 185.5 MB, 607.8 MB, 1.8 GB, 6.1 GB), Kmeans (245.9 MB, 739.9 MB, 2.5 GB, 4.9 GB, 7.4 GB, 9.9 GB, 24.8 GB),Blackscholes (631.7 MB), Dfs (6 MB, 18.1 MB, 60.7 MB, 185.5 MB, 607.8 MB, 1.8 GB, 6.1 GB), Genann (9.1 MB, 36.4 MB, 145.6 MB, 582.4 MB, 873.6 MB, 1.2 GB, 3.6 GB, 6 GB, 10.7 GB, 14.3 GB, 17.9 GB), Liblinear (760.1 MB, 1.5 GB, 3 GB, 6.1 GB, 9.1 GB, 12.2 GB, 15.2 GB, 18.2 GB, 21.3 GB, 24.3 GB, 27.4 GB), LDA (4.9 GB), PCA (5 GB, 10 GB, 15 GB, 20 GB, 25 GB, 29.9 GB, 34.9 GB, 39.9 GB, 44.9 GB, 49.9 GB, 54.9 GB)
Velocity Train +Test Set	Bfs (292 MB, 592.5 MB, 745.5 MB, 1.2 GB, 2.6 GB, 5.2 GB, 7.3 GB, 8.2 GB, 10.7 GB, 13.5 GB, 14.7 GB), Connected Component (6 MB, 18.1 MB, 60.7 MB, 185.5 MB, 607.8 MB, 1.8 GB), Grep (1.1 GB, 5.4 GB, 10.7 GB, 16.1 GB, 21.5 GB, 26.8 GB, 32.2 GB, 37.6 GB, 42.9 GB, 48.3 GB, 53.7 GB), Histogram (104.5 MB, 418.1 MB, 1.4 GB), Linear Regression (108.5 MB, 542.5 MB, 1.1 GB, 2.2 GB, 4.3 GB, 8.7 GB, 13 GB, 17.4 GB, 21.7 GB, 26 GB, 30.4 GB), Nearest Neighbor (411 MB, 822.1 MB, 1.6 GB, 3.3 GB, 6.6 GB, 9.9 GB, 13.2 GB, 16.6 GB, 21.2 GB, 26.3 GB, 30.8 GB),Kmeans (245.9 MB, 739.9 MB, 2.5 GB, 4.9 GB, 7.4 GB, 9.9 GB), Page Rank (6 MB, 18.1 MB, 60.7 MB, 185.5 MB, 607.8 MB, 1.8 GB), Reverse Index (1.01 GB), String Match (108.5 MB, 542.5 MB, 1.1 GB, 2.2 GB, 4.3 GB, 8.7 GB, 13 GB, 17.4 GB, 21.7 GB, 26 GB, 30.4 GB), Word Count (52.5 MB, 105.3 MB, 1.1 GB, 2.1 GB), Blackscholes (631.7 MB), Dfs (6 MB, 18.1 MB, 60.7 MB, 185.5 MB, 607.8 MB, 1.8 GB), Genann (9.1 MB, 36.4 MB, 145.6 MB, 582.4 MB, 873.6 MB, 1.2 GB, 3.6 GB, 6 GB, 10.7 GB, 14.3 GB), Liblinear (760.1 MB, 1.5 GB, 3 GB, 6.1 GB, 9.1 GB, 12.2 GB, 15.2 GB, 18.2 GB, 21.3 GB, 24.3 GB, 27.4 GB), LDA (4.9 GB), PCA (5 GB, 10 GB, 15 GB, 20 GB, 25 GB, 29.9 GB, 34.9 GB, 39.9 GB, 44.9 GB, 49.9 GB, 54.9 GB)
Volume Feature Set	Maximum_Static_Allocation_in_Bytes (x_1), Maximum_Dynamic_Allocation_in_Bytes (x_2)
Velocity Feature Set	Static_Calc_Count (x_1), Static_Call_Count (x_2), Static_Compare_Count (x_3), Static_Condition_Count (x_4), Static_Convert_Count (x_5), Static_Extract_Count (x_6), Static_Fcalc_Count (x_7), Static_Fconvert_Count (x_8), Static_Load_Count (x_9), Static_Ret_Count (x_{10}), Static_Store_Count (x_{11}), Dynamic_Calc_Count (x_{12}), Dynamic_Call_Count (x_{13}), Dynamic_Compare_Count (x_{14}), Dynamic_Condition_Count (x_{15}), Dynamic_Convert_Count (x_{16}), Dynamic_Extract_Count (x_{17}), Dynamic_Fcalc_Count (x_{18}), Dynamic_Fconvert_Count (x_{19}), Dynamic_Load_Count (x_{20}), Dynamic_Ret_Count (x_{21}), Dynamic_Store_Count (x_{22})

dynamic cases due to unknown n. Hence, machine learning regression [28] is employed for predicting maximum RSS. At the end of each basic block, an individual maximum of static and dynamic features is computed.

2) PREDICTOR

The extracted feature set is passed through the regression model for predicting the maximum RSS as depicted in Figure 6. Only those applications are passed whose allocation size is unknown at compile time. The regression model is built offline from the training dataset, comprising of diverse applications expected to possess similar allocation behavior. The regression model is built based on the training set listed in Table 1 using *linear*,[7] *tree*,[8] *random forest*,[9] and *SVM*[10] techniques. The testing has been done using leave one out cross-validation. The results are reported in 2 categories i.e ***seen*** and ***unseen***. ***Seen*** indicates the case, where the test application is present in the training set but with different data sizes. In the case of ***unseen***, no evidence of test application is kept in the training set. Each application is run with almost 10 varying data sizes to construct per machine supervised dataset. The features of each application are extracted with respect to data sizes. The obtained dataset possesses skewness, which is reduced by taking *ln* transformation. The prediction accuracy is evaluated in terms of relative error (MRE), which judges the closeness of predicted values (y_i') to the actual ones (y_i) as given in Equation 2 [33].

$$Mean\ Relative\ Error\ (MRE) = \frac{1}{N}\sum_{i=1}^{N}\frac{|y_i' - y_i|}{y_i} \qquad (2)$$

For the ***seen*** category Figure 7, the lowest mean relative error of 21.4% is observed for SVM. SVM is run with radial kernel and optimal values of parameters ***sigma***, ***C***[11] have been found as 0.3 and 8000, through the *tune* function of R. The maximum accuracy is obtained via SVM due to the non-linear nature of data. Individually, *bfs* shows the least error of 9.7% with SVM. For *dfs* and *genann*, SVM shows greater mean error rates of 71.3% and 24.8%. In the case of *dfs*, the mean error rate is increased due to 381% error for 6 MB size. Similarly, for *genann*, the error rate is high due to

[7] *Linear* builds the relationship between response and explanatory variables using linear functions [54].

[8] *Tree* fits a regression model to each node for predicting the response variable and uses the squared difference for measuring the prediction error.

[9] *Random forest* is an ensemble learning method to predict the output by combining multiple trees.

[10] In *Support Vector Machine (SVM)* regression, an optimal hyperplane is found, which is expressed as a combination of support vectors.

[11] The margin gets smaller with a larger value of parameter C and vice versa. The larger epsilon results in more flat estimates, and selection of fewer selected support vectors. The Radial Basis Function (RBF) kernel can handle the situations where the relation between dependent and independent variable is non-linear by using parameter sigma/gamma [44], [55].

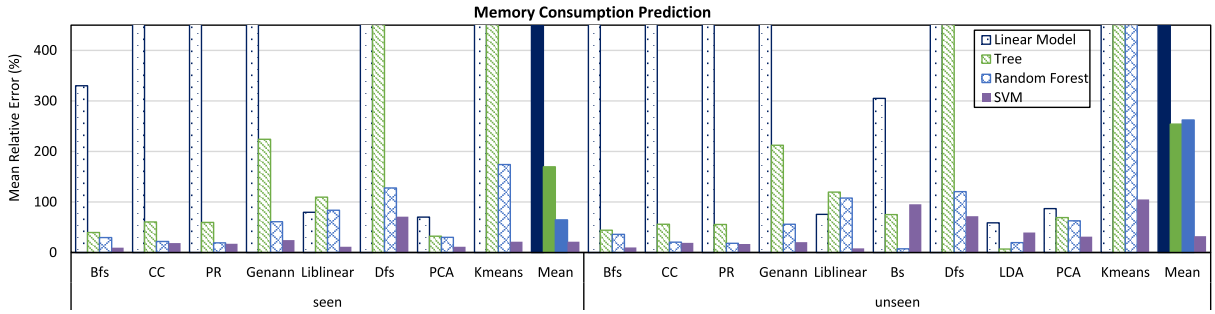

FIGURE 7. Memory consumption prediction.

130.5% error for 9.1 MB size. The error is reported by taking the average of separate readings of a single application with different data sizes. It is observed that error rates are higher with smaller sizes in the test set and vice versa, indicating the unstable and lesser accurate consumption for small sizes.

In the case of ***unseen*** applications, the overall mean relative error is increased for all the techniques as depicted in Figure 7. SVM performs better than others by showing an error rate of 32.4%. SVM is run with radial kernel and optimal values of parameters ***sigma*** and ***C*** are 0.4 and 365. The error rates are increased, as no instance of the same application is present in the training set. For *kmeans*, an error rate of 105% is observed with SVM, which is still lesser than others. SVM depicts the smallest error of 8.1% for *liblinear*, indicating the similarity of *liblinear* memory pattern with the other training samples. For *blackscholes (Bs)*, random forest shows a lesser error of 7.6% as compared to SVM error of 95.7%, indicating the dominance of *Bs* similar applications in the random forest model. The higher error rates are possibly due to two reasons; the characteristics of testing application are not covered by the training dataset, or the model is constructed such that one type of data is preferred over the other, hence giving more accurate results for one and less accurate for the other. The error rates are expected to be reduced by extending the training set with more diverse applications.

Due to the lower error rate, *SVM* has been selected as a default technique for the *volume predictor* of the framework. First, the model will be trained offline to acquire the relevant SVM coefficients. Then, these coefficients will be coded in the *SVM based volume predictor*.

3) DETECTOR

For identifying volume, algorithm 1 is used, which accepts as input the *test application code IR*, *data size*, *available memory amount*, *instrumented executable file*, and *sample datasets*. In case if the dynamic allocation is null, maximum RSS is estimated by static count only. Otherwise, *SVM* regression is used for predicting the maximum RSS. The memory consumed ratio is computed by obtained RSS and available memory. The obtained ratio is compared with the volume threshold. In case if the ratio is greater than the threshold, the volume is present for the considered scenario, otherwise, the volume is absent.

Algorithm 1 Detection of Existence of Volume

Input: Test_Code_IR,Instrumented_Exe_file,Sample_Dataset_a,Sample_Dataset_b, Sample_Dataset_c,Test_Data_Size,Total_Memory_Available
Output: Status of volume existence (FALSE, TRUE)

```
function Volume_Detection(Test_Code_IR,Instrumented_Exe_file,Sample_Dataset_a,Sample_Dataset_b,Sample_Dataset_c,Test_Data_Size,Total_Memory_Available):
    x ← Memory_Feature_Extraction(Test_Code_IR, Instrumented_Exe_file, Sample_Dataset_a,Sample_ Dataset_b,Sample_Dataset_c);
    x1* ← x.f1, x2* ← x.f2;
    if x2*=0 then
        y* ← x1*;
    else
        y* ← Memory_Prediction(x);
    if y*/Total_Memory_Available > θv1 then
        volume ← TRUE;
    else
        volume ← FALSE;
    return volume;
```

B. VELOCITY DETECTION

Velocity means the real-time generation of data, which requires fast processing to meet the deadlines [1], [3]–[5], [9], [56], [57]. The velocity can be detected if the data generation rate is greater than the processing rate, which means the execution time per process exceeds the product of sample count and source sampling period (T_s) as shown in Equation 3. The processing deadlines are missed when the processing time per sample is greater than the sampling period.[12] The processing time is varied with the data size and format. Therefore, the presence of velocity can be determined through application, platform, and data properties namely size, format, and generation rate.

$$Velocity = true, \text{ if } P(t) > Number\ of\ samples * T_s \quad (3)$$

This paper proposes *feature extractor*, *predictor*, and *detector* modules for velocity detection. Only size variation is considered assuming fixed data formats. At compile time, application execution time, source sampling period, and sample count are required for detecting velocity. This work assumes constant rate data generation, hence both the sampling period and the sample count are predetermined at

[12]Sampling period is the reciprocal of sampling rate, which is defined as the number of samples per unit time [57]. Each data source has a predefined sampling rate [58].

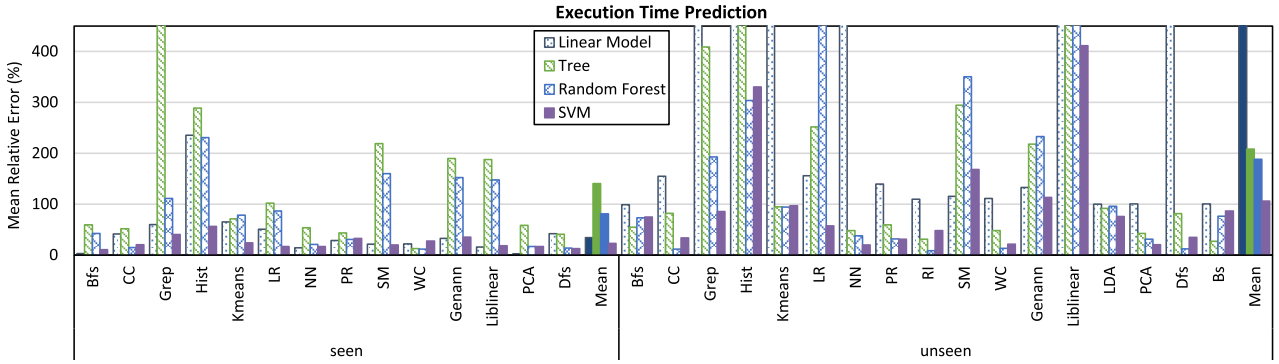

FIGURE 8. Execution time prediction.

compile time, and effort is made for estimating the execution time.

Obviously, the exact time cannot be determined without executing the application, hence rough estimations are preferred at compile time. Static complexity analysis of the application is discouraged due to non-deterministic execution paths and inherent analysis complexity of the real-world applications [30], [34], [35]. In this regard, no compiler based tool is found to precisely analyze the complexity of programs. Instead, in literature profiling and statistical analysis are emphasized for estimating the time [28], [30], [31], [33], [35]. In the case of big data, one-time application profiling is not preferred, due to larger and complex datasets, which consumes time and resources resulting in substantial overhead [28]–[31], [33]. In this situation, per machine regression analysis is a feasible option for predicting time [28], [31], [33], which is dependent on comprehensive training, and expected to be less accurate. However, the overhead and complexity are minimal in comparison to other approaches.

In this work, time prediction is based on data size variations, which is considered along with application features, as time is not constant for different applications operating on the same size. The time estimation requires the control flow information, which is only known at run time. Hence, the proposed framework utilizes both the static and application control flow information [30], [31], [33], which is obtained by executing the instrumented code with three small size samples [35]. The statistics achieved with smaller datasets can be extrapolated for larger sizes [30], [35]. Finally, the predicted time assists in velocity determination.

1) FEATURE EXTRACTOR

The considered features include static and dynamic count of *integer calculation*, *call*, *compare*, *condition*, *integer convert*, *extract*, *float calculation*, *float convert*, *load*, *return*, and *store* instructions. The main motivation behind selecting these features is [31]. Static instructions are executed irrespective of data size, whereas dynamic count varies with respect to data size. The procedure for determining the instructions count is shown in Figure 6. The compiler front end emits the Intermediate Representation (IR), which is analyzed to know the details of all the functions, basic blocks, and instructions present in the code. The IR is instrumented and run with three smaller samples to determine the executed functions and basic block names. The obtained traces are compared to know the static and dynamic basic blocks. A basic block is static if its frequency is the same for all three samples, otherwise, the basic block is dynamic. The further analysis of obtained basic block information yields static and dynamic instruction counts.

Static instruction counts are correct as these are the same for all the data sizes of a single application. While the dynamic counts might contain error as these are extrapolated based on three samples. In this work, the static and dynamic are treated separately, instead of a consolidated count, which is different from the previous works [30], [31], [33]. It is justified for saving the absolute static values from the influence of less correct dynamic values. The proposed approach leads to lesser accuracy in comparison to profiling. However, minimal overhead is involved which is crucial for big data applications [30].

2) PREDICTOR

The train set builds an offline regression model for predicting the execution time of test applications. *ln* transformation is taken for reducing data skewness. For regression, *linear*, *tree*, *random forest*, and *SVM* are used, which is validated with hold one out technique. In the case of ***seen*** (118 samples) Figure 8, test application instances are included in training samples with different data sizes. Overall SVM shows the least mean relative error of 22.8%. Whereas, other techniques show greater error rates. SVM is run with radial kernel and optimal values of parameters ***sigma*** and ***C*** have been found as 0.02 and 300, through the tune function of R. Individually, *grep*, *histogram*, *genann*, and *page rank (PR)* show higher error rates with SVM, due to dissimilar behavior of test and train samples. For *grep*, *histogram*, *linear regression (LR)*, *string match (SM)*, *dfs*, and *kmeans*, SVM performs better than others. For *word count (WC)*, and *connected component (CC)*, random forest is better than SVM. Similarly, for *PCA*, *liblinear*, *WC*, *nearest neighbor (NN)*, and *bfs*, linear model shows lesser error rates as compared to SVM, due to linear

Algorithm 2 Detection of Existence of Velocity

Input: Test_Code_IR,Instrumented_Exe_file,Sample_Dataset_a,Sample_Dataset_b,Sample_Dataset_c,Test_Data_Size,Sample_count,Source_sampling_period
Output: Status of velocity existence (FALSE, TRUE)
function Velocity_Detection(Test_Code_IR,Instrumented_Exe_file,Sample_Dataset_a,Sample_Dataset_b,Sample_Dataset_c, Test_Data_Size,Sample_Count, Source_Sampling_Period):
  $y^* \leftarrow$ *Time_Prediction(Test_Code_IR,Instrumented_Exe_file,Sample_Dataset_a,Sample_Dataset_b,Sample_Dataset_c,Test_Data_Size)*;
  if $y^* >$ *Sample_Count* $*$ *Source_Sampling_Period* **then**
    velocity $\leftarrow$ TRUE;
  else
    velocity $\leftarrow$ FALSE;
  return *velocity*;

Algorithm 3 Detection of Existence of Variety

Input: Test_Code_IR,Test_Data_Format
Output: Status of variety existence (FALSE, TRUE)
function Variety_Detection(Test_Code_IR,Test_Data_Format):
  base_format $\leftarrow$ *Format_Extraction(Test_Code_IR)*;
  base_format_count $\leftarrow$ *base_format.length*;
  het_format_count $\leftarrow$ *Difference(base_format,Test_Data_Format).length*;
  het_ratio $\leftarrow$ (*het_format_count*)/(*base_format_count* + *het_format_count*);
  if *het_ratio* >0 AND *het_ratio* <1 **then**
    variety $\leftarrow$ TRUE;
  else
    variety $\leftarrow$ FALSE;
  return *variety*;

behavior of applications. The error is reported by taking the average of individual applications with different data sizes. With smaller sizes, the error rate is higher and vice versa.

For the ***unseen*** category (121 samples) Figure 8, no instance of test application is present in the training set, hence the prediction errors are increased. In comparison to others, SVM shows a lesser error rate of 105.8%. SVM is run with radial kernel and optimal values of parameters ***sigma*** and ***C*** are 0.0001 and 175. The error rates are relatively lesser for *dfs*, *PCA*, *WC*, *PR*, *reverse index (RI)*, *NN*, and *CC*, due to the coverage of application properties in the training dataset. The error rates are higher with smaller data sizes and vice versa. For example, in the case of *SM*, the error rate is 357.7% with SVM for 108.5 MB size, and it is reduced to 76.7% for 30.4 GB size, giving an overall average of 168%. The error rate is significant, even with a slight drift in predicted value, due to smaller time readings. For example, with actual 0.2 and predicted 0.1 the relative error is 50%. But, with the actual 1000 and predicted 900 the relative error is only 10%. The high error rates are emerged due to smaller values, less affecting the detection accuracy. The error rates can be reduced by including large size samples and diverse applications in the training set.

Due to the lower error rate, *SVM* has been selected as a default technique for the *velocity predictor* of the framework. First, the model will be trained offline to acquire the relevant SVM coefficients. Then, these coefficients will be coded in the *SVM based velocity predictor*.

3) DETECTOR

The time predicted via *SVM* is compared with the product of sampling period and sample count as shown in algorithm 2. In case of greater time, the test application in the given scenario possesses velocity, otherwise, velocity is absent.

C. VARIETY DETECTION

Variety is originated due to heterogeneous data formats [1], [4], [5], [9], [59]. For N representations, structural variety is present if the process heterogeneity ratio $P(h_{st})$ lies between 0 (exclusive) and $P\left(\frac{N-1}{N}\right)$ (inclusive) as per Equation 4. The heterogeneity ratio is the ratio of heterogeneous representations (with respect to a base application) and total representations as per Equation 5. These representations involve difference in data type, data schemas, etc., [59].

$$Variety = true, \text{ if } 0 < P(h_{st}) \leqslant P\left(\frac{N-1}{N}\right) \tag{4}$$

$$P(h_{st}) = \frac{Number\ of\ heterogenous\ representations\ w.r.t\ base}{Total\ number\ of\ representations} \tag{5}$$

Variety can be detected by computing the heterogeneity ratio, which is straightforward to estimate. The data manipulation formats are extracted by analyzing application code, and external data sources. The formats are then compared for computing the heterogeneity ratio.

1) FEATURE EXTRACTOR

The *file read* function calls are statically analyzed to check the placement format of file data. The format vector is constructed by appending the formats of all the read operations of an application.

2) DETECTOR

The variety status is decided through the application and test data format vectors as depicted in algorithm 3. The base format count includes the data manipulation formats used in the application. The application and external source formats are compared for computing heterogeneous format count. The format vector for a specific data source is fixed. The heterogeneous and base format counts compute the heterogeneity ratio. The variety status is true if the heterogeneity ratio lies in the range 0 and 1 exclusive.

IV. RESULT ANALYSIS

A. EXPERIMENTAL SETUP AND BENCHMARKS

The implementation has been done using Low Level Virtual Machine (LLVM) compiler[13] analysis passes and R[14] scripts [35]. LLVM analysis passes have been developed to extract code features. Those features have been processed using R scripts. For validating volume and velocity, only data size variation is considered, and the data formats are

[13]The LLVM Compiler Infrastructure https://llvm.org/.
[14]The R project for statistical computing https://www.r-project.org/.

TABLE 2. Benchmark details.

Category	Application	Description	Input Dataset Format
Graph Mining [52], [60]	**Breadth-First Search (BFS)** [36]	Traverses a graph in a breadthward motion.	Graph generated by specifying the number of nodes.
	Connected Component (CC) [37]	Computes set of connected subgraphs for a given graph.	Graph dataset (vertices and edges files).
	Page Rank (PR) [37]	Measures the graph nodes connectivity by assigning a rank to each node.	Graph dataset (vertices and edges files).
	Depth-First Search (Dfs) [37]	Traverses a graph in a depthward motion.	Graph dataset (vertices and edges files).
Machine Learning [52], [60]	**Kmeans** [36]	Represents the data objects by the centroids of the sub-clusters by dividing a cluster of data objects into K sub-clusters.	Dataset consisted of set of numeric features.
	Nearest Neighbor (NN) [36]	Evaluates the k nearest neighbors by calculating the euclidean distance from the target latitude and longitude.	Hurricanes dataset in the format "year, month, date, hour, num, name, lat, lon, speed, press".
	Gennan [41]	Neural network library for using and training feedforward artificial neural networks (ANN).	Numeric predictive attributes and the class.
	Liblinear [40]	Library for linear regression and classification.	Numeric predictive attributes and the class.
	Latent Dirichlet Allocation (LDA) [40]	Topic modeling algorithm that is used in natural language processing for discovering topics from unordered documents.	Document is represented as a sparse vector of word counts, in the form: [M][term_1]:[count]....[term_N]:[count]
	Principle Component Analysis (PCA) [40]	It is statistical technique for feature extraction in multivariate datasets.	Image data numeric attributes and the class.
Text Search [52], [60]	**Grep** [39]	Searches a file for a particular pattern of characters, and displays the lines containing that pattern.	Text file containing words.
	String Match (SM) [38]	Pattern matches a set of strings against streams of data.	Text files containing the list of encrypted words.
	Word Count (WC) [38]	Counts the frequency of occurrence of each unique word in a text document.	Text files containing words.
Statistical & Mathematical [52], [60]	**Histogram (Hist)** [38]	Computes the RGB (Red Green Blue) histogram of input image.	Bitmap Image files.
	Linear Regression (LR) [38]	Applies linear regression best-fit over data points.	Text files containing the input coordinates.
	Reverse Index (RI) [38]	Recursively traverses html files and extract tags from html pages to build the reverse-index.	Collection of folders that contain html files.
	Blackscholes (Bs) [42]	It analytically computes the prices for a portfolio of European options using the Black-Scholes partial differential equation (PDE).	Numeric array of data of options.
	Bzip2d [43]	Burrows-Wheeler decompression algorithm is implemented.	Compressed file with bz2 extension.

assumed to be constant. The hardware and software configuration details are given in Table 1. The results have been reported in terms of ***seen*** and ***unseen*** application categories. In the ***seen*** category, the testing applications are also part of the train set, but with different data sizes, whereas in the case of ***unseen*** an application is only part of the test set.

For extracting the actionable insights from the avalanche of data, *graph traversal*, *machine learning*, *text analytics*, and *statistics* algorithms are commonly used [61]. In this work, only those applications have been tested to validate the detection of 3Vs (Volume, Velocity, Variety) which are part of standard big data benchmarks, representing *graph mining*, *machine learning*, and *statistics & mathematics* categories [52], [60]. These applications have been selected through well known C/C++ based benchmark suites including Rodinia [36], Graphbig [37], Phoneix [38], CortexSuite [40], cbench [43], genann [41], PARSEC [42], and grep-bench [39].[15] These include *bfs*, *grep*, *kmeans*, *word count*, *page rank*, etc as discussed in Table 2.

The applications *bfs*, *connected component*, *page rank*, and *dfs* operate on large graph datasets containing list of edges and vertices. Similarly, *kmeans*, *nearest neighbor*, *genann*, *liblinear*, *LDA*, and *PCA* deal with large machine learning datasets. It can be noted that *PCA* operates on high dimensionality datasets for extracting features. Finally, *grep*, *wordcount*, and *reverse index* perform the search operation. While *blackscholes*, *linear regression*, *bzip2d*, and *histogram* applications perform statistical & mathematical operations on a large amount of data. It can be observed that most of these applications use text datasets, while, *histogram* uses image dataset, and *reverse index* recursively operates on text files placed in multiple folders. The considered benchmark applications and datasets are diverse in nature, covering maximum possible domains of big data, hence these are sufficient for testing the existence of 3Vs.

[15]For constructing the dataset, diverse applications have been run with maximum 11 random sizes as listed in Table 1.

B. PERFORMANCE METRICS

For measuring the accuracy of detecting the 3Vs, detection accuracy is used as represented by Equation 6.

$$\textit{Detection Accuracy } (DA) = \frac{\textit{Number of Correct Detections}}{\textit{Total Number of Detections}} \tag{6}$$

While detecting the 3Vs, there can be either correct or incorrect detection. The accuracy is computed by dividing the count of those detections that are found to be correct (TP+TN) with the total detection operations performed (TP+TN+FP+FN). Where **true positive (TP)** means a real V is detected, **false positive (FP)** means a false V is detected, **false negative (FN)** means a real V is not detected, **true negative (TN)** means a false V is not detected.

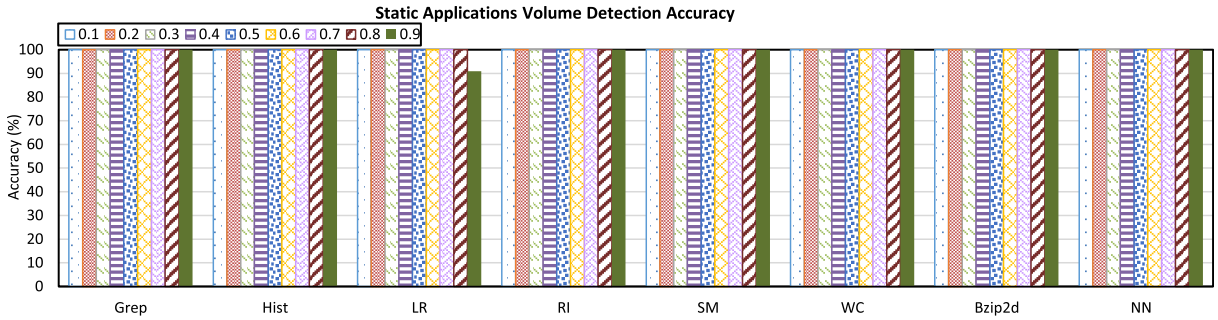

FIGURE 9. Static applications volume detection accuracy.

C. 3Vs DETECTION

1) VOLUME

The memory consumed ratio[16] is compared with the threshold.[17] The detection is considered correct, only if the predicted detection outcome (true/false) is the same as the actual detection status (true/false).[18] Due to the criticality of resource utilization, no error margin is tolerable at the detection stage.[19] The threshold value has a significant impact on the overall detection process. For example, with the threshold of 0.8, volume is absent with actual memory consumed ratio of 0.7. In case, the predicted consumed ratio is 0.1, the volume is correctly predicted despite a high error rate. However, with the predicted ratio of 0.81, the detection is wrong, despite of lower error rate. Hence, the boundary conditions are critical in increasing the false positives (FP) and false negatives (FN), requiring accurate predictions.

Applications *grep*, *nearest neighbor (NN)*, and *bzip2d* show numeric constant in static allocation only, which is due to stack and global contributions, while dynamic allocation is null, indicating the constant memory consumption. Similarly, *histogram (Hist)*, *linear regression (LR)*, *reverse index (RI)*, *string match (SM)*, and *word count (WC)* applications show null dynamic allocation, whereas the static allocation includes the sum of constant numeric and size factor.[20] For these static applications, volume detection accuracy is shown in Figure 9. The detection accuracy is 100% for majority of the thresholds and applications, due to deterministic allocation size which leads to lower errors. Only for *LR* with a threshold of 0.9, the accuracy is reduced to 90.9%, due to boundary value misprediction for size 30.4 GB, with an error rate of only 0.37%.

In contrast to this, *bfs*, *connected component (CC)*, *page rank (PR)*, *kmeans*, *blackscholes*, *dfs*, *genann*, *lda*, and *PCA* applications have both static and dynamic allocations, due to functions whose size can be determined at run time by reading the data file. Hence, for these applications direct computation of memory consumption is not possible, and SVM regression is employed for predicting memory consumption. The volume detection accuracy for these dynamic set of applications is depicted in Figure 10. For *genann*, *PCA*, *kmeans*, the detection accuracy is greater in the ***seen*** category, because the testing applications are also part of the train set, but with different data sizes. For the remaining applications, ***seen*** and ***unseen*** accuracy is almost the same. *Bfs*, *CC*, *PR*, show 100% accuracy for all thresholds except for 0.2. For *bfs* (14.7 GB), the misdetection is due to boundary conditions, where the predicted value is 0.19 for the actual value of 0.21. Similarly, for *CC* and *PR* 607.8 MB, the predicted value is 0.21 and 0.20 for the actual value of 0.18. For *genann*, *liblinear*, *PCA*, *kmeans*, the detection accuracy is less than 100% for more thresholds. *Kmeans* shows lesser accuracy for both ***seen*** and ***unseen***. In the ***seen*** case, *kmeans* accuracy is dropped to 85.7% for 0.4, 0.5, and 0.6 thresholds, due to 1 misdetection (24.8 GB) out of total 7. In this case, the predicted value is noted as 0.357 for the actual value of 0.69. In ***unseen*** case, for 0.1 threshold *kmeans* correctly detects only 245.9 MB, 739.9 MB, 2.5 GB, and misdetects the others, reducing the accuracy to 42.8%. Similarly, for 0.2 and 0.3 the accuracy is recorded to be 57.1% and 71.4%. These misdetections are due to increased prediction errors in the ***unseen*** case. The error rate is noted as 80.2%, 86.8%, 90.1%, and 95.1% for 4.9 GB, 7.4 GB, 9.9 GB and 24.8 GB size.

The combined detection accuracy is reported for both static and dynamic in Figure 11. In the case of ***seen***, the average accuracy of 97.8% can be observed. For the threshold of 0.9, the detection accuracy is highest i.e 98.4%, due to lesser cases having consumption greater than 0.9, thus even the mispredictions are not affecting the overall accuracy. The threshold of 0.2 shows the least accuracy of 95.2%, due to the mix of both true and false samples for this range, facing more

[16]Memory consumed ratio is defined as the proportion of maximum physical memory consumed by a process with respect to total available RAM size.

[17]Volume threshold is the maximum allowed memory consumption ratio. It is expected to be high, at least 0.8 because there is a high likelihood of system performance getting dropped if consumption by a process is reached to 80%.

[18]It means memory consumption is rightly predicted with respect to threshold.

[19]For example, actual memory consumed ratio is 0.79 (volume absent in actual), and predicted is 0.81 (volume present in prediction), by considering this misdetection, volume optimizations are invoked, leading to resource misuse.

[20]The numeric constant is due to the stack and global allocations. Whereas, the size factor is due to the mmap function which allocates memory proportional to the data size.

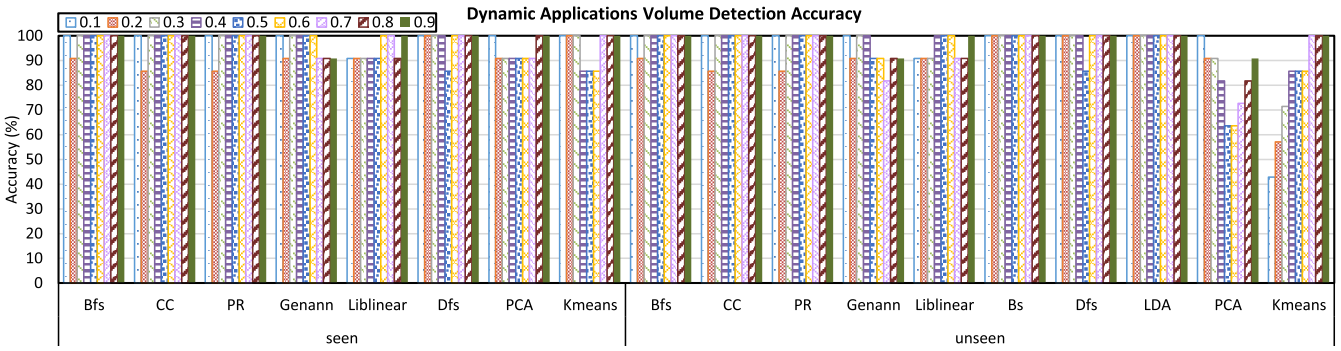

FIGURE 10. Dynamic applications volume detection accuracy.

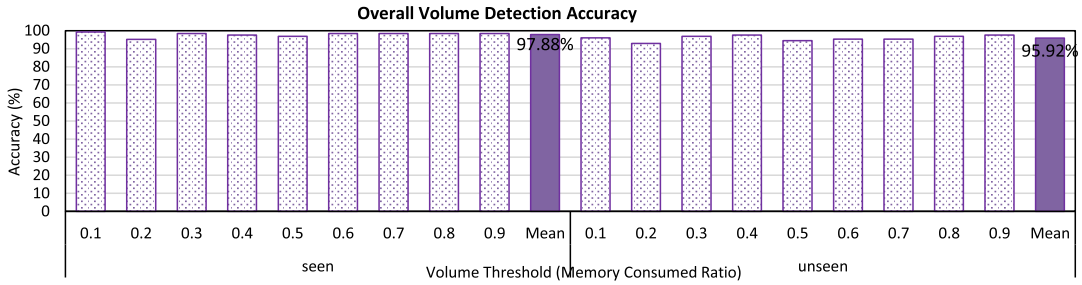

FIGURE 11. Overall volume detection accuracy.

misdetections. With the threshold of 0.8, reasonable accuracy of 98.4% is achieved. The selection of 0.8 is practical as the system performance is harmed for the consumption above it. In the case of ***unseen***, the average detection accuracy is reduced to 95.9%. The detection shows the highest accuracy of 97.6% for the threshold of 0.9, due to lesser consumption cases having values greater than 0.9, hence the mispredictions are not affecting the overall accuracy. The threshold of 0.2 shows the least accuracy of 92.9%, due to the mix of both true and false cases in this range, which increases the misdetections. With the practical threshold of 0.8, the accuracy of 96.8% is achieved. For certain applications, the memory consumption cannot be accurately estimated at compile time. However, with certain constraints, the proposed technique is showing satisfactory results both for ***seen*** and ***unseen*** applications. Hence, the selected feature set is justified for predicting the memory consumption of an application.

2) VELOCITY

The detection is done by considering multiple sampling periods and sample count. In this regard, each sample is assumed to be 1 kB, thus sample count is obtained by dividing data size with 1 kB.[21] The average detection accuracy for individual applications is shown in Figure 12. For all applications (***seen*** & ***unseen***), the detection accuracy is 100% with sampling periods of 0.002s, 0.02s, 0.2s, and 2s, due to slower generation rate. Further decrease in sampling period to 0.2ms, 0.02ms, 2μs still shows 100% accuracy for *connected component (CC)*, *nearest neighbor (NN)*, *word count (WC)*, and *PCA* (***seen*** & ***unseen***). The accuracy is higher due to lower prediction errors and the absence of boundary cases. *String match (SM)* shows 100% accuracy for all thresholds in the ***seen*** case, but the accuracy is dropped to 54.5% for ***unseen*** 0.02ms period. The misdetections occur in SM (108.5 MB, 542.5 MB, 1.1 GB, 2.2 GB, 4.3 GB), due to higher error rates for these sizes. For *bfs* and *liblinear*, the accuracy is 100% for all periods except 0.02ms in both ***seen*** and ***unseen*** cases. In the ***seen*** case, the prediction error is only 10.7%, but the accuracy is 27.2% for 0.02ms, due to boundary conditions. In ***unseen*** case, the prediction error is increased to 74.5%, reducing the accuracy to 18.1% for 0.02ms. Similarly, for *linear regression (LR)*, the accuracy is 100% for all thresholds except for 2μs. In the case of *LR* ***seen***, the accuracy is dropped to 36.3% due to boundary values, despite an error rate of only 16.7%. Whereas, for *LR* ***unseen***, the accuracy is 0% due to an increased error rate of 57.2%. For *page rank (PR)*, the ***unseen*** error rate is 30.9% as compared to the ***seen*** rate of 32.5%, which leads to 100% accuracy of *PR* for all thresholds in case of ***unseen***. Whereas, for ***seen***, the accuracy is dropped to 83.3% for 0.2ms.

For ***seen*** applications, the average detection accuracy is 97.3% as shown in Figure 13. Practically, the sampling periods are lesser, hence contributing to velocity. The accuracy of 98.3% is noted for sampling period of 0.2ms. By decreasing the sampling period to 0.02ms, the detection

[21]The velocity is true, only if the data processing time is greater than generation time. The velocity detection is considered correct, only if the predicted detection outcome is same as actual detection status.

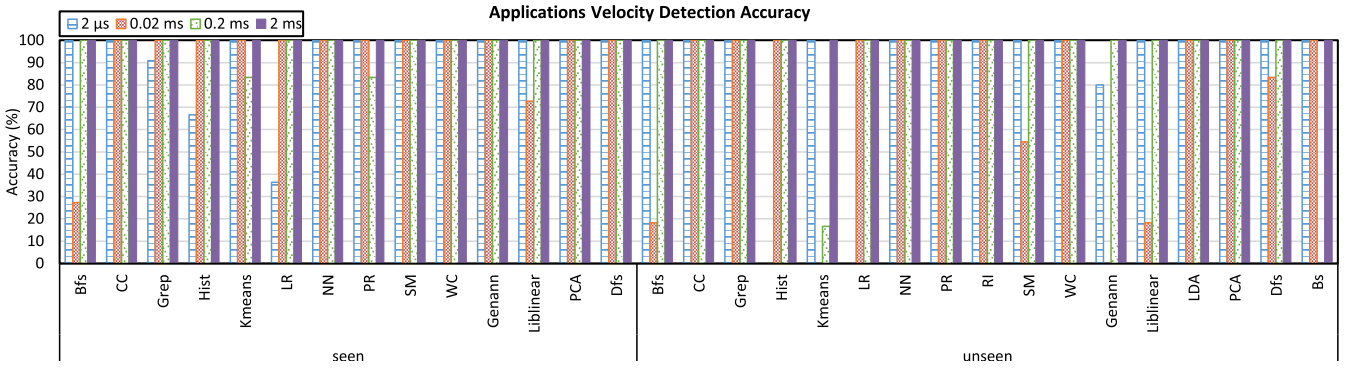

FIGURE 12. Applications velocity detection accuracy.

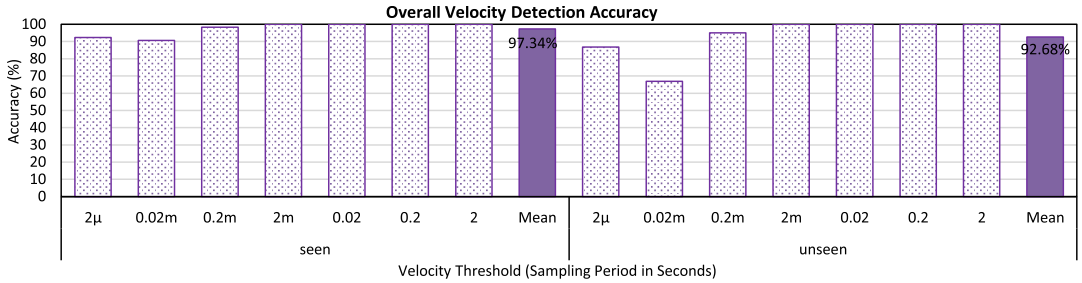

FIGURE 13. Overall velocity detection accuracy.

accuracy is dropped, as there is an equal mix of velocity true and false samples for this value. Hence, the detection can both be true or false instead of constant decisions like the other cases. In the considered situation, the predictions are tougher, as a slight miss can lead to bad accuracy. The performance is lesser for period of 0.02ms, due to misdetections of *bfs*, and *liblinear* samples. The misdetected samples are the boundary cases for the considered sampling period. By further decrease in the sampling period, true samples are increased, improving the overall accuracy as noted for 2μs. For *grep* (5.4 GB), despite of 32% error rate, the status is wrongly detected to be false due to boundary values.

For ***unseen*** applications, the average accuracy is dropped to 92.6% as depicted in Figure 13. For most samples, the velocity is true with sampling period of 2μs. In this situation, the average accuracy of 86.7% is observed, due to *histogram*, *genann*, and *LR* misdetections. For the sampling period of 0.02ms, velocity true and false samples are mixed, hence the detection accuracy is dropped. In this situation, the mean accuracy 66.9% is observed. Further increase in the sampling period to 0.2ms (involving more false cases), improves the average accuracy to 95%. Few samples of *kmeans* and *blackscholes (Bs)* show misdetections. From the above discussion, the significance of correct feature selection is evident for detecting the presence of velocity via SVM.

3) VARIETY

Considering the non-adaptive applications, where the base schema is hardcoded. As per Table 3, when *bfs* application with the base format of {int, int, int, int, int, int, int}, deals with the dataset having format {unsigned long, unsigned long, unsigned int, unsigned long, unsigned long, unsigned long, unsigned int}, the variety is *false* due to null heterogeneity ratio. However, when the same application receives data in {unsigned long, unsigned long, unsigned float, unsigned long, unsigned long, unsigned long, unsigned float} format, the variety status is *true* due to non-null heterogeneity ratio. Similarly, the histogram [38] is intended for bitmap 24 bit images. In case, when jpeg images are input via a new source, heterogeneity ratio becomes greater than 0.

In this manner, the framework detects whether the application with incoming data possesses variety by following this simple yet effective method. The detection method is valid for all the formats and types of data be it basic like integer, float, or complex like image, video, or other type. Although, image and video do have different processing demands, but this doesn't represent software and hardware incapability. In this situation, the variety gets true only, when the given hardware or software is not capable to fulfill the processing demands of these formats. Variety represents the handling deficiencies of hardware and software because a single application needs to deal with different data formats. No matter, how complex the format is, if the infrastructure is capable of handling it, variety is not true. The detection relies on straightforward comparisons, hence there is no margin of error in variety detection even if a diverse range of application areas are considered that involve complex formats.

D. 3Vs OPTIMIZATIONS

The support for volume workloads is depicted through Figure 14 (a). For *page rank* (6.1 GB), the detector indicates

TABLE 3. Variety presence status for heterogeneous data formats.

Application Name	Base Format	Test Data Format	Base Format Count	Heterogeneous Format Count	Heterogeneity Ratio	Variety Status
Bfs	{int, int, int, int, int, int, int}	{unsigned long, unsigned long, unsigned int, unsigned long, unsigned long, unsigned long, unsigned int}	7	0	0	FALSE
Bfs	{int, int, int, int, int, int, int}	{unsigned long, string, unsigned float, unsigned long, unsigned long, unsigned long, unsigned float}	7	3	0.3	TRUE
Grep	{char*}	{char*}	1	0	0	FALSE
Grep	{char*}	{int}	1	1	0.5	TRUE
Histogram	{char* for 24 bit bitmap image file format}	24 bit image file in bitmap format	1	0	0	FALSE
Histogram	{char* for 24 bit bitmap image file format}	jpeg image file	1	1	0.5	TRUE
Kmeans	{int, int, float}	{int, int, float}	3	0	0	FALSE
Kmeans	{int, int, float}	{float, float, int}	3	3	0.5	TRUE
Linear Regression	{char*}	{string}	1	0	0	FALSE
Linear Regression	{char*}	{float}	1	1	0.5	TRUE
Nearest Neighbor	{int, int, int, int, int, char*, float, float, int, int}	{int, int, int, int, int, char*, float, float, int, int}	10	0	0	FALSE
Nearest Neighbor	{int, int, int, int, int, char*, float, float, int, int}	{float, float, int, char*, int, char *,float, float, int, float}	10	4	0.28571	TRUE
String Match	{char*}	{string}	1	0	0	FALSE
String Match	{char*}	{int}	1	1	0.5	TRUE
Word Count	{char*}	{string}	1	0	0	FALSE
Word Count	{char*}	{float}	1	1	0.5	TRUE

the presence of volume, whereas for *grep* (53.7 GB) the volume status is detected as false, due to lower memory consumption. For both the cases, the detection is correct, as the volume is present in *page rank* not in *grep*, despite the lesser data size. Hence, the volume cannot be detected by data size only, and a detector is vital. Similarly, velocity support is depicted through Figure 14 (b). For sampling period of 0.02ms, the detector indicates the presence of velocity for *page rank* (1.8 GB), whereas for *grep* (53.7 GB) the status is detected as false, due to deadline compliance. The detection is correct for both the scenarios. Due to deadline miss, the velocity is present in *page rank* not in *grep*, despite lesser data size of *page rank*. Hence, velocity cannot be detected by data size only, and a detector is vital. Finally, the support for variety workloads is depicted through Figure 14 (c). For *page rank*, the detector indicates the presence of variety, whereas for *grep* the variety status is detected as false, due to null heterogeneity.

Volume can be tackled by offloading *page rank* to a cluster or cloud consisting of larger size RAMs. Also, it can be tackled by performing *garbage collection* operations through libraries or compilers [22]. For velocity, *page rank* can be executed in parallel on *Hadoop*, *Spark*, *Flink*, *Storm*, etc *clusters* for reducing the execution time. Also, the jobs can be run on *hardware accelerators* by generating appropriate machine codes via compilers. Further, compiler optimizations like *parallelization*, *vectorization*, *loop*, *data access*, etc., [23] can be used. Finally, variety can be tackled by invoking NoSQL databases or compiler optimizations like *type inferencing*, *data layout*, etc., [24].

The *detection* technique can be integrated inside the existing tools like Hadoop, Spark, Storm, Flink, MongoDB, etc by means of a shell *script*. Besides, it can be integrated into any commercial compiler by means of a *compiler pass*. In this manner, the optimizations are only enabled if there is a need, promoting the efficient utilization of human, financial, and computing resources. Also, the programmer is freed from the burden of selecting the suitable optimizations. The programmer is expected to run a single *script* or enable a compiler *flag*, which is sufficient to first detect the incoming *3Vs*, and then trigger the relevant *memory consumption*, *time*, and *heterogeneity* optimizations, hence avoiding the user interventions.

V. CASE STUDY

Consider a *real-time airline recommendation system* to recommend the best airline from source to destination [62]. The recommendation is done via Twitter, Facebook, airline web API 1, and airline web API 2 real-time sources. Only three sources are present initially, and the fourth one is added later. These sources generate data at the rate of 5 samples/seconds, 2 samples/seconds, 10 samples/seconds, and 3.3 samples/seconds with different formats. The recommendation application is designed for the three sources.

In the case of Twitter and Facebook, text based tweets and comments are analyzed for extracting the airline sentiments.[22] Airline API 1 acquires airline details in {price (float), departure status (bool), timing delay (time)} format, and airline API 2 uses {weather conditions (text), airport facilities (text), flight historical data (text), price (double)} format. The application fuses user sentiments, price, timing, weather conditions, airport facilities for recommending the airline with the highest positive sentiments, lowest price, and lowest delay counts. Assuming the fusion application presented in Figure 15, is run on (8 GB RAM,

[22]Sentiment analysis is the process of understanding an opinion about a given subject from written or spoken language.

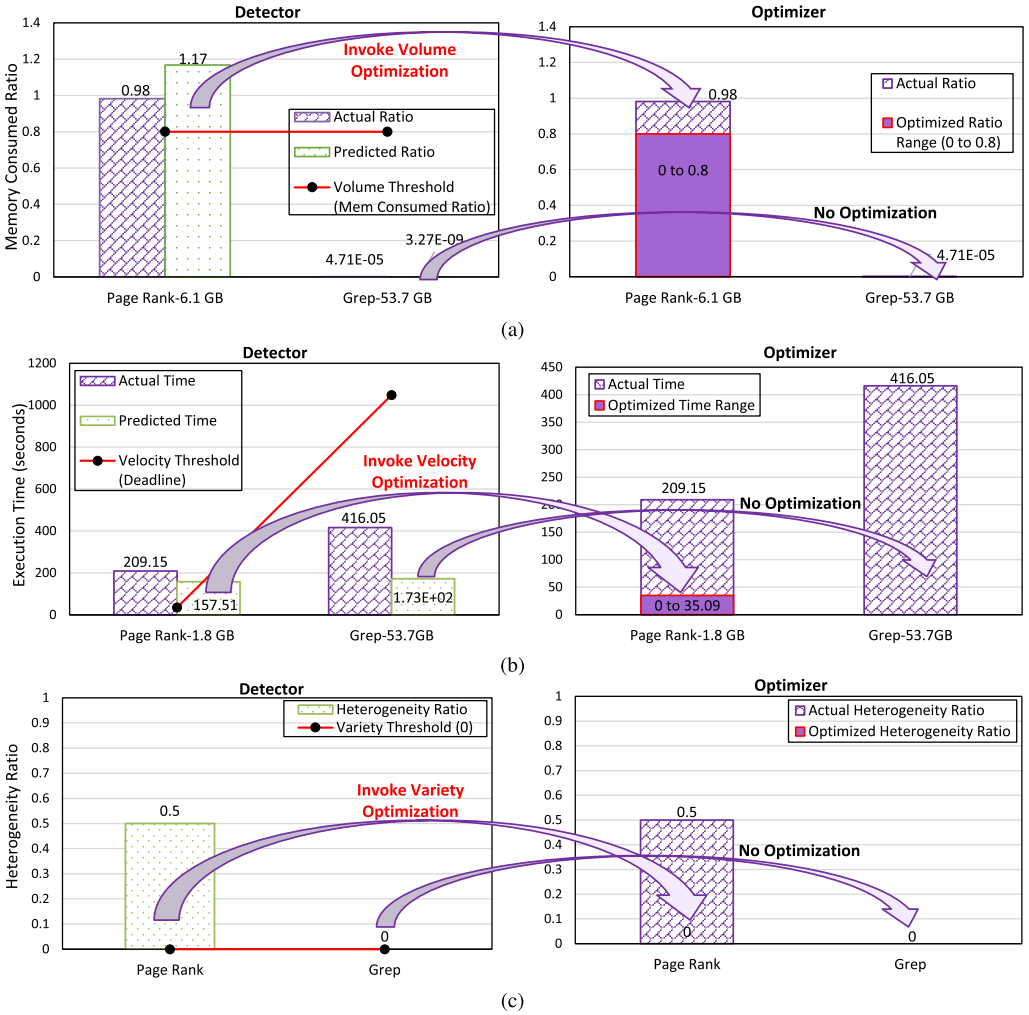

FIGURE 14. Automatic support for 3Vs. (a) Volume. (b) Velocity. (c) Variety.

```
A=malloc(size of source 1);
B=malloc(size of source 2);
 . . . . . .
N=malloc(size of source N);
while(1)
{
A=read source 1;
B=read source 2;
 . . . . . .
N=read source N;
for(i=0;i<ws;i++)
Process and Fuse A, B,.....N; // data of ws sec
}
```

FIGURE 15. Pseudo code of fusion application.

Intel i7-8550U CPU @ 1.80GHz) platform. The application reads samples from N sources, storing in memory, and fusing it. The window size (ws) is the product of sample count and source sampling period, hence for each source (ws/T_S) samples can be processed at one time. The ws is taken as 5000 seconds.

For 3 sources, 25,000, 10,000, and 50,000 samples are collected over 5000s ws, consuming 2 GB, 1 GB, and 4 GB memory. Afterward, source 4 data is expected with 16,667 samples. For the detection of 3Vs via the proposed framework, source data sizes, sampling periods, and data formats are known. The memory consumed ratio is estimated to be 7 GB[23] by adding the data sizes of 3 sources, indicating the presence of volume due to consumption ratio of above 0.8. Similarly, the processing time is predicted to be greater than 5000s, hence velocity is true. The application code is generic enough to read data from multiple sources involving similar or different formats. The processing is done smoothly if the data is emitted in the same format. Otherwise, handling issues are raised due to variety. Consider the application starts reading data from source 4, collecting 16,667 samples. The format of source 4 is compared with the base format, and the heterogeneity ratio is computed to be above 0, thus indicating the presence of variety.

The framework relies on extensive per environment training for accurate predictions of memory and time

[23] 2 GB (source 1 data size) + 1 GB (source 2 data size) + 4 GB (source 3 data size) = 7 GB.

TABLE 4. Comparison with related works.

Characteristics	Works										
	[14]	[15]	[13]	[63]	[16]	[18]	[17]	[31]	[33]	[29]	**Proposed**
Implemented in	Spark	Spark	Spark	Spark	Spark	MapReduce & Spark	Spark	Compiler	Compiler	Compiler	Compiler
Dataset Format	Text	Text & Graph	Text	Text, Graph, & Image	Text & Graph	Text	Image	Text	Text	Text	Text, Graph, & Image
Execution Time Prediction	✓	✓	✓	✓	×	×	✓	✓	✓	×	✓
Memory Consumption Prediction	×	✓	×	×	✓	✓	×	×	×	✓	✓
Heterogeneity Ratio Estimation	×	×	×	×	×	×	×	×	×	×	✓
Generic Prediction Model	×	×	×	×	×	×	×	✓	✓	✓	✓
Big Data Workloads	✓	✓	✓	✓	✓	✓	✓	×	×	×	✓
Big Data (3Vs) Detection	×	×	×	×	×	×	×	×	×	×	✓

consumption. The detection requires prior knowledge of data, application, and platform properties. For real big data workloads, both the application and platform properties are known at compile time, but the data properties are not fixed. Such workloads can only be detected at run time when the real data properties are available. The run time detection is expected to increase overheads.

VI. RELATED WORK

For efficient utilization of big data resources, numerous techniques have been proposed which estimate the job's execution requirements. As it can be observed via Table 4, several works predict execution time [13]–[15], [17], [31], [33], [63], while others estimate memory consumption [16], [18], [29]. It can be observed that in [13]–[18], [63], the prediction is done for big data workloads, whereas in [29], [31], [33], the considered applications are routine ones. In [14], a performance prediction framework *Ernest* is proposed, which can predict the execution time on a hardware configuration, given a job and its input. It has been tested on Spark. In [15], a model is proposed to predict Spark job performance including execution time, memory consumption, and I/O cost. In [13], a machine learning approach is proposed for predicting the execution time of Spark applications. Similarly, an empirical model is proposed in [63], for estimating the completion time of the Spark job on a cloud, with respect to cluster nodes count, input data size, and the number of iterations. The model predicts the cost optimal cluster configuration for executing the Spark job on a cloud under the SLO (Service Level Objective) specified completion deadline.

In [16], a machine learning-based model is presented to predict an application's memory requirement given its service level agreement, which is evaluated on a Spark cluster. Similarly, a machine learning based model is proposed in [17], for predicting the completion time of convolutional neural network (CNN) models implemented in a Spark cluster. The prediction is done by analyzing the effects of data, task, and resource characteristics. In [18], the resource usage parameters (CPU usage, memory usage, read rate, and write rate) of MapReduce and Spark applications are predicted using multivariate LSTM and multiple linear regression methods. It predicts the current resource usage using previous time instant values, whereas our work can directly predict the resource consumption at a specific instance by using application, data, and platform properties. Additionally, in [18] separate models are constructed for each application resource prediction, whereas in our work a single generic model is enough to predict a particular metric for all applications.

Furthermore, machine learning based metrics prediction at compile time is reported in [31], [33]. In this regard, [33] predicts the performance of HPC applications running in the cloud through machine learning for selecting the best cloud configuration before deployment. Similar to our proposed approach, [33] relies on executing the instrumented code with smaller sample and extrapolating the larger size using the linear or neural network regression depending on application complexity. Our work considers big data workloads which are more complicated real-world application codes in comparison to [33], hence the direct complexity analysis is not possible. The large size metrics are extrapolated by observing the growth pattern achieved via three smaller samples. The mean relative error for time predictions reported in [33] is lesser in comparison to the work presented in this paper because the experiments are conducted using simpler benchmark codes that are similar in nature, while this work considers complex big data applications, possessing distinct behavior.

Similarly, in [31], the execution time of functions is predicted using machine learning, relying on instrumentation for computing feature vector, which is tested using simpler kernels involving no external datasets. Also, [31], [33] use total instruction counts, whereas this work treats static and dynamic counts separately, reducing the overall error rate. Moreover, our instrumentation technique is simpler as compared to [31], [33], as only the reachable functions and basic blocks list is generated. For predicting memory consumption, [29] estimates the stack frame size of a given task using static compiler analysis similar to our approach.

As per Table 4, no existing work is found for estimating the heterogeneity ratio. In comparison to existing works, the proposed work appears to be the first framework that successfully predicts all three big data metrics. Additionally, the prediction models proposed in [13]–[18], [63] work for a specific platform like Spark and MapReduce. Whereas, similar to [29], [31], [33], the proposed prediction model is generic because it is implemented at the compiler stage. The proposed model can work with any computational engine, because it extracts core application features via compiler Intermediate

Representation (IR), instead of using abstract application properties present in specialized implementations. Additionally, the proposed model has been tested with different types of datasets like text, graph, and image. Whereas, mostly existing solutions [13], [14], [18], [29], [31], [33] are designed for text datasets only, [17] for image dataset, few [15], [16] for text and graph both, while only [63] considers graph, image, and text. It can be observed via Table 4, the metrics predictions in existing works have been done in a different environment, using different techniques and workloads, hence a direct comparison of these with proposed work is not feasible.

The existing approaches predict individual metrics but none of these detect the big data 3Vs as shown in Table 4. The time, memory, and resource predictions are not enough, instead, these predictions are required to be linked with 3Vs thresholds for big data detection, which is missing from the literature. For the first time, a *generic 3Vs detection technique* has been presented in this paper, that can work with any computational engine, due to the core application features. Hence, our proposed framework can act as an offloading technique for specialized big data hardware and software.

VII. CONCLUSION

The prior cost-benefit analysis of big data is necessary for preventing the computational losses. In this regard, the first *framework* has been proposed for detecting the big data workloads automatically, thus assisting the selective offloading of these applications to dedicated hardware and software solutions. The proposed framework can be useful for detecting the big data (3Vs) characteristics for any application before execution, through *static code features*, *source data sizes*, *data formats*, *sampling periods*, and *platform knowledge*. The detection approach saves the computational resources by restricting the 3Vs (Volume, Velocity, Variety) optimizations to detected workloads only. The framework appears to be simpler yet effective with minimal overheads, lesser programmer interventions, higher usability, and portability. Hence, the framework is generic enough to be easily adapted in third party libraries and compilers, by passing the application, data, and platform properties as input.

REFERENCES

[1] M. Ge, H. Bangui, and B. Buhnova, "Big data for Internet of Things: A survey," *Future Gener. Comput. Syst.*, vol. 87, pp. 601–614, Oct. 2018.

[2] M. Chen, S. Mao, and Y. Liu, "Big data: A survey," *Mobile Netw. Appl.*, vol. 19, no. 2, pp. 171–209, Apr. 2014.

[3] C. Kacfah Emani, N. Cullot, and C. Nicolle, "Understandable big data: A survey," *Comput. Sci. Rev.*, vol. 17, pp. 70–81, Aug. 2015.

[4] P. Wongthongtham, J. Kaur, V. Potdar, and A. Das, "Big data challenges for the Internet of Things (IoT) paradigm," in *Connected Environments for the Internet of Things*. Cham, Switzerland: Springer, 2017, pp. 41–62.

[5] V. Jirkovsky, M. Obitko, and V. Marik, "Understanding data heterogeneity in the context of cyber-physical systems integration," *IEEE Trans. Ind. Informat.*, vol. 13, no. 2, pp. 660–667, Apr. 2017.

[6] T. R. Rao, P. Mitra, R. Bhatt, and A. Goswami, "The big data system, components, tools, and technologies: A survey," *Knowl. Inf. Syst.*, vol. 60, no. 3, pp. 1–81, Sep. 2019.

[7] C. L. Philip Chen and C.-Y. Zhang, "Data-intensive applications, challenges, techniques and technologies: A survey on big data," *Inf. Sci.*, vol. 275, pp. 314–347, Aug. 2014.

[8] M. Chi, A. Plaza, J. A. Benediktsson, Z. Sun, J. Shen, and Y. Zhu, "Big data for remote sensing: Challenges and opportunities," *Proc. IEEE*, vol. 104, no. 11, pp. 2207–2219, Nov. 2016.

[9] X. Jin, B. W. Wah, X. Cheng, and Y. Wang, "Significance and challenges of big data research," *Big Data Res.*, vol. 2, no. 2, pp. 59–64, Jun. 2015.

[10] L. Rodríguez-Mazahua, C.-A. Rodríguez-Enríquez, J. L. Sánchez-Cervantes, J. Cervantes, J. L. García-Alcaraz, and G. Alor-Hernández, "A general perspective of big data: Applications, tools, challenges and trends," *J. Supercomput.*, vol. 72, no. 8, pp. 3073–3113, Aug. 2016.

[11] Y. Zhang, T. Cao, S. Li, X. Tian, L. Yuan, H. Jia, and A. V. Vasilakos, "Parallel processing systems for big data: A survey," *Proc. IEEE*, vol. 104, no. 11, pp. 2114–2136, Nov. 2016.

[12] W. Inoubli, S. Aridhi, H. Mezni, M. Maddouri, and E. Mephu Nguifo, "An experimental survey on big data frameworks," *Future Gener. Comput. Syst.*, vol. 86, pp. 546–564, Sep. 2018.

[13] S. Mustafa, I. Elghandour, and M. A. Ismail, "A machine learning approach for predicting execution time of spark jobs," *Alexandria Eng. J.*, vol. 57, no. 4, pp. 3767–3778, Dec. 2018.

[14] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics," in *Proc. 13th USENIX Symp. Netw. Syst. Design Implement.*, 2016, pp. 363–378.

[15] K. Wang and M. M. H. Khan, "Performance prediction for apache spark platform," in *Proc. IEEE IEEE 17th Int. Conf. High Perform. Comput. Commun. 7th Int. Symp. Cyberspace Saf. Secur., IEEE 12th Int. Conf. Embedded Softw. Syst.*, Aug. 2015, pp. 166–173.

[16] L. Tsai, H. Franke, C.-S. Li, and W. Liao, "Learning-based memory allocation optimization for delay-sensitive big data processing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 6, pp. 1332–1341, Jun. 2018.

[17] R. Myung and H. Yu, "Performance prediction for convolutional neural network on spark cluster," *Electronics*, vol. 9, no. 9, p. 1340, Aug. 2020.

[18] Y. Y. Li, T. V. Do, and H. T. Nguyen, "A comparison of forecasting models for the resource usage of MapReduce applications," *Neurocomputing*, vol. 418, pp. 36–55, Dec. 2020.

[19] R. Dautov and S. Distefano, "Quantifying volume, velocity, and variety to support (Big) data-intensive application development," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2017, pp. 2843–2852.

[20] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang, "In-memory big data management and processing: A survey," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 7, pp. 1920–1948, Jul. 2015.

[21] C. Chen, K. Li, A. Ouyang, Z. Zeng, and K. Li, "GFlink: An in-memory computing architecture on heterogeneous CPU-GPU clusters for big data," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 6, pp. 1275–1288, Jun. 2018.

[22] K. Nguyen, K. Wang, Y. Bu, L. Fang, J. Hu, and G. Xu, "Facade: A compiler and runtime for (almost) object-bounded big data applications," in *Proc. 20th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, New York, NY, USA, 2015, pp. 675–690.

[23] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August, "Compiler optimization-space exploration," in *Proc. Int. Symp. Code Gener. Optim.* Washington, DC, USA: IEEE Computer Society, Mar. 2003, pp. 204–215.

[24] B. Schiller, C. Deusser, J. Castrillon, and T. Strufe, "Compile- and runtime approaches for the selection of efficient data structures for dynamic graph analysis," *Appl. Netw. Sci.*, vol. 1, no. 1, p. 9, Dec. 2016.

[25] S. Sakr, "Big data processing stacks," *IT Prof.*, vol. 19, no. 1, pp. 34–41, Jan. 2017.

[26] L. Belcastro, F. Marozzo, and D. Talia, "Programming models and systems for big data analysis," *Int. J. Parallel, Emergent Distrib. Syst.*, vol. 34, no. 6, pp. 632–652, 2018.

[27] Y. Chan, A. Wellings, I. Gray, and N. Audsley, "A distributed stream library for Java 8," *IEEE Trans. Big Data*, vol. 3, no. 3, pp. 262–275, Sep. 2017.

[28] Z. Wang and M. O'Boyle, "Machine learning in compiler optimization," *Proc. IEEE*, vol. 106, no. 11, pp. 1879–1901, Nov. 2018.

[29] P. Thoman, P. Zangerl, and T. Fahringer, "Static compiler analyses for application-specific optimization of task-parallel runtime systems," *J. Signal Process. Syst.*, vol. 91, nos. 3–4, pp. 303–320, Mar. 2019.

[30] G. Mariani, A. Anghel, R. Jongerius, and G. Dittmann, "Scaling properties of parallel applications to exascale," *Int. J. Parallel Program.*, vol. 44, no. 5, pp. 975–1002, Oct. 2016.

[31] D. Tetzlaff and S. Glesner, "Intelligent prediction of execution times," in *Proc. 2nd Int. Conf. Informat. Appl. (ICIA)*, Sep. 2013, pp. 234–239.
[32] A. Anghel, L. M. Vasilescu, G. Mariani, R. Jongerius, and G. Dittmann, "An instrumentation approach for hardware-agnostic software characterization," *Int. J. Parallel Program.*, vol. 44, no. 5, pp. 924–948, Oct. 2016.
[33] G. Mariani, A. Anghel, R. Jongerius, and G. Dittmann, "Predicting cloud performance for HPC applications before deployment," *Future Gener. Comput. Syst.*, vol. 87, pp. 618–628, Oct. 2018.
[34] T. Rubiano, "Implicit computational complexity and compilers," Ph.D. dissertation, Fac. Sci., Dept. Comput. Sci., Univ. Copenhagen, Copenhagen, Denmark, 2017.
[35] M. Kruse, "Perfrewrite–program complexity analysis via source code instrumentation," 2014, *arXiv:1409.2089*. [Online]. Available: http://arxiv.org/abs/1409.2089
[36] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Oct. 2009, pp. 44–54.
[37] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin, "GraphBIG: Understanding graph computing in the context of industrial solutions," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2015, pp. 1–12.
[38] R. M. Yoo, A. Romano, and C. Kozyrakis, "Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Oct. 2009, pp. 198–207.
[39] *Grep-Bench*, SEEK, Melbourne, VIC, Australia, 2019.
[40] S. Thomas, C. Gohkale, E. Tanuwidjaja, T. Chong, D. Lau, S. Garcia, and M. B. Taylor, "CortexSuite: A synthetic brain benchmark suite," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Oct. 2014, pp. 76–79.
[41] *C Neural Network Library: Genann*, CodePlea, IA, USA, 2019.
[42] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proc. 17th Int. Conf. Parallel Archit. Compilation Techn. (PACT)*, 2008, pp. 72–81.
[43] A. H. Ashouri, G. Palermo, J. Cavazos, and C. Silvano, *Automatic Tuning of Compilers Using Machine Learning*. Cham, Switzerland: Springer, 2018.
[44] M. Awad and R. Khanna, *Efficient Learning Machines: Theories, Concepts, and applications for Engineers and System Designers*. New York, NY, USA: Apress, 2015.
[45] A. Gandomi and M. Haider, "Beyond the hype: Big data concepts, methods, and analytics," *Int. J. Inf. Manage.*, vol. 35, no. 2, pp. 137–144, Apr. 2015.
[46] H. V. Jagadish, "Big data and science: Myths and reality," *Big Data Res.*, vol. 2, no. 2, pp. 49–52, Jun. 2015.
[47] I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, and S. U. Khan, "The rise of 'big data' on cloud computing: Review and open research issues," *Inf. Syst.*, vol. 47, pp. 98–115, Jan. 2015.
[48] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. Mendell, H. Nasgaard, S. Schneider, R. Soulé, and K. L. Wu, "IBM streams processing language: Analyzing big data in motion," *IBM J. Res. Develop.*, vol. 57, nos. 3–4, pp. 1–7, 2013.
[49] V. Kiriansky, Y. Zhang, and S. Amarasinghe, "Optimizing indirect memory references with milk," in *Proc. Int. Conf. Parallel Archit. Compilation*, Sep. 2016, pp. 299–312.
[50] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, and A. Ghodsi, "Apache spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, 2016.
[51] J. Dittrich and J.-A. Quiané-Ruiz, "Efficient big data processing in Hadoop MapReduce," *Proc. VLDB Endowment*, vol. 5, no. 12, pp. 2014–2015, Aug. 2012.
[52] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, "BigDataBench: A big data benchmark suite from Internet services," in *Proc. IEEE 20th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2014, pp. 488–499.
[53] W. R. Stevens and S. A. Rago, *Advanced Programming in the UNIX Environment*. Reading, MA, USA: Addison-Wesley, 2008.
[54] G. A. Seber and A. J. Lee, *Linear Regression Analysis*, vol. 329. Hoboken, NJ, USA: Wiley, 2012.
[55] V. Cherkassky and Y. Ma, "Selection of meta-parameters for support vector regression," in *Proc. Int. Conf. Artif. Neural Netw.* Berlin, Germany: Springer-Verlag, 2002, pp. 687–693.
[56] P. Basanta-Val, N. C. Audsley, A. J. Wellings, I. Gray, and N. Fernandez-Garcia, "Architecting time-critical big-data systems," *IEEE Trans. Big Data*, vol. 2, no. 4, pp. 310–324, Dec. 2016.
[57] S. M. Kuo, B. H. Lee, and W. Tian, *Introduction to Real-Time Digital Signal Processing*. Hoboken, NJ, USA: Wiley, 2006, ch. 1, pp. 1–47.
[58] M. Strohbach, H. Ziekow, V. Gazis, and N. Akiva, "Towards a big data analytics framework for IoT and smart city applications," in *Modeling and Processing for Next-Generation Big-Data Technologies*. Cham, Switzerland: Springer, 2015, pp. 257–282.
[59] W. Mayer, G. Grossmann, M. Selway, J. Stanek, and M. Stumptner, "Variety management for big data," in *Semantic Applications*. Berlin, Germany: Springer Vieweg, 2018, pp. 47–62.
[60] W. Gao, J. Zhan, L. Wang, C. Luo, D. Zheng, F. Tang, B. Xie, C. Zheng, X. Wen, X. He, H. Ye, and R. Ren, "Data motifs: A lens towards fully understanding big data and AI workloads," in *Proc. 27th Int. Conf. Parallel Archit. Compilation Techn.*, Nov. 2018 p. 2.
[61] A. Tumeo and J. Feo, "Irregular applications: From architectures to algorithms [guest editors' introduction]," *Computer*, vol. 48, no. 8, pp. 14–16, Aug. 2015.
[62] S. Chen, W. Huang, M. Chen, J. Zhong, and J. Cheng, "Airlines content recommendations based on passengers' choice using Bayesian belief networks," in *Bayesian Inference*. Rijeka, Croatia: InTech, 2017.
[63] S. Sidhanta, W. Golab, and S. Mukhopadhyay, "Deadline-aware cost optimization for spark," *IEEE Trans. Big Data*, early access, Mar. 29, 2019, doi: 10.1109/TBDATA.2019.2908188.

HAMEEZA AHMED received the B.E. and M.Eng. degrees in computer and information systems from the NED University of Engineering and Technology, Pakistan, in 2012 and 2015, respectively, where she is currently pursuing the Ph.D. degree. Her research interests include big data computing, compiler optimizations, and computer architecture.

MUHAMMAD ALI ISMAIL (Member, IEEE) received the Ph.D. degree in high-performance computing, in 2011. He is currently a Professor and the Chair of the Department of Computer and Information Systems Engineering, NED University of Engineering and Technology, where he is also serving as the Director of the High-Performance Computing Center and the Scientific Director of the Exascale Open Data Analytics Laboratory, National Center in Big Data and Cloud Computing. He has more than 16 years' experience of research, teaching, and administration in both national and international universities. Afterwards, he pursued his Postdoctoral Researcher in automatic design space exploration from ULBS Romania and become a HiPEAC member. He has published more than 65 scientific papers in international journals and conferences along with U.S. patent. His current research interests include computational HPC, big data mining, cluster and cloud computing, multicore processor architecture and programming, machine learning, heuristics, and automatic design space exploration. He is a member of IET. He has won many of the national and international grants of worth above Rs. 200 Million. He was also a recipient of the Research Productivity Award by Pakistan Council for Science and Technology, Ministry of Science and Technology, Government of Pakistan. He is also serving as the Vice Chairman of IET Karachi Network.