J Supercomput (2016) 72:4337–4364
DOI 10.1007/s11227-016-1742-7

ShenZhen transportation system (SZTS): a novel big data benchmark suite

Wen Xiong[1,2] · Zhibin Yu[1] · Lieven Eeckhout[3] · Zhengdong Bei[1] · Fan Zhang[1] · Chengzhong Xu[1,4]

Published online: 4 June 2016

Abstract Data analytics is at the core of the supply chain for both products and services in modern economies and societies. Big data workloads, however, are placing unprecedented demands on computing technologies, calling for a deep understanding and characterization of these emerging workloads. In this paper, we propose ShenZhen Transportation System (SZTS), a novel big data Hadoop benchmark suite comprised of real-life transportation analysis applications with real-life input data sets from Shenzhen in China. SZTS uniquely focuses on a specific and real-life application domain whereas other existing Hadoop benchmark suites, such as HiBench and CloudRank-D, consist of generic algorithms with synthetic inputs. We perform a cross-layer workload characterization at the microarchitecture level, the operating system (OS) level, and the job level, revealing unique characteristics of SZTS compared to existing Hadoop benchmarks as well as general-purpose multi-core PARSEC benchmarks. We also study the sensitivity of workload behavior with respect to input data size, and we propose a methodology for identifying representative input data sets.

This is an extended version of 'SZTS: A Novel Big Data Transportation System Benchmark Suite' by the same authors, published at the 44th Annual International Conference on Parallel Processing (ICPP), September 2015, Beijing, China. This extension includes a general revision, provides more details and analysis, and includes an OS-level characterization, as a complement to the microarchitectural and job-level characterization.

✉ Zhibin Yu
zbyupro@163.com; zb.yu@siat.ac.cn

1 Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Shenzhen, China

2 Shenzhen College of Advanced Technology, University of Chinese Academy of Sciences, Shenzhen, China

3 Ghent University, Ghent, Belgium

4 Wayne State University, Detroit, USA

Keywords Big data · Benchmarking · Performance measurement · MapReduce/hadoop · ShenZhen transportation system (SZTS)

1 Introduction

Information technology is undergoing a data-centric revolution in which companies and governments use analytics on massive data to extract information and discover trends. In fact, data are keys to many products and services in modern economies and governmental policies. Projections, however, show that data are growing by an order of magnitude every year, surpassing improvements in computing devices. Big data analytics is thus placing unprecedented demands on our computer systems. Hence, it is of utmost importance to analyze and understand these big data workloads to drive future computer system research and development. Example big data workloads span different areas of interest in government policy (e.g., city transportation, health care), IT (e.g., social networking, Web 2.0), industry (e.g., manufacturing), and research (e.g., data-intensive scientific discovery).

In this paper, we propose a novel benchmark suite, called ShenZhen Transportation System (SZTS), consisting of five applications that analyze different forms of transportation data in Shenzhen, an 18-million city in the south of China. The applications are written in Hadoop, a popular open-source implementation framework of the MapReduce programming model for big data processing. The SZTS workloads analyze how many people are moving from point A to B using the metro or taxi; the distribution of people and vehicles at a given time and location; identification and prediction of hot spots in the city; mapping GPS measurements to the city grid; and sorting GPS records for temporal correctness. The input data sets of the SZTS benchmark suite are taken from real-life operation; the number of data points are in the range of tens of millions of records per day. SZTS is unique by focusing on real workloads from a real application domain using real-life data, whereas other existing Hadoop benchmark suites are comprised of generic algorithms using synthetic inputs. SZTS is available for download at http://cloud.siat.ac.cn/cloud/szts/szts.php.

To better understand SZTS' unique properties, we perform a cross-layer workload characterization at the microarchitecture, operating system (OS), and job level; and we compare SZTS against other Hadoop benchmarks from HiBench [6] and CloudRank-D [7], as well as general-purpose multi-core benchmarks from PARSEC [9]. Our characterization reveals that SZTS exhibits vastly different behavior from typical general-purpose multi-core benchmarks (PARSEC), with much lower IPC, and much higher instruction cache miss rates and last-level (data) cache miss rates at the microarchitecture level. In addition, the SZTS benchmarks spend a much larger fraction of their execution time (90 %) in OS libraries. While more similar to the other Hadoop benchmarks than to PARSEC, we find SZTS' microarchitecture behavior to be quite unique relative to HiBench and CloudRank-D with fairly low instructions per cycle (IPC), good instruction cache performance (and thus relatively few fetch stalls), and high last-level cache miss rates and high off-chip memory pressure (resulting in a relatively high number of resource stalls). We also find that SZTS exhibits relatively high data processing speed; limited time spent in the *map* stage relative to the *reduce*

stage; fairly large amount of *reduce* output data; and substantial intra-node communication at the job level. Finally, we study the sensitivity to the input data set size, and we find several benchmarks to be very sensitive to their inputs; hence, we propose a methodology to identify representative input data sets.

In summary, we make the following contributions in this paper:

- We propose ShenZhen Transportation System (SZTS), a novel Hadoop benchmark suite composed of five benchmarks representative of the big data transportation system in Shenzhen, China, and which we make publicly available.
- We perform a cross-layer characterization (at the microarchitecture, OS, and job level) of SZTS, and compare its characteristics against other Hadoop benchmarks from the HiBench and CloudRank-D benchmark suites. We also compare against the general-purpose, multi-core PARSEC benchmark suite. We point out that Hadoop/MapReduce workload characterization should be done across layers. In particular, microarchitecture-level characterization reveals important properties that job-level characterization cannot reveal, and vice versa.
- We find that SZTS exhibits very different characteristics compared to PARSEC at the microarchitecture and OS levels. We also find SZTS to exhibit unique workload characteristics relative to HiBench and CloudRank-D, with relatively low IPC, good instruction cache performance, and high last-level cache and off-chip memory pressure, along with high intra-node communication, large reduce outputs, and large reduce processing times.
- We propose a methodology for identifying representative input data sets through input sensitivity analysis.

The rest of this paper is organized as follows. Section 2 describes the Hadoop framework we use in this paper. Section 3 introduces SZTS, our novel big data benchmark suite derived from the Shenzhen transportation system. Section 4 describes the methodology used to characterize the workloads at the job, OS and microarchitecture levels. Section 5 introduces our experimental setup. Section 6 presents our characterization results and analysis. Section 7 discusses input sensitivity and representative input selection. Section 8 discusses related work, and finally, Sect. 9 concludes the paper.

2 Background: big data and MapReduce

Big data applications are often characterized using the four Vs: volume, velocity, variety and veracity. Volume refers to the scale of the data; velocity refers to the analysis of streaming data; variety refers to the different forms of data; and veracity refers to the uncertainty of data. The huge scale of the data, its uncertainty and variety makes big data analytics extremely challenging; the fact that the data are continuously streaming in, as is the case for our application domain (city transportation system), makes big data analytics a grand challenge. Benchmarking big data applications and systems hence comes with its own challenges: the huge volumes of (streaming) data in real-life applications require that sufficiently large input sizes are considered during performance evaluation, as we will see in this paper. In fact, to the best of our knowledge, how

input data set size affects Hadoop application behavior and performance has not been systematically studied before.

Big data applications call for domain-specific languages. MapReduce is a programming model designed for big data processing; a MapReduce program takes a set of key/value pairs as input and produces a set of output key/value pairs. Hadoop is the most popular open-source implementation framework for MapReduce. Users only need to write *map* and *reduce* functions and the rest is handled by the Hadoop runtime. Large input data sets are split into small blocks by the Hadoop Distributed File System (HDFS). The execution of a Hadoop program can be divided into *map* and *reduce* stages. In the *map* stage, each *map* task reads a small block and processes it. When *map* tasks complete, the outputs—also known as intermediate files—are copied to the *reduce* nodes. At the *reduce* stage, *reduce* tasks fetch the key/value pairs from the output files of the *map* stage, which they then sort, merge, and process, to produce the final output. The *map* stage itself can be further divided into *read*, *map*, *collect*, *spill*, and *merge* phases. Similarly, the *reduce* stage can be divided into *shuffle*, *merge*, *reduce*, and *write* phases.

3 ShenZhen transportation system

The SZTS benchmarks are derived from the Smart Urban Transportation System of Shenzhen (SUTS). Before we dig into the details of SUTS, it is worth briefly introducing the Shenzhen city. Shenzhen is a famous international city located in southern China. It covers 2000 square kilometers with a population of approximately 18 million. Shenzhen already built a modern urban transportation infrastructure including 5 subway lines with 118 stations, 936 bus lines, and 30,000 cabs. Each bus and cab is equipped with a GPS device. As Shenzhen hosts a lot of Internet and software companies, it attracts more and more people to work in the city. In fact, Shenzhen was founded in 1979—only 35 years ago—and has grown to become one of the most successful special economic zones in China. As a result, the city transportation faces severe challenges. For example, daily traffic jams during rush hour now take 1.5 h longer than 2 years ago!

SUTS was built with two goals in mind. For one, it provides the government with a tool to develop transportation policies and plan for future infrastructure. Second, it helps citizens organize their daily intra-city trips. To achieve these goals, data generated by all vehicles and passengers must be stored and used as an analysis base for various transportation policies and smart applications. SUTS, therefore, developed a data warehouse to store the data. Currently, there are four types of data: (1) Cellular phone GPS records with fields such as *phoneid*, *owner-name*, *time*, *location*, and *speaking-time*; (2) GPS records with fields such as *cab-id*, *time-stamp*, *latitude*, *longitude*, *speed* and *direction*; (3) smart card transaction records with fields such as *smart-card-id*, *entrance- or exit-flag*, *station-id* and *transaction-account*; (4) cab deal information with fields such as *time-stamp*, *transaction-account*, *distance* and *start-time*. These data are generated by GPS devices (in buses, taxis, and cellular phones) and by machines in metro stations, and is sent to the data warehouse of SUTS at a rate of approximately once every 20 seconds. Table 1 summarizes the amount of data in SUTS.

Table 1 The amount of data in SUTS

Index	Item	Value
1	No. vertices of digital map	73,515
2	No. edges of digital map	101,749
3	No. smart cards	11 million
4	No. GPS records	95 million per day
5	No. smart card records	15 million per day
6	No. taxicab deal records	1.5 million per day

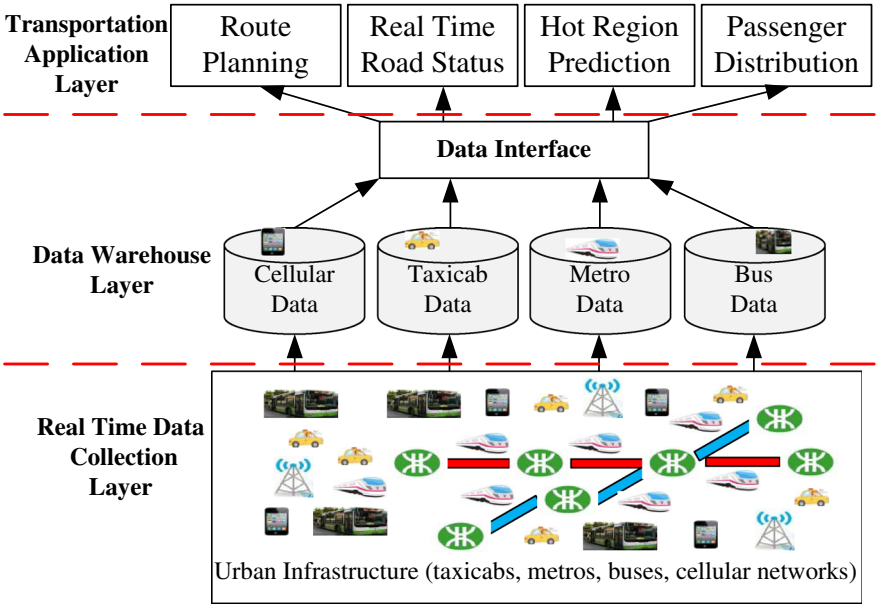

Fig. 1 Architecture overview of the Smart Urban Transportation System of Shenzhen (SUTS)

The three-layer architecture of SUTS is shown in Fig. 1. The bottom layer is the real-time data collection which receives the aforementioned data in an online manner. The middle layer is the data warehouse. A data interface is defined to support the top application layer. A unique feature of SUTS is that it uses a digital map as its user interface, visualizing the distribution or traffic jams of the city. All components of SUTS are implemented in the Hadoop framework and its derivative systems such as Hive and Pig, making up for a typical big data system.

3.1 SZTS benchmark suite

Given the SUTS system as just described, we have distilled five typical benchmarks, of which we now provide more details, and which we refer to as the SZTS benchmark suite.

Sztod The `sztod` program computes how many moving objects start from point A and head for point B in a given time interval in the city. `sztod` has two use-cases in SZTS: (1) Metro system: in this case, `sztod` computes how many people enter station A and exit station B; (2) Taxi system: in this case, `sztod` calculates how many cabs navigate from point A to point B.

Hotregion The `hotregion` program computes the distribution of people, cars, or other vehicles in Shenzhen within a given time interval. For example, `hotregion` can compute how many smart card transactions occur in all metro stations of the Shenzhen city on a special day such as the National Holiday. We can, therefore, easily identify the hot metro stations according to the number of smart card transactions. Another example is to use `hotregion` to compute the real-time taxi distribution in all traffic regions that are specified by their latitude and longitude coordinates on the Shenzhen digital map.

Mapmatching Typically, a GPS (Global Position System) trajectory consists of a sequence of points with latitude, longitude and time-stamp information. However, this information is not accurate due to the measurement error caused by the limitation of GPS devices and sampling error. The observed GPS points, therefore, need to be aligned with the route network on a given digital map [8]. The `mapmatching` program is used to match the observed GPS trajectory with the route on a digital map. Based on `mapmatching`, one can develop a lot of other applications such as identifying traffic hotspots and smart route planning.

Hotspot In SUTS, the hotspots are defined as (1) a shopping center, (2) a train station, (3) a prominent landmark, (4) the airport, and (5) customs. The `hotspot` program aims at acquiring, aggregating, analyzing, visualizing, and predicting traffic jams on the roads surrounding and leading to these hotspots. For instance, `hotspot` analyzes the intensity of traffic flow (free or congested) to identify the traffic phases of each hotspots. In addition, `hotspot` contains a prediction model based on logistic regression to predict the future traffic flow intensities for a given hotspot.

Secsort The original GPS records received from remote GPS devices or smart card readers in SUTS are out-of-order. However, further analyses may need sorted data. The `secsort` program is developed to sort the data according to two keys. What the two keys are depends on the application that uses `secsort`. For example, one smart card record has multiple fields such as time-stamp and metro station name. In a metro application, `secsort` first sorts the records by smart card record, then sorts by the primary time-stamp key, and finally sorts the records in each group by the secondary metro station name key. `secsort` is implemented in Apache Pig.

3.2 SZTS Input Data Sets

The input data sets of SUTS are considerable. The 30,000 cabs generate 4.8 GB of data each day, resulting in 144.8 GB and 1.74 TB data per month and per year, respectively.

The Shenzhen city has approximately 15,000 buses servicing 936 bus lines in between 10,300 bus stations, generating 5 GB data per day. The Shenzhen urban transportation company has sold more than 10 million smart cards, of which 7.3 million are used on a daily basis on average. The total amount of smart card transaction data is 1 GB per day.

The SZTS programs do not change frequently; however, their inputs change every 20 seconds in the most extreme case. Moreover, the amount of data that needs to be processed may vary widely: for example, officers of the Shenzhen government may want to see the daily, monthly, even yearly cab and passenger distributions of the city. This indicates that the same program needs to process a lot of different sizes of input data. However, to the best of our knowledge, no previous work has carefully studied how the input data size affects the behavior of Hadoop programs. We provide such a detailed analysis in this paper, and we provide the SZTS benchmark suite with different input sets for each benchmark.

4 Cross-layer characterization

We analyze the SZTS benchmark suite through a cross-layer characterization involving the job, OS and microarchitecture levels. As will become clear in the results section, a cross-layer characterization reveals insights that a single-layer characterization cannot reveal. Subsequent data analysis is done using principal component analysis (PCA) and cluster analysis.

4.1 Job-level characterization

The metrics we consider at the job level are enumerated in Table 2; these metrics are chosen such as to provide a fairly broad view on data size, processing speed and communication at the job level. *Data Processing Speed (DPS)* measures how fast a program processes data, and is defined as the amount of data processed divided by

Table 2 Job-level metrics

Index	Metric	Description
1	DPS	Data processing speed
2	MOI	(Map output)/(map input)
3	SMI	(Shuffle data)/(map input)
4	TMRS	(Map stage time)/(reduce stage time)
5	TMRF	(Map function time)/(reduce function time)
6	ROI	(Reduce output)/(reduce input)
7	ROMI	(Reduce output)/(map input)
8	TMI	(Temporal write data)/(map input)
9	TRAMI	(Intra-node transmitted data)/(map input)
10	TERMI	(Inter-node transmitted data)/(map input)

the execution time of a program. To quantify how the data size changes after the *map* operation, we define *Map Output/Input ratio (MOI)* as the ratio of the amount of *map* output and input data; *Shuffle/Map Input ratio (SMI)* is defined as the ratio of the amount of data processed by the *shuffle* operation to that processed by the *map* operation. Two metrics—*Time Map/Reduce Stage ratio (TMRS)* and *Time Map/Reduce Function ratio (TMRF)*—quantify the amount of time spent in the *map* stage and function relative to the *reduce* stage and function, respectively. (Note that the *map* stage takes longer than the *map* function since the *map* stage includes more operations such as *spill* and *merge*, as previously mentioned.) Two metrics—*Reduce Output/Input ratio (ROI)* and *Reduce Output to Map Input ratio (ROMI)*—relate to the amount of output data produced by the *reduce* stage. The *Temporal to Map Input ratio (TMI)* metric quantifies the amount of data temporally written to the local file system. Finally, two metrics—*Intra-node to Map Input (TRAMI)* and *Inter-node to Map Input (TERMI)*—quantify the amount of data transmitted between processes within the same node versus between different nodes.

Collectively, the job-level metrics primarily focus on data size changes between the map and reduce stages and functions, and the time required to process the data. HiBench [6] in its characterization, employs three job-level metrics only; we define more metrics to enable a more detailed analysis.

4.2 OS-level characterization

In addition to the job-level characteristics, we also characterize how much time is consumed in the OS and system libraries. This information helps us find where cycles go in the system stack. Hadoop is a framework implemented in Java, which leads to an overall complicated software stack, involving the application, the MapReduce runtime, the JVM and the OS. The complex system stack presents interesting challenges for correlating performance between kernel modules, shared libraries, the runtimes and application binaries.

Programs generally rely on a set of common system libraries such as the C runtime, a mathematical library, a dynamic linker/loader, and the POSIX thread library. However, unlike compute-intensive programs (such as those from the PARSEC benchmark suite), Hadoop benchmarks involve two additional types of libraries: network-related libraries and JVM-related libraries. Analyzing the time spent in these system libraries enables us identifying optimization opportunities. For example, the IBM `libjvm.so` system library leads to a 38 % performance improvement compared to the default version of `libjvm.so` for a set of Hadoop programs according to a study done by Yasin et al. [32]. Such a general approach is usually preferred over *ad hoc* approaches since the latter typically work for a small subset of Hadoop programs only.

The OS and system libraries that we characterize are shown in Table 3. While most libraries are widely known, a couple needs further explanation, such as `vdso` and `anon`. `vdso` is the abbreviation of Virtual Dynamically linked Shared Object and it exports kernel space routines to user space applications using standard mechanisms for linking and loading the standard ELF format. It also helps to reduce the calling overhead on simple kernel routines as well as to select the best system call method on

Table 3 OS and system libraries

OS and system library	Description	Category
vmlinux	Linux kernel	Linux
libc-2.15.so	C runtime library	Linux
libm-2.15.so	Mathematical library	Linux
ld-2.15.so	Dynamic linker/loader	Linux
libpthread-2.15.so	POSIX thread library	Linux
Libnio.so	Network interface	Java network lib
Libnet.so	Network lib	Java network lib
e1000e	Network	Network device driver
Libjava.so	Java library	JVM-related libraries
Libjvm.so	Java virtual machine	JVM-related libraries
Libhadoop.so	Hadoop libraries	JVM-related libraries
vdso	Virtual dynamically linked shared object	JVM-related libraries
Anon	Anonymous executable mappings	JVM-related libraries
Libverify.so	Verification module	Others
Librt-2.15.so	Real-time library	Others

Table 4 Microarchitecture-level metrics

Index	Metric	Description
1	IPC	Instructions per cycle
2	L1ICMPKI	No. L1 icache misses per 1K insns
3	L2ICMPKI	No. L2 icache misses per 1K insns
4	LLCMPKI	No. last-level cache misses per 1K insns
5	BRPKI	No. branches per 1K insns
6	BRMPKI	No. branch misses per 1K insns
7	OCBWUTI	Off-chip bandwidth utilization
8	IFSPKC	No. instruction fetch stall cycles per 1K cycles
9	RSPKC	No. resource related stall cycles per 1K cycles
10	DTLBPKI	No. D-TLB load misses per 1K insns

some architectures. `anon` represents anonymous executable mappings which are not backed by an ELF file, and they occur when programs involve dynamic compilation into machine code using a just-in-time (JIT) compiler.

4.3 Microarchitecture-level characterization

Table 4 shows the metrics we use to characterize our benchmarks at the microarchitecture level; these metrics, when put together, provide a good view on the performance and behavior of individual nodes. We collect these metrics on each node, and then com-

pute the average across all eight nodes (see Sect. 5 for experimental details). We will use these metrics not only to compare SZTS against other Hadoop benchmarks, but also to compare SZTS against (non-Hadoop) general-purpose multi-core benchmarks.

Instructions Per Cycle (IPC) computes the number of useful instructions executed per cycle, and is a measure for performance (higher is better). *L1ICMPKI* and *L2ICMPKI* quantify the number of L1 and L2 instruction cache misses, respectively, per thousand instructions, and are measures for the instruction footprint and locality (lower is better). *LLCMPKI* quantifies the number of last-level cache (LLC) misses per thousand instructions, and is a metric for the data footprint and locality (lower is better). We do not consider L1 and L2 data cache misses as most of these latencies can be hidden by out-of-order execution (as supported by the processors considered in this paper). *BRPKI* is the number of branches per 1000 instructions, and *BRMPKI* is the number of branch misses per 1000 instructions (lower is better). *OCBWUTI* is the off-chip bandwidth utilization per-core measured in bytes per second, and is calculated as follows:

$$(64 \times 2.4 \times 10^9) \times llcm/clkuh,$$

with 64 the number of bytes fetched, 2.4 GHz the frequency of the processor, *llcm* the number of LLC misses and *clkuh* the number of unhalted clock cycles. *IFSPKC* and *RSPKC* are defined as the number of stall cycles due to instruction fetch stalls and other resource stalls, respectively. *DTLBPKI* quantifies the number of D-TLB misses per thousand instructions.

4.4 Data analysis

The amount of data collected during this characterization is substantial: 10 job-level metrics, 15 OS-level metrics, and 10 microarchitecture-level metrics for all 133 benchmark-input pairs in the analysis, as we will discuss later in this paper, yielding 199,500 data points in total. (Note that we collect the data from eight server nodes, leading to a total of over 1.5 million data points.) To facilitate the analysis, we employ a well-established workload characterization methodology using principal component analysis (PCA) and cluster analysis [21].

4.4.1 Principal component analysis

PCA is a statistical data analysis technique that presents a different view on the measured data. It builds on the assumption that many variables, the 35 metrics (microarchitecture, OS and job level metrics) for each workload in our data set, are correlated and these metrics measure the same or similar properties of the benchmark-input pairs. PCA computes new variables, called principal components, which are linear combinations of the original variables. The resulting principal components are uncorrelated or independent.

PCA transforms p variable $X_1, X_2, \ldots X_p$ into p principal components (PCs) $Z_1, Z_2, \ldots, Z_p$ such that:

$$Z_i = \sum_{j=0}^{p} a_{ij} X_j.$$

This transformation has the property that $Var[Z_1] \geq Var[Z_2] \geq \cdots \geq Var[Z_p]$, which means that Z_1 contains the most information while Z_p contains the least. PCA enables reducing the dimensionality of the data set in a controlled way by removing PCs with the lowest variance.

In other words, the input to PCA is a raw data set comprised of behavioral metrics for each of the workloads; the output is a reduced data set with principal components for each of the workloads. The reduced dimensionality enables visualizing the workload space, and in addition, it guarantees that the dimensions are no longer correlated which enables an unbiased analysis. The dimensions (principal components) are linear functions of the original behavioral metrics, enabling a comprehensive interpretation.

4.4.2 Cluster analysis

Although PCA reduces dimensionality, visualizing a multi-dimensional space beyond 2D and 3D remains challenging. We, therefore, employ cluster analysis to visualize the high-dimensional workload space. Clustering analysis is a data analysis technique that groups n workloads based on the measurements of p variables (principal components in this paper). The final goal is to obtain a number of groups, so-called clusters, each containing workloads that have 'similar' behavior. There exist two commonly used types of clustering techniques, namely linkage clustering and K-means clustering. In this paper, we use linkage clustering as it enables easy visualization of the clustering output.

Linkage clustering starts with a matrix of distances between the n workloads (i.e., benchmark-input pairs). As a starting point for the algorithm, each benchmark-input pair is considered a group. In each iteration of the algorithm, the two groups that are the closest to each other (with the smallest distance in the p-dimensional space, also called the linkage distance) are combined to form a new group. As such, close groups are gradually merged until finally all cases belong to a single group. This can be represented in a so-called *dendrogram*, which graphically represents the linkage distance for each group merge at each iteration of the algorithm. Having obtained a dendrogram, it is up to the user to decide how many clusters to retain. This decision can be made based on the linkage distance, as small linkage distances imply strong clustering while large linkage distances imply weak clustering. We use linkage clustering to evaluate the impact of input data size on Hadoop program behavior by analyzing the behavior dis/similarity between workloads.

5 Experimental Setup

Before presenting our experimental results and the corresponding analysis, we first discuss our setup.

5.1 Hardware platform

We use a Hadoop cluster consisting of one master (control) node and eight slave (data) nodes. All the nodes are equipped with two 1 GB/s ethernet cards and are connected through two ethernet switches, as shown in Fig. 2. The A switch is used for global clock synchronization whereas the B switch is used to route Hadoop communication. Each node has three 2 TB disks, 16 GB memory, and two Intel Xeon E5620 multi-core (Westmere) processors. The detailed configuration of each processor is shown in Table 5. Each processor contains 8 cores, with each core being a 4-wide out-of-order superscalar architecture. The operating system running on each node is Ubuntu 12.04, kernel version 3.2.0. The versions of Hadoop and JDK are 1.0.3 and 1.7.0, respectively. The Hadoop security model is disabled, as by default.

We note that our experimental setup is limited in size, and much smaller compared to typical big data systems. Our experimental results confirm that the master node is

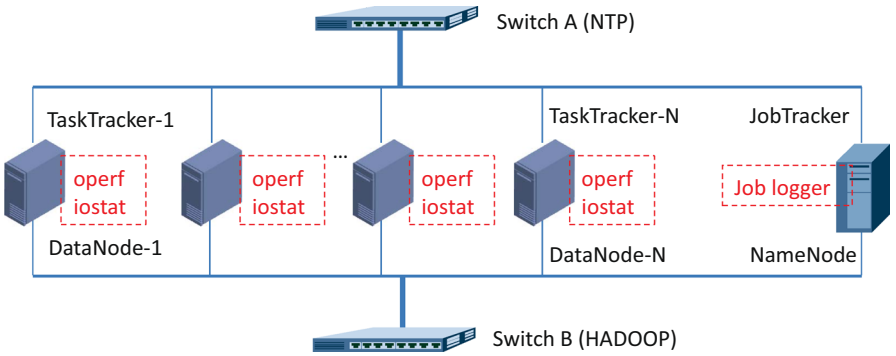

Fig. 2 Our experimental cluster consists of one master node and eight slave nodes. The nodes are connected using two switches. Switch B is used to route Hadoop computation, whereas switch A is used for global clock synchronization

Table 5 Processor configuration

Index	Item	Value
1	CPU type	Intel Xeon E5620
2	# cores	8 cores, 2.4 GHz
3	# threads	16 threads
4	# sockets	2
5	I-TLB	4-way, 64 entries
6	D-TLB	4-way, 64 entries
7	L2 TLB	4-way, 512 entries
8	L1 DCache	32 KB, 8-way, 64 bytes/line
9	L1 ICache	32 KB, 4-way, 64 bytes/line
10	L2 Cache	256 KB, 8-way, 64 bytes/line
11	L3 Cache	12 MB, 16-way, 64 bytes/line

not the bottleneck in the system. In larger systems, one may need to consider several master nodes. The proposed workload characterization can still be applied to larger scale systems with multiple master nodes.

5.2 Measurement tools

To observe the microarchitecture-level characteristics, we employ `oprofile` v0.98 to read the hardware performance counters of the processors via sampling. To collect the job-level events, we use `sysstat` v9.06. In addition, we read the log file of the Hadoop framework every 5 s to observe the Hadoop job features. Finally, we also read */proc/net/dev* every 5 s to obtain the runtime characteristics of the network.

5.3 Benchmarks

The benchmarks considered in this paper are summarized in Table 6. Next to the SZTS benchmarks discussed in Sect. 3, we also consider benchmarks from HiBench [6] and Cloudrank-D 1.0 [7]. The inputs for the latter benchmark suites are synthetically generated. The SZTS benchmarks on the other hand come with real-life inputs. Note that a Hadoop program may consist of multiple jobs. Each job typically consists of a *map* task and a *reduce* task. From this point of view, a Hadoop job is actually a small Hadoop program. Hence, we treat different jobs of the same program as different benchmarks. The *no. jobs* column in Table 6 shows the number of jobs of the corresponding program.

Table 6 Benchmarks and their input data sets

Benchmark	Suite	No. jobs	Input data set (GB)
Terasort	HiBench	1	100, 400,…, 1000
Sort	HiBench	1	20, 40,…, 400
Wordcount	HiBench	1	20, 40,…400, 500
Kmeans	HiBench	2	26, 50, 80, 107, 201
Pagerank	HiBench	2	12, 24, 53, 80, 109
Hive-aggre	HiBench	1	23, 61, 83, 100, 140
Hive-join	HiBench	3	23, 61, 83, 100, 140
Grep	CloudRank-D	1	50, 100, 160, 200, 300
hmm	CloudRank-D	1	50, 100, 160, 200, 300
Nativebayes	CloudRank-D	4	10, 30, 50, 80, 100
Sztod	SZTS	1	20, 50, 100, 160, 200
Hotregion	SZTS	1	50, 100, 160, 200, 300
Mapmatching	SZTS	1	2, 4, 8, 12, 16
Hotspot	SZTS	2	50, 100, 160, 200, 300
Secsort	SZTS	2	50, 100, 160, 200, 300

6 Results and analysis

We now characterize SZTS at the microarchitecture, OS, and job levels, followed by a cross-layer characterization. We provide an input data set sensitivity analysis and representative input selection methodology in the next section.

6.1 Microarchitecture-level characterization

IPC We start off the microarchitecture-level characterization by quantifying IPC in Fig. 3. The average per-core IPC for the Hadoop benchmarks equals 0.62, which is substantially lower than for PARSEC with an average IPC of 0.98. We notice that the program with the lowest IPC among all benchmarks is `mapmatching` from the SZTS benchmark suite. In fact, except for `hotspot-s2`, the SZTS benchmarks seem to have a consistently low IPC, compared to the other Hadoop benchmarks. We explore the reason next.

Cache performance Figure 4 quantifies the L1 and L2 instruction cache misses per 1K instructions of the Hadoop programs and PARSEC benchmarks. The I-cache miss rate appears to be much higher for the Hadoop benchmarks compared to PARSEC, indicating a large code footprint and poor instruction locality. Figure 5 shows the LCC misses per 1K instructions. The Hadoop benchmarks have a generally high LLC MPKI compared to PARSEC, with SZTS's `mapmatching` having the highest LLC MPKI among all the Hadoop programs, indicating a bigger data footprint and limited data locality.

PCA analysis Having established the major difference between the Hadoop benchmarks and PARSEC, we now focus on the Hadoop benchmarks. As noted in Figs. 3 through 5, it is hard to clearly discern the key differences between SZTS and the other Hadoop benchmarks. We, therefore, need a more advanced data analysis and workload characterization methodology, namely PCA. Applying PCA to the microarchitecture-level characteristics, we retain six principal components (PCs), accounting for 88.9 % of the total variance. The first PC explains 32.7 % of total variance, the second 18.7 %, the third 12.2 %, the fourth 11.9 %, the fifth 7.5 %, and the sixth 5.8 %.

In Fig. 6, the factor loadings are presented for the first four principal components. The first principal component is positively dominated by the number of last-level cache misses and the number branches per thousand instructions, and is negatively dominated by IPC, the number of L1 instruction cache misses, fetch stalls and DTLB misses. The second component is positively dominated by the number of resource stalls and off-chip bandwidth utilization, and negatively dominated by the number of L1/L2 instruction cache misses and instruction fetch stalls. This implies that a workload that has a high value along the first component exhibits a relatively large last-level cache miss rate and high branch ratio, but low IPC, relatively few DTLB misses, and relatively few fetch stall cycles and L1 instruction cache misses. A similar interpretation can be given to the second (and other) principal components.

Figure 7 visualizes how different/similar these Hadoop benchmarks are with respect to each other: it shows all the Hadoop workloads as a function of the first (horizontal

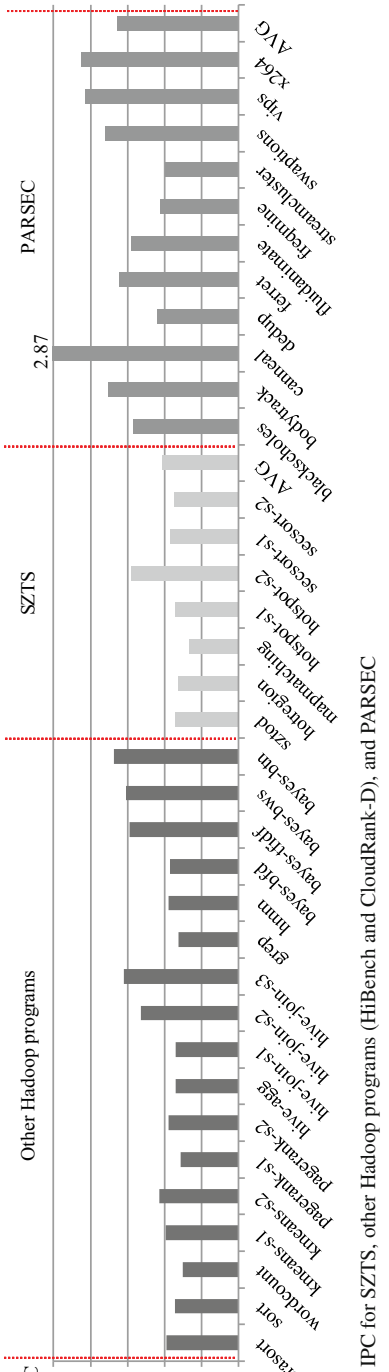

Fig. 3 IPC for SZTS, other Hadoop programs (HiBench and CloudRank-D), and PARSEC

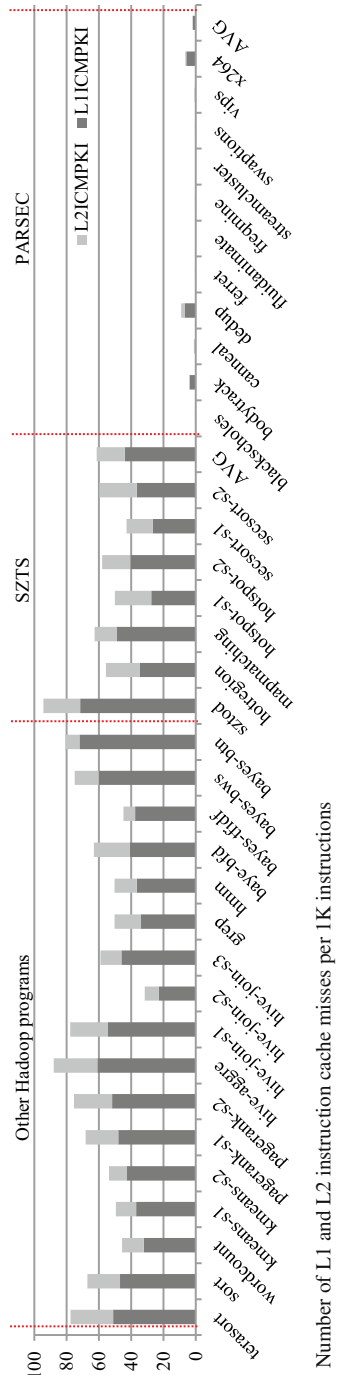

Fig. 4 Number of L1 and L2 instruction cache misses per 1K instructions

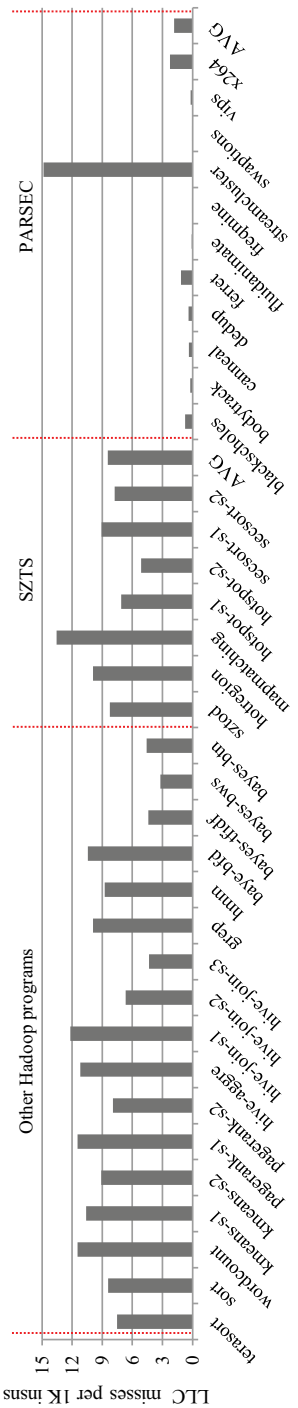

Fig. 5 Number of last-level cache misses per 1K instructions

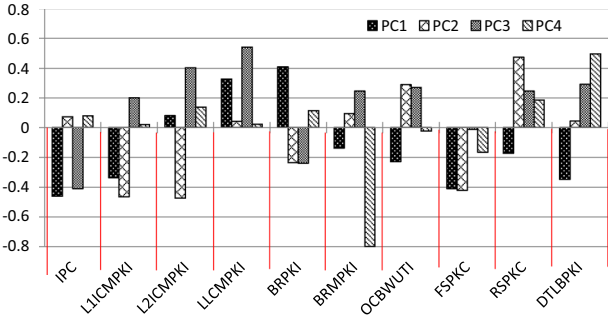

Fig. 6 PCA factor loadings using microarchitecture-level characteristics

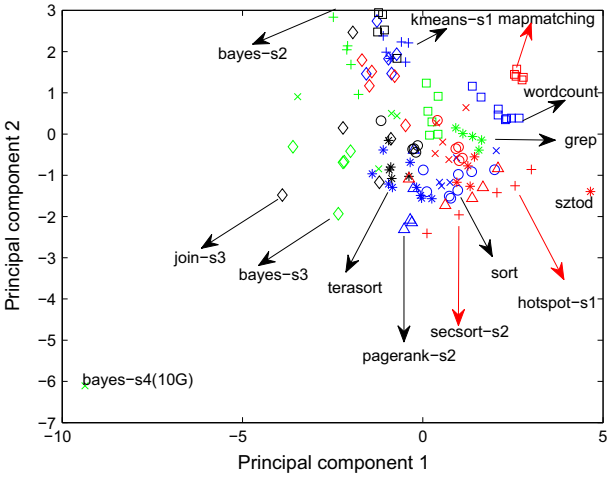

Fig. 7 Workload space (first two principal components) based on the microarchitecture-level characteristics

axis) and second (vertical axis) principal components. The red and green symbols represent the SZTS and CloudRank-D benchmarks, respectively, while the black and blue symbols represent the HiBench benchmarks. The black symbols are written in Hive, differentiating from the other HiBench benchmarks. The different data points per benchmark represent different input data set sizes. There are a number of interesting observations to be made here. For one, we observe that, although SZTS exhibits somewhat similar behavior compared to HiBench and CloudRank-D, we find the SZTS benchmarks to be located in the upper right corner of the workload space, i.e., high values for the two most significant principal components. This suggests that the SZTS benchmarks have a relatively low IPC, fairly good instruction cache performance (with relatively few fetch stalls as a result), and relatively high last-level cache miss rate and off-chip memory bandwidth pressure, which results in a fairly high number of resource stalls. In other words, SZTS is more data intensive with a relatively small code footprint compared to the other Hadoop benchmarks. Second, one of the SZTS benchmarks, namely `mapmatching`, is an outlier workload compared to all the other benchmarks considered here, primarily because of its poor last-level cache behavior

and high off-chip memory bandwidth pressure. Third, we also observe that some benchmarks are quite sensitive to their inputs, while others are not. For example, all of `mapmatching`'s inputs lead to similar behavior—all data points are located close to each other in the workload space. In contrast, the input has a significant impact on the benchmark's behavior, see for example `sztod`—a small input (e.g., `20GB`) leads to isolated behavior in the workload space.

6.2 OS-level characterization

Figure 8 shows the time breakdown for SZTS, the other Hadoop programs and PARSEC, for the OS and system libraries versus user code. The most prevalent OS and system libraries are shown only, accounting for more than 90 % of the total execution time for all benchmarks. Hadoop benchmarks spend more than 90 % of their time in OS and system libraries. More specifically, 37.8, 25.4, and 32.4 % of the time is spent in `libjvm.so`, `anon`, and `vmlinux`, respectively. In contrast, the PARSEC benchmarks spend most of their time in user-level code (denoted as `application` in Fig. 8), not OS and system libraries. PARSEC benchmarks employ `pthreads` and `OpenMP`; the creation, cleanup, synchronization, and switching is done (almost) entirely in user space without system calls [33]. The Hadoop benchmarks on the other hand spend most of their execution time in JVM-related libraries (`libjvm.so` and `anon`) and the OS (`vmlinux`). This is due to the fact that the Hadoop workloads run on top of Sun's JVM which maps Java threads to Light-Weight Processes (LWP) [31]. Moreover, the Hadoop workloads send and receive network messages, and read and write to file—tasks serviced by the operating system.

Apart from the distinction in OS/system-level code versus user-level code, there is another distinction to be made between the Hadoop workloads and the PARSEC benchmarks. Figure 9 shows that Hadoop programs uniformly spend between 1 and 2 % of their total execution time in `libpthread-2.15.so` and `libc-2.15.so`. In contrast, PARSEC benchmarks use these libraries to varying degrees. For example, the time spent in `libc-2.15.so` amounts to 18.5 % of the total execution time for `blackscholes`, whereas time spent in `libpthread-2.15.so` only accounts for 0.03 % of the time, as shown in Fig. 9.

Figure 10 illustrates the fraction of cycles per 1K cycles consumed in the network-related libraries `libnet.so`, `libnio.so`, and `e1000e` for the Hadoop benchmarks. The PARSEC benchmarks spend less than 0.4 % of their time in network-related libraries. While some Hadoop programs such as `wordcount`, `kmeans`, `hmm` and `grep`, also do not spend a large fraction of their execution time in network-related libraries (because most communication is done intra-node), other Hadoop benchmarks spend up to 10 % of their time in networking.

6.3 Job-level characterization

We now turn to job-level analysis. Applying PCA to the job-level characteristics yields five PCs, accounting for 90 % of the overall variance. (The first PC accounts for 45.8 % of the variance; the second PC accounts for 16.5 % of the variance.) Figure 11 shows

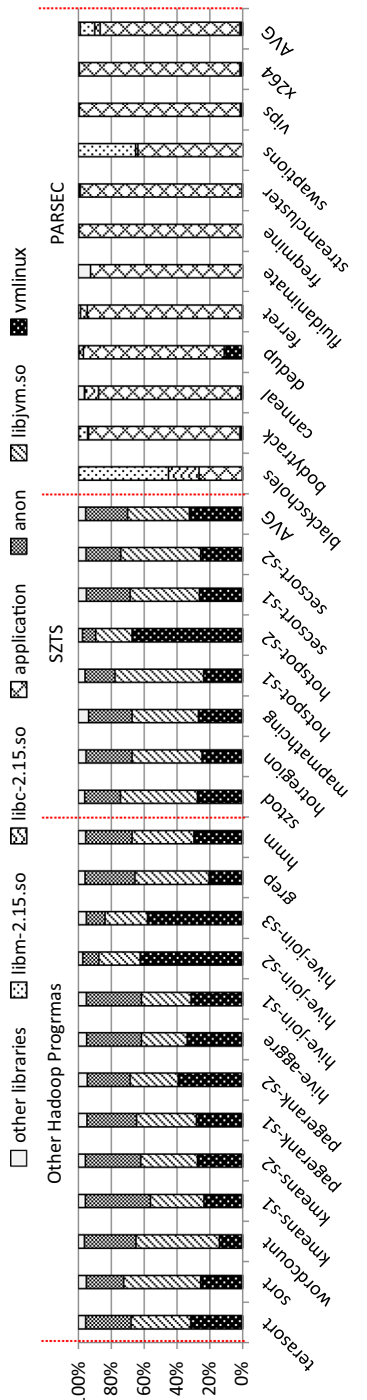

Fig. 8 Cycle breakdown in system (OS libraries) versus user-level code

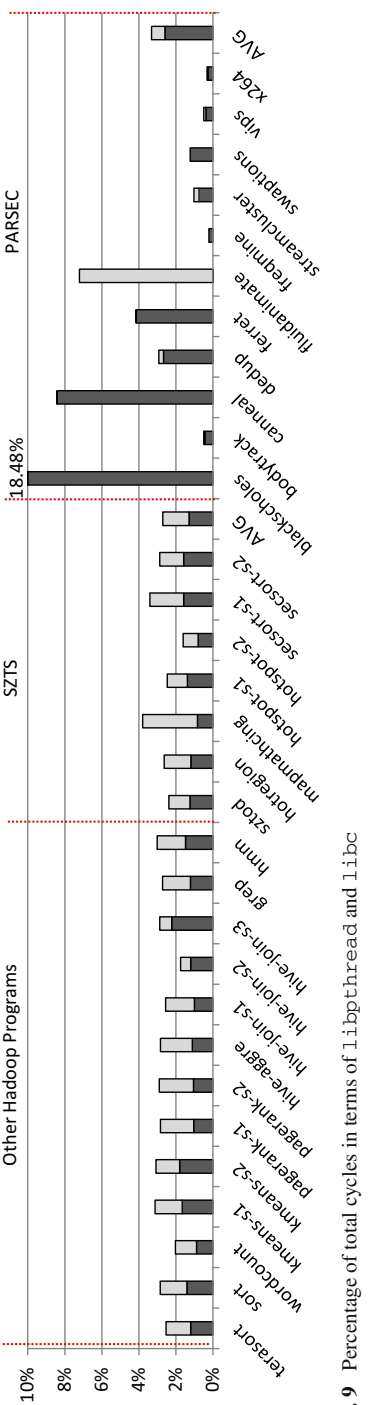

Fig. 9 Percentage of total cycles in terms of `libpthread` and `libc`

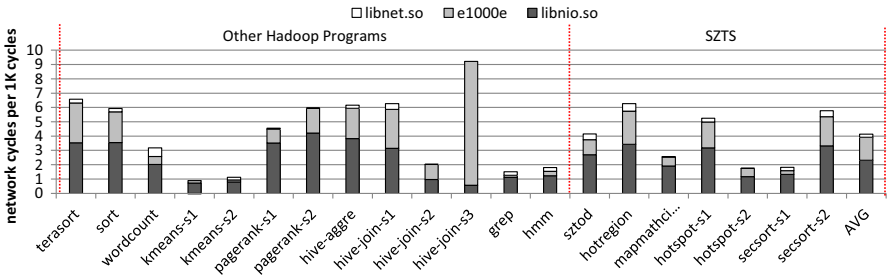

Fig. 10 Network-related cycles per 1K cycles

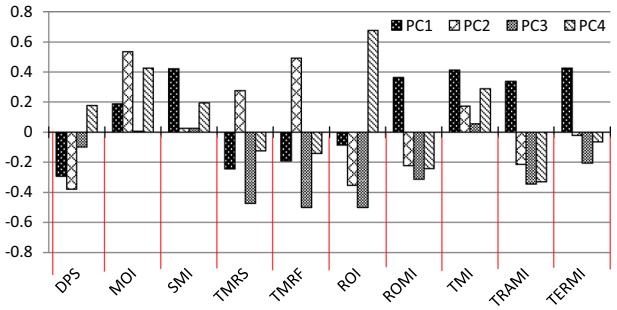

Fig. 11 PCA factor loadings using job-level characteristics

the factor loadings of the first four principal components. PC1 is positively dominated by SMI, TMI, TERMI and TRAMI, and is negatively dominated by DPS, TMRS and TMRF. This implies that if a workload has a higher value along the first principal component, the workload shuffles more data; more temporal data are generated; and more data are transmitted within a node and between nodes. On the other hand, data processing speed tends to be lower; more time is spent in the *reduce* stage/function versus the *map* stage/function.

Figure 12 visualizes the workload space considering the job-level characteristics as a function of the first two principal components. The most notable conclusion here is that SZTS benchmarks have negative values for the second principal component, which suggests that these benchmarks exhibit a relatively high data processing speed (DPS); relatively limited time is spent in the *map* stage compared to the *reduce* stage; a relatively large amount of *reduce* output data is being produced; and a relatively high intra-node communication traffic is observed. Most of the other Hadoop benchmarks are located in the upper half of the workload space (positive value along second principal component), although some appear in the bottom half. This analysis clearly shows that SZTS as a whole exhibits consistent and fairly unique behavior at the job level.

Another key observation to be made here is that the workload space is very different across layers. For example, `mapmatching`, as discussed in the previous section, is the SZTS benchmark with the most distinct behavior at the microarchitecture level, yet it appears right in the center of the workload space at the job level (see Fig. 12). In other words, although this workload exhibits very typical behavior at the job level,

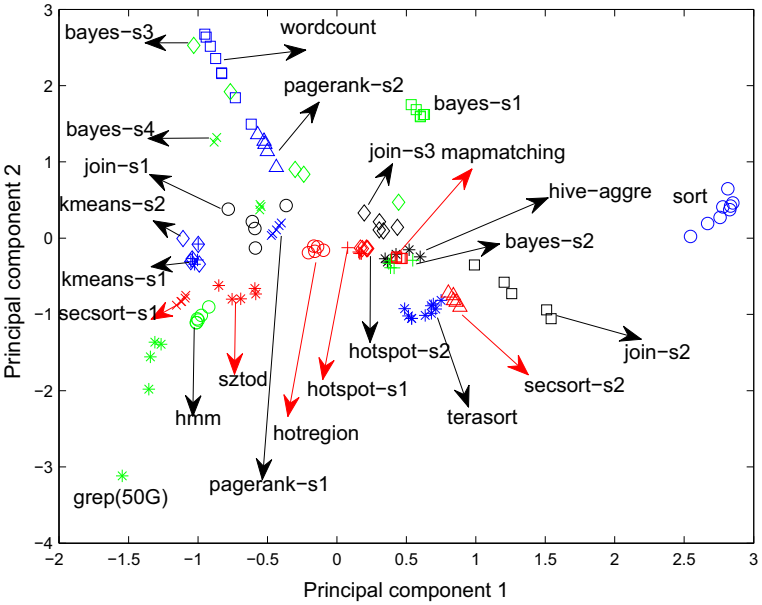

Fig. 12 Workload space (first two principal components) considering the job-level characteristics

it shows very unique behavior at the microarchitectural level. The inverse reasoning can be done for `sort`: we observe very unique behavior at the job level (see the isolated data points on the right-hand side in Fig. 12), but very typical behavior at the microarchitectural level (see the data points close to the origin in Fig. 7). The meta-conclusion is that the characterization of MapReduce/Hadoop workloads needs to be done across layers, to unravel a workload's true characteristic behavior.

6.4 Cross-layer characterization

Applying PCA to the job-level and microarchitecture characteristics yields ten PCs, accounting for 90.5 % of the overall variance. (The first PC accounts for 23.6 % of the variance; the second PC accounts for 15.6 % of the variance.) Figure 13 shows the factor loadings for the first four principal components. The first principal component is positively dominated by SMI, ROMI, TMI, TERMI, TRAMI, L2ICMPKI, and is negatively dominated by DPS, TMRS, TMRF, LLCMPKI, and BRMPKI. This implies that a workload with a high value along the first principal component exhibits a large amount of shuffled data, reduce output, temporal write data, intra/inter-node transmitted data, and high L2 instruction cache miss rate, but low data processing speed, low last-level cache miss rate, low branch misprediction rate, and more time spent in the reduce stage and function compared to the map stage and function.

Combining the microarchitecture and job-level characteristics in one analysis yields the workload space shown in Fig. 14. This visualization and its interpretation reconfirm our findings. SZTS is fairly unique at both the microarchitectural and job level compared to the other Hadoop benchmarks. The SZTS benchmarks appear at the bot-

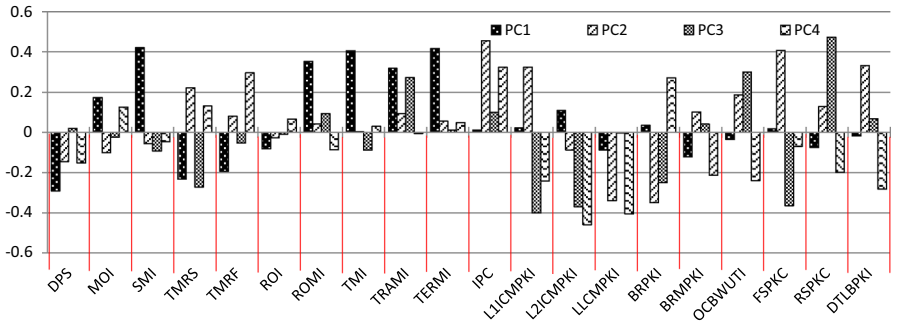

Fig. 13 PCA factor loadings using the job-level and microarchitecture-level characteristics

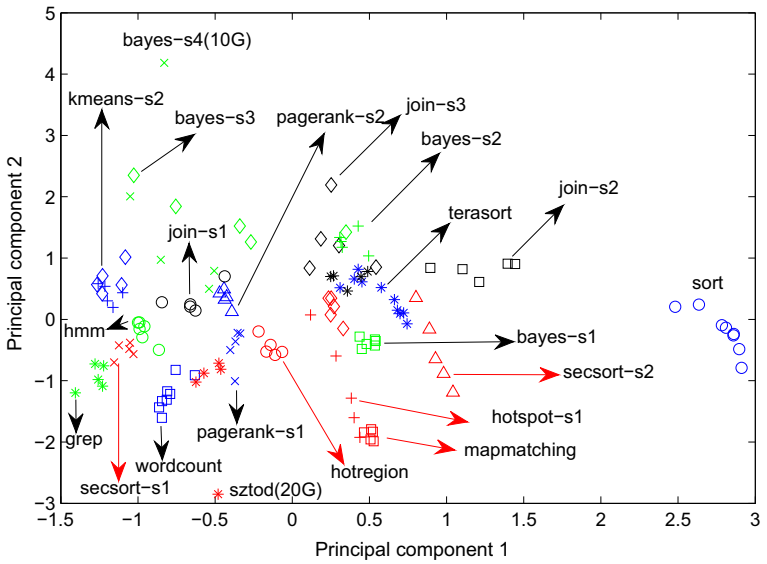

Fig. 14 Workload space (first two principal components) considering both job-level and microarchitecture-level characteristics

tom half which reflects their relatively low IPC, low instruction cache miss rate, high last-level cache and off-chip bandwidth pressure, and limited time spent in the map stage relative to the time spent in the reduce stage. This re-enforces our finding that cross-layer characterization reveals unique workload behaviors and provides a more comprehensive analysis.

7 Input data set sensitivity and selection

Input data size increases rapidly and hence a single Hadoop program needs to run with a lot of different input data sets in the big data era. However, how the input data size of a Hadoop program affects its performance behavior has not yet been fully understood. We, therefore, investigate this issue in this section.

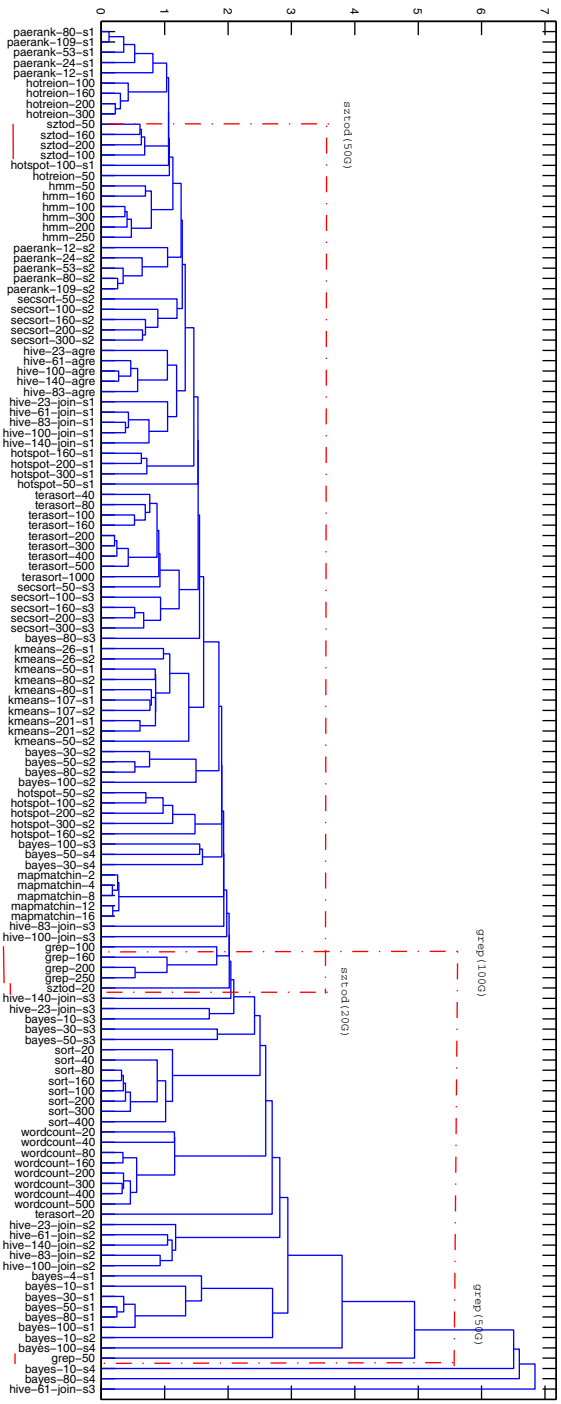

Fig. 15 Dendrogram obtained through cluster analysis using the *complete* linkage rule using cross-layer characteristics

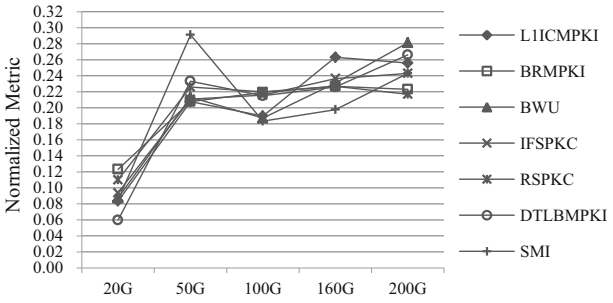

Fig. 16 Various metrics as a function of the input data set size for `sztod`

We use a wide range of input data set sizes, as wide as we possibly can. In particular, for `mapmatching` we are limited to an input set size of 16 GB because that is the size of the Shenzhen city map at the highest resolution. For the other SZTS benchmarks we consider input set sizes ranging from 10 to 300 GB; there are five input set sizes per benchmark uniformly distributed across the entire range.

Figure 15 shows the dendrogram from applying cluster analysis on the workload space built from microarchitecture and job-level characteristics after PCA. (Note that we do not use OS-level characteristics for different input data sets of the same benchmark here because the ratio of time spent in kernel space to total execution time for even different benchmarks is quite similar, see Fig. 8.) Recall that large linkage distances indicate dissimilar behavior, whereas short distances indicate similar behavior. The dendrogram is a useful tool to understand the impact of input data sets on execution behavior. For example, all inputs for `mapmatching` and `hmm` (Hidden Markov Model), as well as for `wordcount` and `sort` (towards the bottom of the dendrogram) lead to similar execution behavior—short linkage distances. However, other programs are quite sensitive to their input, see for example `grep` and `sztod`, as illustrated in the Fig. 15 using the dashed red lines. For `grep`, the smallest input (50 GB) leads to dissimilar behavior compared to the larger inputs (100–250 GB). Similarly, for `sztod`, the smallest input (20 GB) is dissimilar to the larger inputs.

We now zoom in on these two cases, `sztod` and `grep`. Figure 16 shows why the 20 GB input data set of `sztod` cannot be merged to the group of other input data sets (50–200 GB). The seven metrics shown in the graph, including job- and microarchitecture-level characteristics (L1ICMPKI, BRMPKI, BWU, IFSPKC, RSPKC, DTLBMPKI and SMI) suggest significant differences between the 20 GB input data set and the other input data sets. Figure 17 illustrates why the program behavior of the 50 GB input data set of `grep` is significantly different from the other input data sets. The IPC, L1CMPKI and TMRF for all five input data sets is quite similar. However, the ROI is much higher for the 50 GB input data set compared to the others. The same applies to BRPKI, although to a lesser extent. DPS also seems lower for the 50 GB input data set.

Input data set sensitivity analysis is instructive when selecting representative inputs for benchmarking purposes. For example, for `mapmatching`, which appears to be an input-insensitive workload, we could pick any input size and still obtain the same performance results. One strategy could be to pick a small input to reduce experimen-

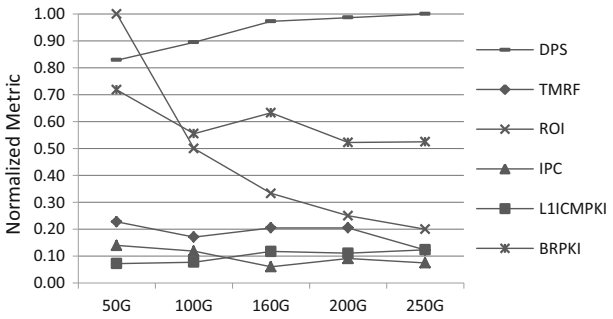

Fig. 17 Various metrics as a function of the input data set size for `grep`

tation time. On the contrary, for `sztod`, we should not pick the smallest input to save experimentation time as it may lead to a misleading characterization compared to the larger, more representative inputs. In other words, for an input-sensitive workload, we need to be very careful when selecting a representative input.

8 Related work

Big data processing has placed unprecedented demands on computing systems, resulting in challenges for evaluating and understanding such systems. Benchmarking big data systems, therefore, attracts significant attention. Although big data system performance evaluation is a recent topic, benchmarking parallel systems has been a long-standing challenge, see for examples [35–43].

The traditional benchmark suite for database systems is TPC [10]. Big data imposes severe challenges on database systems and impels these systems to improve for big data processing, see TPC-DS [10] and BigBench [15]. Another database benchmark suite for big data processing is YCSB which is designed for evaluating Yahoo!'s cloud platform and data storage systems [11]. YCSB mainly consists of online service workloads—so-called 'Cloud OLTP' workloads. More recently, Armstrong et al. released a benchmark suite—LinkBench—based on Facebook's social graph [12]. These benchmark suites focus on database systems which are fairly different from Hadoop-based systems.

Several non-database big data benchmark suites have been proposed as well. Ferdman et al. study the microarchitecture-level characteristics of scale-out programs in the cloud [13]; Wang et al. propose a big data benchmark suite named BigDataBench for evaluating Internet services [14]; and Jia et al. characterize data analysis workloads in data centers [16]. Early Hadoop benchmarks include `GridMix` [17], `Sort` [18], and `TeraSort` [19,20]. The closest related works are HiBench [6] and CloudRank-D [7]. All these prior works focus on a single layer, either the job level or the microarchitecture level. In this work, we characterize the workloads at both levels, and more importantly, we focus on a particular application domain (city transportation) while considering real, deployed big data applications with real-life inputs; other benchmark suites contain (mostly) generic algorithms with (mostly) synthetically generated input data sets.

9 Conclusions

In this paper, we propose a novel big data benchmark suite named SZTS, which we release publicly. The SZTS benchmarks are unique because they are derived from domain-specific real-life programs in city transportation analysis along with real-life data. We comprehensively characterize SZTS via a cross-layer approach including microarchitecture-, OS-, and job-level characteristics. We find SZTS to be very different from traditional multi-core workloads (PARSEC), and while being more similar to other Hadoop benchmark suites (HiBench and CloudRank-D), we still find that SZTS exhibits fairly unique behavior. SZTS typically has unique characteristics at both the microarchitecture level and job level: relatively low IPC, good instruction cache performance, but high last-level cache and off-chip memory bandwidth pressure; limited time spent in the map stage relative to the reduce stage, high data processing speed, and high intra-node communication traffic. We also study the impact of input data size on workload behavior, and find several benchmarks to be very sensitive to input size, which enables identifying representative inputs. The meta-conclusion from the analysis is that cross-layer characterization leads to unique insights and a more comprehensive evaluation.

References

1. Eyal O, Renaud D, Michele C, Richard C, Holger S, Giovanni T (2008) Sindice.com: a document oriented lookup Index for open linked data. Int J Metadata Semant Ontol 3:37–52
2. Luo L (2010) Heuristic artificial intelligent algorithm for genetic algorithm. Key Eng Mater 439:516–521
3. Kevin Sr B, Vuk E, Rainer G, Andrey B, Mohamed E, .Ozcan KC,Shekita F, Eugene J (2011) Jaql: A scripting language for large scale semistructured data analysis, in VLDB
4. Wen MP, Lin HY, Chen AP, Yang C (2011) An integrated home financial investment learning environment applying cloud computing in social network analysis, in ASONAM, pp 751–754
5. Langmead B, Schatz MC, Lin J, Pop M, Salzberg SL (2009) Searching for SNPs with cloud computing. Genome Biol 10(11):R134
6. Huang J, Dai J, Xie T, Huang B (2010) The HiBench Benchmark Suite: Characterization of the MapReduce-Based Data Analysis, in ICDEW, pp 41–51
7. Luo C, Zhan J, Jia Z, Wang L, Lu G, Zhang L, Xu CZ, Sun N (2012) CloudRank-D: benchmarking and ranking cloud computing systems for data processing applications. Front Comp Sci 6(4):347–362
8. Lou Y, Zhang C, Zheng Y, Xie X, Wang W, Huang Y (2009) Map-Matching for Low-Sampling-Rate GPS Trajectories. In: Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, pp 1–10
9. Bienia C, Kumar S, Singh J, Li K (2008) The PARSEC Benchmark Suite: Characterization and Architecture Implications, in PACT, pp 72–81
10. URL http://www.tpc.org/information/benchmarks.asp
11. Cooper BF Silberstein A, Tam E, Ramakrishnan R, Sears R (2010) Benchmarking Cloud Serving Systems with YCSB, in SoCC, pp 143–154
12. Armstrong TG, Ponnekanti V, Borthakur D, Callaghan M (2013) LinkBench: A Database Benchmark based on the Facebook Social Graph
13. Ferdman M, Adileh A, Kocberber O, Volos S, Alisafaee M, Jevdjic D, Kaynak C, Popescu AD, Ailamaki A, Falsafi B (2012) Clearing the Clouds: A Study of Emerging Workloads on Modern Hardware, in ASPLOS, pp 37–48
14. Wang L, Zhan J, Luo C, Zhu Y, Yang Q, He Y, Gao W, jia Z, Shi Y, Zhang S, Zheng C, Lu G, Zhan K, Li X, Qiu B (2014) BigDataBench: A Big Data Benchmark Suite from Internet Services, in HPCA, pp 1–12

15. Ghazal A, Hu M, Rabl T, Raab F, Poess M, Grolotte A, Jacobsen HA (2013) BigBench: Towards an Industry Standard Benchmark for Big Data Analytics, in SIGMOD
16. Jia Z, Wang L, Zhan J, Zhang L, Luo C (2013) Characterizing Data Analysis Workloads in Data Centers, in IISWC
17. GridMix Program. Available in Hadoop source distribution: src/benchmarks/gridmix. URL http://hadoop.apache.org/docs/r1.2.1/gridmix.html
18. Sort program. Available in Hadoop source distribution: src/examples/org/apache/hadoop/examples/sort. URL https://github.com/facebookarchive/hadoop-20/tree/master/src/examples/org/apache/hadoop/examples/terasort
19. TeraSort. URL http://sortbenchmark.org/
20. Hadoop TeraSort program. Available in Hadoop source distribution since 0.19 version: src/examples/org/apache/hadoop/examples/terasort. URL https://github.com/facebookarchive/hadoop-20/blob/master/src/examples/org/apache/hadoop/examples/Sort.java
21. Eeckhout L, Vandierendonck H, De Bosschere K (2003) Quantifying the impact of input data sets on program behavior and its applications. J Instr-Level Parall 5(1):1–33
22. Hoste K, Eeckhout L (2007) Microarchitecture-independent workload characterization. IEEE Micro 27(3):63–72
23. Baru C, Bhandarkar M, Nambiar R, Poess M, Rabl T (2013) Benchmarking big data systems and the BIGDATA TOP100 list. Big Data J 1(1):60–64
24. URL http://clds.sdsc.edu/wbdb2014.de
25. URL http://prof.ict.ac.cn/bpoe_5_vldb
26. Baru C, Bhandarkar M, Nambiar R, Poess M, Rabl T (2013) Setting the direction for big data benchmark standards. selected topics in performance evaluation and benchmarking. Lect Notes Comp Sci 7755:197–208
27. Yanpei C, Ganapathi A, Griffith R, Katz R (2011) The case for evaluating MapReduce performance using workload suites, in MASCOTS, pp 390–399
28. URL http://www.fis.unipr.it/doc/oprofile-0.9.7/internals.html
29. Silberschatz A, Gagne G, Galvin PG. Operating System Concepts, Ninth Edition, Chapter 4
30. URLhttp://en.wikipedia.org/wiki/Light-weight-process
31. Zhou Z. Understanding the JVM Advanced Features and Best Practices, Second Edition, Chapter 12
32. Yasin A, Ben-Asher Y, Mendelson A (2014) Deep-dive Analysis of the Data Analytics Workload in CloudSuite, in IISWC
33. URL http://parsec.cs.princeton.edu
34. Islam NS, Rahman MW, Jose J, Rajachandrasekar R, Wang H, Subramoni H, Murthy C, Panda DK (2012) High performance RDMA-based design of HDFS over InfiniBand, in High Performance Computing. Networking, Storage and Analysis, SC
35. Bhandarkar SM, Arabnia HR (1995) The REFINE multiprocessor: theoretical properties and algorithms. Parall Comp Elsevier 21(11):1783–1806
36. Arabnia HR, Oliver MA (1987) Arbitrary rotation of raster images with SIMD machine architectures. Int J Eurograp Assoc (Comp Graph Forum) 6(1):3–12
37. Arabnia HR (1990) A parallel algorithm for the arbitrary rotation of digitized images using process-and-data-decomposition approach. J Parall Distrib Comp 10(2):188–193
38. Arabnia HR, Smith JW (1993) "A Reconfigurable Interconnection Network For Imaging Operations And Its Implementation Using A Multi-Stage Switching Box"; Proceedings of the 7th Annual International High Performance Computing Conference. The 1993 High Performance Computing: New Horizons Supercomputing Symposium, Calgary, Alberta, Canada, pp 349–357
39. Arabnia HR (1995) "A Distributed Stereocorrelation Algorithm". In: Proceedings of Computer Communications and Networks (ICCCN'95), IEEE, pp 479–482
40. Bhandarkar S M, Arabnia HR, Smith JW (1995) "A Reconfigurable Architecture for Image Processing and Computer Vision", International Journal of Pattern Recognition and Artificial Intelligence (IJPRAI) (special issue on VLSI Algorithms and Architectures for Computer Vision, Image Processing, Pattern Recognition and AI), Vol 9, no 2, pp 201–229
41. Bhandarkar SM, Arabnia HR (1995) The hough transform on a reconfigurable multi-ring network. J Parall Distrib Comp 24(1):107–114

42. Arabnia HR, Bhandarkar SM (1996) Parallel stereocorrelation on a reconfigurable multi-ring network. J Supercomp (Springer Publishers) 10(3):243–270
43. Arif Wani M, Arabnia HR (2003) Parallel edge-region-based segmentation algorithm targeted at reconfigurable multi-ring network. J Supercomp 25(1):43–63