Kill Two Birds with One Stone: Auto-tuning RocksDB for High Bandwidth and Low Latency

Yichen Jia
Computer Science and Engineering
Louisiana State University
yjia@csc.lsu.edu

Feng Chen
Computer Science and Engineering
Louisiana State University
fchen@csc.lsu.edu

***Abstract*—Log-Structured Merge (LSM) tree based key-value stores are widely deployed in data centers. Due to its complex internal structures, appropriately configuring a modern key-value data store system, which can have more than 50 parameters with various hardware and system settings, is a highly challenging task. Currently, the industry still heavily relies on a traditional, experience-based, hand-tuning approach for performance tuning. Many simply adopt the default setting out of the box with no changes. *Auto-tuning*, as a self-adaptive solution, is thus highly appealing for achieving optimal or near-optimal performance in real-world deployment.**

In this paper, we quantitatively study and compare five optimization methods for auto-tuning the performance of LSM-tree based key-value stores. In order to evaluate the auto-tuning processes, we have conducted an exhaustive set of experiments over RocksDB, a representative LSM-tree data store. We have collected over 12,000 experimental records in 6 months, with about 2,000 software configurations of 6 parameters on different hardware setups. We have compared five representative algorithms, in terms of throughput, the 99th percentile tail latency, convergence time, real-time system throughput, and the iteration process, etc. We find that multi-objective optimization (MOO) methods can achieve a good balance among multiple targets, which satisfies the unique needs of key-value services. The more specific Quality of Service (QoS) requirements users can provide, the better performance these algorithms can achieve. We also find that the number of concurrent threads and the write buffer size are the two most impactful parameters determining the throughput and the 99th percentile tail latency across different hardware and workloads. Finally, we provide system-level explanations for the auto-tuning results and also discuss the associated implications for system designers and practitioners. We hope this work will pave the way towards a practical, high-speed auto-tuning solution for key-value data store systems.

I. Introduction

In today's data centers, Log-Structured Merge (LSM) tree [46] based key-value data stores (e.g., LevelDB [24] and RocksDB [4]) are being widely deployed for high-speed data processing. Due to its complex internal structures, a modern key-value data store offers a number of configurable parameters (e.g., buffer size, thread number, table size, etc.), allowing users to tune system performance for different hardware and workloads. After years of optimizations, such a set of configurable parameters becomes indispensable for users to gain fine-grained customizability for performance tuning. For example, RocksDB, a highly popular key-value data store in industry, exposes over 50 tunable parameters to system administrators in its latest release [4].

Appropriately configuring a key-value store is crucial to the runtime performance. Each configuration parameter controls a certain aspect of the system behavior, such as parallelism degree, I/O size, event-triggering frequency, etc. A selected configuration profile in effect determines the observed performance. Further considering the highly diverse workload and hardware properties in real-world deployment, a configuration that works optimally in one particular scenario may not work equally well in another. In other words, it is difficult to have a universally optimal setting for all cases. For this reason, performance tuning in the deployment of key-value data stores is an important but notoriously tedious, time-consuming, and case-by-case work.

In the current practice, the industry still heavily relies on a traditional, experience-based *hand-tuning* approach, which significantly increases the administration cost and delays the time to deploy. *Auto-tuning*, as an automatic self-adaptive approach, is thus highly appealing. As a general system solution, auto-tuning was originally proposed to overcome the unscalability of manual tuning. It has been studied in various scenarios, such as cloud storage, databases, parallel systems, and many others [5], [20], [34], [38], [40]. These prior works focus on optimizing for one objective function, such as throughput, power consumption, or monetary cost, etc. However, such a solution cannot readily satisfy the need for quick deployment of a key-value store system.

A unique challenge for auto-tuning key-value data stores is that we must achieve *multiple* Service Level Objectives (SLOs) [36], [49], simultaneously. In a typical enterprise-class application scenario, the data store system needs to guarantee to achieve two important but sometimes conflicting optimization goals, *throughput* and *latency*[1]. Such a requirement makes auto-tuning in LSM-tree based key-value data stores even more challenging.

In this paper, we have quantitatively studied and compared five representative optimization methods for auto-tuning the performance of RocksDB, a highly popular LSM-tree based key-value store optimized for flash SSDs, to meet its unique

[1]The main optimization targets for the latency SLOs are typically the 95th, 99th or even 99.9^{th} percentile tail latency [16], [28], [36], [49], [55].

Quality of Service (QoS) requirements. To the best of our knowledge, this is the first work that presents quantitative analysis on the efficacy and efficiency of auto-tuning algorithms on RocksDB and provides system-level explanations for the auto-tuning process. Our study has been conducted in two stages.

- **Stage 1: Data collection**: In order to quantitatively demonstrate the ability of the auto-tuning algorithms, we have evaluated RocksDB with over 2,000 software configurations, 4 hardware setups, and 3 representative workloads. We have executed more than 12,000 experimental runs over 6 months. The performance metrics and the related information are maintained in an MySQL [3] database for offline references. The information includes hardware and workload details, parameter settings, throughput, tail latency, etc. In this paper, we focus on optimizing for two major SLOs, *throughput* and *tail latency* (the $99th$ percentile). The methodology that we have used in this paper can also be applied to achieve other optimization goals, such as power consumption and monetary cost, etc.
- **Stage 2: Algorithm Analysis**: We select five optimization algorithms for auto-tuning, namely Genetic Algorithms (GA) [8], Non-dominated Sorting Genetic Algorithm II (NSGA-II) [19], Speed-constrained Multi-objective Particle Swarm Optimization (SMPSO) [44], ϵ-constraint Method (ECM) [56], and Weighted Sum Method (WSM) [43], and apply them to the collected real experimental data to find the (near-)optimal configurations. In our study, the selected five algorithms cover both single- and multi-objective optimization methods, and represent three common techniques (prior, posteriori, and no-preference methods as discussed in Section II) used to solve multi-objective optimization problems. Our experimental results show that, although their efficacy differs, all the five algorithms are able to eventually converge to provide stable performance, if given enough time. However, the best algorithm differs according to the QoS requirements. We also find that the multi-objective optimization algorithms can achieve a good balance among multiple goals. We present 12 findings in this work, which summarize our key observations and understandings on the auto-tuning algorithms, and also provide the optimization recommendations for RocksDB.

It is worth noting that our focus is not to find a good parameter setting for a specific hardware or system setup, which may vary on different platforms. Rather, our main objective is to compare and understand the practical efficacy of applying auto-tuning algorithms, especially the multi-objective optimization algorithms, on RocksDB and to gain important insight on how they behave in real-world deployment. We hope that our findings and the associated system implications will pave the way for system designers and practitioners towards developing a practical, efficient, and effective auto-tuning solution for key-value data stores.

The rest of the paper is organized as follows. Section II gives the background. Section III introduces the methodology. Section IV presents the comparative analysis on the five auto-tuning algorithms. Section V discusses the impact of hyper-parameters. Section VI describes the system implications. Section VII discusses the limitations of this work and the future work. Related work is presented in Section VIII. The final section concludes this paper.

II. Background

A. LSM-tree based Databases

LSM-tree based data stores are widely used nowadays in industry, such as LevelDB [24], RocksDB [4], and Apache Cassandra [2], etc. LSM-tree based key-value stores have two important components: in-memory write buffers and on-disk *Sorted Sequence Table* (SST) files. The incoming traffic is first accumulated in write buffer, which is implemented as a *skiplist* [47], and then becomes *Immutable Table* (ImmuTable) when the write buffer is full. Then the ImmuTables are flushed to the underlying storage device for persistence. This design guarantees that the storage only receives large, sequential I/Os, benefiting the I/O performance. Meanwhile, a *Write Ahead Log* (WAL) is maintained for the recovery purpose. SST data files are organized into multiple levels (from Level-0 to Level-N) of increasing size. Level-0 files are special, since they may have overlapping key ranges, while files in each of the other levels have non-overlapping key ranges. A background merging process, called *Compaction*, routinely runs to remove the deleted and obsolete key-value items. For example, when the number of files in Level-0 reaches a predefined threshold, multiple Level-0 files are merged with the Level-1 files that have overlapping key ranges with these Level-0 files. Once completed, the input Level-0 and Level-1 files are deleted and replaced by the newly generated Level-1 files. The compaction processes at the other levels are similar. A *Manifest* file maintains the metadata of the SST files.

B. Multi-objective Optimization Algorithms

Multi-objective Optimization (MOO) methods deal with optimization problems that have multiple conflicting goals. As defined in prior work [17], the multi-objective optimization problem can be generalized as follows:

$$\underset{x}{Minimized} : F(x) = [F_1(x), F_2(x), ..., F_k(x)]^T \quad (1)$$

Subject to:

$$x_i^L \leq x_i \leq x_i^U; i = 1, 2, ..., n$$

$$g_j(x) \leq 0; j = 1, 2, ..., m$$

where k is the number of targets, n is the number of variables, and m is the number of constraints. $x \in R^n$ is a collection of variables $x_i \in [x_i^L, x_i^U]$, $g_j(x)$ are the constraint functions, and $F(x) \in R^k$ is a collection of objective functions $F_i(x)$.

Typically, there is no single solution that can simultaneously optimize all the objectives. A solution is called *Pareto optimal* [51], [54], if no objective can be improved without degrading the other objectives. If there is no additional information about user-specific preference, all the Pareto optimal solutions are considered equally good.

According to the articulation of preferences, the multi-objective optimization algorithms can be classified into three

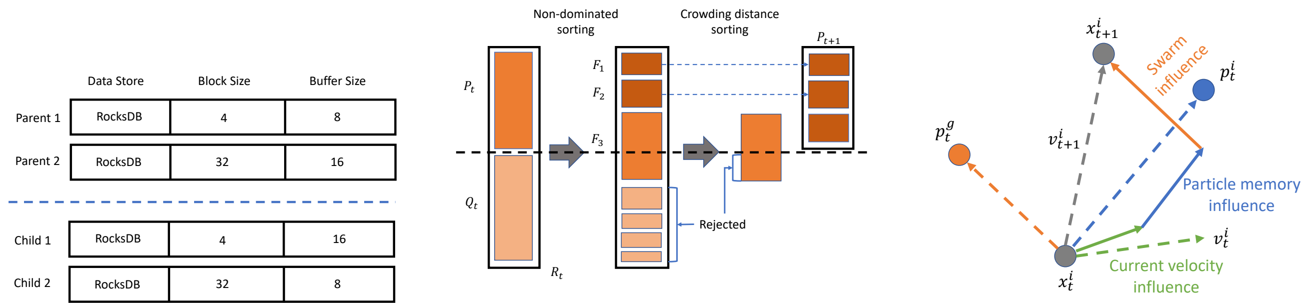

Fig. 1: Illustration of Crossover in GA. Fig. 2: An Illustration of NSGA-II [19]. Fig. 3: Position Update in PSO [52].

main categories: (1) Algorithms with a *priori articulation* of preferences: These methods allow users to specify their preferences in terms of the relative importance of different objectives. (2) Algorithms with a *posteriori articulation* of preferences: When the users cannot provide an explicit preference function, it is effective to allow users to choose from a collection of possible solutions. (3) Algorithms with a *no articulation* of preferences: when the users cannot define their preference explicitly, this group of methods assume that all the objectives are equally important.

C. Applied Methods

Multi-objective optimization (MOO) has been extensively studied in the fields of science and engineering [10], [15], [18], [27], [29], [33], [45], [48], [51]. In this work, we choose five mature and widely deployed auto-tuning algorithms for our study, including one single-objective algorithm (GA) and four multi-objective optimization algorithms (NSGA-II, SMPSO, ECM, and WSM).

Genetic Algorithm (GA). Single-objective genetic algorithm [8] belongs to the family of evolutionary algorithms that are designed based on the natural selection process and have been extensively studied [9], [29], [41]. The initial population is randomly generated. The population size is defined according to the nature of the problem. The fittest portions of the current population are selected based on the user-defined fitness function to create a new generation, during each successive generation. This process guarantees that better genes are inherited with higher probability. The next generation is generated through a combination of genetic operators: crossover and mutation. The selection process is repeated until reaching the termination condition. Figure 1 illustrates an example of the cross-over process in GA, in which a cross point is selected and the tails of the two parents (Buffer Size) are swapped to generate new offsprings.

Non-dominated Sorting Genetic Algorithm (NSGA-II). As one of the most well known and widely deployed MOO algorithms [57], NSGA-II [19] can find a diverse set of solutions with fast non-dominated sorting and diversity preservation. The process of NSGA-II is illustrated in Figure 2.

Firstly, the population is initialized randomly with a predefined population size. All the chromosomes are sorted in the Pareto front based on Pareto Non-dominated sets. The chromosomes in the Pareto front are ranked based on euclidean distance or I-dist between solutions, which are defined in NSGA-II. Typically, solutions that are far from others will have a higher probability to be selected to improve diversity and to avoid a crowded solution set. Then the best ones in the current population are put into the mating pool. During the mating process, tournament selection, crossover and mating are conducted to generate offsprings. The offsprings and the current population are combined and sorted to pick the best N chromosomes into new population. The selection process continues until reaching the maximum number of generations. Finally, the highest ranked Pareto optimal solutions from the latest population are chosen as the final solutions.

Speed-constrained Multi-objective Particle Swarm Optimization (SMPSO). Particle Swarm Optimization (PSO) algorithm [52] simulates the social behaviors of animals, such as insects, herds, fishes, and birds. These swarms cooperate to find food by sharing information among members. In comparison to other optimization algorithms, PSO needs to adjust fewer parameters and thus is simpler for deployment while still providing good performance [7]. Specifically, as Figure 3 shows, PSO updates the new position of a particle by combining the optimal position of the swarm and that of its own, as well as its velocity. Particles keep updating their states constantly until they reach the termination condition. As defined in prior work [52], a particle i is defined by its position vector, x_i, and its velocity vector, v_i. Every iteration, each particle's position is updated according to Equation 2 and Equation 3:

$$v^i_{t+1} = \omega v^i_t + c_1 r^t_1 (p^g_t - x^i_t) + c_2 r^t_2 (p^i_t - x^i_t) \tag{2}$$

$$x^i_{t+1} = x^i_t + v^i_{t+1} \tag{3}$$

where p^g_t and p^i_t respectively denote the best group position and the best particle position, ω denotes inertia weight, c_1 and c_2 represent two positive constants, and r_1 and r_2 represent two random parameters within [0, 1]. Speed-constrained Multi-objective Particle Swarm Optimization (SMPSO) [44] introduces a velocity constraint mechanism which restricts the value within the variable ranges and vanishes the erratic movements. SMPSO is believed to deliver higher accuracy with less time than traditional PSO.

Epsilon Constrained Method (ECM). According to prior work [56], ϵ-constrained Method selects and minimizes a primary objective by expressing the other objectives with inequality constraints:

$$\underset{x}{Minimize} : F_p(x) \tag{4}$$

subject to $F_i(x) \leq \epsilon_i$, for $i = 1...k, i \neq p$. For example, in key-value data store, we may maximize the throughput by meeting the latency SLO requirement. This bounded function method is robust and efficient for converting a multi-objective optimization problem into a single-objective one [42].

Weighted Sum Method (WSM). WSM is one of the most common approaches [42]. According to prior work [43], the weighted sum method realizes the multi- to single-objective problem conversion by constructing a simple weighted sum of all the objectives:

$$\underset{x}{Minimize} : F(x) = \sum_{i=1}^{k} (\omega_i \times F_i(x)) \tag{5}$$

As Equation 5 shows, the single final target $F(x)$ is the weighted sum of each individual target $F_i(x)$. The challenge of this approach is how to determine the weighting coefficients to each of the objectives. Besides, summing up two objectives, such as throughput and tail latency, requires the normalization to be semantically meaningful.

The above summarizes the five methods briefly. More details can be found in their related papers [8], [19], [43], [52], [56].

D. Exploitation and Exploration

The *exploitation* and *exploration* tradeoff is well-known in auto-tuning systems to acquire new knowledge and to maximize the uncertain payoffs. Exploitation means to probe a limited portion of the search space, expecting to improve the existing promising solution. This operation tries to leverage the vicinity of the current candidate to figure out a better solution. On the other hand, exploration means to probe a large search space to avoid being trapped into a local optimum. The tradeoff between exploitation and exploration is among our interests in this paper to illustrate the effectiveness and efficiency of the selected algorithms.

III. METHODOLOGY

In this section, we describe the details of hardware environments, workload characteristics, parameter space and our implementation of the optimization algorithms.

Hardware. Our experiment platform is an Intel W2600CR server with 32 HT virtual cores on two 8-core Intel Xeon E5-2690 2.9GHz processors and 128 GB memory. We use an 800GB Intel 750 PCIe SSD and a 240GB Intel 530 SATA SSD as storage devices in our experiments. In order to collect performance data with different hardware setups, we have defined four machine configurations as listed in Table I. Based on their computing, memory, and storage capabilities, we categorize the four hardware setups as *Baseline*, *CPU-plus*, *MEM-plus* and *STOR-minus* to illustrate the effect of different computer components to the performance.

Software and Workloads. We use Ubuntu 14.04 with Linux Kernel 4.4.0, Ext4 file system and RocksDB 5.17.0 in our experiments. In order to cover different workload patterns, we have enhanced RocksDB's default benchmarking tool, *db_bench*, to generate three workloads following typical key distributions. (1) *Zipfian*. It is a distribution pattern following true Zipf's law [23], [30], [58], where a small portion of items receives most of the requests, and the rest items are requested rarely. (2) *Hotspot*. A majority (80%) of its GET requests access a relatively small portion (20%) of the entire data set. (3) *Random*. All the records in the database are accessed randomly with an equal probability. Among the three distributions, Zipfian has the most skewed access pattern.

The working-set size has an obvious effect to the duration of experiments. Since we mainly focus on exploring a large parameter space within a practical time period, we choose to set the working set size to be 100 GB. This large working set can guarantee that all experiments can be finished within a reasonably long time (six months in our case). We have also repeated part of the experiments when the working set is 200 GB to validate our findings. For each test, after warming up the system, we run the experiment for 180 seconds, which is long enough to collect stable evaluation results. Our experimental results show a wide range of performance numbers and are suitable for applying and evaluating auto-tuning algorithms.

Parameter Space. Compared with LevelDB, RocksDB has adopted several schemes particularly optimized for fully exploiting the rich internal parallelism features of flash SSDs [13], [14]. For example, multiple immutable Memtables are used to avoid write stalls. Flushes and compaction operations are multi-threaded with separated thread pools and execution priorities. Since RocksDB is optimized for parallel operations, users are suggested to parallelize requests at the application level. Furthermore, RocksDB keeps multiple files in Level 0 and triggers the compaction process when the number of Level-0 files reaches a predefined threshold. RocksDB also keeps a read cache to accelerate READ operations.

Because of the aforesaid rich features, RocksDB maintains over 50 tunable parameters. Apparently we are unable to exhaustively study all combinations of the 50 parameters. Since our goal is to study the efficacy of applying different optimization methods for auto-tuning RocksDB, we desire to study a parameter space that is large enough to cover the important parameters. Based on the observations from prior work [22], [40], we select and focus on six most important parameters that have a significant impact to performance. Table II shows the parameters and the value range of each parameter.

Experiments and Implementations. Our experimental studies have been performed in two stages.

Stage 1: Exhaustive Experiments. We first run experiments with an exhaustive combination of the six configurations as listed in Table II, with the selected workloads and machine setups (see Table IV). Over a period of 6 months, we have completed over 12,000 experimental runs. After that, we store all the system configurations, workload and hardware

Machine	Server ID	CPU Model	vCores	Memory	Storage	Connection	OS
Baseline	M1	Intel Xeon 2.9GHz	4	8GB	Intel 750 SSD	PCIe	Ubuntu 14.04
CPU-plus	M2	Intel Xeon 2.9GHz	32	8GB	Intel 750 SSD	PCIe	Ubuntu 14.04
MEM-plus	M3	Intel Xeon 2.9GHz	32	16GB	Intel 750 SSD	PCIe	Ubuntu 14.04
STOR-minus	M4	Intel Xeon 2.9GHz	32	16GB	Intel 530 SSD	SATA	Ubuntu 14.04

TABLE I: Experimental Machine Configurations.

Parameter	Abbr.	Values	Description
Write Buffer (x32MB)	WB	1,2,4,8,16	The number of write buffers. Each buffer is 32MB.
Concurrent Threads	CT	1,2,4,8	The number of application-level requesting threads.
Flush Writers	FW	1,2,4	The maximum number of background flush operations.
Read Cache Size (GB)	CS	1,2,4	The size of read cache.
Cleanup File Num	CF	1,2,4,8	The number of Level-0 files when background compaction process is triggered.
Concurrent Compactor	CC	1,2,4	The maximum number of background compaction operations.

TABLE II: RocksDB Parameter Space.

information, and the benchmarking results in an MySQL [3] database for the emulation in Stage 2.

Stage 2: Auto-tuning Emulation. We emulate the process of auto-tuning key-value storage systems by running the optimization algorithms and querying MySQL for the evaluation results. In this way, we can avoid running the experiments against RocksDB each time. Since we focus on multi-objective optimization approaches, we choose to optimize both *throughput* and *latency* (the 99th percentile tail latency) simultaneously in all our experiments. We believe that the same methodology can be used when other objectives, such as power consumption, are considered.

We have implemented a client to make use of Platypus [1], which is an open source Python library for multi-objective optimization, for all of our experiments. We further convert the parameters in RocksDB into the algorithm-related ones. For example, we define write buffer number as *gene*, and each configuration as a *chromosome*. A sufficient number of configurations are measured as the evolution process continues.

The above-said experiment process provides two benefits. First, we only need to complete an experiment for each configuration once (Stage 1). In the algorithm evaluation (Stage 2), we can simply run the algorithms and query the MySQL database to collect the corresponding data without actually running the experiments. This significantly saves the experimental time and allows us to repeat this evaluation process quickly. Second, since we have already completed all the experiments exhaustively, we can know the global optimal configurations, which allows us to quantitatively measure how close each algorithm can reach the global optimum.

IV. Experimental Results

A. Motivations

In this section, we first demonstrate the parameter space and its effect on the two key performance metrics, throughput and latency. Due to space constraint, we select three scenarios (Zipfian and Hotspot workloads running on two hardware setups, M1 and M3) for illustration, as shown in Figure 4, 5, and 6. Each point represents one RocksDB configuration. In the figures, we also mark the configurations that deliver the maximum throughput, the minimum latency, and the default configuration. We can obtain several important clues from the figures. (1) The RocksDB configurations have a significant performance impact. For example, as Figure 4 shows, the achievable maximum throughput is 211 kop/s, while the tail latency is around 254 μs; the lowest achievable tail latency is 49 μs, while the throughput is only 82 kop/s. There is a clearly a tradeoff between these two goals. (2) The configurations are clustered in several groups, rather than uniformly distributed. This means that the parameters do not have an equal effect on the performance, and some parameters could have a dominant effect on the performance. (3) The default configuration cannot achieve the optimum, in all three cases. Also, we can find in the figures that the default setting tends to optimize for tail latencies rather than high throughput. As shown in Figure 4, the default setting achieves an average tail latency of about 83 μs, while the throughput is only about 60 kop/s. (4) The effect of different RocksDB configurations varies significantly across different hardware setups and workloads, which is clearly illustrated in the figures with the distinct shapes of the clouds of configuration points.

These figures show that although we may not find an optimal solution for both goals simultaneously, there is enough tradeoff and optimization space to fit the specific preferences of users. In the following section, we will discuss the performance of the five optimization algorithms for auto-tuning RocksDB.

B. Comparative Analysis

In this section, we compare the five optimization algorithms, whose configurations are shown in Table III. We use the terms, *optimal* and *near-optimal*, to represent configurations that provide 100% and 95% of the maximum achievable throughput respectively and restrict the 99th percentile tail latency within 250 μs as the QoS requirement in our experiments. For brevity, we focus on comparing and analyzing the auto-tuning behaviors with the PCIe SSD on M3 and the SATA SSD on M4 in this section.

As shown in Figure 7 and Figure 8, given enough time, all the five algorithms can converge to relatively stable performance. Specifically, as Figure 7 shows, on M3 with the PCIe SSD, GA, NSGA-II, SMPSO, and ECM can achieve comparable peak throughout (about 205 kop/s) with minimum 318 minutes. Accordingly, the corresponding 99th percentile

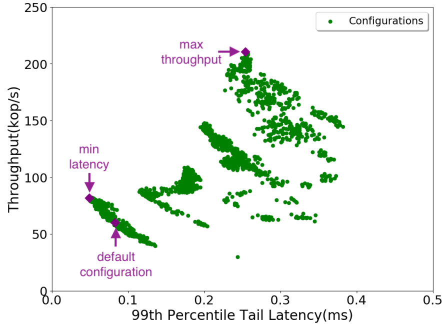

Fig. 4: Zipfian, M3.

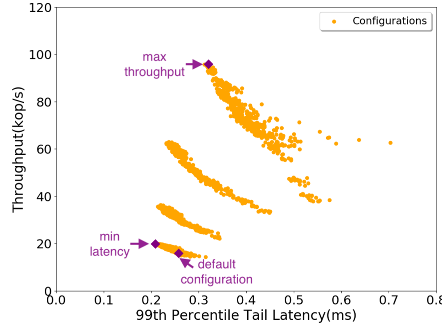

Fig. 5: Hotspot, M3.

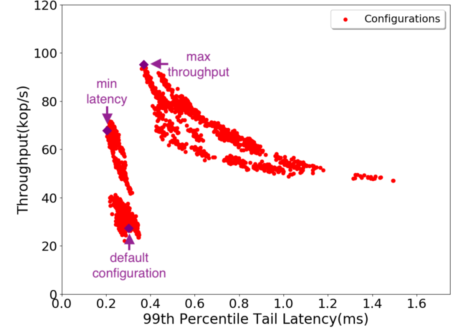

Fig. 6: Zipfian, M1.

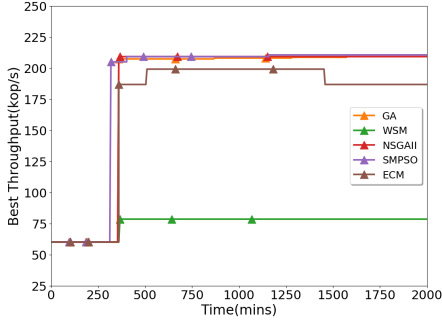

Fig. 7: Throughput vs. Time on PCIe SSD.

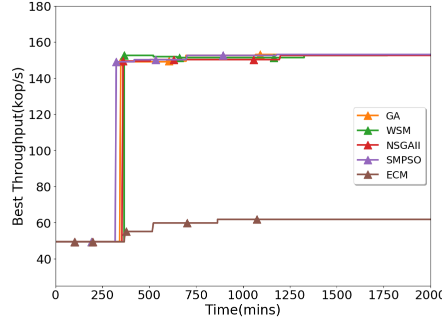

Fig. 8: Throughput vs. Time on SATA SSD.

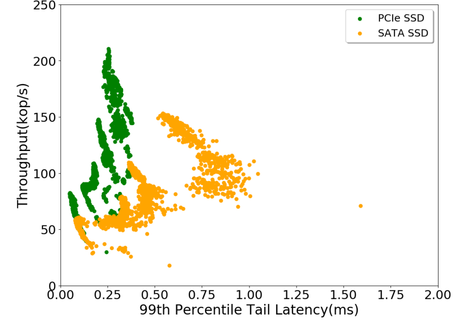

Fig. 9: Performance on PCIe vs. SATA SSD.

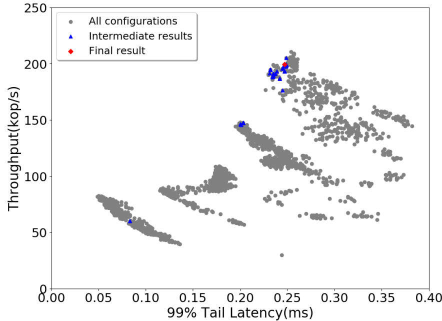

Fig. 10: Intermediate and Final Results.

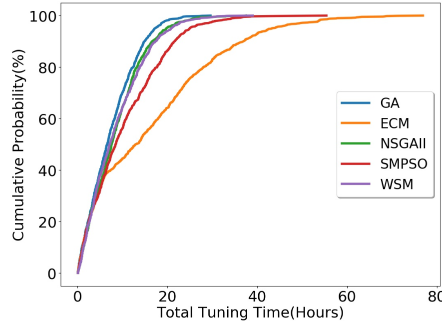

Fig. 11: CDF of Tuning Time on PCIe SSD.

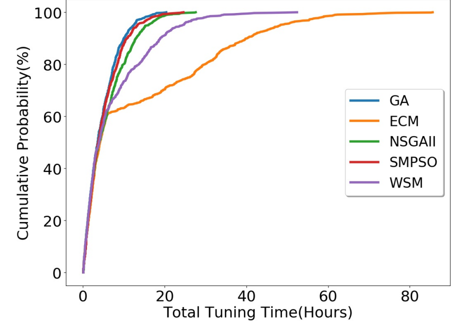

Fig. 12: CDF of Tuning Time on SATA SSD.

Algorithms	Optimizations Targets	QoS Requirements
GA	Throughput	No consideration
NSGA-II	Throughput and latency are treated equally	No consideration
SMPSO	Throughput and latency are treated equally	No consideration
ECM	Throughput	250μs
WSM	throughput/current_max_throughput - latency/current_max_latency	Considered in the combined target

TABLE III: Auto-tuning Algorithm Configurations.

tail latency is about 250 μs. WSM shows a totally different behaviors. We have combined the throughput and the 99th percentile tail latency into one single target using the weighted sum method. The single final target is defined as

$$T = \frac{throughput}{curr_max_throughput} - \frac{latency}{curr_max_latency} \quad (6)$$

We use $curr_max_throughput$ and $curr_max_latency$ to denote the maximum throughput and latency respectively until the current step. Since the normalized throughput (first component in Equation 6) and latency (second component in Equation 6) make positive and negative contributions to the combined final target respectively, we try to increase the throughput and decrease the latency by maximizing the combined goal. We can see that the maximum throughput achieved by WSM is about 78 kop/s and the corresponding 99th percentile tail latency is around 64 μs. In this case, WSM with the defined optimization target does not find the Pareto optimal configuration. There exists one better configuration, which has a higher throughput (82 kop/s) and a lower corresponding 99th percentile tail latency (49 μs), as Figure 4 shows.

Compared to the auto-tuning process with PCIe SSD on M3, the behaviors of these algorithms with SATA SSD on M4 show different trends. In Figure 8, GA, WSM, NSGA-II, and SMPSO exhibit similar behaviors in finding (near-)optimal configurations. It takes them at least 324 minutes to achieve the maximum throughput which is about 150 kop/s and the related 99th percentile tail latency is around 542 μs. However, the maximum throughout achieved by ECM is only about 62 kop/s and the corresponding 99th percentile tail latency is about 240 μs. The reason is that in order to meet the QoS requirements (250 μs in our experiments), some parameters in RocksDB, such as the number of concurrent threads, have to be set significantly smaller than other algorithms. Thus, the maximum throughput achieved by ECM is remarkably lower than the other approaches. The performance differences of ECM as shown in Figure 7 and Figure 8 is because the much faster PCIe SSD provides a larger search space than SATA SSD while constraining the 99th percentile tail latency within 250 μs. As a consequence, ECM achieves higher throughput on PCIe SSD by comparing more possible genes (parameters) and chromosomes (configurations). It means that based on the Quality of Services requirements provided by the users, we should choose the proper auto-tuning approach to identify the best configuration.

Finding#1: All of the algorithms can converge to stable performance if given enough time. Most of them can reach the (near-)optimal configuration eventually.

Finding#2: The performance features of underlying hardware, such as storage devices, have non-trivial impact on the auto-tuning behavior of the optimization algorithms.

We have also noticed that WSM shows very different behaviors on PCIe and SATA device as Figure 7 and Figure 8 show. To give a direct impression on the differences, we have illustrated the effect of the storage device to performance in Figure 9. Each dot in Figure 9 represents one configuration and the data with different devices are marked in different colors. We can see in Figure 9 that the tail latency range on SATA SSD (78 μs–1.6 ms) is significantly larger than that on PCIe SSD (49 μs–382 μs), while the throughput range difference is relatively smaller. As a consequence, the weight of latency component in the combined target, as Equation 6 shows, becomes significantly weaker on SATA SSD. Thus, WSM tends to tune the system to achieve near-maximum throughput on SATA SSD, and in contrast, to probe the system to produce moderate throughput on PCIe SSD.

Finding#3: ECM tends to achieve a relatively lower throughput than other algorithms on a slower storage hardware. On the contrary, our defined WSM produces a lower throughput on a faster storage hardware.

Finding#4: ECM and WSM behave differently on PCIe and SATA SSDs, and the reasons are distinct: ECM has to meet the QoS requirements while WSM optimizes the combined final target.

To have a better understanding on the auto-tuning process, Figure 10 shows the intermediate and final tuning results of ECM on machine M3. We can see that the algorithm has gone through 23 intermediate configurations before finding the near-optimal configuration. By constraining the 99th percentile tail latency within 250 μs, the maximum throughput ECM achieves is about 199 kop/s. Compared with the peak throughput (about 210 kop/s) we have observed, ECM can achieve 95% of the maximum throughput while meeting users' latency requirements. Note that we have also observed experiments that produce no solutions when the QoS requirements are set extremely low (e.g., 20μs).

Finding#5: ECM improves its solutions based on its intermediate results, but may produce no solutions when the users' QoS requirements are not properly set.

We have also compared the speed of the five auto-tuning methods in finding the near-optimal configurations on PCIe (see Figure 11) and SATA (see Figure 12) SSDs. The Y axis shows the percentage of total runs (1,000) that can find near-optimal configurations within certain period of time (X axis). Apparently, the faster the better. Figure 11 shows that ECM takes longer than the other algorithms to find the near-optimal configuration on PCIe SSD. Specifically, 90% of the experiments take 37 hours for ECM to achieve near-optimal performance, compared to less than 21 hours for GA, NSGA, WSM, and SMPSO. Similarly, on SATA SSD, as Figure 12 shows, ECM remains to be the most time-consuming approach among the five algorithms—90% of the experiments take about 40 hours to find the near-optimal configuration, while the other algorithms take less than 20 hours.

Based on our observations, ECM is the most time consuming approach on both PCIe and SATA SSDs. The reason is that the population in ECM is updated with only the configurations that meet the QoS requirements. In our experiments, only offsprings that constraint the latency within 250 μs are picked for the next-generation population. As a consequence, the gene diversity is reduced because of more strict selection conditions. Since gene diversity is one of the key factors that determine auto-tuning time, the total time taken by ECM is the longest among all the algorithms. This unique operation in ECM, as a consequence, prolongs the tuning duration.

Finding#6: ECM is the most time consuming algorithm among the five algorithms, although it meets the QoS requirements to the best.

C. Instantaneous Performance

Besides comparing the performance of near-optimal configurations, another aspect to consider is the instantaneous

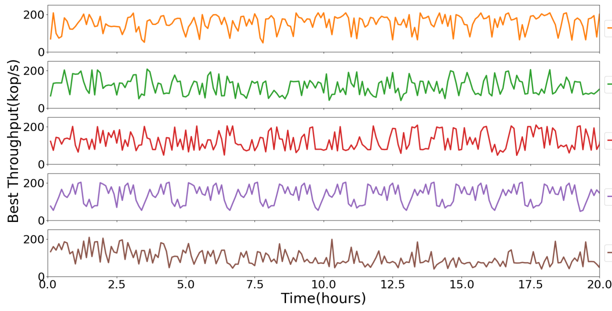

Fig. 13: Instantaneous Throughput.

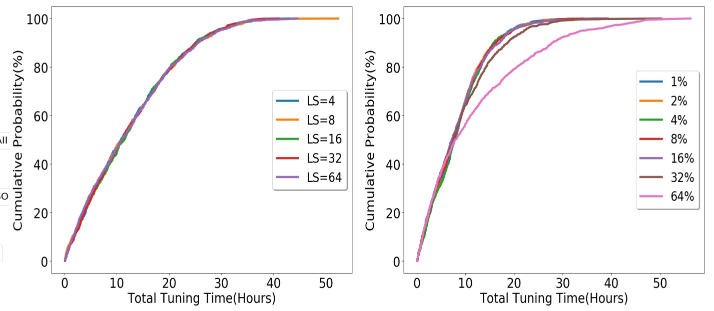

Fig. 14: Impact of Leader Size. Fig. 15: Impact of Mutation Rate.

performance during the auto-tuning process. To find a satisfactory configuration, a reasonably large exploration space is necessary, but it is desirable to avoid under-performing configurations as much as possible. A better algorithm is able to spend less time on bad configurations.

Figure 13 shows the instantaneous throughput (Y-axis) over time (X-axis) for one run for each method on PCIe SSD on machine M3. We can see that GA and SMPSO are the two methods that perform the best in terms of instantaneous throughput. They occasionally pick a configuration worse than the current one during the auto-tuning process, and they both have the ability to discard these unpromising configuration and evolve based on the satisfactory ones. Specifically, only 16% and 26% of the overall auto-tuning time show a throughput that is below 100 kop/s for GA and SMPSO, respectively. In contrast, the throughput of ECM and NSGA-II drops frequently, because ECM and NSGA-II have tried noticeably more "bad" configurations. In specific, 45% and 32% of the auto-tuning time undergoes a throughput lower than 100 kop/s for ECM and NSGA-II, respectively.

GA works by assigning the probability of surviving to the next generation based on the fitness value (i.e., throughput). Configurations with lower throughput values have a lower chance to be picked as parents, thus their genes (parameter values) have a lower chance to survive in the next generation. SMPSO can recognize and discard the unpromising configurations, because it chooses new configurations by considering the previous one, group best, and particle best solutions (see Section II). The combination of these three components help SMPSO keep focusing on the promising configurations.

The throughput degradation of ECM and NSGA-II is due to the fact that the two algorithms, as typical multi-objective optimization methods, consider both throughput and latency simultaneously when selecting the next generation genes. All Pareto optimal configurations are treated equally good and selected. The difference between ECM and NSGA-II is that ECM also considers the latency constraints, and only the configurations that can meet the latency requirements will be selected. As a consequence, they do not always try to maximize one single target, such as throughput in Figure 13.

Different from the other four algorithms, WSM shows a totally different behaviors. WSM achieves significantly lower instantaneous throughput than the others. Specifically, 77% of the tested configurations provide a throughput lower than 100 kop/s. It means that WSM gives configurations that provide moderate throughput more opportunities to survive in the next generation. That is because in WSM, the optimization targets have been combined into a single target as defined in Equation 6. WSM tries to balance multiple goals, including throughput and tail latency, rather than a single target.

> Finding#7: When exploring the configuration space, the MOO algorithms choose the candidate configurations for testing by making different tradeoff decisions between multiple objectives, which leads to distinct instantaneous and final performance.

D. Summary: Guidance for Algorithm Selection

The five selected optimization algorithms show different behaviors when auto-tuning RocksDB with various workloads and hardware setups. Our observations show that no algorithm outperforms the others in all the cases. Although the main goal of this work is not to identify certain universally good algorithm, we can provide some rules of thumb as guidance for selecting an algorithm for auto-tuning.

First, ECM should be among the top picks when users can provide specific QoS requirements. ECM is the most time-consuming method among the selected five approaches, however, it outperforms the others in finding the proper solutions when specific QoS requirements are set. Second, both NSGA-II and SMPSO can be considered when users cannot provide explicit preferences for their optimization objectives. They can achieve similar performance in finding the optimal or near-optimal configurations with multiple hardware settings, although they simulate different evolutionary processes. Third, WSM is generally a sub-optimal choice when multiple optimization objectives are semantically different. The behaviors of WSM are hard to predict when auto-tuning RocksDB, since it shows significantly different behaviors with various hardware settings in our experiments. Fourth, GA should be evaluated before being applied for multi-objective optimizations. Simply optimizing one single objective, such as throughput, may not be sufficient when other metrics, such as power consumption, also need to be considered.

V. Impact of Hyper-Parameters

In the multiple-objective optimization methods, the hyper-parameters, i.e., the optimization algorithms' own parame-

ters, play an important role in determining the total auto-tuning time. In this section, we discuss the impact of hyper-parameters from the perspective of exploitation and exploration (see Section II-D).

A. Effect of Exploitation: Leader Size in SMPSO

In SMPSO, *leader size* means the number of the best particles at current stage, which determines the selection pressure for the next movement. Typically, a higher selection pressure, caused by a smaller leader size, pushes the search toward exploitation and expects a shorter tuning time. However, according to Figure 14, leader size does not have a noticeable influence on the overall tuning time. We have conducted the experiments by changing the leader size from 4 to 64 and the total tuning time remains almost the same. For example, 90% of the experiments can reach the near-optimal configuration within 25.6 hours. This is because some alleles (parameter values) play a dominant role in determining the performance. These alleles will be selected even though only four leaders are selected during the selection process. These surviving alleles further produce offsprings which achieve good performance. We have a detailed discussion about alleles in Section VI-1.

B. Effect of Exploration: Mutation Rate in NSGA-II

Figure 15 shows that mutation rate in genetic algorithm plays a significant role for determining the total tuning time. For example, when mutation rate is 1%, the total time consumed to find the (near-)optimal configuration is 15.9 hours for 90% of the experiments. As we increase the mutation rate to 64%, the tuning time increases to 28 hours for 90% of the runs. Typically, a mutation operator modifies genes in each individual in a random manner with a given probability. Thus the increasing structural diversity of the population pushes the search toward exploration. However, a high mutation rate (64%) causes the "good" gene disappear easily in the next generation, while a relative low mutation rate achieves a better balance for gene diversity and stability.

> Finding#8: Since only a few alleles play dominant role in determining the system performance, a relatively small exploitation space is sufficient for finding a satisfactory configuration, while a large exploration space may cause performance degradation when auto-tuning RocksDB.

VI. Insight and System Implications

We have studied the auto-tuning behaviors of popular multi-objective optimization methods for RocksDB on SATA and PCIe SSDs. Despite successful deployment of these algorithms on auto-tuning complex key-value systems, little is known how and why some approaches work better than others for certain target. In this section, we attempt to open the "black-box" algorithms and gain insight into their internals based on the behaviors of these algorithms and our knowledge about the key-value systems. We further present several important system implications for designers and practitioners to effectively optimize RocksDB based on their Quality of Services requirements.

1) Alleles: It is expected that as the auto-tuning process moves forward, there will be some alleles (parameter values) dominant in the population (configurations). We present the alleles of genetic algorithm in Figure 16 as an example to demonstrate the evolution of each system parameter. The Y-axis shows 6 genes (parameters) as listed in Table II, while each row represents one allele (parameter value). The X-axis shows the evolution over the first 30 generations. Each cell is colored based on the frequency of alleles appearing in each generation. The darker colors indicate more frequent alleles.

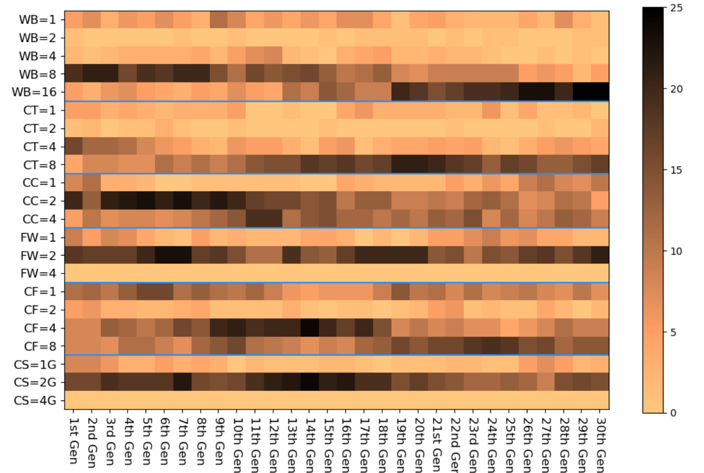

Fig. 16: Alleles of Genetic Algorithm.

In the first generation, the number of write buffers being 8 and concurrent thread number being 4 are dominant. However, as the evolution proceeds, write buffer number being 16 and concurrent thread number being 8 becomes more dominant than other alleles. Since GA simulates the natural selection process where alleles with better fitness are more likely to survive, this indicates that GA prefers more write buffers and more concurrent threads when optimizing the throughput. For some other parameters, such as concurrent compactors and cleanup file number, their alleles become more diverse as the evolution proceeds, which means that they do not have significant impact to the overall throughput, compared to write buffer size and concurrent thread number.

> Finding#9: As auto-tuning proceeds, the most dominant alleles become clear for most parameters, such as write buffer, concurrent threads and flush writers, etc.

2) Importance of Parameters: Based on the prior observations, an interesting question is what is the impact of each parameter to the overall performance of RocksDB. To answer this question, we have quantitatively studied the correlations between each parameter and two performance metrics (throughput and the 99th percentile tail latency).

As the parameters we have studied in this work are all discrete numbers, whereas the throughput is continuous, we

Metric	Machine ID	Device	CPU Cores	MEM	WL	WB	CT	CC	FW	CF	CS
Throughput	M1	PCIe SSD	4	8 GB	Zipfian	0.91	0.79	-	-	-	-
Throughput	M1	PCIe SSD	4	8 GB	Random	0.90	0.96	-	-	-	-
Throughput	M2	PCIe SSD	32	8 GB	Zipfian	0.91	0.87	-	-	-	-
Throughput	M3	PCIe SSD	32	16 GB	Zipfian	0.91	0.74	-	-	-	-
Throughput	M3	PCIe SSD	32	16 GB	Hotspot	0.92	0.89	-	-	-	-
Throughput	M4	SATA SSD	32	16 GB	Zipfian	0.93	0.68	-	-	-	-
99th Percentile Tail Latency	M1	PCIe SSD	4	8 GB	Zipfian	0.92	0.65	-	-	-	-
99th Percentile Tail Latency	M1	PCIe SSD	4	8 GB	Random	0.77	0.86	-	-	-	-
99th Percentile Tail Latency	M2	PCIe SSD	32	8 GB	Zipfian	0.81	0.82	-	-	-	-
99th Percentile Tail Latency	M3	PCIe SSD	32	16 GB	Zipfian	0.66	0.89	-	-	-	-
99th Percentile Tail Latency	M3	PCIe SSD	32	16 GB	Hotspot	0.84	0.70	-	-	-	-
99th Percentile Tail Latency	M4	SATA SSD	32	16 GB	Zipfian	0.54	0.85	-	-	-	-

TABLE IV: Importance of Parameters (measured by R^2).

have taken a widely used approach to calculate the correlation between discrete and continuous values [12]. We illustrate with the Concurrent Threads (CT) as an example. We set Concurrent Threads with 4 values (1,2,4,8) in our experiments. We convert this parameter into 4 binary variables: x_1, x_2, x_3 and x_4. If the thread number is set to be 1, we assign $x_1 = 1$ and x_2, x_3 and x_4 are set to be 0. Let Y represent the corresponding throughput values. We then do a linear regression with Ordinary Least Squares (OLS) on Y and x_1, x_2, x_3 and x_4. R^2 is a commonly used metric to measure how data fits a regression line [11], [53]. In our approach, R^2 measures the the correlations between the selected parameter and the received performance (throughput and tail latency). Typically, $R^2 > 0.6$ is an indicator that the parameter has significant impact on the performance. Parameters with highest R^2 are colored green in Table IV. To find the second important parameter, the same process is applied to the remaining parameters, but with the first important parameter being fixed. For example, we calculate R^2 respectively by setting CT to be 1, 2, 4 and 8. We take the highest value as the R^2 value for this parameter. The second important parameters are colored blue.

We have conducted experiments with different hardware setups and workloads. As Table IV shows, the correlated parameters remain the same across different platforms. Concurrent thread number and write buffer play the most important roles in determining throughput and tail latency. However, we also note that the number of concurrent threads affects the tail latency more significantly than the throughput, while the number of write buffers has a stronger influence on the throughput than on the tail latency.

> Finding#10: Write buffer and concurrent thread number play dominant roles in determining the system performance across hardware and workloads.

3) Implications to RocksDB Design: To further look into RocksDB's system design and verify our findings, we have plotted the 99th percentile tail latency and throughput to illustrate the effect of concurrent threads, write buffers and compaction threads on M3 in Figure 17, 18, and 19. Each dot on the figures represents one configuration.

We can see in Figure 17 that configurations with more threads tend to produce a higher maximum throughput at a cost of a higher tail latency. However, the performance ranges have intersections when thread number differs. Furthermore, we find that the corresponding set of dots (purple) for 8 threads has a higher variance than the other sets. That means that when more threads are used, other parameters tend to play an increasingly more important role in determining the performance.

Figure 18 shows the performance with different numbers of write buffers. We can see that more write buffers provide a relatively higher throughput, while not necessarily increasing the tail latency. Also, the changes of throughput and tail latency when changing write buffer number is not as significant as the changes with the number of concurrent threads.

Finally, Figure 19 shows the performance difference when changing the number of background compaction operations. We can observe an obvious throughput improvement when increasing compaction operation number from 1 to 2. However, the throughput does not have an obvious change when the parameter value further increases from 2 to 4. That is because compaction is an expensive background operation in RocksDB, and the further increase of the background operations has a diminishing effect on the foreground processes. That explains why this parameter is not chosen as the most influential ones in Table IV. Our observations about the performance trend in this Section (Sec VI-3) are consistent with our analysis about the optimization algorithms in Section VI-1 and Section VI-2. These findings are helpful for system designer to optimize and tune the system.

> Finding#11: Selecting a proper user-level parallelism degree and setting suitable write buffer size will be the top choices for system designers and practitioners to balance multiple objectives.

> Finding#12: Other parameters, such as the number of flush and compaction processes, have noticeable impact to the performance, but the tuning space can be restricted.

VII. LIMITATIONS AND FUTURE WORK

In this paper, we have presented the first comprehensive study on the multi-objective optimization algorithms on RocksDB. Auto-tuning the increasingly complex key-value system is a challenging task. There are several works that

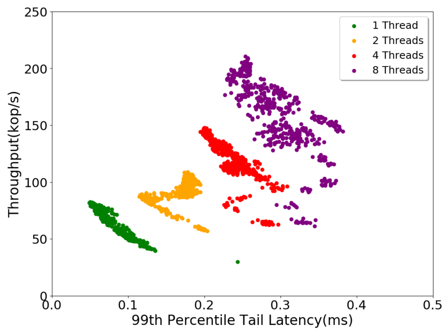

Fig. 17: Effect of Concurrent Threads.

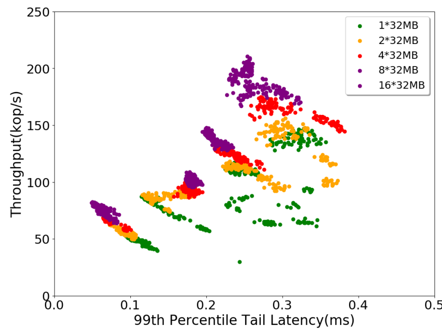

Fig. 18: Effect of Write Buffer.

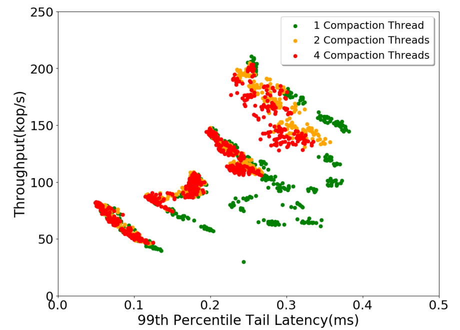

Fig. 19: Effect of Compaction Threads.

remain worth being explored in the future. (1) *Optimization targets.* In this work, we mainly focus on throughput and tail latency, two major optimization objectives in real-world key-value system deployment. Other metrics, such as power consumption and capital cost, are also of practical interests. We plan to expand the work to also consider these optimization objectives in the future work. (2) *Workloads and parameter space.* In this paper, we collect the experimental data with synthesized workloads, following the representative zipfian, hotspot and random distributions. We plan to further explore the possibility of repeating real enterprise workloads on our platform. Moreover, we currently focus on six most influential parameters in terms of performance. When including power consumption as our optimization objective, the parameter selection may need to change. We plan to extend the parameter space, according to the optimization target and our evaluation results, in the future. (3) *Algorithm improvement.* In this paper, we have discussed the impact of hyper-parameters in the algorithms. Based on our findings, we plan to improve the traditional multi-objective optimization algorithms to make them more robust for different workloads, hardware, and optimization targets, by integrating them with new techniques, such as penalty function or re-initialization, etc.

VIII. Related Work

Auto-tuning computer systems has been extensively studied. Prior works can be roughly divided into two categories, single- and multiple-objective auto-tuning.

Single Objective Tuning. Auto-tuning technologies have been widely studied to maximize one single objective, such as throughput, in complex computer systems. For example, Behzad *et al.* [9] propose to apply GA to HDF5 applications to improve I/O performance. More recently, Li *et al.* [37] aim to optimize Lustre with neural network-based deep reinforcement learning. Aken *et al.* [5] use supervised and unsupervised machine learning methods to identify (near-)optimal configurations for database management systems. Alipourfard *et al.* [6] try to find the best configuration for big data analytics in cloud. Besides, GA is also used for other purposes, such as storage system provisioning [50] and recovery [32]. Rafiki [40] tires to tune the parameters of NoSQL database, such as Cassandra and ScyllaDB, for HPC and dynamic metagenomics workloads. Cao *et al.* [11] compare multiple black-box single-objective auto-tuning approaches for storage systems. All these prior works try to optimize one single goal. Our work presents the first study of multi-objective auto-tuning for RocksDB.

Multiple Objective Tuning. Multi-objective optimization (MOO) approaches have been widely studied in various fields [10], [15], [18], [27], [29], [33], [45], [48], [51]. Prior studies also have applied multi-objective auto-tuning techniques on computer systems [21], [25], [31], [35]. Durillo *et al.* have discussed the advantages and drawbacks of existing single-objective and multiple-objective auto-tuning algorithms [21]. Gschwandtner *et al.* have applied multi-objective auto-tuning to parallel applications for optimizing execution time, energy and resource usage simultaneously [25]. Jordan *et al.* introduce framework to auto-tune compiler and run-time components to optimize run-time and efficiency [31]. Kofler *et al.* try to enhance traditional multi-objective algorithm with region division [35]. PSLO [36] attempts to enforce the tail latency and throughput SLOs for consolidated virtual machine (VM) storage by coordinating I/O concurrency level and arrival rate for each VM issue queue. Starfish [26] aims to tune the Hadoop software stack for big data analytics by considering the resource utilization, time and monetary cost, etc. ACIC [39] performs performance/cost predictions by using machine learning techniques for cloud systems and HPC applications to balance execution time and monetary cost. Compared to traditional computer systems, key-value systems are designed particularly for providing high-speed data services, which demands to meet several strictly defined SLOs, such as throughput and tail latency, etc. Some special storage management mechanisms, such as the compaction process, in the underlying LSM-tree data structure also make key-value system's performance behaviors unique and sensitive to the system configurations. In this paper, we focus on auto-tuning RocksDB, a popular key-value data store system, and have gained important insight, providing important guidance to system designers and practitioners for future optimizations.

IX. Conclusion

Auto-tuning system configurations for key-value stores is important for achieving crucial performance goals, such as throughput and tail latency. In this paper, we have conducted a comprehensive study to understand the behaviors of multi-objective optimization algorithms for auto-tuning RocksDB. We have also discussed the instantaneous performance and the impact of hyper-parameters of the auto-tuning approaches

from the perspective of exploitation and exploration. Based on our observations, we have also presented the associated system implications for designers and practitioners in future optimizations. We believe the methodology developed in this work can also be applied to other optimization targets and systems.

Acknowledgments

We thank our shepherd, Dr. Changwoo Min, and the anonymous reviewers for their insightful comments and valuable suggestions. This work was partially supported by the U.S. National Science Foundation under Grants CCF-1453705, CCF-1629291, and CCF-1910958.

References

[1] Platypus. https://github.com/Project-Platypus/Platypus, 2019.

[2] Apache Cassandra. http://cassandra.apache.org/, 2020.

[3] MySQL. https://www.mysql.com/, 2020.

[4] RocksDB. https://rocksdb.org/, 2020.

[5] D. V. Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*, pages 1009–1024, New York, NY, USA, 2017. ACM.

[6] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *Proceedings of 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*, pages 469–482, Boston, MA, Mar. 2017. USENIX Association.

[7] B. Almeida and V. Leite. Particle Swarm Optimization: A Powerful Technique for Solving Engineering Problems. 12 2019.

[8] D. Beasley, D. R. Bull, and R. R. Martinz. An Overview of Genetic Algorithms : Part 1, Fundamentals. *University Computing*, 15(2):58–69, 1993.

[9] B. Behzad, H. V. T. Luu, J. Huchette, S. Byna, Prabhat, R. Aydt, Q. Koziol, and M. Snir. Taming Parallel I/O Complexity with Auto-tuning. In *Proceedings of 2013 International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*, pages 1–12, Nov 2013.

[10] L. T. Bui, S. Alam, L. T. Bui, and S. Alam. *Multi-Objective Optimization in Computational Intelligence: Theory and Practice*. IGI Global, Hershey, PA, USA, 1 edition, 2008.

[11] Z. Cao, V. Tarasov, S. Tiwari, and E. Zadok. Towards Better Understanding of Black-box Auto-tuning: A Comparative Analysis for Storage Systems. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '18)*, pages 893–907. USENIX Association, July 11-13 2018.

[12] G. Casella and R. L. Berger. *Statistical Inference*, volume 2. Duxbury Pacific Grove, CA, 2002.

[13] F. Chen, D. A. Koufaty, and X. Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives. In *Proceedings of the 11th International Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '09)*, pages 181–192, New York, NY, USA, 2009. ACM.

[14] F. Chen, R. Lee, and X. Zhang. Essential Roles of Exploiting Internal Parallelism of Flash Memory based Solid State Drives in High-Speed Data Processing. In *Proceedings of the 17th IEEE International Symposium on High-Performance Computer Architecture (HPCA '11)*, San Antonio, TX, February 12-16 2011.

[15] D. Craft, T. Halabi, H. Shih, and T. Bortfeld. Approximating Convex Pareto Surfaces in Multiobjective Radiotherapy Planning. *Medical Physics*, 33:3399–407, 10 2006.

[16] J. Dean and L. A. Barroso. The Tail at Scale. *Communications of the ACM*, 56:74–80, 2013.

[17] K. Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley Sons, Inc., USA, 2001.

[18] K. Deb and R. Datta. Hybrid Evolutionary Multi-objective Optimization and Analysis of Machining Operations. *Engineering Optimization*, 44(6):685–706, 2012.

[19] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, April 2002.

[20] S. Duan, V. Thummala, and S. Babu. Tuning Database Configuration Parameters with iTuned. In *Proceedings of the 2009 Very Large Data Bases (VLDB '09)*, August 24-28 2009.

[21] J. Durillo and T. Fahringer. From Single-to Multi-objective Auto-tuning of Programs: Advantages and Implications. *Scientific Programming*, 22(4):285–297, Oct. 2014.

[22] Facebook. DBBench. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools, 2019.

[23] B. F.Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud computing (SoCC '10)*, Indianapolis, Indiana, June 10-11 2010.

[24] S. Ghemawat and J. Dean. LevelDB. https://github.com/google/leveldb, 2020.

[25] P. Gschwandtner, J. J. Durillo, and T. Fahringer. Multi-Objective Auto-Tuning with Insieme: Optimization and Trade-Off Analysis for Time, Energy and Resource Usage. In F. Silva, I. Dutra, and V. Santos Costa, editors, *Proceedings of 2014 Europe Parallel Processing (Euro-Par '14)*, pages 87–98, Cham, 2014. Springer International Publishing.

[26] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. Cetin, and S. Babu. Starfish: A Self-tuning System for Big Data Analytics. In *Proceedings of 5th Biennial Conference on Innovative Data Systems Research (CIDR '11)*, pages 261–272, 01 2011.

[27] N. Hochstrate, B. Naujoks, and M. Emmerich. SMS-EMOA: Multiobjective Selection based on Dominated Hypervolume. *European Journal of Operational Research*, 181:1653–1669, 02 2007.

[28] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding Up Distributed Request-response Workflows. In *Proceedings of the ACM Conference on SIGCOMM (SIGCOMM '13)*, pages 219–230, New York, NY, USA, 2013. ACM.

[29] S. Jena, P. Patro, and S. S. Behera. Multi-Objective Optimization of Design Parameters of a Shell Tube type Heat Exchanger using Genetic Algorithm. *International Journal of Current Engineering and Technology*, 3(4):1379–1386, 2013.

[30] Y. Jia, Z. Shao, and F. Chen. SlimCache: Exploiting Data Compression Opportunities in Flash-based Key-value Caching. In *Proceedings of 2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '18)*, pages 209–222. IEEE, 2018.

[31] H. Jordan, P. Thoman, J. J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer, and H. Moritsch. A Multi-objective Auto-tuning Framework for Parallel Codes. In *Proceedings of 2012 International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*, pages 1–12, Nov 2012.

[32] K. Keeton, D. Beyer, E. Brau, A. Merchant, C. Santos, and A. Zhang. On the Road to Recovery: Restoring Data After Disasters. *SIGOPS Oper. Syst. Rev.*, 40(4):235–248, Apr. 2006.

[33] I. Y. Kim and O. L. de Weck. Adaptive Weighted-sum Method for Bi-objective Optimization: Pareto Front Generation. *Structural and Multidisciplinary Optimization*, 29:149–158, 2005.

[34] A. Klimovic, H. Litz, and C. Kozyrakis. Selecta: Heterogeneous Cloud Storage Configuration for Data Analytics. In *Proceeding of 2018 USENIX Annual Technical Conference (USENIX ATC '18)*, pages 759–773, Boston, MA, 2018. USENIX Association.

[35] K. Kofler, J. J. Durillo, P. Gschwandtner, and T. Fahringer. A Region-Aware Multi-Objective Auto-Tuner for Parallel Programs. In *Proceedings of 2017 46th International Conference on Parallel Processing Workshops (ICPPW' 17)*, pages 190–199, Aug 2017.

[36] N. Li, H. Jiang, D. Feng, and Z. Shi. PSLO: Enforcing the Xth Percentile Latency and Throughput SLOs for Consolidated VM Storage. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*, pages 28:1–28:14, New York, NY, USA, 2016. ACM.

[37] Y. Li, K. Chang, O. Bel, E. L. Miller, and D. D. E. Long. CAPES: Unsupervised Storage Performance Tuning Using Neural Network-based Deep Reinforcement Learning. In *Proceedings of 2017 International Conference for High Performance Computing, Networking, Storage and Analysis (SC' 17)*, pages 42:1–42:14, New York, NY, USA, 2017. ACM.

[38] Z. L. Li, C.-J. M. Liang, W. He, L. Zhu, W. Dai, J. Jiang, and G. Sun. Metis: Robustly Tuning Tail Latencies of Cloud Systems. In *Proceedings*

of 2018 USENIX Annual Technical Conference (USENIX ATC '18), pages 981–992, Boston, MA, 2018. USENIX Association.

[39] M. Liu, Y. Jin, J. Zhai, Y. Zhai, Q. Shi, X. Ma, and W. Chen. ACIC: Automatic Cloud I/O Configurator for HPC Applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*, pages 1–12, 2013.

[40] A. Mahgoub, P. Wood, S. Ganesh, S. Mitra, W. Gerlach, T. Harrison, F. Meyer, A. Grama, S. Bagchi, and S. Chaterji. Rafiki: A Middleware for Parameter Tuning of NoSQL Datastores for Dynamic Metagenomics Workloads. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference (Middleware '17)*, pages 28–40, New York, NY, USA, 2017. ACM.

[41] S. Mardle, S. Pascoe, and M. Tamiz. An Investigation of Genetic Algorithms for the Optimization of Multi-objective Fisheries Bioeconomic Models. *International Transactions in Operational Research*, 7(1):33–49, 2000.

[42] R. T. Marler and J. S. Arora. Survey of Multi-objective Optimization Methods for Engineering. *Structural and Multidisciplinary Optimization*, 26:369–395, 2004.

[43] R. T. Marler and J. S. Arora. The Weighted Sum Method for Multi-objective Optimization: New Insights. *Structural and Multidisciplinary Optimization*, 41:853–862, 2010.

[44] A. J. Nebro, J. J. Durillo, J. Garcia-Nieto, C. A. Coello Coello, F. Luna, and E. Alba. SMPSO: A New PSO-based Metaheuristic for Multi-objective Optimization. In *Proceedings of 2009 IEEE Symposium on Computational Intelligence in Multi-Criteria Decision-Making(MCDM '09)*, pages 66–73, March 2009.

[45] G. O. Odu and O. E. Charles-Owaba. Review of Multi-criteria Optimization Methods – Theory and Applications. *IOSR Journal of Engineering (IOSRJEN)*, 3:01–14, 2013.

[46] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The Log-structured Merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[47] T. Papadakis. Skiplist. https://en.wikipedia.org/wiki/Skip_list, 1993.

[48] M. Sessarego, K. Dixon, D. Rival, and D. Wood. A Hybrid Multiobjective Evolutionary Algorithm for Wind-turbine Blade Optimization. *Engineering Optimization*, 47(8):1043–1062, 2014.

[49] C. Stewart, A. Chakrabarti, and R. Griffith. Zoolander: Efficiently Meeting Very Strict, Low-Latency SLOs. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC '13)*, pages 265–277, San Jose, CA, 2013. USENIX Association.

[50] J. D. Strunk, E. Thereska, C. Faloutsos, and G. R. Ganger. Using Utility to Provision Storage Systems. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST '08)*, pages 21:1–21:16, Berkeley, CA, USA, 2008. USENIX Association.

[51] E. Triantaphyllou, B. Shu, S. N. Sanchez, and T. Ray. Multi-Criteria Decision Making: An Operations Research Approach. *Encyclopedia of Electrical and Electronics Engineering*, 15:175–186, 1998.

[52] D. Wang, D. Tan, and L. Liu. Particle Swarm Optimization Algorithm: An Overview. *Soft Computing*, 22:387–408, January 2018.

[53] Wikipedia. Coefficient of Determination. https://en.wikipedia.org/wiki/Coefficient_of_determination.

[54] Wikipedia. Multi-objective Optimization. https://en.wikipedia.org/wiki/Multi-objective_optimization.

[55] Z. Wu, C. Yu, and H. V. Madhyastha. CosTLO: Cost-effective Redundancy for Lower Latency Variance on Cloud Storage Services. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI '15)*, pages 543–557, Berkeley, CA, USA, 2015. USENIX Association.

[56] Z. Yang, X. Cai, and Z. Fan. Epsilon Constrained Method for Constrained Multiobjective Optimization Problems: Some Preliminary Results. In *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation (GECCO Comp '14)*, pages 1181–1186, New York, NY, USA, 2014. ACM.

[57] Y. Yusoff, M. Ngadiman, and A. Zain. Overview of NSGA-II for Optimizing Machining Process Parameters. *Procedia Engineering*, 15, 08 2011.

[58] G. K. Zipf. Relative Frequency as a Determinant of Phonetic Change. *Harvard Studies in Classical Philology*, 40:1–95, 1929.