Dremel: Adaptive Configuration Tuning of RocksDB KV-Store

CHENXINGYU ZHAO, University of Washington, USA
TAPAN CHUGH, University of Washington, USA
JAEHONG MIN, University of Washington, USA
MING LIU, University of Wisconsin-Madison, USA
ARVIND KRISHNAMURTHY, University of Washington, USA

LSM-tree-based key-value stores like RocksDB are widely used to support many applications. However, configuring a RocksDB instance is challenging for the following reasons: 1) RocksDB has a massive parameter space to configure; 2) there are inherent trade-offs and dependencies between parameters; 3) right configurations are dependent on workload and hardware; and 4) evaluating configurations is time-consuming. Prior works struggle with handling the curse of dimensionality, capturing relationships between parameters, adapting configurations to workload and hardware, and evaluating quickly.

In this work, we present a system, Dremel, to adaptively and quickly configure RocksDB with strategies based on the Multi-Armed Bandit model. To handle the massive parameter space, we propose using fused features, which encode domain-specific knowledge, to work as a compact and powerful representation for configurations. To adapt to the workload and hardware, we build an online bandit model to identify the best configuration. To evaluate quickly, we enable multi-fidelity evaluation and upper-confidence-bound sampling to speed up identifying the best configuration. Dremel not only achieves up to ×2.61 higher IOPS and 57% less latency than default configurations but also achieves up to 63% improvements over prior works on 18 different settings with the same or less time budget.

CCS Concepts: • **Information systems** → **Key-value stores**; • **Computing methodologies** → *Modeling methodologies*; *Active learning settings.*

Additional Key Words and Phrases: RocksDB, configuration management, multi-armed bandit

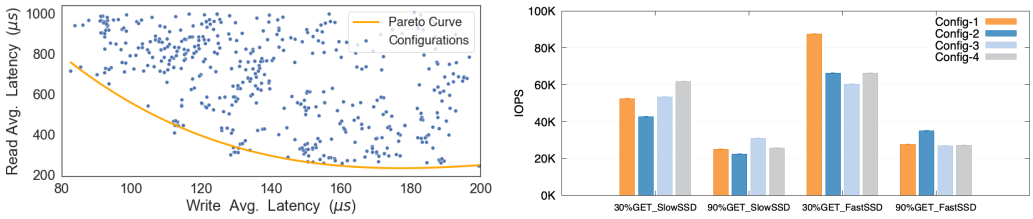
ACM Reference Format:
Chenxingyu Zhao, Tapan Chugh, Jaehong Min, Ming Liu, and Arvind Krishnamurthy. 2022. Dremel: Adaptive Configuration Tuning of RocksDB KV-Store. *Proc. ACM Meas. Anal. Comput. Syst.* 6, 2, Article 37 (June 2022), 30 pages. https://doi.org/10.1145/3530903

1 INTRODUCTION

Persistent key-value stores based on the Log-Structured Merge-tree (LSM-Tree) are widely used in Facebook RocksDB [28], Google LevelDB [32], Amazon DynamoDB [20], Alibaba X-Engine [35], Apache Cassandra [47], WiredTiger [56], TiDB [34], and more. Among them, RocksDB is an industry-standard open-source implementation of the LSM-Tree that provides high performance and versatility with flexible tunability. However, as cited from the developers' tuning guide, optimally configuring RocksDB is not trivial even for experts [12, 25, 28] due to the following challenges.

Authors' addresses: Chenxingyu Zhao, cxyzhao@cs.washington.edu, University of Washington, Seattle, WA, USA; Tapan Chugh, tapanc@cs.washington.edu, University of Washington, Seattle, WA, USA; Jaehong Min, jaehongm@cs.washington.edu, University of Washington, Seattle, WA, USA; Ming Liu, mgliu@cs.wisc.edu, University of Wisconsin-Madison, Madison, WI, USA; Arvind Krishnamurthy, arvind@cs.washington.edu, University of Washington, Seattle, WA, USA.

2476-1249/2022/6-ART37
https://doi.org/10.1145/3530903



(a) Trade-off curve and sub-optimal configurations

(b) No config is always best

Fig. 1: Challenges for Tuning RocksDB

Challenge 1 – Massive configuration space: For RocksDB (v6.27.3), its header file *option.h* supports up to 83 parameters to configure. Further, most parameters have a wide range of possible values, sometimes from large continuous ranges (e.g., write_buffer_size can vary from KB to MB). The resulting curse of dimensionality makes it hard for tuning methods to search such a large configuration space with a limited time budget.

Challenge 2 – Fundamental configuration trade-offs: RocksDB is, essentially, a data structure for organizing and accessing data, facing well-known and inherent trade-offs between read performance, write performance, and space costs [17, 18, 33, 38]. Figure 1(a) shows the trade-off between read and write performance as we randomly select about 500 configurations to run over the same workload. We observe that there is an optimal Pareto curve for the read-write performance trade-off. Beyond the Pareto curve, we cannot improve read or write performance without hurting the other. It is worth noting that many configurations don't achieve Pareto optimally (i.e., they are not on the Pareto curve). Besides striking a good trade-off, configuration tuning is also responsible for filtering out sub-optimal configurations. Many RocksDB parameters control the above trade-off, and identifying the optimal configuration in such settings is challenging.

Challenge 3 – Being adaptive to workloads and hardware: The need to support diverse workloads and hardware further complicates configuration tuning. As Figure 1(b) shows, we select four configurations and four different workload/hardware settings (30%/90% GET ratio, fast/slow storage device). We can observe that the best configurations for these settings are different. No one configuration is always best.

For RocksDB, there are three scenarios in which adaptivity is desired: 1) First case is the initial setup of a RocksDB instance. RocksDB is widely deployed for production applications [12, 24, 25] that have different workload characteristics, infrastructure environments, and resource requirements. Configuring RocksDB corresponding to a specific application is a crucial step to launch a new instance. 2) Workload changes are common in production. Based on Facebook's deployment experience [12], instances show a strong diurnal pattern of 24-hours. For example, for serving a social network, GET/PUT ratio usually reaches a peak of 4:1 at about 17:00 (people read content during off-work time). For working hours, GET/PUT ratio is usually 2:1. 3) Hardware heterogeneity is also common. LSM-based KV-stores are increasingly deployed over diverse hardware environments like cloud environment [35], disaggregated storage [55], and ZNS SSD [58]. Further, hardware conditions can also change [55], so the system needs to adapt to current hardware characteristics.

Challenge 4 – Expensive evaluation: A typical approach to evaluating a configuration is executing the desired workload (e.g., inserting/reading millions of kv-pairs) and measuring a metric (e.g., IOPS). This is expensive in terms of time cost, typically lasting tens of minutes for each configuration. Having a large configuration space further increases the evaluation cost.

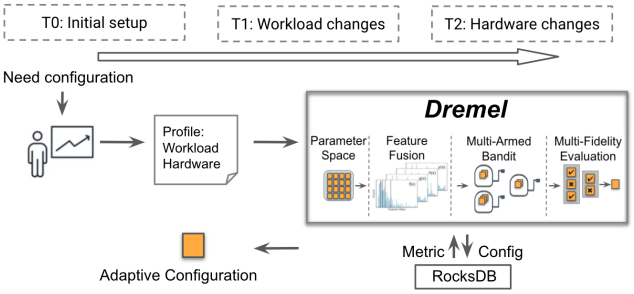


Fig. 2: Architecture of Dremel- a system to adaptively configure RocksDB.

Prior work and limitations: One line of work tries to navigate the design and tuning of LSM-tree-based KV-stores by modeling the operation cost with analytical primitives or closed-form expressions [17–19, 38, 51]. However, these models encode very few or none of the factors affected by workload and hardware, so they are of limited value in recommending an appropriate configuration for a given deployment setting. The second line of work tries to tune configurations by modeling the relationship between parameters and performance as a black-box function and then optimizing parameters by learning to fit the black-box function with some general optimization framework like Bayesian Optimization and Reinforcement Learning [1, 49, 67]. Given the massive number of configurations, inherent complexities, and expensive evaluation costs, it is difficult to fit the black-box function well with limited optimization steps. The third line of work applies the data-driven approach like collecting a large number of traces and then applying machine-learning or optimization algorithms over training sets [40, 67]. Considering the dimensionality of the parameter space and the effects from workload/hardware, collecting data is extremely time-consuming (e.g., needing six months [40]), which is not practical for most cases.

Our goal is to design a system that can quickly identify a RocksDB configuration that achieves high performance while adapting to specific workload and hardware conditions. Our system, ***Dremel***, addresses the above challenges in the following ways (illustrated in Figure 2):

• First, we use insights from prior characterizations of storage systems that identify a fundamental trade-off to strike between read, write, and space costs. We use domain-specific knowledge to distill a large number of raw features into a small number of *fused features* that govern the trade-off between read, write, and space costs. These fused features can then express configurations more concisely and reduce the search space.

• Second, to allow RocksDB to be adaptive to workloads and hardware conditions, we adopt online tuning following the principle of Lazy Tuning [37]; if one executes tuning decisions after actual workload and hardware are seen, tuning decisions can better correspond to the specific setting. Our online tuning is based on a Multi-Armed Bandit model to identify good configurations represented by fused features. Each arm is a cluster of configurations that have the same normalized feature values. Then, we formalize the process of identifying good configurations as the problem of best-arm identification, a classical bandit problem. For the identified best-arm comprising a cluster of configurations, we sample configurations from them by Upper-Confidence-Bound sampling.

• Third, to speed up the evaluation of configurations, we use the strategy of Multi-fidelity evaluation. Multi-fidelity evaluation is an approximation strategy that stops poor configurations early and saves time to explore more promising configurations.

Our evaluations show that Dremel achieves up to ×2.61 higher IOPS than default configurations while tuning to maximize IOPS and 57% less latency while tuning to minimize latency. We compare Dremel against four common tuning approaches on 18 different settings. While maximizing IOPS, Dremel achieves up to 63% improvement over the prior work with the same or less time budget.

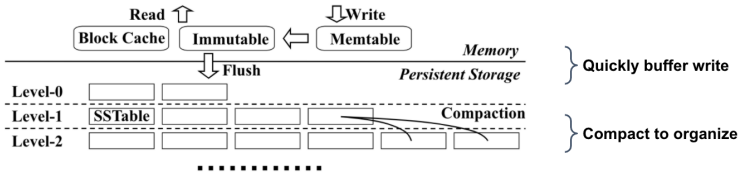


Fig. 3: LSM-Tree Architecture. Memory component and Level 0 are to quickly buffer write and persist data in a sequential manner. Other levels of storage component are organized by compaction to serve read.

2 BACKGROUND

2.1 LSM-tree-based KV-stores

Architecture: A Log-structured Merge-tree (LSM-Tree) [59] (illustrated in Figure 3) is a data structure to store key-value pairs in persistent storage (e.g., SSD) and consists of two main components: 1) Memory component organizes data as *Memtables* that are in-memory data structures (e.g., skip-lists) to buffer recent updates. Once the Memtable's capacity is reached, it becomes immutable and is replaced by a new one. Immutable Memtable waits for a background job to flush it out to the persistent storage. 2) Storage component organizes data as Sorted Sequence Tables (*SSTables*) that are files of sorted key-value pairs. When an Immutable Memtable is flushed out, its entries are sorted and persisted as an SSTable. In persistent storage, SSTables are structured in a series of levels $L_0, L_1, ..., L_n$. SSTables within the same level are maintained in a sorted order, with a disjoint key-range for each SSTable file (except in L_0, where key-ranges of SSTables are allowed to overlap with each other). Generally, from lower levers to higher levels (from L_0 to L_n), the number of SSTables grow exponentially with a multiplier factor (denoted by T).

Operations: LSM-Tree allows two types of operations:

1) Client operations: LSM-Tree supports GET/PUT interfaces to read/write KV-pairs with the specified key. For write requests, Memtables immediately buffer the write requests in memory. When Memtables become full, recent write requests are sorted as an SSTable and flushed to storage in sequential batches. By doing this, LSM-Tree exploits the high sequential write bandwidth of persistent storage. For read requests, data retrieval starts from Memtables and a BlockCache, where LSM-Tree caches data in memory for reads. Upon a cache miss, it will lookup multiple SSTables (from L_0 to high levels) until finding the key. To speed up searching, each SSTable has a corresponding Bloom Filter [9] to test whether an entry exists. If Bloom Filter reports negative, we can save the cost of loading the SSTable from storage.

2) Background operations: LSM-Tree has two background operations, Flushing and Compaction, executed by threads disjoint from client operations. Flushing is to flush out Memtables to L_0 in storage. To reduce the processing cost of flushing, SSTables in L_0 are allowed to have overlapping key ranges between each other. Compaction is to merge lower-level SSTables into higher levels. Because lower-level SSTables store newer data, the LSM-Tree deletes obsolete entries (i.e., those updated by newer values) during compaction and reduces the number of SSTables. Specifically, for compaction from L_i to L_{i+1} (L_i->L_{i+1} for short), one SSTable from L_i is selected and then is merged with the SSTables from L_{i+1} that have overlapping key-ranges. After merging, newly generated SSTable is put in L_{i+1}.

Amplification Factors and Trade-offs: LSM tree is, essentially, an *Access Method* [33, 38] that is used to organize and access data. For Access Methods, there is a fundamental trade-off to strike between the read cost (R), the update/write cost (U), and the memory/storage overhead (M), also known as the *RUM Trade-off* [38]. Performance tuning of RocksDB can be viewed as striking a good RUM trade-off that is appropriate for workloads and hardware. Specifically, for an LSM-Tree, there are three amplification factors to write, read, and space costs.

Write Amplification (WA) is defined as the ratio between the total bytes of data written to the storage and the total bytes of user data written to the LSM-Tree. Compaction is the major cause for WA. For L_i->L_{i+1} compaction (L_{i+1} is T times larger than L_i), merging one SSTable results in reading and writing T overlapping SSTables from L_{i+1} on average, which incurs a WA factor of T.

Read Amplification (RA) is defined as the number of disk reads per read request. Read requests are first served by the block cache. Upon a cache miss, a read is processed by searching the SSTables from persistent storage. Before the actual disk IO of an SSTable, a Bloom Filter tests if the requested entry exists in the SSTable. If the Bloom Filter reports positive, the block containing the entry is loaded to the block cache. Thus, RA is determined by the number of SSTables searched, the false positive rate of Bloom Filters, and the block size.

Space Amplification (SA) is defined as the ratio between the size of data on the disk and the actual user data size. LSM-Tree relies on compaction to execute garbage collection (GC) to remove obsolete entries in the LSM-Tree.

2.2 Multi-Armed Bandit

Multi-Armed Bandit (MAB) problem [4, 11, 27, 62] has a long history and has recently received attention in various applications such as hyperparameter tuning for machine learning algorithms, online advertising, and medical trials [13, 21, 26, 30, 42, 50, 57, 63]. MAB is, essentially, a simple but powerful framework for a decision-making process based on observations.

Model: MAB model is comprised of k arms. At each round, a player (agent) selects an arm to pull and receives a reward. For i-th arm ($1 \leq i \leq k$), its reward follows a probability distribution with the mean as μ_i, which is unknown to the player. An arm with the highest mean reward is called the *best arm*. At round t, the player decides which arm to pull depending on the history of observations (selections and rewards) up to time $t-1$. Under the setting of best-arm identification, the player's goal is to identify the best arm as rapidly as possible.

Exploration-Exploitation Trade-off: When making selections, the player faces an Exploration-Exploitation trade-off. On the one hand, the player needs to explore more arms to get more information. On the other hand, the player needs to exploit the seemingly most rewarding arms to reduce uncertainty and concentrate reward bounds. One example strategy [65] for striking the exploration-exploitation trade-off is to choose arm x_t at round t by upper-confidence-bound: $x_t = \text{argmax}\ \mu_{t-1}(x) + \sigma_{t-1}(x)(x \in [k])$ where $\mu_{t-1}(x)$ is the posterior mean reward for arm x based on observations up to time $t-1$ and $\sigma_{t-1}(x)$ is the standard deviation. It always greedily selects the arm with a reasonable upper bound. It strikes the exploration–exploitation trade-off by preferring both two types of arms: arms which are promising to achieve high rewards (large $\mu_{t-1}(x)$) and arms whose reward distribution is uncertain (large $\sigma_{t-1}(x)$);

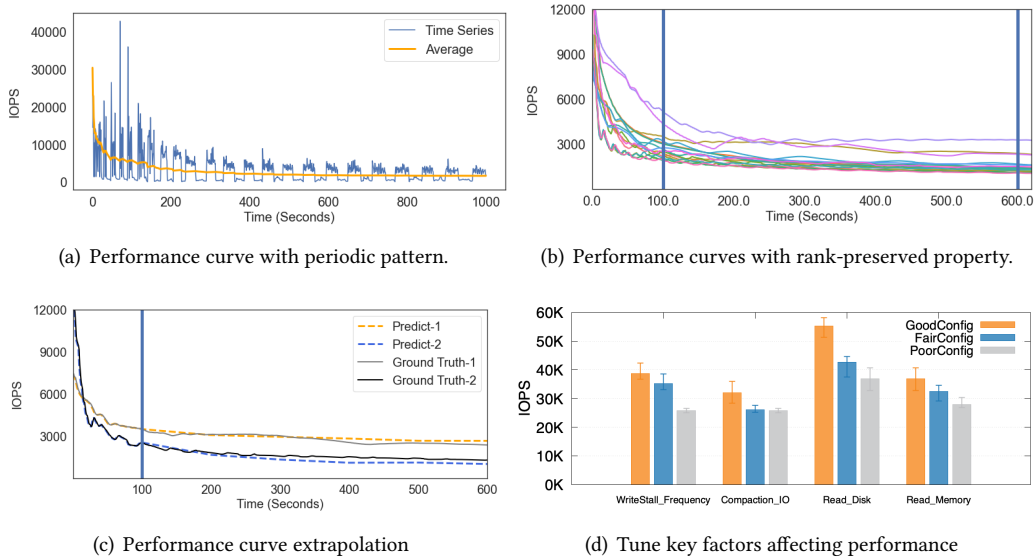
Arm Evaluation: In practice, pulling an arm once and receiving the reward is usually time-consuming. For example, for hyperparameter tuning in machine learning algorithms, pulling one arm (one hyperparameter configuration) means executing cross-validation on a large training set. In many cases, cheap approximations to evaluation are available. For example, a machine learning model's performance can be approximated by training models on a subset of data rather than the whole dataset. Thus, we can apply the *Early-Stopping* strategy [31, 45] to terminate evaluation of poor arms early to reduce the overall time-cost of evaluation.

3 MOTIVATING EXPERIMENTS

3.1 Setting

We use the tool db_bench [28], RocksDB's standard tool, to measure RocksDB's performance and gather insights regarding its behavior. We use the YCSB [14] workload generator implemented by

Balmau et al. [7]. For motivating experiments, we use a write-intensive (70% PUT) and low-skewed workload (Zipf skewness is 0.3) to increase the frequency of compactions and stress the persistent storage component of the LSM-Tree. We use the performance metric *Input/Output Operations Per Second (IOPS)*, which is a critical metric used to measure RocksDB's performance.



(a) Performance curve with periodic pattern.

(b) Performance curves with rank-preserved property.

(c) Performance curve extrapolation

(d) Tune key factors affecting performance

Fig. 4: Motivating Experiments

3.2 RocksDB Performance Curves

Periodicity of Performance Degradation: As Figure 4(a) shows, the IOPS time-series curve (blue line, reporting IOPS every second) exhibits a periodic pattern after the initialization phase while the IOPS average curve (yellow one, average from the start) converges to a stable status. Note that there is an initialization phase for building the structure of an LSM-Tree. The faster the request rate is (e.g., with more application threads), the shorter the initialization phase is. After the initialization phase, we notice that for some intervals, IOPS drop significantly, almost to 0. Prior works on RocksDB [7, 48, 53, 64, 68] also observe a similar pattern which is related to the mechanisms of RocksDB: There is a fixed trigger condition for both flushing and compaction based on the number of filled buffers and L_0 SSTables. Given the application request rate is stable, the filling rate of Memtables and L_0 are also relatively stable, and flushing and compaction would be triggered periodically. Another periodic condition is that flushing and compaction incur write stalls, which arise in two situations: 1) Memtables fill up due to slow flushing. Flushing, which writes Memtables out to the persistent storage, competes for IO bandwidth with compaction and read requests. If flushing is slowed down due to contention on IO bandwidth, more Memtables are filled up but not flushed out in time, which triggers a write stall. 2) L_0 might fill up due to slow compaction. If compaction is too slow, more SSTables will accumulate in L_0 resulting in a write stall. *Observation: RocksDB exhibits periodic performance degradation due to flushing and compaction operations as well as write stalls incurred by Memtables and L_0 being full.*

Rank Preserved: As Figure 4(b) shows, we observe that the average IOPS curves start converging after a short-term initialization phase, implying that long-term performance could be approximated by a short-term observation. We validate such an approximation with the following two methods.

First, we propose a *Rank-Preserved* property for RocksDB configurations: Given n configurations and a reasonable selection of time points $t_1 \leq t_2$, the configuration ranks at time t_2 (sorted by a performance metric such as IOPS) are the same as the ranks at time t_1. We examine its validity using experiments. As Figure 4(b) shows, we sort configurations at two time points, 100 seconds and 600 seconds. We observe that top-3 configurations at 600 seconds are the same as the top-3 at 100 seconds. Further, nine of the top-10 at 600 seconds are among the top-10 at 100 seconds.

Second, inspired by the work on predicting the learning curves for ML training [23, 41, 46], we examine whether we can extrapolate the RocksDB performance curve from the first part of the workload to its remainder. We apply the curve prediction tool [23], which integrates a probabilistic learning curve model, to extrapolate the performance of the first 100 seconds to the performance of a longer workload of about 600 seconds. As Figure 4(c) shows, the extrapolation fits the ground truth very well. *Observation: The performance curves of RocksDB configurations have the rank preserved property. Further, long-term performance can be predicted by extrapolating short-term behavior.*

3.3 Factors Affecting Performance

We now use previous observation and domain-specific knowledge to explore the relevance of certain factors on RocksDB performance. For storage systems, read/write (GET/PUT for RocksDB) performance directly determines IOPS. The performance of read requests is often governed by how often reads are served from memory data structures (i.e., cache hits) or storage (i.e., cache misses). Thus, for reads, we identify two factors: Read-Memory and Read-Disk frequency. For writes, building on the discussion from the previous experiment (Section 3.2), we identify two factors: WriteBuffer-Full-Frequency and Compaction-IO. WriteBuffer-Full-Frequency captures the frequency of write stalls caused by Memtables and L_0 SSTables filling up. Compaction-IO captures the compaction behavior of LSM-Tree and affects write requests from two aspects in contrasting ways. 1) Compaction-IO competes for bandwidth with flushing, thus causing backlogs and write stall; 2) Freeing up L_0 SSTables by compacting them with L_1 SSTables could help avoid write stalls.

We first associate RocksDB configuration parameters with these four factors based on our analysis of RocksDB. For Read-Memory frequency, we select parameters related to managing memory allocations. For Read-Disk frequency, we select parameters related to controlling the level structure of the LSM-Tree. For WriteBuffer-Full-Frequency, we select parameters that control the size of Memtables and L_0. For Compaction-IO, we select parameters related to compaction triggers and the size of levels (More details are in Section 4.1). The four factors capturing Read/Write performance significantly affect IOPS. To demonstrate it, we tuned only those parameters that affect the factor while fixing others for each factor and found Good/Fair/Poor configurations ranked by IOPS. Figure 4(d) shows that the performance varies significantly depending on the configuration. *Observation: Carefully crafted composite features can help navigate the performance trade-offs.*

4 DESIGN

Figure 5 presents the overview of Dremel. The various possible values of RocksDB parameters define the parameter space through which we generate configurations. After generating configurations, we use a rule-based filter to remove ineligible configurations due to resource constraints and domain-specific requirements. For configurations passing the filter, we compute numeric values of *fused futures* for each configuration and conduct quantile bucketing to normalize the numeric values. Next, configurations with the same bucketed features form an arm of a multi-armed bandit model. After building the multi-armed bandit model, we perform the exploration over arms by evaluating configurations sampled from arms. To speed up evaluation, we conduct the multi-fidelity evaluation to identify the best arm from which Dremel finally recommends a performant configuration.

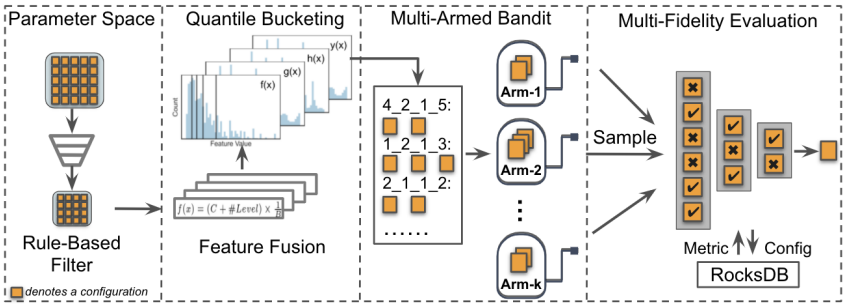


Fig. 5: An overview of the Dremel workflow. Parameter Space (Section 4.1) defines RocksDB configurations. Rule-based Filter (Section 4.2) removes ineligible configurations. Feature Fusion (Section 4.3) and Quantile Bucketing generates a powerful representation for each configuration. Multi-Armed Bandit (Section 4.4) is built by fused features. By Multi-Fidelity Evaluation (Section 4.5), we identify the best arm of multi-armed bandit and select the best configuration. Sampling strategy (Section 4.6) is to initialize the Multi-Fidelity Evaluation.

4.1 Parameter Space

To define the parameter space (presented in Table 1), we first identify which RocksDB parameters significantly affect the performance metrics. We use three methods to identify important parameters from over fifty candidates: 1) One-way analysis of variance: for each candidate, we vary its value and compute the variance of observed metrics like IOPS and latency. We sort candidates by variance and choose ones with a larger variance which means higher impacts. 2) We analyze key components and mechanisms of LSM-Tree design and RocksDB implementation. We pick parameters related to three key mechanisms: LSM-tree memory component management, LSM-tree storage component management, and multi-threading parallelism. 3) We refer to the RocksDB tuning guide [28] and other literature [1, 17, 18, 24, 40]. By using these three methods, we produce the parameter space.

4.2 Rule-Based Filter

After generating all possible configurations through the parameter space, we use a rule-based configuration filter to remove configurations that will not meet the resource constraint rules and RocksDB-specific requirements. We generate these rules based on the RocksDB codebase, the official tuning guide [28], prior works [17, 18, 24], and human experience. We list these rules as two separate sets (Resource constraints rules and RocksDB-Specific Rules) and explain their implications, respectively. We consider resource constraints rules first:

- Rule 1: Worst-case space amplification is $O(\sum_i^{\#Level} \frac{1}{T^i})$ (notations defined in Table 1)
- Rule 2: Worst-case read amplification is $O(\#Level + \#L_0File)$
- Rule 3: Worst-case write amplification is $O(T \times \#Level)$
- Rule 4: Memory budget is O(MemtableUsage + BlockCacheUsage + BloomFilterUsage)

Rule-1 is to bound the disk space usage of RocksDB. We use an analytical model for the worst-case space amplification, similar to prior works [17, 18, 24]. For RocksDB, the worst case of space amplification is that entries stored in last level have newer entries with the same keys stored in levels 0 to $L-1$. For such a case, the space amplification factor is $O(\sum_i^{\#Level} \frac{1}{T^i})$. By Rule-1, T (max_bytes_for_level_multiplier) should not be too small, and we filter out configurations with T smaller than four.

Rule-2 is to bound the read cost. For RocksDB, the worst case of read amplification is that one read request results in searching all files in level 0 and searching one file for each level from 1 to L (e.g., due to false positives in bloom filters). By Rule-2, level 0 should not be too large, so the rule-based filter will filter out configurations with P (level0_stop_writes_trigger) larger than 24.

Class	Name	Notation	Description	Range	Default
$\mathcal{D}$	level0_file_num_compaction_trigger	C	Number of files in level-0 when compactions start	[2,16]	4
$\mathcal{D}$	level0_slowdown_writes_trigger	D	Number of files in level-0 that will slow down writes	[6,20]	8
$\mathcal{D}$	level0_stop_writes_trigger	P	Number of files in level-0 that will trigger put stop	[10,24]	16
$\mathcal{D}$	max_bytes_for_level_multiplier	T	A multiplier to compute max bytes for level	[4,16]	10
$\mathcal{D}$	max_bytes_for_level_base	S	Max bytes for level-1	[64,2048]MB	256MB
$\mathcal{D}$	target_file_size_multiplier	R	A multiplier to compute target level-N file size	[1,8]	1
$\mathcal{D}$	target_file_size_base	F	Target file size at level-1	[32,128]MB	64MB
$\mathcal{D}$	num_levels	L	The total number of levels	[2,10]	10
$\mathcal{M}$	max_write_buffer_number	Q	The number of in-memory memtables	[2,8]	6
$\mathcal{M}$	write_buffer_size	W	Number of bytes to buffer in memtable before compacting	[32,80]MB	64MB
$\mathcal{M}$	min_write_buffer_number_to_merge	M	The minimum number of write buffers that will be merged together	[1,4]	2
$\mathcal{M}$	bloom_bits	E	Bloom filter bits per key	10	10
$\mathcal{M}$	cache_size	O	Number of bytes to use as a cache of uncompressed data	[384,1024] MB	616MB
$\mathcal{M}$	block_size	B	Number of bytes in a block	[2048,16384]B	4096B
$\mathcal{P}$	max_background_compactions	H	The maximum number of concurrent background compactions	[1,8]	1
$\mathcal{P}$	max_background_flushes	U	The maximum number of concurrent background flushes	[1,4]	1
·	op_num	N	Number of operations to do	·	·
·	key_size	K	Size of each key	·	·
·	value_size	V	Size of each value	·	·

Table 1: RocksDB parameter space. The first column denotes the classification of parameters: $\mathcal{D}$ refers to LSM-tree storage component management; $\mathcal{M}$ refers to LSM-tree memory component management; $\mathcal{P}$ refers to multi-threading parallelism. For convenience, the names and descriptions of parameters keep the same as RocksDB source code [28]. Note that the last three parameters N, K, V are not RocksDB configuration parameters that are listed for later usage. We discuss ranges for parameters in Section 4.2.

Rule-3 is to bound the write cost. For RocksDB, the worst case of write amplification is that one entry gets involved in $O(T)$ compactions per level, and the cost is amortized across entries within the same SSTable. By Rule-3, T (max_bytes_for_level_multiplier) should not be too large, and the rule-based filter will filter out configurations with T larger than 16.

Rule-4 is to bound the memory usage. For RocksDB, the primary consumers of memory are Memtables, BlockCache, and bloom filters. We set the memory budget as a constant (e.g., M_{budget} is 1 GB for our experiments), and the filter removes configurations whose usage exceeds the budget.

Next, We consider RocksDB-specific rules:

- Rule 5: level0_compaction_trigger < level0_slowdown_trigger < level0_stop_trigger
- Rule 6: max_bytes_for_level_multiplier > target_file_size_multiplier
- Rule 7: min_write_buffer_number_to_merge < max_write_buffer_number
- Rule 8: max_bytes_for_level_base ≈ L0 size

We also add a few rules that capture the relationships between the different RocksDB parameters. Rule-5 expresses the constraints on parameters used by RocksDB's rate-limiting mechanism. The threshold for trigger compactions should be smaller than the thresholds for rate limiting. Rule-6 controls the number of SSTables within a level and prevents scenarios such as one level only has a single SSTable. Rule-7 captures the semantics of the threshold for the number of merging Memtables. Rule-8 is from the tuning guide, and it is to ensure that L0->L1 compactions are fast.

4.3 Feature Fusion

Although the rule-based filter removes some ineligible configurations, the dimensionality of the parameter space is still high, and the curse of dimensionality still leads to a massive number of

Fused Features	Asymptotic Worst-Case Analysis	Closed-form Expression
Compaction-Frequency (Write Amplification)	$O(\frac{\#CompactThreads}{\#Level} \times \sum_{i=1}^{\#Level} L_{(i-1)\rightarrow i} Frequency)$	$\frac{H}{L} \times \frac{T+1}{F} \times \frac{R - \frac{1}{R^{\#Level-1}}}{R-1}$
WriteBuffer-Full-Frequency	$O(\frac{1}{L_0 Size} + \frac{1}{MemtableSize} \times \frac{1}{\#FlushThreads})$	$\frac{1}{C \times M \times W} + \frac{1}{Q \times W \times U}$
Read-SSTable-Cost (Read Amplification)	$O(\frac{\#L_0 File + \#Level}{\#EntriesPerBlock})$	$(C + \#Level) \times \frac{1}{B}$
BlockCache-Hit-Rate	$O(\frac{\#EntriesInBlockCache}{\#Entries})$	$M_{budget} - Q \times W$

Table 2: Fused Features. Notations in the last column are defined in Table 1.

candidate configurations (e.g., more than millions in our setting). As described earlier, RocksDB performance is governed by the constituent read/write performance, and the settings of the various parameters control the performance trade-offs between read and write performance. We now use domain-specific knowledge and distill fused features that control read/write performance. By leveraging fused features, we can reduce the search space associated with tuning configurations.

Based on the characterization experiments in Section 3, we propose the four fused features shown in Table 2. This work focuses on striking a good trade-off between read and write performance, which directly affects performance metrics. For space costs, as mentioned before in Section 4.2, it is governed by one parameter, the level size multiplier T. We use a rule-based filter to bound it within an acceptable range.

For characterizing write performance, we use two fused features *Compaction-Frequency* and *WriteBuffer-Full-Frequency* by revisiting the analysis in motivating experiments (Section 3): RocksDB performance is highly affected by compaction operations as well as write stalls incurred by Memtables and level-0 being full. For characterizing read performance, we use *Read-SSTable-Cost* and *BlockCache-Hit-Rate* by revisiting previous analysis: read performance is often governed by how often reads are served from memory data structures (i.e., cache hits) or the cost of accessing SSTables (i.e., cache misses). Next, we introduce these features:

4.3.1 *Feature-1: Compaction Frequency.*

Compaction-Frequency captures how frequently compaction happens, which could slow down the flush process and cause write operations to stall as a cascading effect. Also, Compaction-Frequency represents the write amplification factor because compaction is the main cause of write amplification. To derive the expression, we first derive the asymptotic worst-case analysis following a similar line of reasoning as the seminal work that introduced LSM-Trees [59]:

Given the level design of RocksDB, the worst-case for compaction frequency occurs when compaction operations for every level happen at the same time.[1] So the maximum Compaction-Frequency is the following sum: $\sum_{i=1}^{\#Level} L_{(i-1)\rightarrow i} Frequency$ here $L_{(i-1)\rightarrow i} Frequency$ means the compaction frequency for L_{i-1}->L_i compaction operations. RocksDB implements multi-threaded compaction, so the number of compaction threads limits the maximum number of compaction jobs. Thus, we multiply the sum of the level compaction rates by the term $\frac{\#CompactThreads}{\#Level}$ to encode the multi-threading mechanism. Hence, the asymptotic worst-case Compaction-Frequency is $O(\frac{\#CompactThreads}{\#Level} \times \sum_{i=1}^{\#Level} L_{(i-1)\rightarrow i} Frequency)$.

With the asymptotic worst-case analysis, we now derive the closed-form expression of Compaction-Frequency. First, we derive the expression of the number of levels $\#Level$.[2] Using one estimate, we assume the last level of RocksDB holds $N \times \frac{T-1}{T}$ entries (the same approximation as prior work

[1] In practice, due to some SSTables involved in more than one compaction, compaction operations with dependencies are not initiated at the same time, so our big-O analysis provides the upper bound.

[2] RocksDB has one parameter *num_levels* which is the upper limit of the total number of levels. Here, we derive the actual number of levels given the workload.

[17–19]). Using another estimate, the last level of RocksDB is $T^{\#Level}$ times larger than Level-0, which holds $\frac{C \times F}{K+V}$ entries. Hence,

$$N \times \frac{T-1}{T} = \frac{C \times F}{K+V} \times T^{\#Level} \rightarrow \#Level = log_T \frac{N \times (K+V)}{C \times F} \frac{T-1}{T} \tag{1}$$

Next, we derive the term $\sum_{i=1}^{\#Level} L_{(i-1)\rightarrow i} Frequency$. We make two assumptions, which are similar to Theorem 3.1 in the seminal work of LSM-Tree [59]: 1) All entries are never deleted until they arrive at the last level of LSM-Tree. 2) When RocksDB is at a steady rate, the rate at which compaction operations migrate entries from L_{i-1} to L_i is the same for all levels. We denote this rate as γ, and this could also be viewed as the rate of newly ingested data at a level. For L_{i-1} -> L_i compaction, the input rate from L_{i-1} is γ. To perform the compaction, we would then have to read $T \times \gamma$ amount of data from L_i and write out $(T+1) \times \gamma$ to L_i. Hence,

$$L_{(i-1)\rightarrow i} IORate \rightarrow 2 \times (T+1) \times \gamma \tag{2}$$

Given the IO rate, the IO size for each compaction determines the compaction frequency. Generally, RocksDB takes one SSTable from L_i for L_i->L_{i+1} compaction. Thus, the I/O size for each compaction is determined by the SSTable size for each level. Hence,

$$\sum_{i=1}^{\#Level} L_{(i-1)\rightarrow i} Frequency \rightarrow \sum_{i=1}^{\#Level} \frac{L_{(i-1)\rightarrow i} IORate}{SSTable_i Size} \tag{3}$$

For $SSTable_i Size$, we can derive its expression from the definition of F (target_file_size_base) and R (target_file_size_multiplier):

$$\sum_{i=1}^{\#Level} \frac{1}{SSTable_i Size} = \sum_{i=1}^{\#Level} \frac{1}{(SSTable_1 Size \times R^{i-1})} = \frac{K+V}{F} \sum_{i=1}^{\#Level} \frac{1}{R^{i-1}} = \frac{K+V}{F} \frac{R - \frac{1}{R^{\#Level-1}}}{R-1} \tag{4}$$

Finally, according to Equations 1, 2, 3, and 4, we can derive the close-formed expression for Compaction-Frequency and ignore γ, K, and V, which are workload-dependent and are constant across all configurations. Hence,

$$O(\frac{\#CompactThreads}{\#Level} \times \sum_{i=1}^{\#Level} L_{(i-1)\rightarrow i} Frequency) \rightarrow \frac{H}{L} \times \frac{(T+1)}{F} \times \frac{R - \frac{1}{R^{\#Level-1}}}{R-1} \tag{5}$$

4.3.2 *Feature-2: WriteBuffer-Full-Frequency.*

WriteBuffer-Full-Frequency captures the frequency of Memtables being full and L_0 being full. Assuming that the write request rate is ω, the rate at which Memtables fill up is proportional to $\frac{\omega}{MemtableSize}$. For RocksDB supporting multi-threaded flushing, the rate at which Memtables are flushed out is proportional to $\#FlushThreads$. Considering both the speed of filling and flushing, the frequency of Memtables being full is proportional to $\frac{1}{MemtableSize} \times \frac{1}{\#FlushThreads}$, where ω is the same constant for all configurations and is thus ignored. By applying the same analysis for $L0$, the frequency of L_0 being full is $\frac{1}{L_0 Size}$. Due to the overlapping SSTables in L_0, L_0->L_1 compaction is conducted by a single thread, and there is no multi-threading term for L_0. Note that there is no parameter to precisely control the L_0 size for RocksDB. We estimate the L_0 size by the approximation method from the RocksDB tuning guide [28]:

$$\#L_0 File \approx level0\_file\_num\_compaction\_trigger \rightarrow L_0 Size \approx C \times M \times W \tag{6}$$

Then we can derive its expression from the definition of parameters:

$$\frac{1}{L_0Size} + \frac{1}{MemtableSize} \times \frac{1}{\#FlushThreads} \rightarrow \frac{1}{C \times M \times W} + \frac{1}{Q \times W \times U} \tag{7}$$

4.3.3 *Feature-3: Read-SSTable-Cost.*

Read-SSTable-Cost captures the worst-case read cost when reads miss in the BlockCache and are served by SSTables. The worst-case scenario occurs when the bloom filter reports false positives, which leads to SSTable accesses. Also, Read-SSTable-Cost represents the read amplification factor because disk reads due to false positives is a primary factor driving read amplification. For the worst case, all files in L_0 might be accessed as they have overlapping key ranges. After searching all of the files in L_0, only one SSTable is touched for each L_i (i>0) as there is no overlapping between SSTables at the same level L_i (i>0). The unit for disk read is a block, so the cost is amortized by the number of entries in the block.

Thus, Read-SSTable-Cost is $O(\frac{\#L_0File+\#Level}{\#EntriesPerBlock})$. By Equation 1 for $\#Level$, Equation 6 for $\#L_0File$, and the definition of $\#EntriesPerBlock$, we derive the closed-form expression where we ignore K and V, which are workload-dependent and constant across all configurations:

$$O(\frac{\#L_0File + \#Level}{\#EntriesPerBlock}) \rightarrow (C + \#Level) \times \frac{1}{B/(K+V)} \rightarrow (C + \#Level) \times \frac{1}{B} \tag{8}$$

4.3.4 *Feature-4: BlockCache-Hit-Rate.*

BlockCache-Hit-Rate captures the cache efficiency for read requests and is determined by the percentage of entries cached, that is, $O(\frac{\#EntriesInBlockCache}{\#Entries})$. $\#EntriesInBlockCache$ is determined by the size of BlockCache. We can derive the closed-form expression, where we ignore N, K, and V as before:

$$O(\frac{\#EntriesInBlockCache}{\#Entries}) \rightarrow \frac{M_{budget} - Q \times W}{K+V} \times \frac{1}{N} \rightarrow M_{budget} - Q \times W \tag{9}$$

4.4 Multi-Armed Bandit Model

After feature fusion, each configuration has four numeric values corresponding to the fused features. Next, we discuss how to construct the Multi-Armed Bandit model by using fused features.

4.4.1 *Normalize feature values by Quantile Bucketing.*

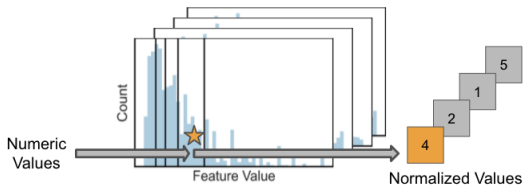


Fig. 6: Quantile Bucketing. The example numeric value of the first feature lies in the 4-th bucket.

We normalize numeric feature values by Quantile Bucketing, as shown in Figure 6. By the definitions of the fused features, it is likely that configurations with similar feature values are likely to have similar behaviors. For example, configurations with similar numerical values for BlockCache-Hit-Rate are likely to have a similar cache-hit rate. We, therefore, use Quantile Bucketing to normalize numerical feature values into a small number of buckets. In particular, for the i-th feature, we bucketize the numerical values such that each bucket has the same number of configurations. For example, Figure 6 shows bucketing with 5-quantiles for the first feature. We then take each configuration and normalize its numerical feature values with the indices of the buckets the values

lie in. For example, Figure 6 shows we normalize a configuration's first feature as 4 because its numeric value lies in the 4-th bucket (2/1/5 for second/third/fourth features, respectively).

We have now converted the large parameter space into a small fused feature space (e.g., 5×5×5×5 as Figure 6 shows). Fused feature space has much less dimensionality, which can enable the efficient search method described next.

4.4.2 *Multi-Armed Bandit.*

Now we build a Multi-Armed Bandit model in which each arm is a cluster of configurations whose normalized features are the same for all four fused features. For example, as Figure 6 shows, we construct an arm which is a cluster of configurations whose normalized features are (4,2,1,5).

We define our Multi-Armed Bandit model as follows: A player is given n arms, indexed by $[n] = [1, 2, 3, ..., n]$. For example, with 5-quantile bucketing, our Multi-Armed Bandit model has $5^4 = 625$ arms. The i-th arm is associated with M configurations $\{C_i^j | \forall j_1, j_2, 1 \leq j_1, j_2 \leq M, NormalizedFeatures(C_i^{j_1}) = NormalizedFeatures(C_i^{j_2})\}$. The i-th arm is also associated with a reward, which is performance metrics (e.g., IOPS) while running configuration C_i^j. At every round t, the player pulls one arm (i.e., runs one configuration from it) and observes performance metrics as its reward. Note that each arm has a bunch of configurations, and we assume that we randomly sample one configuration from this arm to run (We will discuss later other policies for sampling configurations in Section 4.6). Our goal is to identify the best arm (i.e., the arm having the maximal expected reward, that is, best performance). Next, we describe our policy on how to evaluate arms and then identify the best arm.

4.5 Multi-fidelity Evaluation

We have now built a Multi-Armed Bandit model with n arms. Next, we present our policy on how to explore the n arms to identify the best one.

4.5.1 *Successive Halving.*

Evaluating RocksDB configurations is usually expensive because workload traces for evaluating configurations are generally in the order of tens of minutes (even hours) [12, 22, 25]. We need to replay the workload trace for each configuration, so even evaluating a limited number of configurations incurs a significant time cost.

Based on the characteristics of the RocksDB performance curves (Section 3.2), we enable the *multi-fidelity evaluation* for RocksDB configurations by using a cheap approximation to expensive longer duration testing. We refer to the conventional method of evaluating configurations by running the whole workload as the single fidelity method (In our setting, fidelity means the duration of testing). Alternatively, we pursue a multi-fidelity method where we terminate testing earlier for configurations performing poorly on a small subset of the workload. For promising configurations, we assign a longer evaluation time. By doing this, we speed up the process of identifying the best arm. The rationale for the multi-fidelity evaluation is that configurations performing poorly are also more likely to perform worse over complete runs, given the nature of the RocksDB performance curves we discussed in motivating experiments (Section 3).

Algorithm 1: Successive Halving

Input: n arms, round $\mathcal{R}$, time points $\mathcal{T}[.]$, rank function $RANK(.)$

Initialize $S_0 \leftarrow$ sample n configurations one configuration per arm

for $r = 0$ *to* $\mathcal{R}$ **do**

  run each configuration $i \in S_r$ until arriving at time point $\mathcal{T}[r]$

  $S_{r+1} \leftarrow$ set of $\lceil \frac{|S_r|}{2} \rceil$ configurations with top-50% $RANK(.)$ values

Output: configurations in S_R

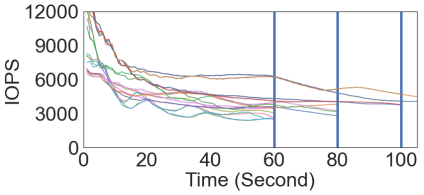


Fig. 7: Runtime of Successive Halving

Specifically, we apply the Successive Halving algorithm [44] to implement multi-fidelity evaluation (As Algorithm 1 shows). Basically, each round runs all configurations with a time budget, collects performance metrics for all configurations, performs an early stop for the worst half, and repeats until one configuration remains. Figure 7 shows one example of executing Successive Halving where each vertical line represents the halving step.

Our Successive Halving algorithm executes halving with a rank function. We can flexibly define the rank function to achieve the desired objective. We give three examples: 1) If the single objective is to maximize IOPS, the rank is given by sorting IOPS such that the higher the IOPS, the higher is the ranking. 2) If the single objective is to minimize latency, the rank is given by sorting latency such that the lower the latency, the higher is the ranking. 3) If multiple objectives are sought (say to achieve high IOPS and low latency), for each configuration, we have two ranks, IOPS_rank and Latency_rank. We take the lower one of these two ranks. By doing this, we only select the configurations with latency and IOPS both being ranked in top-%50 for each halving step.

4.5.2 *Selection of Time Points.*

As Figure 7 shows, at the starting stage of the workloads, the IOPS curve has not yet converged to a stable state due to the structure of the LSM-Tree (i.e., the persistent storage component has not achieved a stable state). We conduct the Successive Halving until the performance curve exhibits less fluctuation. We use one metric *Windowed Variance* to measure convergence: for time point t, its windowed variance is defined as the variance of IOPS reported per second during the last k seconds. When to converge is affected by the rate of request. In our practice, we control the request rate by the number of application threads. Fixing the number of application threads, then we apply the windowed variance to monitor the fluctuation. If the fluctuation is less than a threshold (e.g., 10% of average IOPS), we select such a time point to execute Successive Halving.

4.6 Sample Configuration from Arms

In the previous section, we just randomly sample one configuration when pulling each arm. Next, we present two alternative sampling methods that take into account the performance variance of configurations within the same arm.

4.6.1 Heuristic Sampling. We pick configuration from a specific arm according to the rank of the numeric feature values. Although configurations within the same arm have the same normalized feature values, their numeric values before normalization have slight differences. We sort configurations by the numeric value of one fused feature (WriteBuffer-Full-Frequency by default) and then pick the configuration with the smallest value. The rationale to select the smallest WriteBuffer-Full-Frequency is that the configuration could have the least frequency of write buffer being full. With such a *Heuristic Sampling* method, we sample one configuration per arm. Next, we present a method to sample more configurations from an arm.

4.6.2 Upper Confidence Bound Sampling. We propose a sampling method based on the concept of Upper Confidence Bound (UCB) [5, 10, 42, 43] to sample configurations for Successive Halving. We associate the i-th arm with a list I_i recording metrics (e.g., IOPS) of ever-tried configurations from the i-th arm. UCB for the i-th arm is defined as: $UCB_i = Mean(I_i) + Stddev(I_i)$. UCB sampling works as follows: Basically, we initialize UCB for each arm by running one configuration (heuristically sampling) from the arm and collecting performance metrics. Note that we cannot calculate the stddev with only one data point, so we use the stddev across all arms initially (denoted as global stddev). Next, we sample more configurations from the top-k arms ranked by UCB, evaluate them, collect metrics, and update their UCB. We can repeat the process multiple times to sample more configurations, thus relying on UCB Sampling to strike a good exploration-exploitation trade-off and find a better configuration. To integrate heuristic sampling and UCB sampling into the workflow: we replace the initialization set of Successive Halving (line 1 in Algorithm 1) with the heuristic sampling output (As Algorithm 2 in Appendix A.2 shows). Then we evaluate configurations sampled heuristically and then resort to UCB sampling to explore additional configurations within the arms.

5 EVALUATION

5.1 Setting

Testbed. Our testbed comprises a cluster with x86 servers as client nodes and Stingray PS1100R, a SmartNIC-based disaggregated storage solution, as storage nodes. Each server has Intel Xeon processors and 96GB of memory running CentOS 7.4. Storage nodes use Samsung DCT983 960GB NVMe SSDs. We use Intel SPDK (v19.07) as a storage IO stack, supporting rate limits on IO bandwidth. We use RocksDB (v6.15), which is compatible with the SPDK environment.

Implementation. We implement all components of Dremel into a tuning controller that is independent of the RocksDB instance. The tuning controller sends the configuration to the RocksDB instance and fetches the performance log after testing. For Quantile Bucketing, we use two buckets by default for each fused feature. For Successive Halving and UCB Sampling, we describe its parameters later for specific experiments.

Comparison Baselines. We compare Dremel against four baselines described as follows:

• Single-Task Bayesian Optimization (SingleTaskBO) [6] is a classical Bayesian Optimization (BO) method that uses BO to solve the optimization problem of a black-box function, which in our case is optimizing an objective such as IOPS. The BO method incorporates the RocksDB parameters into a Gaussian Process model to navigate the search space. We use an efficient implementation of the Gaussian Process in BoTorch [6].

• Multi-Task Bayesian Optimization (MultiTaskBO) [1, 16] is an extension of BO that leverages the concept of multi-task learning to speed up the search process. This method utilizes a Gaussian Process kernel supporting multi-task learning on the data originating from decomposing main objective into sub-objectives. We implement the method with GPyTorch [29] for the multi-task learning kernel and BoTorch [6] for the BO framework.

• RandomSearch [8] is used to randomly sample configurations in the parameter space. We also use rule-based filters to remove some ineligible configurations due to resource constraints.

• TuningGuide [28] is released by developers of RocksDB to provide some suggestions and general principles for tuning RocksDB based on developers' understanding and experience. In our experiments, we view the performance metric under the guide's configuration as the base value (1.0) to normalize other baselines for comparison.

Workloads. We use the tool *db_bench* [28], RocksDB's standard tool, to benchmark performance. We use the YCSB workload generator implemented by Balmau et al. [7]. We use 16 threads as application threads that generate requests quickly to drive RocksDB into a stable state quickly. By

default, we set the key size as 16 Bytes and the value size as 1024 Bytes, which are representative of real traces. We use preloading (e.g., running *fillseq* for two minutes) to simulate the scenario of adapting on the fly when workload changes occur in an existing instance. For the scenario of optimizing for an initial setup, there is no preloading phase. We vary the SSD bandwidth by using a rate limiter to simulate the scenario of hardware changes. We evaluate 18 settings with different workload characteristics and hardware conditions described as follows:

• Synthetic workloads: For the scenario of initial setup (i.e., no preloading), we consider 9 workloads with varying Zipf skewness [0.3, 0.6, 0.9] and varying GET request proportions [30%, 60%, 90%] (i.e., the remainder requests are PUTs). We limit the SSD bandwidth to 100 MB/s. For the scenario of adapting on the fly (i.e., with preloading), we have 3 workloads with varying GET proportions [30%, 60%, 90%], while we set Zipf skewness as 0.9 and limit the SSD bandwidth as 100 MB/s. For the scenario of hardware changes, we consider 3 workloads with varying GET proportions [30%, 60%, 90%], while we increase the SSD bandwidth to 200 MB/s and set Zipf skewness as 0.9.

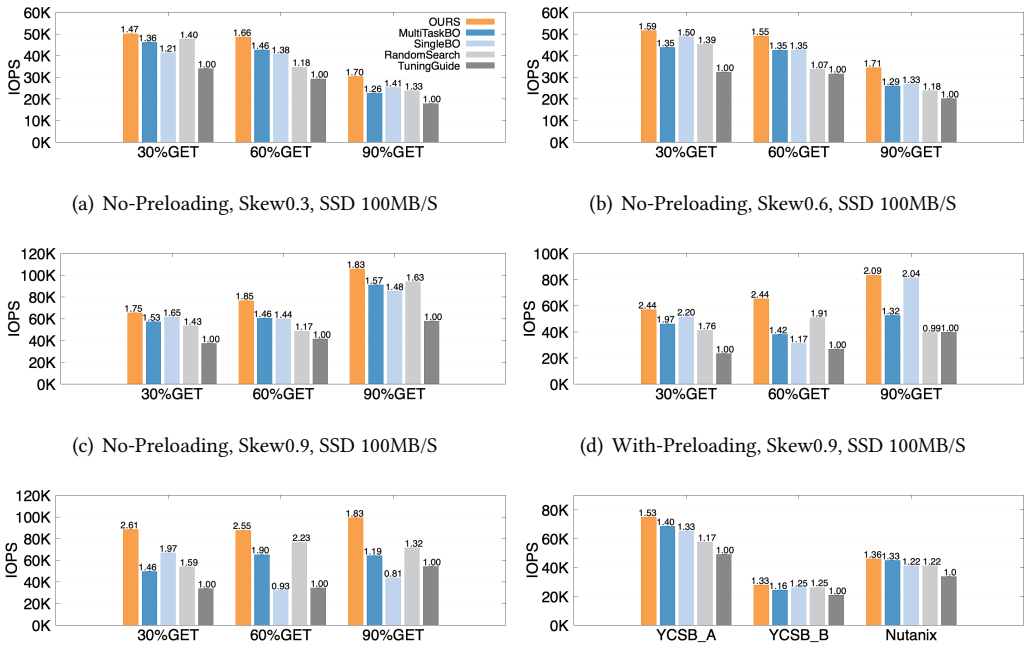
• Production workloads: We have three production workloads: 1) We use YCSB-A [14], corresponding to 50%GET and 50%PUT with the Zipf skewness of 0.5 and SSD bandwidth set to 200 MB/s. 2) We use YCSB-B, a read-intensive workload corresponding to 95%GET and 5%PUT with the Zipf skewness of 0.5 and SSD bandwidth set to 200 MB/S. 3) We use the Nutanix workload sampling from the production deployment [7] corresponding to 57% GET, 41% PUT, and 2% SCAN with the Zipf skewness of 0.5 and SSD bandwidth set to 100 MB/s.

Metrics. In this work, we use Input/Output Operations Per Second (IOPS) and 99% read tail latency. Besides performance metrics, we also report the tuning time.

5.2 Efficacy of Dremel

5.2.1 Optimizing for IOPS.

In this experiment, we apply Dremel to produce the best-identified configuration to maximize IOPS for 18 workload settings while comparing against four other methods with the same time budget. For Dremel, we execute the quantile bucketing with two buckets for each fused feature, i.e., we have $2^4 = 16$ arms. First, heuristic sampling obtains 16 arms, and then UCB sampling obtains top-5 configurations per round for three rounds. So we have $16 + 3 \times 5 = 31$ configurations to evaluate for Dremel. For successive halving, by analyzing the windowed variance, we set time points as [60s, 80s, 100s] to conduct halving. Thus, the total tuning time for Dremel is $60 \times \frac{31}{2} + 80 \times \frac{31}{4} + 100 \times \frac{31}{4} = 2325$ seconds. We assign the same time budget for the other four comparison methods. For the configuration finally recommended by Dremel and others, we take the average IOPS for running the workload for 120 seconds. So the time of each optimization step for MultiTaskBO, SingleBO, and RandomSearch is 120 seconds.



(a) No-Preloading, Skew0.3, SSD 100MB/S

(b) No-Preloading, Skew0.6, SSD 100MB/S

(c) No-Preloading, Skew0.9, SSD 100MB/S

(d) With-Preloading, Skew0.9, SSD 100MB/S

(e) With-Preloading, Skew0.9, SSD 200MB/S

(f) Production Setting

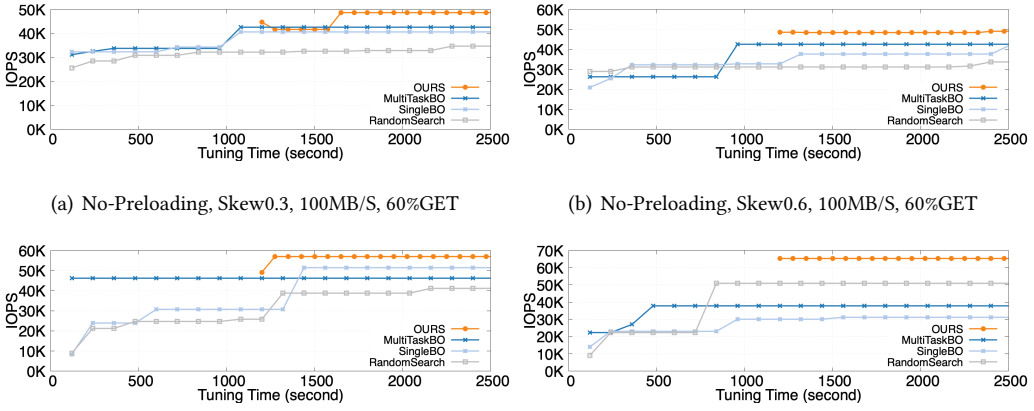
Fig. 8: Maximize IOPS for 18 different settings of workload and hardware. Three settings for each figure.

Figure 8 presents the IOPS comparisons for different settings of workloads and hardware. Dremel achieves the largest improvements on all 18 settings compared with other methods using the same time budget. Dremel improves IOPS over the tuning guide's configuration by up to 2.61×. Figures 8(a), 8(b), and 8(c) present results for settings with no preloading. It demonstrates that Dremel can generate efficient configurations for initial setups before launching RocksDB instances. Figure 8(d) demonstrates that Dremel can generate efficient configurations for on-the-fly adapting when workload changes for an existing instance (i.e., with preloading data). Figure 8(e) demonstrates Dremel can effectively adapt to different hardware conditions, e.g., where we increase SSD bandwidth to 200 MB/S. Figure 8(f) presents results for three settings from a production environment. Note that the Nutanix workload involves 3% Scan operations (the remainder is GET/PUT), which shows that Dremel can work with more types of operations besides GET and PUT. Moreover, the Nutanix workload has a different value size (400 Bytes, 1024 Bytes for other workloads).

5.2.2 *Tuning Time.*

We demonstrate how quickly Dremel finds the best-identified configuration. We plot the tuning process for the settings used in the previous experiments. We present 4 representative settings here. More results are in Appendix A.3.1.

Figure 9 presents the tuning time for Dremel and other methods. Lines of Dremel start from 1200 seconds rather than 0 like others. This is because Dremel needs to play each arm at least once before identifying the best arm, and we have at least 16 arms to conduct successive halving on ($60 \times \frac{16}{2} + 80 \times \frac{16}{4} + 100 \times \frac{16}{4} = 1200$ seconds). After finishing the evaluation of configurations from heuristic sampling (1200 seconds), UCB sampling starts to pull more configurations from promising arms. Note that we can terminate the tuning for Dremel before starting UCB sampling if the time budget is extremely limited. As Figure 9(b) and 9(d), IOPS achieved by heuristic sampling



(a) No-Preloading, Skew0.3, 100MB/S, 60%GET

(b) No-Preloading, Skew0.6, 100MB/S, 60%GET

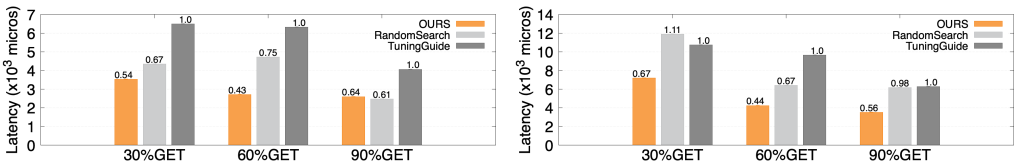
(c) With-Preloading, Skew0.9, 100MB/S, 30%GET

(d) With-Preloading, Skew0.9, 100MB/S, 60%GET

Fig. 9: Tuning time comparison for 4 settings of workload and hardware.

is sufficiently high. If we allow for more tuning time, UCB sampling can further improve the IOPS (see Figures 9(a) and 9(c)). We observe that each step of Dremel uses less time compared with others because Dremel applies the multi-fidelity evaluation to speed up the process. Also, note that in Figure 9(a), IOPS reduces a little bit after we start UCB sampling as the successive halving process could introduce errors due to approximation. But, this error is eliminated after evaluating more configurations. We quantify such errors later (Section 5.3.3).

5.2.3 *Optimization for Latency.*



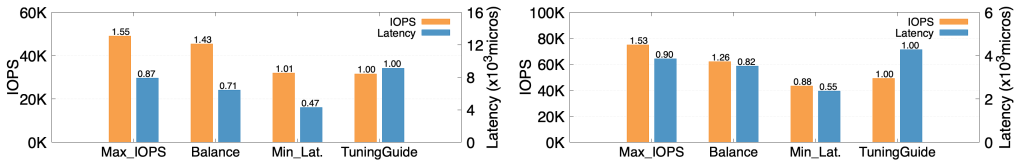
(a) No-Preloading, Skew0.9, SSD 100MB/S

(b) With-Preloading, Skew0.9, SSD 100MB/S

Fig. 10: Minimize read latency (99%) for 6 different settings of workload and hardware.

In this experiment, we apply Dremel to minimize read latency (99%). Dremel can flexibly switch the objective from maximizing IOPS to minimizing latency by assigning different rank functions for successive halving. For BO-based approaches like MultiTaskBO, their design is highly coupled with the objective. For new objectives, MultiTaskBO needs to design a new decomposition of tasks, which is beyond the scope of the original work. Therefore, for this experiment, we only compare against RandomSearch and TuningGuide. Figure 10 shows that Dremel can lower the read latency to 43% of that under the default configuration for both preloading and no-preloading settings. More results are in Appendix A.3.2.

5.2.4 Achieve both low-latency and high-IOPS . This experiment demonstrates that Dremel can recommend a configuration to simultaneously achieve low-latency and high-IOPS. Besides optimizing for a single objective, Dremel also supports multi-objective optimization by defining a rank function that balances multiple objectives simultaneously. Figure 11 presents three types of configurations against the default one. Max_IOPS and Min_Latency are single-objective optimizations (maximizing IOPS and minimizing latency, respectively). *Balance* is the configuration that strikes a trade-off



(a) No-Preloading, Skew0.6, Read 60, 100MB/S (b) YCSB-A

Fig. 11: Achieve both low-latency and high-IOPS. One setting for each figure.

between high-IOPS and low-latency. *Balance* is better than the default TuningGuide in terms of both latency and IOPS. In Figures 11(a) and 11(b), *Balance* is better than Max_IOPS or Min_Latency in terms of one metric (latency and IOPS, respectively).

5.3 Evaluate Components of Dremel

5.3.1 *Impact of Feature Fusion.*

Workload	Avg. IOPS	Inter-Arm STDDEV	Intra-Arm STDDEV
30%GET	34065	1322.76	7407.66
60%GET	29319	1148.48	6058.23
90%GET	17937	877.81	3137.99

Table 3: Impact of Feature Fusion. No preloading, Zipf skewness is 0.3, and SSD is 100 MB/S.

In this experiment, we evaluate the utility of feature fusion. We view feature fusion as a form of clustering over configurations, with each arm now representing a cluster of configurations. An effective clustering approach would group configurations with high similarities in terms of features' definitions and leave out configurations with low similarity. For our default setting (two buckets for each feature), we have 16 arms. We randomly sample five configurations from each arm to evaluate its IOPS, and we define two metrics to evaluate the quality of the clustering. 1) For each arm, Inter-Arm STDDEV is the standard deviation of IOPS values of five configurations from the same arm. We take the average over all the arms. 2) Intra-Arm STDDEV is the standard deviation of IOPS values of 5 configurations randomly sampled regardless of arms. By definition, a good clustering approach can achieve a low Inter-Arm STDDEV and high Intra-Arm STDDEV. We conduct the experiments in three different settings. Table 3 reports the standard deviation values and average IOPS over all the configurations as a reference. It shows that feature fusion achieves a much lower Inter-Arm STDDEV than Intra-Arm STDDEV, which means feature fusion is useful to cluster configurations.

To further evaluate the utility and importance of feature fusion, we conduct experiments by taking out one of the fused features while running Dremel. Figure 12 shows experiments using the subset of fused features, where No-WFF, No-CF, No-RSS, No-BCH mean that we exclude WriteBuffer-Full-Frequency, Compaction-Frequency, Read-SSTable-Cost, or BlockCache-Hit-Rate, respectively. As Figure 12 shows, using all four features can achieve the highest IOPS on all workload settings. Excluding one of the fused features weakens the efficacy of Dremel on tuning configurations, although using subsets of features can shorten the tuning time.

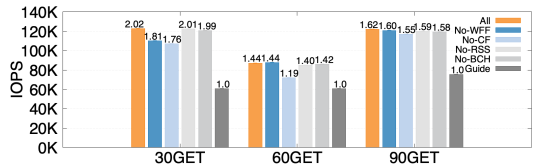


Fig. 12: Using subsets of fused features. Skew 0.9, No-Preloading, SSD 200MB/S

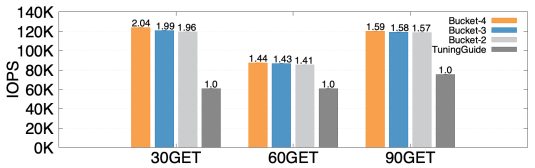
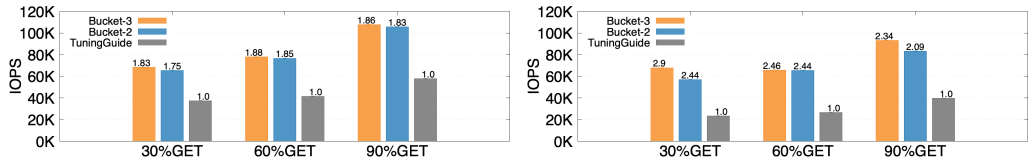


Fig. 13: Varying the number of buckets. Skew 0.9, No-Preloading, SSD 200MB/S

5.3.2 *Impact of Bucket Number.*

In this experiment, we evaluate Dremel's design choice on the number of buckets for quantile bucketing, which determines the number of arms to pull and configurations to evaluate. For our default setting with the number of buckets set as two, Dremel has 2^4 arms. If we increase the number of buckets to three, Dremel has 3^4 arms. Heuristic sampling pulls one configuration per arm. Thus, more arms mean more configurations to evaluate, and more arms increase the probability of finding better configurations but need a longer time to evaluate. Figure 13 and 14 presents maximizing



(a) No-Preloading, Skew0.9, 100MB/S

(b) With-Preloading, Skew0.9, 100MB/S

Fig. 14: IOPS comparison for quantile bucketing with two or three buckets. Three settings for each figure.

IOPS by Dremel with varying the number of buckets from 2 to 4 for quantile bucketing. We use TuningGuide as a baseline for reference. As expected, Dremel with four or three buckets for quantile bucketing performs better than the 2-bucket setting in terms of IOPS. However, the 4-bucket setting needs about 16x times longer evaluation time than the 2-bucket setting. (The 3-bucket setting is 5x longer than the 2-bucket setting.) The 4-bucket setting needs about five hours to tune configurations for one workload/hardware setting. The 2-bucket setting just needs about a half-hour. The 2-bucket setting sufficiently maximizes IOPS compared with the baseline while needing less tuning time.

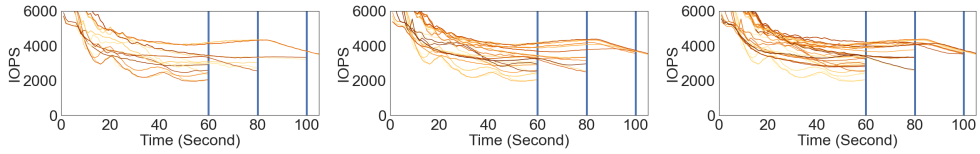
5.3.3 *Impact of Multi-fidelity Approximated Evaluation.*

Workload	Best-Seen IOPS	Achieved IOPS	Achieved Percents
NoPreloading_skew0.3_30%GET	55214	50189	90.8%
NoPreloading_skew0.3_60%GET	49633	48656	98.0%
NoPreloading_skew0.3_90%GET	30430	30430	100%

Table 4: Accuracy of multi-fidelity approximated evaluation. SSD bandwidth is 100 MB/S.

In this experiment, we measure the error incurred by the approximation of successive halving. Successive halving conducts an early-termination strategy to stop configurations that are performing poorly, and this could erroneously terminate some configurations that would perform better when run for a longer time. We quantify the accuracy of successive halving. For our default setting, Dremel evaluated 31 configurations in total (16 for heuristic sampling and 15 for UCB sampling). We identify Best-seen IOPS as the IOPS achieved by the best one among the 31 configurations when all of them are run sufficiently long. Achieved IOPS is from the configuration that is recommended by Dremel. We define the achieved percentages as the ratio between the Achieved IOPS and Best-seen IOPS. 100% is perfect, which means there is no error incurred by the approximated evaluation. Table 4 reports the results from several settings from previous experiments. From the last column, we can see that the error is acceptable, and for some cases, Dremel identifies the best configuration. More results are in Appendix A.3.3.

5.3.4 *Impact of Upper-Confidence-Bound Sampling.*



(a) UCB-0 (b) UCB-10 (c) UCB-20

Fig. 15: Runtime of Upper-confidence-bound sampling. No-Preloading, skewness 0.6, SSD 100MB/S, 30%GET

In this experiment, we measure the impact of Upper-Confidence-Bound sampling. The idea of Upper-Confidence-Bound sampling is to sample more configurations from promising arms after evaluating all arms once. Figure 15 presents one example of UCB sampling runtime trace (one application thread). From left to right, the figures show Dremel samples 0, 10, 20 more configurations by UCB sampling. As we increase the number of UCB samples from left to right, more configurations are evaluated. Among the additional configurations from UCB sampling, most of them are from regions with a higher confidence bound to achieve greater IOPS as UCB sampling navigates Dremel to explore the promising arms more intensively. Thus, UCB sampling can utilize the tuning time budget more efficiently. More results are in Appendix A.3.4.

6 DISCUSSION

6.1 Applicability to Other Systems

Besides RocksDB, Dremel is also applicable to other systems that use LSM-tree-based storage. For example, we discuss below how we could apply Dremel to Apache Cassandra and Apache Flink. Further, the design of the Dremel pipeline is sufficiently general for tuning some other systems, especially storage systems that embody the *RUM tradeoff* [38] between the read cost (R), the update/write cost (U), and the memory/storage overhead (M).

Apache Cassandra [3] is a NoSQL distributed database whose storage engine uses an LSM-tree-based structure. Cassandra's memory component and persistent storage component support operations similar to RocksDB. Also, Cassandra supports leveled compaction policy similar to RocksDB. So Dremel's fused features and tuning workflow are directly applicable to Cassandra, with the difference limited to the parameter names of Cassandra configurations compared with RocksDB parameters. Table 6 in Appendix A.1 shows parameters of Cassandra configurations corresponding to RocksDB parameters. For parameters that are not tunable in Cassandra, we can replace them with constants while applying fused features, which could reduce the power of fused features representing configurations but not affect the correctness.

Apache Flink [2] is a framework and processing engine for stateful computations over data streams, and it uses key-value stores, such as RocksDB, to manage the computations. In fact, RocksDB is the *de facto* store for Flink's state management. Thus, Dremel's fused features and workflow are applicable for tuning the performance of Flink's state engines. Table 6 in Appendix A.1 shows parameters used by Flink and their correspondence to RocksDB parameters.

Generality and Limitation: To generate adaptive configuration, Dremel firstly distills a large number of raw features into a small number of *fused features* and then expresses configurations more concisely so as to reduce the search space. With the condensed search space, Dremel builds a Multi-Armed Bandit model and enables the effective evaluation of configurations. Besides RocksDB, Dremel's workflow is applicable to tune other systems, especially storage systems, with a condensed configuration space. The only issue is that feature fusion requires some domain-specific expertise for other systems. A possible way to derive fused features is using learning methods to distill a large set of features, which is our future work.

6.2 Mechanisms to Adapt Configurations On-the-fly

Besides the initial setup of the RocksDB instance, Dremel also targets adaptivity in the form of on-the-fly adaptation of configurations. For the initial setup phase, the RocksDB instance has not been launched yet. So operators can directly use the configuration produced by Dremel to configure the new instance. For the on-the-fly adaptation, the running instance already has one configuration being used. Next, we discuss two mechanisms to update a configuration on-the-fly, which are orthogonal to Dremel's workflow components that recommend an efficient configuration.

Dynamic Adjustment	Instance Migration
level0_file_num_compaction_trigger, level0_slowdown_writes_trigger, level0_stop_writes_trigger, min_write_buffer_number_to_merge, max_background_compactions, max_background_flushes	max_bytes_for_level_multiplier, max_bytes_for_level_base, target_file_size_multiplier, target_file_size_base, num_levels, max_write_buffer_number, write_buffer_size, bloom_bits, cache_size, block_size

Table 5: Two mechanisms to on-the-fly update configuration parameters.

Dynamic Adjustment: For parameters that are thresholds to trigger operations such as compaction and flushing, we can dynamically update the values from the old configuration to the new one. (See left column of Table 5). Specifically, a configuration file can maintain the latest parameters values, and this file can be queried periodically to determine when operations should be triggered. Implementing the dynamic adjustment of parameters in RocksDB is part of our future work.

Instance Migration: For parameters that control the level structure of the LSM-tree and the layout of SSTables (see right column of Table 5), we launch an instance with the new configuration and migrate data from the old instance to the new one. In practical deployment [25, 34], data migration can be done without stopping the service. Note that RocksDB works as a single node library, and applications using RocksDB generally set up multiple replicas of the instance for the purpose of fault tolerance. Therefore, data migration can be done incrementally without interrupting the application's running service.

7 RELATED WORK

LSM-Tree-based KV-stores modeling and optimization. Prior work has explored worst-case closed-form expressions to model read/write/space cost and amplification factors of LSM-Tree while trying to navigate the design of new LSM-Tree-based KV-stores [17–19, 38, 39, 51]. However, these models encode very few or none of the factors affected by workload and hardware. Further, these models do not consider optimizations enabled by RocksDB implementation like multi-threading. The fused features that we propose not only capture the cost factors but also encode the mechanisms implemented by RocksDB. Further, we integrate these factors into a multi-armed bandit model to be adaptive to workload settings and the hardware conditions. Another line of work is to propose optimizations for LSM-Tree-based KV-stores: optimizing data structures [15, 48, 52, 61], scheduling operations [7, 60], and exploiting emerging hardware [36, 55, 68]. We appreciate these ideas, but the new mechanisms further increase the complexity of tuning LSM-Tree-based KV-stores and motivate the need for automatic tuning.

Tuner for RocksDB and other databases. Several RocksDB tuners have been proposed: Alabed et al. [1] uses multi-task Gaussian Process to model RocksDB, which is then incorporated in a Bayesian Optimization loop to find the configuration that maximizes IOPS. As we discussed before, due to the massive parameter space and intrinsic complexities, black-box learning like BO is not sufficient to capture the relationship between the parameters and system performance with a limited time budget. Further, the objective (IOPS) is set in stone with the specific definition of tasks, and this requires new task designs for new objectives. Luo et al. [54] introduces a memory tuner that tunes

the memory allocation between the write memory and the buffer cache. They focus on the memory component, which is just one part of the whole complex system. Jia et al. [40] apply multi-objective optimization algorithms for auto-tuning RocksDB. Their method pays a much higher time-cost by collecting over 12,000 experimental records in 6 months. Besides tuners for NoSQL databases like RocksDB, researchers present several tuners for SQL database such as BO-based OtterTune [67] and ResTune [70], RL-based CDBTune [69] and QTune [49], and Neural-Network-based iBTune [66]. Compared with these studies, a key difference is that Dremel injects domain-specific knowledge of LSM-Tree-based RocksDB into fused features to build an efficient bespoke model.

Bandit algorithms for hyper-parameter tuning. For hyperparameter tuning of machine learning algorithms, bandit-based approaches have gained lots of attention [5, 23, 42, 43, 46, 50] because of high-efficiency and robust models. HyperBand [50] formulates hyper-parameter tuning as a pure-exploration bandit problem. Kandasamy et al. [43] applies multi-fidelity approximations to the experiment. Auer et al. [5] shows how confidence bounds can be used to deal with an exploitation-exploration trade-off. We borrow some concepts from the ML setting and also tailor our multi-armed bandit model and multi-fidelity evaluation to our RocksDB setting.

8 CONCLUSION

This paper presents Dremel, a configuration tuning system that can quickly identify a RocksDB configuration that achieves high performance while adapting to specific workload and hardware conditions. Dremel integrates several strategies: feature fusion to handle the curse of dimensionality, online tuning with multi-armed bandit models to enable adaptation to workload and hardware conditions, and quick evaluation with multi-fidelity approximation and upper-confidence-bound sampling. We design, implement, and evaluate Dremel on 18 settings with diverse workloads and hardware conditions. We also evaluate the key strategies we proposed. Our evaluations show Dremel achieves better end-to-end performance even with less tuning time than four common approaches and that all components of Dremel are required to achieve the design goals. This work does not raise any ethical issues.

ACKNOWLEDGMENTS

We thank Ajay Mahimkar and the anonymous reviewers for their helpful comments. This research was supported by Cisco.

REFERENCES

[1] Sami Alabed and Eiko Yoneki. 2021. High-Dimensional Bayesian Optimization with Multi-Task Learning for RocksDB. In *Proceedings of the 1st Workshop on Machine Learning and Systems*. 111–119.

[2] Apache. 2021. Flink. https://flink.apache.org/.

[3] Apache. 2022. Cassandra. https://cassandra.apache.org/.

[4] Jean-Yves Audibert, Sébastien Bubeck, and Rémi Munos. 2010. Best arm identification in multi-armed bandits.. In *COLT*. Citeseer, 41–53.

[5] Peter Auer. 2002. Using confidence bounds for exploitation-exploration trade-offs. *Journal of Machine Learning Research* 3, Nov (2002), 397–422.

[6] Maximilian Balandat, Brian Karrer, Daniel R. Jiang, Samuel Daulton, Benjamin Letham, Andrew Gordon Wilson, and Eytan Bakshy. 2020. BoTorch: A Framework for Efficient Monte-Carlo Bayesian Optimization. In *Advances in Neural Information Processing Systems 33*. http://arxiv.org/abs/1910.06403

[7] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 753–766. https://www.usenix.org/conference/atc19/presentation/balmau

[8] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *Journal of machine learning research* 13, 2 (2012).

[9] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[10] Sébastien Bubeck and Nicolo Cesa-Bianchi. 2012. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *arXiv preprint arXiv:1204.5721* (2012).

[11] Sébastien Bubeck, Rémi Munos, and Gilles Stoltz. 2009. Pure exploration in multi-armed bandits problems. In *International conference on Algorithmic learning theory*. Springer, 23–37.

[12] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*. 209–223.

[13] Yu-Zhen Janice Chen, Stephen Pasteris, Mohammad Hajiesmaili, John Lui, Don Towsley, et al. 2021. Cooperative Stochastic Bandits with Asynchronous Agents and Constrained Feedback. *Advances in Neural Information Processing Systems* 34 (2021).

[14] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[15] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2020. From wisckey to bourbon: A learned index for log-structured merge trees. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 155–171.

[16] Valentin Dalibard, Michael Schaarschmidt, and Eiko Yoneki. 2017. BOAT: Building auto-tuners with structured Bayesian optimization. In *Proceedings of the 26th International Conference on World Wide Web*. 479–488.

[17] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 79–94.

[18] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better space-time trade-offs for LSM-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data*. 505–520.

[19] Niv Dayan and Stratos Idreos. 2019. The log-structured merge-bush & the wacky continuum. In *Proceedings of the 2019 International Conference on Management of Data*. 449–466.

[20] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review* 41, 6 (2007), 205–220.

[21] Rémy Degenne and Vianney Perchet. 2016. Anytime optimal algorithms in stochastic multi-armed bandits. In *International Conference on Machine Learning*. PMLR, 1587–1595.

[22] Diego Didona, Nikolas Ioannou, Radu Stoica, and Kornilios Kourtis. 2020. Toward a better understanding and evaluation of tree structures on flash ssds. *arXiv preprint arXiv:2006.04658* (2020).

[23] Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. 2015. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Twenty-fourth international joint conference on artificial intelligence*.

[24] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB.. In *CIDR*, Vol. 3. 3.

[25] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience. In *19th {USENIX} Conference on File and Storage Technologies ({FAST} 21)*. 33–49.
[26] Yihan Du, Siwei Wang, Zhixuan Fang, and Longbo Huang. 2021. Continuous Mean-Covariance Bandits. *Advances in Neural Information Processing Systems* 34 (2021).
[27] Eyal Even-Dar, Shie Mannor, Yishay Mansour, and Sridhar Mahadevan. 2006. Action Elimination and Stopping Conditions for the Multi-Armed Bandit and Reinforcement Learning Problems. *Journal of machine learning research* 7, 6 (2006).
[28] Facebook. 2021. RocksDB. https://github.com/facebook/rocksdb.
[29] Jacob R Gardner, Geoff Pleiss, David Bindel, Kilian Q Weinberger, and Andrew Gordon Wilson. 2018. Gpytorch: Blackbox matrix-matrix gaussian process inference with gpu acceleration. *arXiv preprint arXiv:1809.11165* (2018).
[30] Arnob Ghosh and Vaneet Aggarwal. 2020. Model free reinforcement learning algorithm for stationary mean field equilibrium for multiple types of agents. *arXiv preprint arXiv:2012.15377* (2020).
[31] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and David Sculley. 2017. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*. 1487–1495.
[32] Google. 2021. LevelDB. https://github.com/google/leveldb/.
[33] Joseph M Hellerstein, Michael Stonebraker, and James Hamilton. 2007. *Architecture of a database system*. Now Publishers Inc.
[34] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.
[35] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An optimized storage engine for large-scale E-commerce transaction processing. In *Proceedings of the 2019 International Conference on Management of Data*. 651–665.
[36] Haoyu Huang and Shahram Ghandeharizadeh. 2021. Nova-LSM: A Distributed, Component-based LSM-tree Key-value Store. In *Proceedings of the 2021 International Conference on Management of Data*. 749–763.
[37] Stratos Idreos, Manos Athanassoulis, Niv Dayan, Demi Guo, Mike S Kester, Lukas Maas, and Kostas Zoumpatianos. 2015. Past and future steps for adaptive storage data systems: From shallow to deep adaptivity. In *Real-Time Business Intelligence and Analytics*. Springer, 85–94.
[38] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, et al. 2019. Design Continuums and the Path Toward Self-Designing Key-Value Stores that Know and Learn.. In *CIDR*.
[39] Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S Kester, and Demi Guo. 2018. The data calculator: Data structure design and cost synthesis from first principles and learned cost models. In *Proceedings of the 2018 International Conference on Management of Data*. 535–550.
[40] Yichen Jia and Feng Chen. 2020. Kill Two Birds with One Stone: Auto-tuning RocksDB for High Bandwidth and Low Latency. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 652–664.
[41] Yuchen Jin, Tianyi Zhou, Liangyu Zhao, Yibo Zhu, Chuanxiong Guo, Marco Canini, and Arvind Krishnamurthy. 2021. AutoLRS: Automatic Learning-Rate Schedule by Bayesian Optimization on the Fly. *arXiv preprint arXiv:2105.10762* (2021).
[42] Kirthevasan Kandasamy, Gautam Dasarathy, Junier Oliva, Jeff Schneider, and Barnabás Póczos. 2016. Gaussian process optimisation with multi-fidelity evaluations. In *Proceedings of the 30th/International Conference on Advances in Neural Information Processing Systems (NIPS'30)*.
[43] Kirthevasan Kandasamy, Gautam Dasarathy, Barnabas Poczos, and Jeff Schneider. 2016. The multi-fidelity multi-armed bandit. *Advances in neural information processing systems* 29 (2016), 1777–1785.
[44] Zohar Karnin, Tomer Koren, and Oren Somekh. 2013. Almost optimal exploration in multi-armed bandits. In *International Conference on Machine Learning*. PMLR, 1238–1246.
[45] Aaron Klein, Stefan Falkner, Simon Bartels, Philipp Hennig, and Frank Hutter. 2017. Fast bayesian optimization of machine learning hyperparameters on large datasets. In *Artificial Intelligence and Statistics*. PMLR, 528–536.
[46] Aaron Klein, Stefan Falkner, Jost Tobias Springenberg, and Frank Hutter. 2016. Learning curve prediction with Bayesian neural networks. (2016).
[47] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.
[48] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 447–461.
[49] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2118–2130.

[50] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2017. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research* 18, 1 (2017), 6765–6816.

[51] Hyeontaek Lim, David G Andersen, and Michael Kaminsky. 2016. Towards accurate and fast evaluation of multi-stage log-structured designs. In *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*. 149–166.

[52] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)* 13, 1 (2017), 1–28.

[53] Chen Luo and Michael J Carey. 2019. On performance stability in LSM-based storage systems (extended version). *arXiv preprint arXiv:1906.09667* (2019).

[54] Chen Luo and Michael J Carey. 2020. Breaking down memory walls: adaptive memory management in LSM-based storage systems. *Proceedings of the VLDB Endowment* 14, 3 (2020), 241–254.

[55] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. 2021. Gimbal: Enabling Multi-Tenant Storage Disaggregation on SmartNIC JBOFs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 106–122. https://doi.org/10.1145/3452296.3472940

[56] MongoDB. 2021. WiredTiger. https://github.com/wiredtiger/wiredtiger.

[57] Conor Newton, Ayalvadi Ganesh, and Henry Reeve. 2022. Asymptotic Optimality for Decentralised Bandits. *SIGMETRICS Perform. Eval. Rev.* 49, 2 (jan 2022), 51–53. https://doi.org/10.1145/3512798.3512817

[58] Gijun Oh, Junseok Yang, and Sungyong Ahn. 2021. Efficient Key-Value Data Placement for ZNS SSD. *Applied Sciences* 11, 24 (2021), 11842.

[59] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[60] Fengfeng Pan, Yinliang Yue, and Jin Xiong. 2017. dCompaction: Delayed compaction for the LSM-tree. *International Journal of Parallel Programming* 45, 6 (2017), 1310–1325.

[61] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 497–514.

[62] Herbert Robbins. 1952. Some aspects of the sequential design of experiments. *Bull. Amer. Math. Soc.* 58, 5 (1952), 527–535.

[63] Abishek Sankararaman, Ayalvadi Ganesh, and Sanjay Shakkottai. 2019. Social Learning in Multi Agent Multi Armed Bandits. *Proc. ACM Meas. Anal. Comput. Syst.* 3, 3, Article 53 (dec 2019), 35 pages. https://doi.org/10.1145/3366701

[64] Subhadeep Sarkar, Dimitris Staratzis, Ziehen Zhu, and Manos Athanassoulis. 2021. Constructing and analyzing the LSM compaction design space. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2216–2229.

[65] Niranjan Srinivas, Andreas Krause, Sham M Kakade, and Matthias Seeger. 2009. Gaussian process optimization in the bandit setting: No regret and experimental design. *arXiv preprint arXiv:0912.3995* (2009).

[66] Jian Tan, Tieying Zhang, Feifei Li, Jie Chen, Qixing Zheng, Ping Zhang, Honglin Qiao, Yue Shi, Wei Cao, and Rui Zhang. 2019. ibtune: Individualized buffer tuning for large-scale cloud databases. *Proceedings of the VLDB Endowment* 12, 10 (2019), 1221–1234.

[67] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1009–1024.

[68] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based {KV} Stores with Matrix Container in {NVM}. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*. 17–31.

[69] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, et al. 2019. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*. 415–432.

[70] Xinyi Zhang, Hong Wu, Zhuo Chang, Shuowei Jin, Jian Tan, Feifei Li, Tieying Zhang, and Bin Cui. 2021. Restune: Resource oriented tuning boosted by meta-learning for cloud databases. In *Proceedings of the 2021 International Conference on Management of Data*. 2102–2114.

A APPENDIX

A.1 Parameter Name Reference

Table 6 shows names of parameters from Cassandra and Flink configurations corresponding to RocksDB parameters.

Param. in RocksDB	Notation	Param. in Cassandra	Param. in Flink
level0_file_num_compaction_trigger	C	min_threshold	hbase.hstore.compactionThreshold
level0_slowdown_writes_trigger	D	not applicable	not applicable
level0_stop_writes_trigger	P	not applicable	not applicable
max_bytes_for_level_multiplier	T	fanout_size	not applicable
max_bytes_for_level_base	S	not applicable	state.backend.rocksdb.compaction.level.max-size-level-base
target_file_size_multiplier	R	not applicable	not applicable
target_file_size_base	F	sstable_size_in_mb	state.backend.rocksdb.compaction.level.target-file-size-base
num_levels	L	not applicable	not applicable
max_write_buffer_number	Q	memtable_cleanup_threshold	state.backend.rocksdb.writebuffer.count
write_buffer_size	W	memtable_heap_space_in_mb	state.backend.rocksdb.writebuffer.size
min_write_buffer_number_to_merge	M	not applicable	state.backend.rocksdb.writebuffer.number-to-merge
bloom_bits	E	bloom_filter_fp_chance	state.backend.rocksdb.bloom-filter.bits-per-key
cache_size	O	cache_size_in_mb	state.backend.rocksdb.block.cache-size
block_size	B	not applicable	state.backend.rocksdb.block.blocksize
max_background_compactions	H	concurrent_compactors	state.backend.rocksdb.thread.num
max_background_flushes	U	memtable_flush_writers	not applicable

Table 6: Parameter names in Cassandra and Flink corresponding to RocksDB parameters

A.2 Pseudo code

We describe the detailed algorithm of Upper-Confidence-Bound Sampling here (Algorithm 2). To integrate Upper-Confidence-Bound Sampling with heuristic sampling, we just initialize n configurations with heuristic sampling (line 1 of Algorithm 2).

Algorithm 2: Upper-Confidence-Bound Sampling

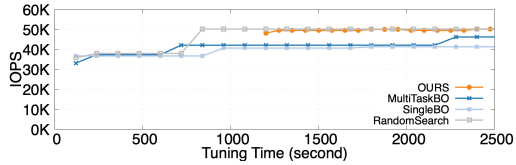
Input: n arms, R, K

1 $S \leftarrow$ sample n configs one config per arm
2 $I_i \leftarrow$ collect IOPS for each arm by runing configs in U_0
3 $\delta \leftarrow stddev(I_1 \cup I_2 \cup ... \cup I_n)$; /* global stddev */
4 $UCB_i \leftarrow mean(I_i) + \delta$
5 **for** $r = 1$ *to* R **do**
6 **for** *i-th arm in top-k ranking by UCB* **do**
7 $S.append($ one more config from i-th arm $)$
8 $I_i.append($ IOPS collected by running new-append config$)$
9 $UCB_i \leftarrow mean(I_i) + stddev(I_i)$; /* $stddev(I_i)$ is local stddev */

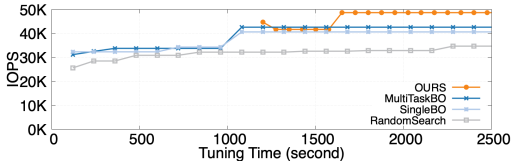
Output: config set S

A.3 Additional experiments

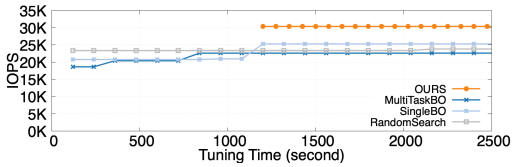
A.3.1 Tuning Time Comparison. As supplement materials of experiments in Section 5.2.2, Figure 16 presents the tuning time for Dremel and other methods on 18 settings. With the same time budget (2500s), Dremel achieves the best results on 18 settings compared against other methods. Dremel can finish the process at 1200s without executing UCB sampling. We can see that IOPS achieved by heuristic sampling is sufficiently high in experiments of Figures 16(c), 16(e), 16(f), and more. After 1200s, UCB sampling further optimize the IOPS for several experiments of Figures 16(a), 16(b), 16(e), 16(q), and more. In this experiment, we comprehensively evaluate Dremel's ability to maximize IOPS with the same or less time budget on diverse settings.



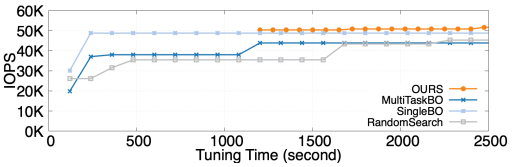
(a) No-Preloading, Skew0.3, 100MB/S, 30%GET



(b) No-Preloading, Skew0.3, 100MB/S, 60%GET



(c) No-Preloading, Skew0.3, 100MB/S, 90%GET



(d) No-Preloading, Skew0.6, 100MB/S, 30%GET

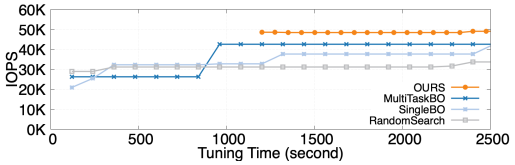
A.3.2 Optimization for Latency. As supplement materials of experiments in Section 5.2.3, Figure 17 shows that Dremel can lower the read latency to 63% of that under the default configuration for 15 settings with varying workloads and hardware.

A.3.3 Impact of Multi-fidelity Evaluation. As supplement materials for experiments in Section 5.3.3, Table 7 reports more results on the accuracy of the approximation of successive halving. From the last column, for most cases, Dremel achieves the best-seen IOPS (accuracy is 100%).

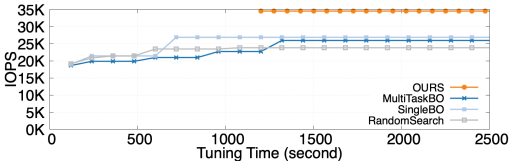
Workload	Best-Seen IOPS	Achieved IOPS	Achieved Percents
N_skew0.3_30%GET	55214	50189	90.8%
N_skew0.3_60%GET	49633	48656	98.0%
N_skew0.3_90%GET	30430	30430	100%
N_skew0.9_30%GET	65425	69371	94.3%
N_skew0.9_60%GET	76757	76757	100%
N_skew0.9_90%GET	105718	105718	100%
Y_skew0.9_30%GET	57074	57074	100%
Y_skew0.9_60%GET	65376	65376	100%
Y_skew0.9_90%GET	83399	83844	99.4%

Table 7: Accuracy of multi-fidelity approximated evaluation. SSD bandwidth is 100 MB/S. In the first column, N denotes no-preloading while Y denotes preloading.

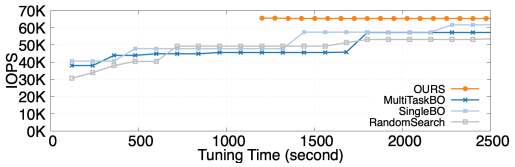
A.3.4 Impact of Upper-Confidence-Bound Sampling. As supplement materials for experiments in Section 5.3.4, Figure 18 presents one example of UCB sampling runtime trace (one application thread) under the setting of preloading. We can see that successive halving works well for the preloading setting. From left to right, Dremel selects more configurations from promising arms with higher confidence bounds. It demonstrates that UCB sampling is applicable for both with-proloading and no-preloading settings.



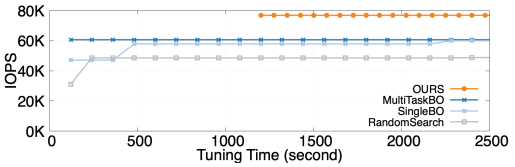
(e) No-Preloading, Skew0.6, 100MB/S, 60%GET



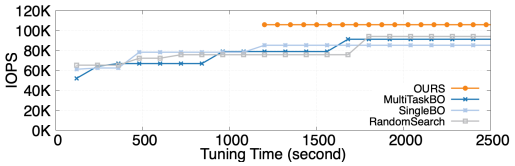
(f) No-Preloading, Skew0.6, 100MB/S, 90%GET



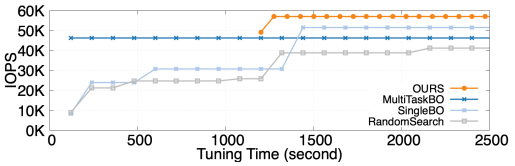
(g) No-Preloading, Skew0.9, 100MB/S, 30%GET



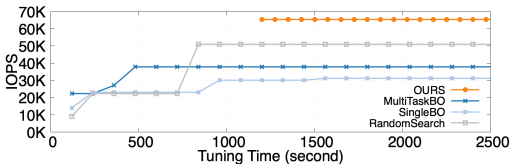
(h) No-Preloading, Skew0.9, 100MB/S, 60%GET



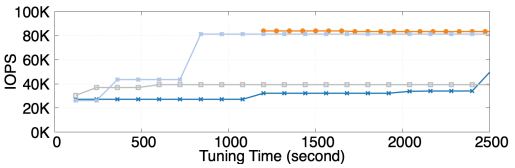
(i) No-Preloading, Skew0.9, 100MB/S, 90%GET



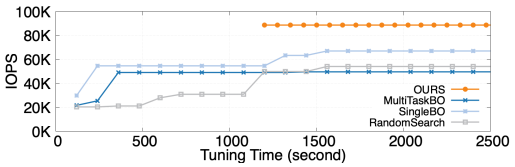
(j) With-Preloading, Skew0.9, 100MB/S, 30%GET



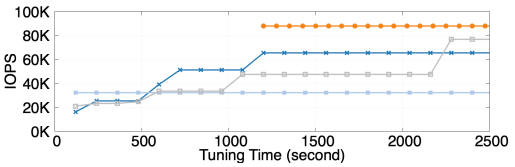
(k) With-Preloading, Skew0.9, 100MB/S, 60%GET



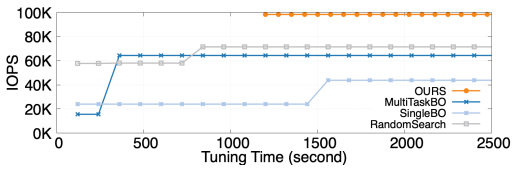
(l) With-Preloading, Skew0.9, 100MB/S, 90%GET



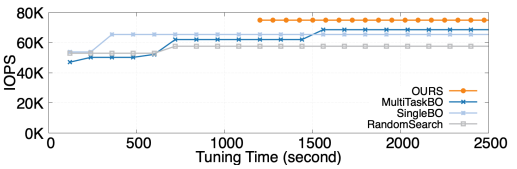
(m) With-Preloading, Skew0.9, 200MB/S, 30%GET



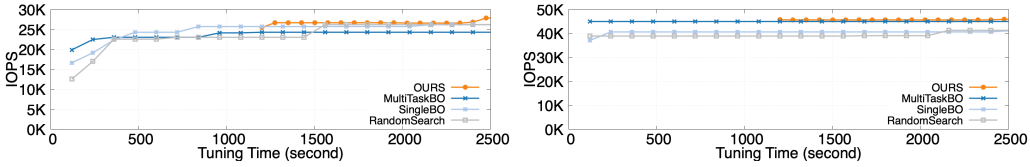
(n) With-Preloading, Skew0.9, 200MB/S, 60%GET



(o) With-Preloading, Skew0.9, 200MB/S, 90%GET

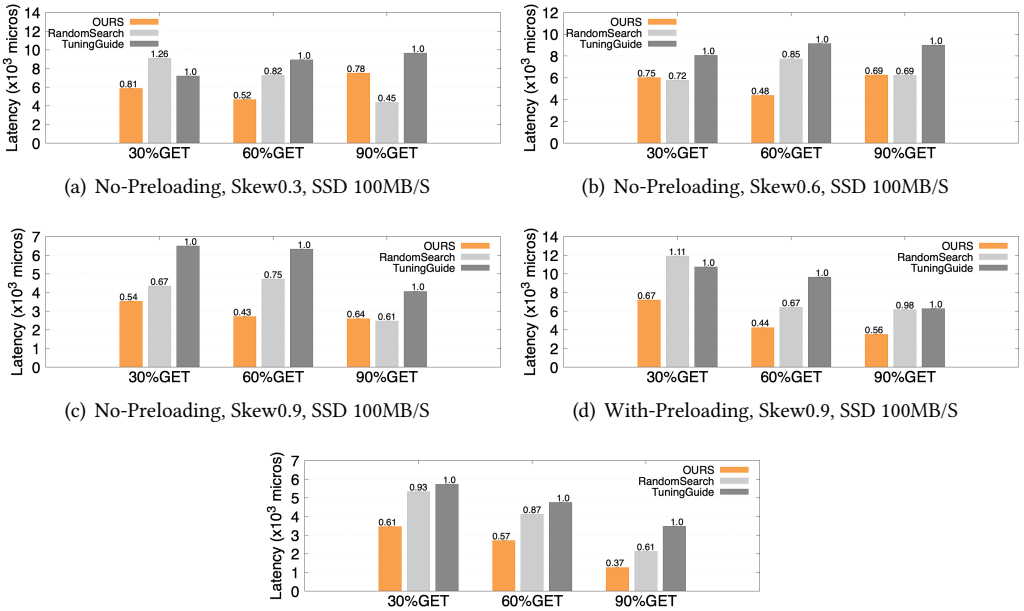


(p) YCSB-A Setting



(q) YCSB-B Setting (r) Nutanix Setting

Fig. 16: Tuning time comparison for 18 settings of workload and hardware. One setting for each figure.

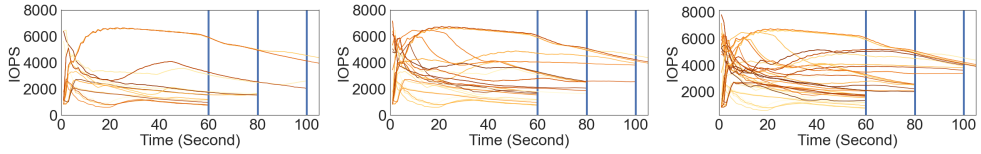


(a) No-Preloading, Skew0.3, SSD 100MB/S (b) No-Preloading, Skew0.6, SSD 100MB/S

(c) No-Preloading, Skew0.9, SSD 100MB/S (d) With-Preloading, Skew0.9, SSD 100MB/S

(e) With-Preloading, Skew0.9, SSD 200MB/S

Fig. 17: Minimize read latency (99%) for 15 different settings of workload and hardware.



(a) UCB-0 (b) UCB-10 (c) UCB-20

Fig. 18: With-Preloading, Skew0.9, 100MB/S, 90%GET

Received February 2022; revised March 2022; accepted April 2022