When Database Meets New Storage Devices: Understanding and Exposing Performance Mismatches via Configurations

Haochen He
National University of Defense Technology, China
hehaochen13@nudt.edu.cn

Erci Xu
National University of Defense Technology, China
xuerci@nudt.edu.cn

Shanshan Li*
National University of Defense Technology, China
shanshanli@nudt.edu.cn

Zhouyang Jia
National University of Defense Technology, China
jiazhouyang@nudt.edu.cn

Si Zheng
National University of Defense Technology, China
si.zheng1009@gmail.com

Yue Yu
National University of Defense Technology, China
yuyue@nudt.edu.cn

Jun Ma
National University of Defense Technology, China
majun@nudt.edu.cn

Xiangke Liao
National University of Defense Technology, China
xkliao@nudt.edu.cn

ABSTRACT

NVMe SSD hugely boosts the I/O speed, with up to GB/s throughput and microsecond-level latency. Unfortunately, DBMS users can often find their high-performanced storage devices tend to deliver less-than-expected or even worse performance when compared to their traditional peers. While many works focus on proposing new DBMS designs to fully exploit NVMe SSDs, few systematically study the symptoms, root causes and possible detection methods of such performance mismatches on existing databases.

In this paper, we start with an empirical study where we systematically expose and analyze the performance mismatches on six popular databases via controlled configuration tuning. From the study, we find that all six databases can suffer from performance mismatches. Moreover, we conclude that the root causes can be categorized as databases' unawareness of new storage devices characteristics in I/O size, I/O parallelism and I/O sequentiality. We report 17 mismatches to developers and 15 are confirmed.

Additionally, we realize testing all configuration knobs yields low efficiency. Therefore, we propose a fast performance mismatch detection framework and evaluation shows that our framework brings two orders of magnitude speedup than baseline without sacrificing effectiveness.

PVLDB Reference Format:
Haochen He, Erci Xu, Shanshan Li, Zhouyang Jia, Si Zheng, Yue Yu, Jun Ma, and Xiangke Liao. When Database Meets New Storage Devices: Understanding and Exposing Performance Mismatches via Configurations. PVLDB, 16(7): 1712 - 1725, 2023.
doi:10.14778/3587136.3587145

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/TimHe95/S3M.

*Corresponding author.

Proceedings of the VLDB Endowment, Vol. 16, No. 7 ISSN 2150-8097.
doi:10.14778/3587136.3587145

1 INTRODUCTION

Database Management System (DBMS) practitioners are eternally in pursuit of better performance. One straightforward approach is to leverage the direct benefits of hardware advancement. A notable example is the NVMe SSD, which can deliver up to 6GB/s throughput and 10μs-level latency [13, 34, 42], far beyond the performance of the SATA SSDs and HDDs.

However, simply shoehorning NVMe SSDs into existing DBMSs may not always have the desired effect. Various user reports indicate that upgrading to NVMe SSDs can yield minimal improvement, or even have negative impacts on performance [22, 25, 28, 29]. For example, one user claims that, under the same setup on MySQL, NVMe SSD only delivers half of the SATA SSD performance [7].

We discover that the reason behind the *performance mismatch* in these reports is the inconsistency between the DBMS behavior and device characteristics. Specifically, for NVMe SSDs, the drastic changes made both internally (e.g., adopting novel NAND architecture [67, 68]) and externally (i.e., NVMe interface) not only offer better performance but also fundamentally alter the I/O characteristics. For example, in NVMe SSD, random I/O speed is close to that of sequential I/O rather than lagging severely behind [13, 34, 42]. However, decades of DBMS development are mostly built on and optimized for the I/O patterns of traditional devices like HDD. Hence, these optimizations may now lead to performance degradation on NVMe SSD.

Regarding the previous report [7], we find out that the root cause is the size of DBMS write requests. Normally, DBMS sets the page (i.e., write unit) size relatively small (e.g., 4KB or 8KB) to be consistent with the physical sector size in HDD (512B to 4KB). Meanwhile, to increase performance, storage devices are equipped with caches. But the size of a cache line may be large (e.g., 32KB). If the DBMS write (followed by `fsync`) is smaller than the cache line, as shown in Figure 1, data has to be padded and written, which may cause side-effects (e.g., write amplification). By increasing the database page size to 32KB in this example, we are able to speed up MySQL performance on NVMe SSD by 15.3%, but it is still slower than that on SATA SSD. Therefore, such performance mismatch may not be able to simply solved by configuration tuning [88, 95, 97].

In this paper, we focus on studying and exposing the performance mismatches between DBMSs and new devices. And our target is to

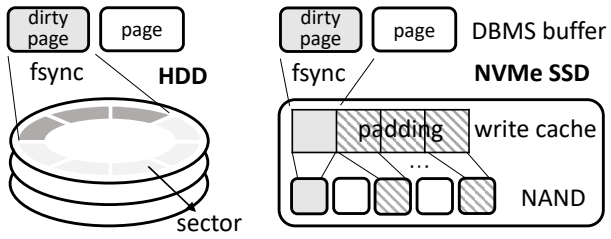

Figure 1: The I/O pattern of DBMS mismatches underlying NVMe SSD's internal I/O property.

help developers adapt DBMSs to the characteristics of NVMe SSDs. Specifically, we begin with a comprehensive empirical study on performance mismatches covering three tiers of devices (i.e., HDD, SATA SSD, NVMe SSD) by comparing the performance between different configurations. The reasons are two-fold: first, databases have become increasingly configurable [94]. For example, MySQL InnoDB storage engine has 529 tuning knobs, covering nearly all aspects of functionalities. Second, the configurations provide an ideal controlled environment for verifying the impacts on different storage devices. Additionally, we filter out I/O-related knobs via static analysis to improve efficiency.

As a result, the study reveals that all the six DBMSs in our study have suffered from performance mismatches. Furthermore, by quantitatively analyzing the I/O stack, we discover that the mismatches of I/O size, parallelism, and sequentiality between DBMS and devices are the culprits.

Our paper makes the following contributions:

- We perform a comprehensive study on performance mismatch in six popular databases and concluded their root cause patterns.
- We designed and implemented a testing framework which leverages configurations to trigger the root cause patterns to expose performance mismatches, and uses taint analysis and light-weight dynamic monitoring to reduce the testing costs.
- We identify 17 new mismatches for the six DBMSs in total, and 15 of them have been confirmed by developers.

2 EXPERIMENT DESIGN

While anecdotes on DBMS performance mismatch with NVMe SSDs are abundant in the field, no thorough study on this issue has yet been conducted. Here, to quantitatively measure the impact and identify the root cause, we conduct an extensive study on performance mismatch in DBMS. At a high level, to expose potential performance mismatches, we run the same sets of workloads on the targeted DBMS atop different devices (e.g., NVMe SSD and HDD), and check if the outcome is expected (i.e., faster on NVMe SSD). We also tune the DBMS configurations to achieve different setups (e.g., with/without specific optimization) of the DBMS environment.

In this section, we begin by introducing the study platform, including the targeted DBMS, the workloads, and the candidate devices. We go on to discuss the methodology in detail, including the configuration selection, performance mismatch checking, and root cause reasoning.

Table 1: Storage devices used in the study.

Label	Model	Capacity	Interface
NVMe SSD_A	Western Digital SN850	500GB	NVMe
NVMe SSD_B	Samsung 980 Pro	500GB	NVMe
SATA SSD_A	Western Digital SA510	500GB	SATA
SATA SSD_B	Samsung 860 Evo	500GB	SATA
HDD_A	Dell Exos 15Krpm	600GB	SAS
HDD_B	Seagate BarraCuda	2TB	SATA

2.1 Test Platform

Databases. We choose six DBMSs (i.e., MySQL, PostgreSQL, SQLite, MariaDB, MongoDB and Redis) for this study. Our selection rationale is based on their popularity [32, 35], code accessibility (for root cause analysis), and community activeness (for submitting issues and discussing with developers). We use the latest (as of the beginning of our test) versions of these databases (see Column 2 in Table 3) to ensure the inclusion of recent updates and optimizations for storage devices.

Storage Devices. Table 1 lists the devices used in the study. Our study covers three tiers of devices, including HDD, SATA SSD, and NVMe SSD. To avoid biased conclusions led by specific drive models, for each tier, we select two representative device models from two vendors. All drives are popular off-the-shelf products and in brand-new condition.

Workloads. We use the two most popular DBMS benchmarks (i.e., YCSB and TPC, including TPC-C, TPC-DS and TPC-E) to generate our test workloads. For each benchmark, we set the dataset size to 100GB in order to exert enough I/O pressure on devices without incurring much background noise (e.g., garbage collection and wear leveling tend to be influential when the SSD utilization rate is high). The remaining settings of the benchmarks are left at their default values.

System settings. We set the maximum memory for all databases under test to 16GB. We run all databases on the Linux vanilla storage stack with ext4 file systems. Note that we disable the consistency configurations (i.e., `noatime` and `nobarrier`) and `journal` to reduce the influence from the kernel storage stack.

2.2 Methodology

In this section, we detail the design of our experiments. Figure 2 shows the experiment framework: first, we generate test cases for configurations; then, we collect runtime information to identify the performance mismatch; last, we reason about the root causes manually based on the collected information and other auxiliary experiments.

Enumerating all combinations of storage devices, configuration knobs, knobs combinations and knob values is infeasible, consuming a huge amount of time. The major reason is DBMSs usually have a large number of configuration knobs. Our main target is to expose I/O related performance mismatches by testing, so we only need to test I/O related knobs.

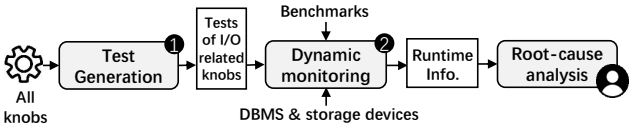

Figure 2: The framework to expose and understand performance mismatch.

In this section, we first elaborate on how we select I/O-related knobs via taint analysis; subsequently, we describe the generation of values for the I/O-related knobs and the performance tests.

I/O-related Knobs Identification. The key characteristic of I/O-related knobs is that they can influence I/O-related syscalls. Therefore, we divide this step into two: first, we identify I/O-related syscalls; then, we use taint analysis to identify knobs that have control-flow and/or data-flow influence on I/O-related syscalls.

I/O-related syscalls. Many syscalls are related to I/O (e.g., `write`, `unlink`). We manually investigated every Linux syscall (335 found in kernel version 5.4.0) by reading the official manual, followed by cross-checking, and filter out 21 that may affect I/O size, parallelism, and sequentiality. These syscalls can be further categorized into four series: 1) *read* series (e.g., `pread`), 2) *write* series (e.g., `pwrite`), 3) *sync* series (e.g., `fsync`), and 4) the syscalls that control the number of threads/processes (DBMS may use separate threads/processes, such as `clone`, to issue I/O requests). Table 2 shows these syscalls.

Connecting knobs to I/O-related syscalls. The input to this step is the source code of the target DBMS and its configuration knobs, and the output is I/O-related knobs. First, we use an existing tool [96] to locate the program variables corresponding to knobs (i.e., knob variables). Subsequently, we determine whether the knob variables have connections with I/O-related syscalls.

Note that existing works [38, 39, 43, 60, 70] have focused on building the connections between specific functions and knob variables. Though effective, these approaches have some limitations. Dynamic methods [39, 43, 60] are dependent on specific inputs, such as bug-triggering input. For their part, static methods can be Java-specific [70] or only handle basic data-flows [38]. Notably, this is insufficient because 1) DBMSs are usually large-scale (e.g., MySQL has over three million lines of code), meaning that the data-flow can be complicated such as those indicated by pointers and field-sensitive analysis, and 2) besides data-flow, where the knob goes straight to one of the arguments of the syscall (e.g., "`pwrite(.., count=knob, ..)`"), control-flow connections also exist: a knob can determine if a syscall can be executed (e.g., "`if( knob) fsync()`"), or how many times a syscall can be executed (e.g., "`while(knob) pwrite(..)`"). To solve the issues, we propose a static taint analysis approach that can handle complicated data- and control-flow in large-scale C/C++ systems.

Data-flow connections. We conduct taint-analysis starting from knob variables, and detect if any of key syscalls are tainted. We support inter-procedure, field-sensitive taint analysis and also have supports for pointer analysis. Moreover, the traditional data-flow is typically based on the Use-Define Chain [19] (e.g., "`a = knob; b = a`"). During our study, we also found cases beyond the Use-Define Chain patterns, for example, "`if(knob) a+=1; else a-=1`", which indicates that the value of "`a`" is dependent on "`knob`" in every path. Therefore, we extend the traditional data-flow by supporting this new data-flow pattern. Specifically, the example above will generate an IR instruction "`%a = phi i1 [ %a_plus, %knob.true ], [ %a_minus, %knob.false ]`", where `%knob.true` and `%knob.false` indicates which branch the program really goes to. Therefore, if the knob variable can decide, i.e., dominate [9], at least one (but not all) of the branches, the knob variable can have a data flow going to the variable `%a`. And such cases are tainted.

Table 2: The 21 syscalls may affect I/O size, parallelism, and sequentiality.

read series	**write series**
read pread64 readv preadv	write pwrite64 writev pwritev
preadv2 io_getevents	pwritev2 io_getevents
io_submit madvise open mmap	io_submit madvise open mmap
sync series	**thread series**
fsync fdatasync syncfs	clone (pthread_create)
sync_file_range fcntl	fork

Control-flow connection. The control-flow can be very complex due to the presence of plentiful code structures, including but not limited to, `if`, `switch`, `for`, `while`, `break`, `return`, and their intersection and/or nesting. We handle all these scenarios by computing if a certain syscall (or its wrapper) is on the path that is dependent on the knob variable. Note that the typical *control-dependency* [50] only covers immediate dependency. Take the code fragment "`if (knob){ if(x) foo(); else bar();} fun();`" as an example, function `foo` and `bar` only get executed if the knob variable is true. But the immediate dependencies of `knob` are only `if(x)` and `fun`. Thus, we extend the control-dependency algorithm to handle multiple layers of dependency by making the control-dependency *transitive* [41].

Knob value generation. After the previous step, the number of knobs that need to be tested can be significantly reduced. It is however still difficult to test all values for numeric I/O-related knobs. For example, the value range of the knob `max_worker_processes` from PostgreSQL covers five orders of magnitude. So we manually inspect the numeric I/O-related knobs, and find that all of them can be categorized into two types: first, knobs that control the concurrency of I/O (e.g., the maximum background writer threads DBMS can has); second, knobs that control I/O timing, for example, how long an I/O should delay after a specific event. These knobs are typically tested exponentially [56, 89], so we generate one value at each order of magnitude for these knobs. As for enum knobs, we generate tests for every value (few enum knobs have more than four values [56, 92]). Meanwhile, we also use existing methods to extract configuration constraints [92] in order to avoid generating mis-configurations.

Pruning search space. Note that the search space of the combination of configuration knobs is known to be enormous [55, 65]. Many configuration sampling techniques [45, 63, 66, 76] have been proposed to reduce the search space. But none of the techniques can be applied directly, because they change two or more knobs at

each sampled configuration, while we need to observe the performance change after tuning one specific knob (so that if the change implies a performance mismatch, we can confirm that the mechanism or the optimization behind the knob may be the culprit). Therefore, we change one knob at each time. Given a DBMS with 3 knobs c_1, c_2, c_3 with value range $R_1 = \{0, 1\}, R_2 = [0, 100], R_3 = \{$"a", "b"$\}$. We will generate 8 combinations: $c_1 = 0, c_2 = \cdot, c_3 = \cdot$; $c_1 = 1, c_2 = \cdot, c_3 = \cdot$; $c_1 = \cdot, c_2 = 0, c_3 = \cdot$; $c_1 = \cdot, c_2 = 1, c_3 = \cdot$; $c_1 = \cdot, c_2 = 10, c_3 = \cdot$; $c_1 = \cdot, c_2 = 100, c_3 = \cdot$; $c_1 = \cdot, c_2 = \cdot, c_3 =$ "a" and $c_1 = \cdot, c_2 = \cdot, c_3 =$ "b" where "$\cdot$" represents for the default value. For dependent configurations (i.e., the value of one knob affecting another), we resort to official guides and previous practice: first, if the official documents explicitly point out that "c_3 works only when c_1 is turned on", we manually set the constraint for these knobs (e.g., $c_1 = 1, c_2 = \cdot, c_3 =$ "a" and $c_1 = 1, c_2 = \cdot, c_3 =$ "b"); then, we leverage existing tools [48, 92] to extract the configuration constraints.

Test Oracle. We compare the performance (i.e., latency and throughput) before and after we change the value of a particular configuration knob. If the performance comparison shows one of the performance mismatch symptoms, we record it as a potential mismatch and determine the root cause. Specifically, we apply a ***heuristic rule*** to determine mismatch symptoms: after changing the value of a knob, the performance change in higher-tier devices (NVMe SSD > SATA SSD > HDD) is counter-intuitive compared with that in lower-tier ones. For example, after turning on `fsync` in MySQL, the TPC-C performance dropped by 1.1x in HDD_A but drop by 8.7x in NVMe SSD_B. The intuition behind this rule is that: the counter-intuitive performance change implies that the optimization/mechanism controlled by the knob is not adaptive to the newer device, which indicates a performance mismatch to a large extent.

Identifying root cause. After a performance mismatch is exposed, we first diagnose the I/O path from DBMS to device driver via `blktrace` [4], Linux kernel event tracing [11], and eBPF [10]. Next, we verify the issue by using `fio` [15] to create a DBMS-independent test suite and subsequently pin down the root cause using control-variable analysis [8]. For example, we assume several factors can be the cause of a specific performance mismatch, then we change every factor (i.e., `fio` arguments or other system settings) to observe which factor(s) contribute to the performance drop. After we finish the performance mismatch analysis in one DBMS, we also perform cross-checks to determine whether the same mismatch exists in other DBMSs.

3 RESULTS & ANALYSIS

In this section, we first present an overview of the testing results obtained by our study and high-level observations on the performance mismatches (§ 3.1). Then, we elaborate on the categorization and in-depth root cause analysis of performance mismatches (§ 3.2-3.4).

Note that in the previous section, we assume only I/O related knobs can expose mismatches so we design a method to filter these knobs out. To validate the assumption, we still generate tests for every knob and draw findings from their test results.

3.1 Results Overview

The left part of Table 3 presents the basic information about the tests generated for our study following the methodology outlined in §2. In total, we generate 66,092 tests that consider both write-intensive (W), and read-intensive (R) workloads. These tests cover a total of 1,543 knobs in the six DBMSs. Note that some read-intensive benchmarks (e.g., TPC-H) contain many sub-tests (e.g., 22 different tests for TPC-H), while the write-intensive workloads do not; thus, there are more tests for read-intensive workloads.

The right part of Table 3 contains the overall test results of our study. First, the whole testing procedure costs eight months (510 machine days). As a result, 432 tests violate the heuristic rule (§ 2.2); these tests expose 123 potential performance mismatches, including 10 mismatches were identified by our cross-checking ("**# PPM**"). These 20 cases do not violate the rule but are discovered during the process of cross-checking between DBMS (i.e., *false negatives* of the rule). For the other 103 cases, a potential performance mismatch is at least exposed by four tests (e.g., knob values *on* and *off* on devices *A* and *B*), and also, some tests expose multiple mismatches. Through manual inspection, we determine how many of the 123 potential mismatches have ever been reported by DBMS users in public areas (e.g., StackOverflow) by searching for specific keywords (e.g., "NVMe SSD", "performance", "slow"). Our results show that similar reports have been made in public areas for 10/123 cases ("**# Public**"). Moving to the next column, the 123 mismatches touch 56 different knobs ("**# Knobs touch**"). Note that, on average, one knob exposes 2.2 (123/56) potential mismatches. This is mainly because different triggering workloads for one touched knob are counted separately. Moreover, if any of the two knobs (within the 56 touched knobs) have dependencies, they will produce three potential mismatches (i.e., exposed by `Knob-A`, `Knob-B`, and `Knob-A & Knob-B`). Finally, we measure the extent to which performance mismatches can affect performance ("**Penalty**"). We use an example to show how the penalty is calculated: if, after changing a knob, the query latency in HDD degrades to 1.3x, while the latency in NVMe SSD degrades to 2.6x, then the performance penalty of this mismatch is $100\% \cdot 2.6/1.3 = 200\%$. As the table shows, the performance penalty caused by these mismatches varies, ranging from 12% to 215%. We do not expose any performance mismatches by the read-intensive workload in SQLite, because few SQLite optimizations on read operations can be controlled by configurations. From the results above, three observations can be made regarding performance mismatches:

- **Previously unknown.** Only 10/123 mismatches could be found in public areas, while the rest of them are new (we categorize and reported these mismatches to developers, see Table 4 in §4 for details). Unlike performance bugs [62], which can be observed at version upgrades (i.e., performance regressions), performance mismatches may only be observed when upgrading the devices, which is a rarer scenario that remains under-considered in current in-house testing practice.
- **Severe impacts.** Among the studied DBMSs, MySQL and PostgreSQL are seriously affected, while SQLite experiences the smallest impacts. This does not however imply that

Table 3: Summary of testing results in our study.

Test Information					Results					
DBMS	**Version**	**# Knobs**	**Workload**	**# Tests**	**Cost**	**# Violation**	**# PPM**	**# Public**	**# Knobs touch**	**Penalty**
MySQL	8.0.22	529	R	21,233	124d	69	13 (2)	2	8	14%-94%
			W	2,915	42d	52	19 (0)	4	12	12%-146%
PostgreSQL	12.11	256	R	8,965	55d	55	12 (4)	1	5	78%-215%
			W	1,230	19d	41	13 (3)	0	8	16%-122%
SQLite	3.36.0	74	R	2,836	21d	-	0 (0)	0	-	-
			W	389	7d	14	4 (1)	0	3	21%-33%
MariaDB	10.6.3	418	R	16,851	90d	55	11 (2)	1	8	12%-42%
			W	2,313	31d	44	13 (0)	1	12	16%-192%
MongoDB	5.0.0	169	R	4,045	51d	31	8 (2)	0	2	-
			W	1,906	26d	24	8 (2)	1	2	43%-67%
Redis	6.0	97	R	2,305	29d	29	10 (2)	0	2	-
			W	1,104	15d	18	12 (2)	0	3	22%-84%
Total	-	1,543	-	66,092	510d	432	123 (20)	10	56†	12%-215%

 # Knobs: number of all knobs; **Workload**: "W" write-intensive (TPC-C and YCSB-A & F), "R" read-intensive (TPC-E/H/DS and YCSB-B~E); **# Tests**: number of tests conducted; **Cost**: time consumption in machine days; **# Violation**: number of tests that violate the heuristic rule in § 2.2; **# PPM**: number of potential performance mismatches exposed (may duplicate), $_{(\cdot)}$: PPMs found by cross-check (i.e., *false negatives* of the rule); **# Public**: number of PPMs can be found in public areas; **# Knobs touch**: number of knobs touched by those PPMs; **# Penalty**: performance penalty in new devices of those PPMs; †de-duplicated sum.

SQLite has fewer mismatches. By contrast, when we manually perform the mismatch cross-checking between DBMSs, we confirm (following consultation with developers) that SQLite has the same types of mismatches as the other two DBMSs.

- **Write sensitive.** Comparing "R" rows with "W" rows in Table 3, we can observe that the write-intensive workloads are more likely to trigger performance mismatches. This is consistent with our further root cause analysis, which holds that database developers make more performance mismatches in writes than in reads (i.e., mismatch in write size § 3.2).

Category of performance mismatches. We manually analyze the root causes of the 61 potential performance mismatches, and find that most (100/123) of them fall into three categories, in which DBMSs mismatch the I/O characteristics of new devices in three critical [46, 47, 57, 64] aspects that are closely related to performance:

- **Size** in **write** (§ 3.2). DBMSs issue too small force-write requests, causing SSD's write cache to work in an inefficient way.
- **Parallelism** in **write** and **read** (§ 3.3). DBMSs issue almost-serialized I/O requests, causing the internal parallelism of new devices to be largely wasted.
- **Sequentiality** in **write** and **read** (§ 3.4). DBMSs usually consume substantial amounts of resources (CPU, memory) converting random I/O to sequential; while the speed gap between the two types of I/O is small in NVMe SSDs, making the conversion less fruitful or even wasteful.

In §3.2-3.4, we elaborate on the three types of mismatches with cases studies.

False positives. The testing framework uses a coarse-grind heuristic rule to identify the mismatches. However, the rule can be inaccurate because it does not leverage mismatch root causes. As a result, the rule produces nine *false positives*. The nigh cases do not belong to any of the three types of mismatches, even though they violate the rule. The reason lies in the fact that the corresponding knobs restrict I/Os by force to prevent the system from being stuck. For example, MySQL allows for the use of `io_capacity` to throttle the I/Os on slow devices (e.g., HDD) to prevent the entire system from becoming stuck due to intensive I/O, which results in the under-utilization of the capacity of new devices (e.g., NVMe SSD).

False negatives. Ten false negatives are found by our manual cross-checking (see Column "# PPM" in Table 3), rather than by rule violation. The reason is that mismatches do not necessarily cause a violation of the rule. Taking the case of PostgreSQL (details in §3.3) as an example, the mismatch always exists no matter how an I/O related knob (whose use is adapting for new storage devices) is changed, and the performance also remains the same (i.e., rule not violated).

We find all performance mismatches are triggered by I/O-related knobs, meaning that only testing I/O-related knobs will not produce false negatives.

3.2 Mismatch in I/O Size

Unaligned writes (e.g., triggered by frequent DBMS flush) can be harmful to SSD, whose write cache has to align the writes in some inefficient ways.

Symptom. DBMS can adjust the flush frequency to achieve higher performance (i.e., fewer flushes; when reliability is guaranteed by hardware) or higher reliability (i.e., more flushes, typically the default setting) via configuration knobs. Intuitively, more

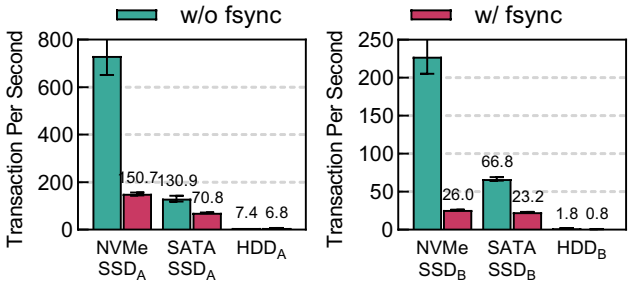

Figure 3: Performance drop of MySQL on NVMe SSD, SATA SSD and HDD after aggressively issue `fsync`.

flushes cause fewer writes to be buffered and merged. Thus, a high flush frequency can incur more small I/Os, leading to performance drops. However, we discover that NVMe SSDs suffer much more from the frequent-flush penalty than SATA SSDs/HDDs. For example, in our tests, after increasing the flush frequency via `innodb_flush_method=fsync` and `innodb_doublewrite=1` in MySQL, the average latency of an individual TPC-C query experiences severe drops on NVMe SSDs (4.9-8.7x), while we only observe (1.8-2.9x) drops on SATA SSDs and HDDs (1.1-2.25x), as shown in Figure 3.

Root cause diagnosis. One of the reasons is that the small writes are smaller than the size of SSD's cache line, which is typically large (e.g., 32KB), causing the SSD to fill the line with dummy data or wait for a timeout. As shown in Figure 4 (see NVMe SSD$_A$, SATA SSD$_A$ and SATA SSD$_B$), in our raw device experiments, if `fsync` is disabled (green lines), the write caches work as expected, i.e., improve write performance (I/Os per second, IOPS for short). Otherwise (`fsync` is enabled, red lines), the write caches work counter-productively and the IOPS is very sensitive to the size of write (i.e., the caches work normally only when write size is aligned to 32KB in NVMe SSD$_A$ and at least 32KB in SATA SSD$_B$). Another reason is that `fsync` can be extremely harmful to NVMe SSD$_B$, causing two more orders of magnitude of performance degradation. In summary, recall that more DBMS flushes incur more small writes (whose size usually equals to the DBMS page size, e.g., 8KB), which fit well with the HDDs while causing performance penalties to SSDs.

One might also speculate as to whether the slowdown is caused by OS. To answer this, we first confirm that I/O is the bottleneck for DBMS performance, then the OS imposes less than 10% overhead on I/O via DBMS internal profiling and `blktrace` respectively.

Developers' feedback. In our discussions with developers, they admit that simply increasing the DBMS page size (i.e., the unit of writes) can not solve the mismatch problem. The reasons are as follows: first, changing page sizes requires mandatory rebooting [5, 6] or even re-compiling the DBMS [31]; second, large DBMS page sizes may cause more unchanged data to be written to the devices during a page flush; third, DBMSs (e.g., SQLite) only use pages to organize data, but not metadata and logs. In this case, such an I/O can incur a mismatch in I/O size. For example, SQLite flushes a fixed-length (4KB) log header for *every* page being flushed. Worse still, Redis does not use the page to organize memory, so the size of write can be small and arbitrary (e.g., after OS padding, Redis issues many 4KB and 8KB writes). To optimize the behavior above, the developers are considering both re-designing the data file format and adding an option to let users place the log on another device as a workaround. PostgreSQL and Redis developers are also rethinking and redesigning their I/O subsystem to be more adaptive to the new characteristics of NVMe SSDs.

3.3 Mismatch in I/O Parallelism

Although DBMSs can use multiple threads to issue I/O concurrently, the internal parallelism of NVMe SSD still remains under-utilized because DBMSs often, if not always, issue I/Os in a synchronous manner.

Symptom. DBMSs allow users to specify multiple I/O workers via configurations. An SSD, especially an NVMe SSD, has multiple structural levels of parallelism [26, 46], which can serve I/Os concurrently. Therefore, intuitively, sending I/O requests to NVMe SSDs in parallel should yield higher performance. For example, increasing I/O worker threads from 1 to 64 via `write_io_threads` in MySQL improve TPC-C performance by three times. However, we find that is not true for all DBMS. For example, increasing `effective_io_concurrency` in PostgreSQL does not benefit performance and even hurts performance significantly (see Figure 5). We observe similar cases with PostgreSQL in 4/6 DBMSs (others are MongoDB, Redis and SQLite). And even though MySQL can be benefit from concurrent I/O workers, the storage device's throughput (i.e., 285MB/s on average) is still far from the speculation (i.e., 2200MB/s random write [42]).

Note that the throughput increases together with the number of workers on HDD in MySQL. It occurs because a single MySQL worker can reject I/Os to slow devices (e.g., HDD) due to memory considerations [23]; these rejected I/Os have to be retried repeatedly, but can be served by more workers.

Root cause diagnosis. The root cause is as follows: I/Os are issued in a synchronous manner by DBMSs design (while asynchronously issuance does occur, this is only for specific workloads), under-utilizing the parallelism of NVMe SSD. The ideal parallelism is realized by sending I/Os to 64K nvme-queues with a maximum depth of 64K each [26, 46]. However, DBMSs issue I/Os either to only one queue and/or until the previous one has finished (i.e., the queue depth is 1 most of the time) most (i.e., more than 95%) of the time. Worse still, DBMSs fail to provide a better way to issue I/O to alleviate such under-utilization. For example, PostgreSQL can only issue paralleled I/Os for specific workloads, e.g., paralleled "index bitmap scan" via specifying higher `effective_io_concurrency` and for other workloads, the OS overhead may affect performance; MySQL uses `libaio`, an asynchronous I/O library, but in a synchronous manner (with buffered I/O) [17].

Interestingly, we also find that 5/6 (except for MariaDB) of the DBMSs have never changed their synchronous I/O engines since they were first released. This is possibly because DBMSs need to keep the I/O order with `fsync`, and ordered asynchronous I/O is less efficient. Meanwhile, HDDs (who have only one magnetic I/O head) and SATA SSDs have one or few concurrent I/O queues, so

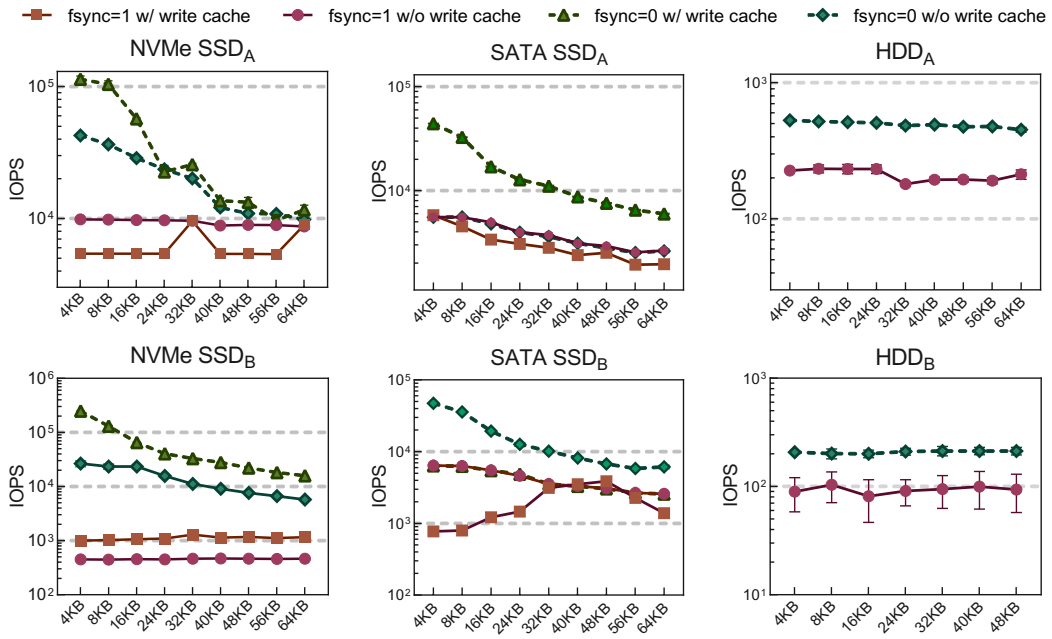

Figure 4: IOPS over different settings and I/O sizes in NVMe SSD, SATA SSD and HDD.

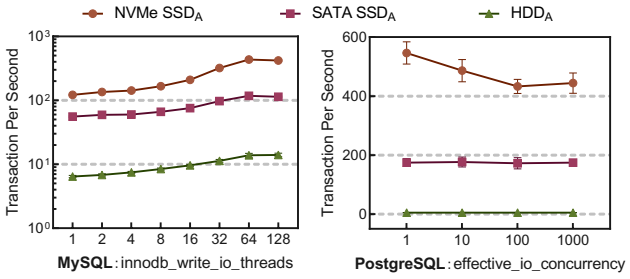

Figure 5: TPC-C throughput on NVMe SSD, SATA SSD and HDD with different I/O concurrency.

ordered synchronous I/O is a natural fit. But NVMe SSDs supports up to 64K I/O queues, using the ordered synchronous I/O is a waste of concurrency [73]. In fact, recent works [73, 74, 90] have proposed approaches to increase the concurrency level for asynchronous I/O for NVMe SSDs.

Developers' feedback. After discussions with the developers, MySQL, PostgreSQL, Redis and MongoDB admit that the current database I/O subsystems are far from highly concurrent. Thus, they are planning to make improvements by switching to new I/O interfaces like `io_uring` and redesigning the I/O subsystems (MariaDB is already doing so). Moreover, PostgreSQL developers are going to increase the number of scenarios in which many workers can work in parallel. However, SQLite developers operate on the principle of making the database as simple as possible (i.e., single-thread synchronous I/O); as a result, they are unwilling to make big changes.

3.4 Mismatch in I/O Sequentiality

Transforming random I/Os into sequential I/Os may hurt performance in NVMe SSDs, because the gap between random and sequential I/O speed is very narrow, while the cost of the transformation is high.

Symptom. Random-to-sequential I/O transformation is a common DBMS optimization that can be tuned by knobs (typically turned on by default). In traditional devices like HDDs, I/O is far slower than CPU and memory, and the gap between random and sequential I/O is huge; intuitively, therefore, such a transformation will be fruitful. Notably, we observe that the opposite is true in SSDs. For example, when switching on "change buffer" optimization via `innodb_change_buffering`, MySQL will buffer all updates to index data (which are usually randomly located in the device), merge them, and write them down to the device sequentially. As a result, the throughput of TPC-C in MySQL on HDD_A is improved by 63.0% as shown in the left of Figure 6. However, in NVMe SSD_A, there is a slight performance drop of 3.5%. In addition to write operations, we observe that random read transformations have a similar symptom.

Root cause diagnosis. The cost of random-to-sequential transformation outweighs its benefits in NVMe SSD. Regarding these benefits, we find that the theoretical maximum performance gain of transforming random I/O to sequential I/O in HDDs reaches two orders of magnitude (measured with `fio`), while the maximum gain in NVMe SSDs is slight because the random I/O speed in NVMe SSD is close to the sequential I/O speed [69]. As for the cost, I/O speed in NVMe SSD (μs level latency) is much higher, making the CPU is both increasingly valuable and more likely to impact system performance [69, 91]. Thus, the cost of the transformation is higher in NVMe SSD. For example, in this case of MySQL (see the

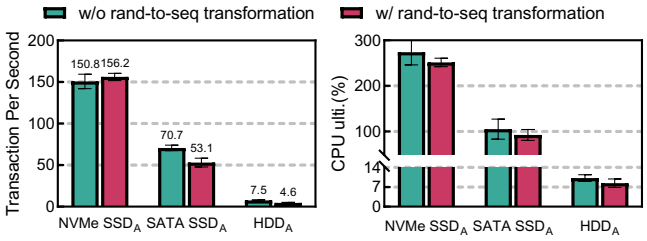

Figure 6: The performance change (left) after turning on an rand-to-seq optimization in MySQL on NVMe SSD_A and HDD_A, and the CPU cost of that optimization (right).

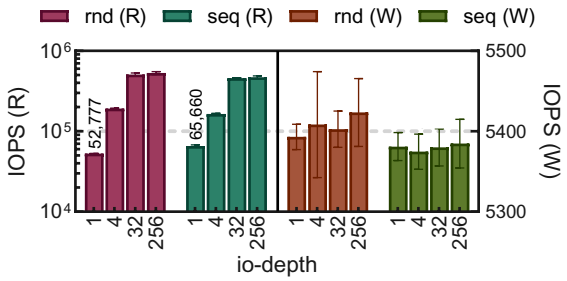

Figure 7: The speed gap between random and sequential read (left) and write (right) of NVMe SSD_A over different io-depth.

right part of Figure 6), the absolute CPU cost increases by 23.6% in NVMe SSDs, while the increase in HDD is only 1.8%. MariaDB had already change the default value of `innodb_change_buffering` since 2022-Feb. due to the reason above.

One may doubt that the high random speed is achieve when the queue depth is high, otherwise (§3.3), there may still be a gap between sequential and random I/O. Our `fio` experiments on raw device show low parallelism does not cause the speed gap. As shown in figure 7, we use `fio` parameter `numjobs=4`, `bs=16KB` and `fsync=1` to simulate MySQL's behavior (by default, MySQL has four writer threads and four reader threads; and the InnoDB page size is 16KB). The results show that there is a gap (19.6%) between random and sequential read when the level of parallelism is low, while the there is nearly no gap for write. As for other settings, we leave other DBMS configurations to default values (16KB DBMS page size, 8KB SSD page size, 4KB HDD page size) and run 16 concurrent TPC-C users to conduct the experiment.

Developers' feedback. During our discussions, even though turning off the random-to-sequential optimization for new devices is the most straightforward solution, developers also admit that the mismatch can still be a problem, as these optimizations are usually turned on by default without warning users about the impacts on new devices. They are accordingly considering related improvements in future versions.

3.5 Summary

We manually combine similar cases and report 17 issue reports in total to the DBMS developers, 15 of them are confirmed, as shown in table 4. Overall, performance mismatches in I/O size and parallelism

Table 4: New performance mismatches found in the study.

Category	DBMS	ID	# Dup.	Verify	CPD.
Size	MySQL	#102514	7	✓	◯
	PostgreSQL	#vYnS[†]	6	✓	-
	SQLite	#vYqK	1	✓	-
	MariaDB	#28909a	5	✓	-
	MongoDB	#9537	3	✓	-
	Redis	#10882	3	✓	◯
Parallelism	MySQL	#105982	5	✓	-
	PostgreSQL	#vYp3	1	✓	◯
	SQLite	#vYqg	-		-
	MariaDB	#28909b	3	✓	-
	MongoDB	#9508	2	✓	-
	Redis	#10881	2	✓	-
Sequentiality	MySQL	#103551	2	✓	-
		#107362	3		◯
		#103272	1	✓	◯
	PostgreSQL	#yVIz	1	✓	◯
	MariaDB	#26790	3	✓	◯

[†]Converted by `tinyurl.mobi`; **# Dup.**: number of duplicated mismatches exposed by different knobs; **Verify**: verified by developer; **CPD.**: *CP-Detector* [56]; ◯: mismatch can be identified the by symptom, without hints on root cause. -: mismatch missed.

exist in all six DBMSs. Less cases are exposed in sequentiality mismatch, because some DBMSs does not allow users to control related behavior (e.g., read-ahead) via configuration knobs. In fact, many DBMS can conduct random-to-sequential transformation silently. For example, MongoDB, Redis, SQLite3 and PostgreSQL uses OS's page cache, which conducts pre-fetch automatically. But changing OS's setting can be harmful to other services (rather than DBMS) running on the system.

3.6 Revising the Experiment Design

Using the rule in the study as the test oracle hurts both effectiveness and efficiency: on one hand, the rule requires a comparison of testing results between two tiers of devices (e.g., after changing a knob, the performance drops in NVMe SSD but increases in HDD); on the other hand, the rule can produce false positives and negatives (see §3.1). Therefore, based on our deeper understanding of performance mismatches obtained from our earlier tests, we identify mismatches by root cause patterns.

Specifically, we first record the key runtime information via eBPF [10] (including CPU utilization, nvme-queue utilization, and the sizes and offsets of I/O requests) from the beginning (t_{beg}) of every performance test to the end (t_{end}) of that test. To eliminate noises, only information produced by the target database server process (and processes created by the server) is recorded. The root cause patterns are as follows:

Mismatch in I/O size. The severity of I/O size mismatch can be reflected by the number of unaligned writes. Therefore, we check the number of unaligned writes before and after the knob tuning to

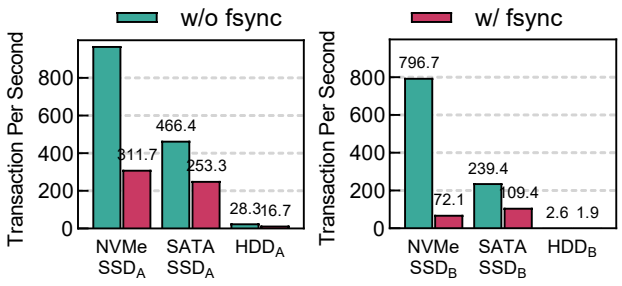

Figure 8: Performance drop of OracleDB on NVMe SSD, SATA SSD and HDD after aggressively issue `fsync`.

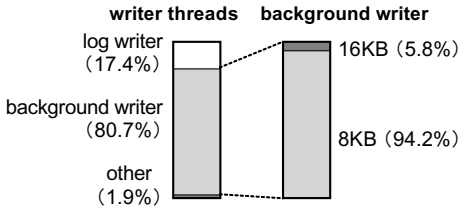

Figure 9: The distribution of the amount (left) and sizes (right) of write requests issued by OracleDB.

determine if a mismatch in I/O size exists. To achieve this, we obtain both the size of I/Os issued to the device and the SSD's preferred write size. Given that the latter one is usually not publicly available, we conduct experiments (e.g., as in Figure 4) to make an estimation. On the other hand, directly extracting the I/O sizes from syscalls issued by databases may not be appropriate, as the OS can change the sizes of I/Os that are received from databases. Thus, we observe I/O sizes just before I/Os are sent to the device (i.e., at the blk-mq layer).

Mismatch in I/O parallelism. The severity of I/O parallelism under-utilization can be reflected by multi-queue utilization. Therefore, we check the utilization rates of queues before/after tuning to determine the existence of underutilized I/O parallelism. The utilization rate [14] $util_i$ of the i^{th} queue is defined as: $util_i = \int_{t_{\text{beg}}}^{t_{\text{end}}} depth_i \, \mathrm{d}t$, where $depth_i$ means the number of I/O requests in the i^{th} queue during time period dt. we focus on two types of underutilized I/O parallelism, as we concluded in §3.3: 1) the *util* of one queue dominates (i.e., is larger than the sum of) all the other queues' *util*s, or 2) the depth of every queue is one (if not zero) during 95% of the time. Note that the underutilized I/O parallelism mostly exists in default settings, meaning that knobs play a different role here: if changing any knob cannot help DBMSs get rid of the multi-queue under-utilization, producing a mismatch alert.

Mismatch in I/O sequentiality. The severity of over-sequentialization can be reflected by the rate of seq-to-random before/after the tuning and the existence of a subsequent performance drop. Unlike the other two types of mismatches, the root causes of which can be observed at kernel level, DBMSs can conduct sequentialization on many levels (query planner level; database buffering level; OS level). One important common characteristic of the sequentialization is the increase in both CPU and sequentiality. Therefore, we take this as an indicator of the sequentialization. Note that the rate of seq-to-random is calculated on the bottom of the OS layer (i.e., blk-mq layer) to cover all levels of sequentialization.

4 SCALABILITY ANALYSIS

In previous sections, we design a testing-based method and find three types of performance mismatches exist in six open-source DBMSs. In this section, we analyze the scalability of our method from three aspects:

- Can we expose performance mismatches in commercial DBMS?
- Can existing methods expose performance mismatches?
- How accurate does our method in identifying I/O-related knobs and how is its cost-efficiency?

For implementation, the program analysis is implemented using the LLVM [18] and is conducted on top of the LLVM Intermediate Representation (IR) of the source code. The block layer tracing is implemented using the BPF Python library [3]. The queue monitoring is implemented using Linux kernel tracing events. All experiments are conducted on a machine with 24-core 3.6GHz CPU, 64GB memory, and a 5.4.0 Linux kernel.

4.1 Performance Mismatches in Commercial DBMS

We choose Oracle Database [27] (version 19c) as the target (OracleDB for short), who is the top-ranked [32, 35] relational DBMS. We use the same workloads as in the study, which are both representative and the most commonly used in existing DBMS research [46, 64, 69, 77, 84]. We use the same storage devices as in §2.1. As for configuration knobs, we choose the knobs in the core module of OracleDB; thus, we get 366 knobs in total as the input. Since OracleDB is close-source, we can not use our taint-analysis and test all the knobs. In total, we generate 14,198 tests for exposing mismatches.

As a result, we find that OracleDB has severe performance mismatches in both I/O size and parallelism. Figure 8 shows the performance change after switching the knob `COMMIT_WAIT` from `NO_WAIT` to `WAIT` (i.e., frequent `fsync`) on different storage devices. From severer to slighter, the impacts are: NVMe SSDs (3.1-11.0x), SATA SSDs (2.0-2.2x) and HDDs (1.4-1.7x). We monitor the distribution of write requests issued to the device when testing NVMe SSD$_A$. Figure 9 shows the result: the background writer is the major source of write requests, and most of the requests are 8KB (i.e., OracleDB's page size), which can be unaligned with the size of the cache line.

Figure 11 (left) shows that as the number of background writer processes increases, the TPC-C performance increases in SSDs and remain unchanged in HDD. This result seems intuitive because SSDs support paralleled I/O so can benefit from paralleled writers. To figure out the reason for the performance gain, we monitor the *total* transactions (accumulated) over time on SSDs (right two subplots), and we can observe that when there is only one writer, the transactions have to wait for a period of time periodically (see the plains of the dashed lines, indicating these waits

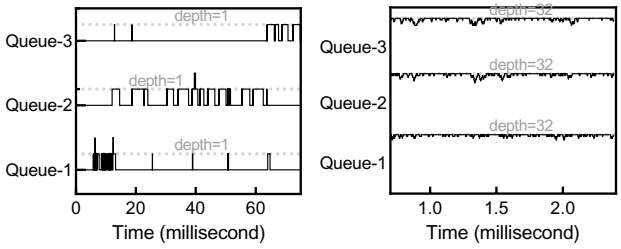

Figure 10: NVMe queue depth over time when OracleDB is running TPC-C (left) and in ideal case (right, obtained by `fio`).

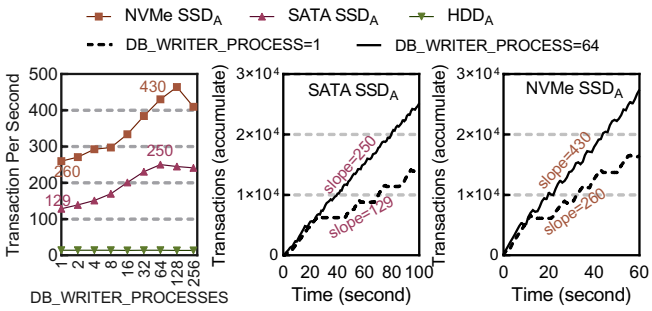

Figure 11: The performance change over different writer processes of OracleDB (left) and the cause analysis of the performance change (middle and right).

last for 5-20 seconds). This is because checkpoints are triggered, so this only writer has to process the checkpoint flush [79], causing normal transactions' waiting. Increasing the number of writers can alleviate the problem in SSD because checkpoint flush and transactions can be written in parallel, but multiple writers can not make normal transactions faster. As we can see from figure 10, even though NVMe SSD has queues with 64K max depth, the actual depth is only 1 almost all the time. Therefore, even though the performance on NVMe SSD$_A$ reaches 464 TPS, the throughput of the device is only 81MB/s, far from its specification [42]. We use `fio` (`io-depth=32`, `numjobs=4`, `bs=8KB`) to exploit the potential of NVMe SSD$_A$ (right of figure 10), and as the queues get filled properly, the throughput reaches 792MB/s. In OracleDB, the knob `PARALLEL_DEGREE_POLICY` has a similar symptom.

We do not observe performance mismatch in sequentiality. In our experiments, the only knob that affects I/O sequentiality significantly is `DB_FILE_MULTIBLOCK_READ_COUNT`. But OracleDB will automatically determine the default value of the knob according to the device, so we do not consider the case as a mismatch.

4.2 Comparison with existing methods

We compare our method with existing performance mismatches detection methods. Since no existing work focuses directly on performance mismatches, and considering that mismatches resemble performance bugs in terms of their symptoms (e.g., both cause performance drop), we choose to compare with *CP-Detector* [56], a testing framework that can expose performance bugs by configurations. It leverages the performance intuition of tuning a knob to detect if the actual performance violates the intuition. For example, if the performance drops after turning on an optimization, *CP-Detector* will report a potential bug.

As shown in table 4, *CP-Detector* can detect 7/17 performance mismatches that are detected by us. For example, in case #10082, using pre-fetching (`random_read_ahead=1`), the actual performance violates the intuition of the knob (i.e., optimization should not harm performance), so *CP-Detector* raises the alarm. For the other cases, however, the actual performance does not violate the intuition defined by *CP-Detector*. For example, in Redis#10081, increasing `io_threads` neither degrades performance (missed by *CP-Detector* because it only reports a bug if using more I/O threads degrades performance), nor alleviates the inefficient queue utilization. This result reveals the main difference between *CP-Detector* and us: we detect mismatches where DBMSs *do not fully utilize* the benefits provided by new devices, while *CP-Detector* detects performance bugs where DBMSs *mistakenly use* resources. By comparing discussions with developers on fixing between *CP-Detector* and us, we also find that developers tend to fix mismatches in future versions, while performance bugs are usually fixed in existing versions. *In conclusion, our method can effectively expose performance mismatches, and outperforms the existing methods.*

4.3 Efficiency and Accuracy

Efficiency in Reducing Testing Cost. We identify I/O-related knobs to reduce the testing time and apply root-cause-based rules to preclude the need to test multiple devices. Thus, we evaluate 1) the time saved for testing, 2) the time consumption in identifying I/O-related knobs, and 3) the overhead of dynamic monitoring, which collects information for the root cause patterns.

As shown in the right part of figure 12, filtering out the I/O-related knobs reduces the testing time by at least an order of magnitude for all six DBMSs. The cost of the filtering process ranges from 0.6-5.5 hours for the six DBMSs, as shown in the left part of Figure 12. This time is restricted by the scale of the code, the number of knobs, and the performance of the LLVM library; furthermore, the overhead is far less than the testing time saved, indicating our method's efficiency. The current bottleneck is that each test must be repeated many times to eliminate unstable results [75]. Notably, some approaches have been proposed [58, 87] to alleviate this problem, which can further help to reduce the cost. The overhead of dynamic monitoring is low. For all the six DBMSs, the overhead of dynamic monitoring is no larger than 5% and mostly negligible; *In conclusion, we can largely reduce the testing cost without sacrificing effectiveness.*

Accuracy on Identifying I/O-related Knobs. We use taint analysis to identify I/O-related knobs to reduce the cost. However, taint analysis can be inaccurate. Therefore, we first evaluate the accuracy on the task of identifying I/O-related knobs, then evaluate the impacts of the inaccuracies on the evaluated DBMSs. To obtain the ground truth of I/O-related knobs, we label them manually (I/O-related or not). Each knob is separately labeled by at least three authors who have more than three years of experience in configuration research. These authors refer to the official documents

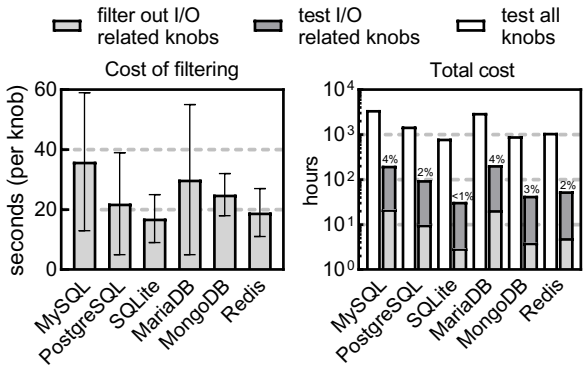

Figure 12: The cost efficiency, demonstrated via the distribution of time spent on identifying I/O-related knobs (left); the total testing time with/without identifying those knobs (right). The percentage label stands for the overhead of dynamic monitoring.

and source code to obtain the labels and discuss divergences until agreements are reached.

As shown in Table 5, our method identifies 165 out of 1,543 I/O-related knobs. The precision of I/O-related knob identification reaches 65.3%-76.4%, and the recalls reach 83.4%-100%. Note that we honors recall more than precision, because *false positives* only incur more testing costs, but *false negatives* may cause knobs that to be missed that might expose a performance mismatch. The precision for MongoDB is the lowest among the six DBMS because there are eight I/O-unrelated knobs that share the same program variable with an I/O-related knob (i.e., the different bits of the variable represent different knobs), meaning that they are falsely connected with I/O-related syscalls. Moreover, the static taint analysis could be inaccurate, which may cause some I/O-unrelated caller functions to become connected with the knob variable. For example, the database error log functions typically incur only a few I/Os, so the knobs controlling the error log are not I/O-related, however, they still need to call the write syscalls and are thus falsely identified by us. The reason for the missed I/O-related knobs related to complicated pointers that cannot be handled by our method.

Regarding the impact of these inaccuracies, nine missed I/O-related knobs can expose performance mismatches, but expose the same mismatches shown in Table 4. For example, `buffer_alignment` exposes the same mismatch as MongoDB#9537, and `mmap_all` exposes the same mismatch as MongoDB#9508. Thus, the impact caused by false negatives of I/O-related knobs may be masked, because multiple knobs may trigger the same performance mismatch (see column "**# Dup.**" in Table 4). On the other hand, false positives on the I/O-related knob identification will lead to higher testing costs. The false positive rate is 23.6%-34.7%, which indicates the amount of extra testing cost can be incurred. *In conclusion, false positives when identifying the I/O-related knobs cause some extra testing costs, while the negative impact of false negatives may be mitigated because multiple knobs can trigger the same mismatch.*

Table 5: Precision and recall in identifying I/O-related knobs.

DBMS	# Knobs	# I/O knobs	Precision	Recall
MySQL	529	43	0.656	0.871
PostgreSQL	256	30	0.749	0.834
SQLite	74	3	0.658	1.0
MariaDB	418	42	0.764	0.862
MongoDB	169	28	0.653	0.901
Redis	97	19	0.717	0.919
Total	1,543	165	0.700	0.898

 # Knobs: number of knobs in the core module of the DBMS.

5 POSSIBLE SOLUTIONS

For researchers. Unaligned writes to NVMe SSDs cause unnecessary waiting (§3.2), and under-paralleled I/O requests cause the I/Os to be served in a synchronous manner inside SSDs (§3.3). Thus, researchers can consider adding an additional device-sensitive layer [61], which works like a network switch that makes I/O requests more adaptive before sending the requests to underlying devices. The incoming and outgoing ports of this switch layer are the per-core request queues (in block layer) and nvme-queues respectively. In this way, the switch layer can dispatch the I/O requests to the idle queues to avoid under-paralleled I/O, as well as re-organize (e.g., merge) write requests when possible to avoid small writes to the NVMe SSDs.

For developers. *First, DBMSs should drop their legacy I/O interfaces.* Most DBMSs use synchronized I/O syscalls (MySQL uses `libaio`, but in an inefficient way [17, 24]), and rely heavily on OS page cache, the scalability of which is very limited on paralleled I/O [2]. Alternatively, since version 5.1, Linux kernel has proposed `io_uring` [12, 16], a new I/O interface that is particularly designed for fast devices. Some DBMSs [20, 30] are already developing upon `io_uring`, and the others [21, 24, 33] are planning so.

Second, DBMSs should try to make the optimizations configurable. For example, SQLite developers uphold the design principle that writing a series of sequential pages of data to a file is faster than doing it randomly [37, 54], although they do not create any knob to control such behavior. PostgreSQL performs pre-fetching *silently* using `fadvise` without making it configurable [2]. Adding knobs to these optimizations can help developers expose more potential performance mismatches and further improve the DBMSs. Moreover, DBMSs can add knobs to create workarounds for potential mismatches that are left out to production environment. For example, adding knobs to provide users with alternative choices to store different DBMS files, including data, index, logs, etc., into different directories (i.e., devices).

6 DISCUSSION

Limitations. Our method has several limitations that we plan to address in future work. First, the studied devices are consumer-level SSDs, meaning that our findings may not scale to enterprise-level SSDs. Notably, enterprise-level SSDs surpass consumer-level devices in every aspects, suggesting that some mismatches (e.g.,

unparalleled I/O, random-to-sequential-transformation) may become more severe. In the future, we will extend our study on performance mismatches to a wider range of storage devices, including enterprise-level SSDs, RAID, and cloud storage. Second, our method cannot expose mismatches that are not triggered by configurations. Even in highly configurable DBMSs, many I/O-related optimizations and mechanisms (e.g., using B-tree or LSM-tree) are not configurable. Our future work will explore alternative techniques to expose mismatches that cannot be triggered by configurations.

Fixing complexity of mismatches. We submitted 17 reports to developers, they admitted that fixing those mismatches could be challenging. For example, convincing developers to use `io_uring` as the I/O interface is easier than implementing it. Changing the I/O interface brings new challenges including but not limited to: how to ensure reliability (e.g., ACID [1]) with the new interface; some DBMSs may have their own layer of I/O engine, such that changing to another interface may break the existing design [40]. On the other hand, some DBMSs may intentionally make the DBMS simple, meaning that their developers tend to maintain an attitude of neutrality to performance mismatches [36].

7 RELATED WORK

Configuration tuning. Many works [88, 95, 97] optimize performance through configuration tuning. Note that our work is different from these works in three aspects: 1) different *audiences*. Tuning helps *end-users* to find configurations that produce good performance in production, while we help *developers* expose performance mismatches via configurations in-house; 2) different *scenarios*. Tuning aims to find the right values for a selective set of important knobs to adapt to *different workloads* in a *fixed* hardware setup, while we aim to expose performance mismatches when *hardware settings change*; 3) different *aims*. Tuning only focuses on producing better performance, so it treats knobs as a black-box without explaining what happens, while we aim at exposing mismatches that are triggered by knobs and categorizing the potential root causes.

SSD-aware optimization. Many works have studied the I/O properties and implemented possible optimizations of these properties. A recent study [57] concludes SSD-specific rules from file system level to application level. Our method has two differences: first, while they use a SATA SSD simulator, we study real SATA SSDs and NVMe SSDs, so mismatches that only manifest in NVMe SSDs (§3.2) can not be exposed by their work; second, their rules are drawn from DBMSs that use default configurations, while mismatches can also be triggered by non-default configurations (§3.2, §3.4); moreover, another work [91] studies the positive implications of NVMe SSD on DBMSs, while we aim at finding the performance mismatches that cause negative impacts on DBMSs. Some works propose completely new designs of DBMS components or data structures (e.g., B-Trees [72, 83], LSM [44, 53, 69, 82, 86], query optimizer [52]) targeting specific SSD properties (e.g., low latency [69, 80], parallelism [46, 52, 59], write amplification [49], property informed by learning internal parameters [64]). While our work has a different target: we focus on *exposing* the mismatches of the existing widely used DBMSs and help developers to reason about performance mismatches.

Detecting performance bugs. Many works have been proposed to detect a wide range of performance problems. One group of works focuses on loop-related or synchronization-related performance bugs, they detect specific patterns of inefficient loops [51, 78], redundant loads [85], or inefficient synchronizations [81, 93]. These methods focus on application-level performance problems alone, but do not consider the potential mismatch between application and underlying devices. The other group of works focuses on detecting configuration-related performance problems. *X-ray* [43] diagnoses the most suspicious configuration knob given an existing performance problem, while *LearnConf* [71] and *Voilet* [60] detect configuration knobs or their combinations that may lead to time-consuming operations. These approaches can also select I/O-related knobs as we do, but they are either Java-exclusive [60] or reliant on heavy symbolic execution [60]. *CP-Detector* [56] exposes performance bugs using the intuition of tuning configurations. As the evaluation shows, *CP-Detector* exposes performance bugs where changing the bug-triggering knob makes the performance drop, while we identify mismatches that cause specific inefficient patterns and may not cause the performance drop.

8 CONCLUSION

In this paper, we find that shoehorning new storage devices into the existing DBMS may cause performance mismatches which can have a severe impact on performance. Performance mismatches are rarely studied and/or detected. To fill this gap, we conduct a comprehensive study of performance mismatches to understand their symptoms, root causes, and triggering conditions. In the study, we propose a method that leverages configurations to detect performance mismatches. We find that performance mismatches can be divided into three types based on their root cause, and we carry out an in-depth analysis of the root cause patterns. Compared with baseline methods, our method is more efficient and can detect more performance mismatches.

ACKNOWLEDGEMENTS

This work was funded by National Natural Science Foundation of China (NSFC) No.62272473 and No.62202474. We thank Yu Jiang, Yanyan Jiang and Tingting Yu for their helpful advice.

REFERENCES

[1] ACID (atomicity, consistency, isolation, durability). https://en.wikipedia.org/wiki/ACID. [Accessed Jan. 2023].
[2] Asynchronous IO for PostgreSQL. anarazel.de/talks/2020-01-31-fosdem-aio/aio.pdf. [Accessed Jan. 2023].
[3] bcc Python Developer. https://github.com/iovisor/bcc/blob/master/docs/tutorial_bcc_python_developer.md. [Accessed Jan. 2023].
[4] blktrace. https://linux.die.net/man/8/blktrace/. [Accessed Jan. 2023].
[5] Changing page size in MySQL. https://dev.mysql.com/doc/refman/8.0/en/innodb-parameters.html#sysvar_innodb_page_size. [Accessed Jan. 2023].
[6] Changing page size in SQLite. https://sqlite.org/pragma.html#pragma_page_size. [Accessed Jan. 2023].
[7] Cloud NVMes: the blind side of them. https://ragainis.lt/cloud-nvmes-the-blind-side-of-them-da927d09b378. [Accessed Jan. 2023].
[8] Control variable. https://en.wikipedia.org/wiki/Control_variable. [Accessed Jan. 2023].
[9] Dominator. https://en.wikipedia.org/wiki/Dominator_(graph_theory). [Accessed Jan. 2023].
[10] eBPF: extended berkeley packet filter. https://ebpf.io/. [Accessed Jan. 2023].
[11] Event Tracing. https://www.kernel.org/doc/html/v4.19/trace/events.html. [Accessed Jan. 2023].

[12] Glauber Costa. How io_uring Will Revolutionize Programming in Linux. https://thenewstack.io/how-io_uring-and-ebpf-will-revolutionize-programming-in-linux/. [Accessed Jan. 2023].

[13] Intel. Intel® SSD 760p Series. https://www.intel.com/content/www/us/en/products/sku/134583/intel-ssd-760p-series-256gb-m-2-80mm-pcie-3-1-x4-3d2-tlc/specifications.html. [Accessed Jan. 2023].

[14] iostat: Report Central Processing Unit (CPU) statistics and input/output statistics for devices and partitions. https://github.com/sysstat/sysstat/blob/master/man/iostat.in. [Accessed Jan. 2023].

[15] Jens Axboe. Flexible I/O Tester. https://fio.readthedocs.io/en/latest/index.html. [Accessed Jan. 2023].

[16] Jonathan Corbet. Ringing in a new asynchronous I/O API. https://lwn.net/Articles/776703/. [Accessed Jan. 2023].

[17] Linux-native asynchronous I/O library. http://lse.sourceforge.net/io/aio.html. [Accessed Jan. 2023].

[18] LLVM Programmer's Manual. https://llvm.org/docs/ProgrammersManual.html. [Accessed Jan. 2023].

[19] LLVM use-def-chains. https://llvm.org/docs/ProgrammersManual.html#iterating-over-def-use-use-def-chains. [Accessed Jan. 2023].

[20] MariaDB. https://mariadb.org/. [Accessed Jan. 2023].

[21] MongoDB. https://www.mongodb.com/. [Accessed Jan. 2023].

[22] MySQL 8.0 is 36 times slower than MySQL 5.7 in Samsung 970 Pro NVMe. https://bugs.mysql.com/bug.php?id=93734/. [Accessed Jan. 2023].

[23] MySQL. Configuring the Number of Background InnoDB I/O Threads. https://dev.mysql.com/doc/refman/8.0/en/innodb-performance-multiple_io_threads.html. [Accessed Jan. 2023].

[24] MySQL. Using Asynchronous I/O on linux. https://dev.mysql.com/doc/refman/8.0/en/innodb-linux-native-aio.html. [Accessed Jan. 2023].

[25] MySQL with NVMe SSD is slower than expected. https://gcore.de/en/help/linux/mysql-nvme-slow.php.

[26] NVM Express Explained. https://nvmexpress.org/wp-content/uploads/2013/04/NVM_whitepaper.pdf. [Accessed Jan. 2023].

[27] Oracle Database. https://www.oracle.com/database/. [Accessed Jan. 2023].

[28] Poor IO performance - NVMe Samsung 950 Pro. https://askubuntu.com/questions/698395/. [Accessed Jan. 2023].

[29] Postgres slower on NVMe then on Sata SSD. https://www.reddit.com/r/PostgreSQL/comments/5nrha9/postgres_slower_on_nvme_then_on_sata_ssd/. [Accessed Jan. 2023].

[30] PostgreSQL. Asynchronous I/O for PostgreSQL (Working in Progress). https://github.com/anarazel/postgres/tree/aio. [Accessed Jan. 2023].

[31] PostgreSQLCO.NF Parameter Documentation. https://postgresqlco.nf/doc/en/param/. [Accessed Jan. 2023].

[32] Red-9. Database Engine Popularity Rankings. https://red9.com/database-popularity-ranking. [Accessed Jan. 2023].

[33] Redis. https://redis.io/. [Accessed Jan. 2023].

[34] Samsung. Samsung 980 Pro NVMe SSD. https://www.samsung.com/semiconductor/minisite/ssd/product/consumer/980pro/. [Accessed Jan. 2023].

[35] SOLID IT. DB-Engines Ranking. https://db-engines.com/en/ranking. [Accessed Jan. 2023].

[36] SQLite Is Serverless. https://sqlite.org/serverless.html. [Accessed Jan. 2023].

[37] SQLite. Performance Related Assumptions. https://www.sqlite.org/fileio.html#fs_performance. [Accessed Jan. 2023].

[38] TaintChecker: a clang static checker that carries out tainting analysis. https://github.com/franchiotta/taintchecker/. [Accessed Jan. 2023].

[39] Taintgrind: a valgrind taint analysis tool. https://github.com/wmkhoo/taintgrind. [Accessed Jan. 2023].

[40] The SQLite OS Interface or "VFS". https://www.sqlite.org/vfs.html.

[41] Transitive relation. https://en.wikipedia.org/wiki/Transitive_relation. [Accessed Jan. 2023].

[42] Western Digital. WD_BLACK SN850 NVMe SSD. https://www.westerndigital.com/products/internal-drives/wd-black-sn850-nvme-ssd/. [Accessed Jan. 2023].

[43] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2012.

[44] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. TRIAD: creating synergies between memory, disk and log in log structured key-value stores. In *2017 USENIX Annual Technical Conference (ATC)*, 2017.

[45] Supratik Chakraborty, Daniel J Fremont, Kuldeep S Meel, Sanjit A Seshia, and Moshe Y Vardi. Distribution-aware sampling and weighted model counting for sat. In *Proceedings of the Association for the Advancement of ArtiĄcial Intelligence (AAAI)*, 2014.

[46] Feng Chen, Binbing Hou, and Rubao Lee. Internal parallelism of flash memory-based solid-state drives. *ACM Trans. Storage*, 12(3), 2016.

[47] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2009.

[48] Qingrong Chen, Teng Wang, Owolabi Legunsen, Shanshan Li, and Tianyin Xu. Understanding and discovering software configuration dependencies in cloud and datacenter systems. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 362–374, 2020.

[49] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard P. Spillane, Amy Tai, and Rob Johnson. Splinterdb: Closing the bandwidth gap for nvme key-value stores. In *2020 USENIX Annual Technical Conference (ATC)*, 2020.

[50] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. A simple, fast dominance algorithm. *Software Practice & Experience*, 4, 2001.

[51] Monika Dhok and Murali Krishna Ramanathan. Directed test generation to detect loop inefficiencies. In *European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2016.

[52] Pedram Ghodsnia, Ivan T. Bowman, and Anisoara Nica. Parallel I/O aware query optimization. In *International Conference on Management of Data (SIGMOD)*, 2014.

[53] X. Guo, X. Liu, E. Zhu, X. Zhu, M. Li, X. Xu, and J. Yin. Adaptive self-paced deep clustering with data augmentation. *IEEE Transactions on Knowledge and Data Engineering*, 2019.

[54] Sibsankar Haldar. *SQLite Database System Design and Implementation*. Sibsankar Haldar, 2016. [Accessed Jan. 2023].

[55] Xue Han, Tingting Yu, and David Lo. Perflearner: Learning from bug reports to understand and generate performance test frames. In *International Conference on Automated Software Engineering (ASE)*, 2018.

[56] Haochen He, Zhouyang Jia, Shanshan Li, Erci Xu, Tingting Yu, Yue Yu, Ji Wang, and Xiangke Liao. Cp-detector: Using configuration-related performance properties to expose performance bugs. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020.

[57] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The unwritten contract of solid state drives. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*, 2017.

[58] Sen He, Glenna Manns, John Saunders, Wei Wang, Lori Pollock, and Mary Lou Soffa. A statistics-based performance testing methodology for cloud applications. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*, 2019.

[59] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Chao Ren. Exploring and exploiting the multilevel parallelism inside ssds for improved performance and endurance. *IEEE Trans. Computers*, 62(6), 2013.

[60] Yigong Hu, Gongqi Huang, and Peng Huang. Automated reasoning and detection of specious configuration in large systems with symbolic execution. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2020.

[61] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. Rearchitecting linux storage stack for μs latency and high throughput. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.

[62] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[63] M. F. Johansen, Ø. Haugen, and F. Fleurey. An algorithm for generating t-wise covering arrays from large feature models. In *Proceedings of the International Software Product Line Conference (SPLC)*, 2012.

[64] Aarati Kakaraparthy, Jignesh M. Patel, Kwanghyun Park, and Brian Kroth. Optimizing databases by learning hidden parameters of solid state drives. *Proc. VLDB Endow.*, 13(4), 2019.

[65] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. Distance-based sampling of software configuration spaces. In *IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019.

[66] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. Distance-based sampling of software configuration spaces. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, 2019.

[67] Ali Khakifirooz, Sriram Balasubrahmanyam, Richard Fastow, Kristopher H Gaewsky, Chang Wan Ha, Rezaul Haque, Owen W Jungroth, Steven Law, Aliasgar S Madraswala, Binh Ngo, et al. A 1Tb 4b/Cell 144-Tier Floating-Gate 3D-NAND Flash Memory with 40MB/s Program Throughput and 13.8 Gb/mm 2 Bit Density. In *Proceedings of the 68th IEEE International Solid-State Circuits Conference (ISSCC)*, 2021.

[68] Chulbum Kim, Doo-Hyun Kim, Woopyo Jeong, Hyun-Jin Kim, Il Han Park, Hyun-Wook Park, JongHoon Lee, JiYoon Park, Yang-Lo Ahn, Ji Young Lee, et al. A 512-Gb 3-b/Cell 64-stacked WL 3-D-NAND flash memory. *IEEE Journal of Solid-State Circuits*, 53(1), 2017.

[69] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[70] Chi Li, Shu Wang, Henry Hoffmann, and Shan Lu. Statically inferring performance properties of software configurations. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)*, 2020.

[71] Chi Li, Shu Wang, Henry Hoffmann, and Shan Lu. Statically inferring performance properties of software configurations. In *Fifteenth EuroSys Conference (EuroSys)*, 2020.

[72] Yinan Li, Bingsheng He, Jun Yang, Qiong Luo, and Ke Yi. Tree indexing on solid state drives. *Proc. VLDB Endow.*, 3(1), 2010.

[73] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. Write dependency disentanglement with horae. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2020.

[74] Xiaojian Liao, Youyou Lu, Zhe Yang, and Jiwu Shu. Crash consistent non-volatile memory express. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, 2021.

[75] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. Taming performance variability. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2018.

[76] D. Marijan, A. Gotlieb, S. Sen, and A. Hervieu. Practical pairwise testing for software product lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, 2013.

[77] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. SFS: random write considered harmful in solid state drives. In *Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST)*, 2012.

[78] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *International Conference on Software Engineering (ICSE)*, 2015.

[79] OREILLY. DBWR—the Database Writer. https://www.oreilly.com/library/view/oracle-database-administration/1565925165/ch10s02s01s01.html. [Accessed Jan. 2023].

[80] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In *2016 USENIX Annual Technical Conference (ATC)*, 2016.

[81] Michael Pradel, Markus Huggler, and Thomas R. Gross. Performance regression testing of concurrent classes. In *International Symposium on Software Testing & Analysis (ISSTA)*, 2014.

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

[83] Hongchan Roh, Sanghyun Park, Sungho Kim, Mincheol Shin, and Sang-Won Lee. B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives. *Proc. VLDB Endow.*, 5(4), 2011.

[84] Jason Sewall, Jatin Chhugani, Changkyu Kim, Nadathur Satish, and Pradeep Dubey. PALM: parallel architecture-friendly latch-free modifications to B+ trees on many-core processors. *Proc. VLDB Endow.*, 4(11), 2011.

[85] Pengfei Su, Shasha Wen, Hailong Yang, Milind Chabbi, and Xu Liu. Redundant loads: A software inefficiency indicator. In *International Conference on Software Engineering (ICSE)*, 2019.

[86] TunLi, WanweiLiu, JuanChen, XiaoguangMao, and XinjunMao. Towards connecting discrete mathematics and software engineering. *TSINGHUA SCIENCE AND TECHNOLOGY*, 25(3), 2020.

[87] Alexandru Uta, Alexandru Custura, Dmitry Duplyakin, Ivo Jimenez, Jan Rellermeyer, Carlos Maltzahn, Robert Ricci, and Alexandru Iosup. Is big data performance reproducible in modern cloud networks? In *17th USENIX symposium on networked systems design and implementation (NSDI)*, 2020.

[88] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM international conference on management of data (SIGMOD)*, 2017.

[89] Teng Wang, Zhouyang Jia, Shanshan Li, Si Zheng, Yue Yu, Erci Xu, Shaoliang Peng, and Xiangke Liao. Understanding and detecting on-the-fly configuration bugs. In *Proceedings of the 45th International Conference on Software Engineering (ICSE)*, 2023.

[90] Youjip Won, Jaemin Jung, Gyeongyeol Choi, Joontaek Oh, Seongbae Son, Joo Young Hwang, and Sangyeun Cho. Barrier-enabled io stack for flash storage. In *USENIX Conference on File and Storage Technologies (FAST)*, 2018.

[91] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. Performance analysis of nvme ssds and their implication on real world databases. In *Proceedings of the 8th ACM International Systems and Storage Conference (SYSTOR)*, 2015.

[92] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. Do not blame users for misconfigurations. In *ACM SIGOPS 24th Symposium on Operating Systems Principles (SOSP)*, 2013.

[93] Tingting Yu and Michael Pradel. Syncprof: Detecting, localizing, and optimizing synchronization bottlenecks. In *International Symposium on Software Testing & Analysis (ISSTA)*, 2016.

[94] Bohan Zhang, Dana Van Aken, Justin Wang, Tao Dai, Shuli Jiang, Jacky Lao, Siyuan Sheng, Andrew Pavlo, and Geoffrey J. Gordon. A demonstration of the ottertune automatic database management system tuning service. *Proc. VLDB Endow.*, 11(12), 2018.

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

[96] Shulin Zhou, Xiaodong Liu, Shanshan Li, Wei Dong, Xiangke Liao, and Yun Xiong. Confmapper: Automated variable finding for configuration items in source code. In *2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 2016.

[97] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC)*, 2017.