Debugging Performance Issues in Mobile Data Management

Carl Nuessle
University at Buffalo
carlnues@buffalo.edu

Oliver Kennedy
University at Buffalo
okennedy@buffalo.edu

Lukasz Ziarek
University at Buffalo
lziarek@buffalo.edu

ABSTRACT

Embedded databases like SQLite are used extensively by smartphone apps for storing persistent state. However, if misused, embedded databases can represent significant performance bottlenecks for app developers. These can result from simple database misconfigurations like missing indexes or poor choice of concurrency mode. However, mobile platforms like Android introduce a host of new challenges for database performance tuning. In this paper, we explore in depth one of these challenges: CPU frequency scaling, fine-grained adjustments to CPU performance that reduce power consumption. Android adjusts frequencies according to a heuristic policy called the governor. In this paper, we explore the relationship between governor and embedded database performance across a range of workloads. Specifically, we show that certain governors, including Android defaults, can significantly penalize some types of workloads. In one case, the stated governor name and policy proves to be misleading of actual performance. We also identify a common use case where the new default governor often exhibits worse performance than the previous system default. We outline the findings of our study, and present a clear decision process for debugging governor-based performance issues in mobile databases.

KEYWORDS

Mobile Platforms, Android, Database Performance

ACM Reference Format:
Carl Nuessle, Oliver Kennedy, and Lukasz Ziarek. 2018. Debugging Performance Issues in Mobile Data Management. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Databases are used extensively in smartphone applications (apps) to store persistent state, with typical smartphones averaging over two queries per second [14]. Despite their extensive use, databases can be performance bottlenecks, leading to apps with poor responsiveness [20]. In some cases, performance issues result from simple database configuration issues (e.g., not enabling SQLite's WAL mode [18]). However, performance issues also result from the platform itself. Smartphone hardware and operating systems rely heavily on heuristic optimizations for managing scarce resources like power or bandwidth. Poor choices from these heuristics can lead to significant performance penalties for apps.

To help app developers better understand how these issues arise, we introduce MDPerf, a performance debugger specifically targeting mobile data management. While MDPerf explores database configuration parameters, it also considers the interplay between an app's data management needs and the heuristics governing the mobile platform on which the app runs. In this paper we focus on one specific heuristic unique to mobile platforms: the governor. A governor is the part of the operating system that manages CPU usage, deciding which of the competing processes will be run, how soon, for how long, and most importantly how fast. Specifically, the governor has fine-grained control over the CPU's frequency, selecting from tens of frequencies to trade off between performance and power consumption. The Android system has a modular governor — The Nexus 5 and 6, for example have a choice of six available governors, each representing a different policy [1].

This paper aims to help developers to debug governor-based database performance issues. Specifically, we make the following contributions: (1) We study database performance on a range of governors and workloads representative of real-world app usage. (2) We explore unexpected results from this study. (3) We give a decision process for debugging governor-related data-performance in smartphone apps.

2 STUDY DESIGN

We now discuss our governor measurement study. All tests were run on a stock version of Android AOSP 5.1.1 system [8], instrumented as described below, and with the governor adjusted as indicated in each experiment. Our benchmark workloads are derived from the result of a previous study [14] and are designed to mimic workloads encountered on mobile devices as discussed below.

2.1 Benchmarking Environment

All tests were run on Nexus 5 devices with 2GB RAM and a quad-core 2.3 GHz CPU (quality bin 2 [19]). Software and libraries used in tests included SQLite 3.8.6.1 and the Android Youtube app version 13.10.59 [9]. The Youtube app is the second-most installed app on the Android platform at a 71% share.[12]

Benchmark workloads were run in a self-contained Android "app" that runs in two phases. The first time the app is run, it preloads an initial database and then exits. On the second run, the app replays one of several pre-generated workloads — the choice to replay rather than to randomly reproduce was made to ensure repeatability across different test settings.

All trace data was collected using the Linux ftrace framework. Android was instrumented at the ftrace framework level to record I/O operations and context switches to and from the measurement app. The benchmark app injected additional ftrace events at the start and stop of each delay period to record per-query latency. From these events we calculate two metrics: latency, or total time time spent responding to a DB request, and CPU time, or time the app spends on-core processing the request. We have confirmed that the overhead of logging within the Android kernel is negligible, while app-level logging added a nearly fixed 1s overhead to the CPU time of trial, roughly 0.6ms per query.

Conference'17, July 2017, Washington, DC, USA
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/0000001.0000001

Overview. Our study is based on of 64 trials, spanning 8 workloads, 5 governor choices, and 2 load levels, as described below.

2.2 Workloads

The benchmark itself consisted of 8 workloads (A-*H*). In a previous study [14], we found overwhelming evidence that smartphone apps use SQLite like a key-value store. Hence, the first 6 workloads (A-F) are the six canonical YCSB [5] workloads: a mix of read, write, update, append, and scan operations implemented over a key-value store. We supplement these workloads with 2 additional workloads: *G* and *H*. Both use the same data set, and consist of a distribution of database insert, upsert, and select operations proportional to typical app behavior seen in our prior study. Select queries in the two workloads involve, respectively, 1-dimensional (*G*) or 2-dimensional (*H*) range scans.

A second insight from our previous study was that database queries on smartphones, unlike those on traditional servers, are intermittent in nature. Specifically, inter query delays follow a long-tail distribution that we model as a lognormal distribution with a mean of 6.67 ms. Accordingly, our benchmark app pauses by sleeping in between operations for the appropriate amount of time.

2.3 Governors

We evaluated all 6 governor choices [1] available on the Nexus 5: (1) **Conservative**, where frequency scaling happens slowly, (2) **Interactive**, where frequency scales up rapidly and down using a timer delay, (3) **Ondemand**, which triggers frequency scaling based on per-task work queues, (4) **Performance**, where the CPU runs at the highest frequency modulo thermal limiting, (5) **Powersave**, designed to run the CPU at the lowest frequency available, and (6) **Userspace**, an option allowing apps to directly set frequencies

Frequency gradations on the Nexus 5 phone consist of 14 discrete steps between 300 MHz and 2.265 GHz, inclusive. All governors were run with otherwise default intra-governor settings. *Ondemand is the default governor for the Nexus 5* device that we used, while *Interactive is the default for the Nexus 6* [1]. The Userspace governor defaults to the maximum frequency and performs identically to the Performance governor under default conditions. We therefore omit our results for the Userspace governor.

2.4 Non-Database Load

In contrast to their server-class counterparts, mobile databases typically share system resources with other applications. An application executing queries on SQLite may be doing other things, or the user may be using multiple applications simultaneously. Since governor behavior is based on active system load, we wish to understand the effects of load on database performance.

Specifically, we run our benchmark under two load conditions. First, for each governor we ran the benchmark alone, where it essentially had system resources to itself. Second, for a subset of the governors, we ran it simultaneously with a well-known popular app, Youtube, where it had to compete for resources. We selected this app because it is a popular and commonly used Android app, and because video playback and network streaming provides a representative way to stress system resources.

The Performance and Powersave choices are upper and lower bounds for available performance, given that the CPU is essentially pinned to the highest or lowest available frequency. The three intermediate governors — Conservative, Interactive, and Ondemand — represent the interesting middle ground where CPU frequency fluctuates based on system usage, so we focus our evaluation under load on these three governors.

2.5 Implementation Challenges

Our benchmarking system encountered some challenges. We ran Youtube to produce a resource-limited environment in which our benchmark could run. A deliberate policy of the Android system, however, is periodically to kill background tasks that it determines are consuming resources, such as our benchmark, in order to maximize user experience. Getting complete runs thus often involved repeated trials. While we could have minimized this behavior by running the benchmark as a system service, this would not have mimicked the environment enjoyed by a typical app.

Additionally, when running the benchmarks tests, it was a matter of time until the test devices themselves started exhibiting thermal stress, with display flicker, touchscreen unresponsiveness, and random app launches. Thermal problems are a known issue with Android devices, particularly during periods of long screen usage and during battery charging [10, 17]. Both of these factors were present when running our app; the solution was inevitably a prolonged cooldown period. We observed identical patterns of both the benchmark killing and thermal limit problems on different but identically configured devices, to rule out a possible hardware glitch.

3 STUDY ANALYSIS

Before we discuss results specific to governors, we outline the general performance characteristics of queries. First, reads are universally fast, with typical latencies of around 1ms. Update performance is bimodal, typically taking either 1ms or 10ms. Inserts are universally expensive, taking around 10ms. Unlike the formulaic performance of the other operations, scans are highly variable, but typically fall in the range of 1ms to 10ms latency.

General Governor Behavior. To get a better sense of the governors and their effect on query processing times, consider the scatter plots presented in Figure 1 and Figure 2. These two figures show the predominantly IO-bound workload F (Figure 1), and the predominantly CPU-bound workload *G* (Figure 2)[1]. On both workloads, the Powersave/Performance governors are stable, while the Conservative governor adapts frequencies slowly throughout the workload, with 3 discrete steps seen in Workload F and 1 step in Workload *G*. By comparison, the Interactive governor adapts frequently during the workload leading to highly variable performance, while the Ondemand governor fluctuates more predictably, ramping up periodically when its utilization threshold is crossed before slowly ramping back down.

Performance Under Load. When additional load is added to the system, as shown in the 3 graphs in the lower left of both figures, we see a marketed shift in performance for the governors. The most

[1]Similar graphs for the remaining workloads, excluded in the interest of space, may be found at http://pocketdata.info

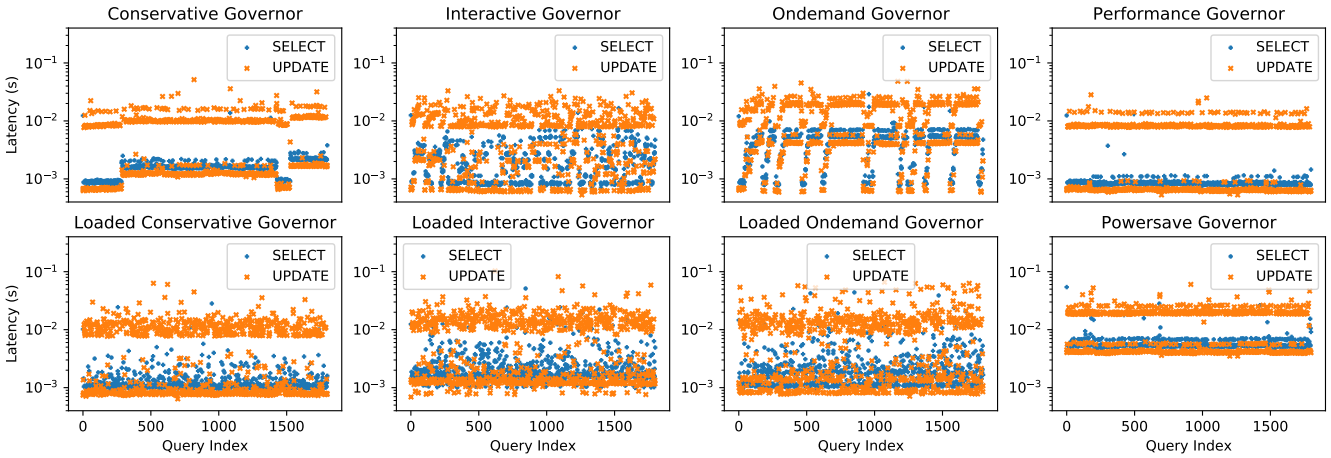


Figure 1: Per-Query Latencies on Workload F

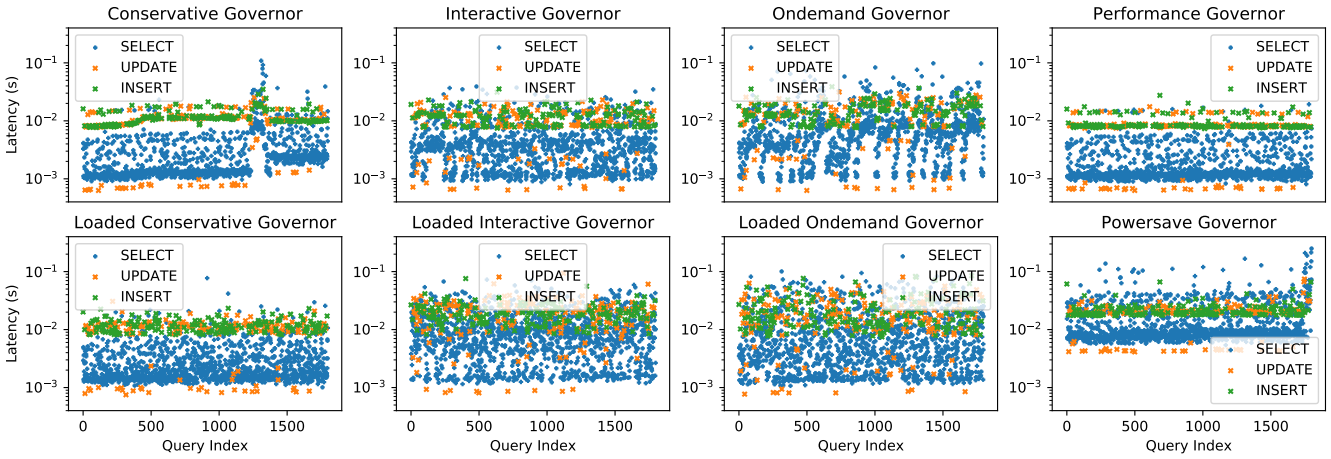


Figure 2: Per-Query Latencies on Workload *G*

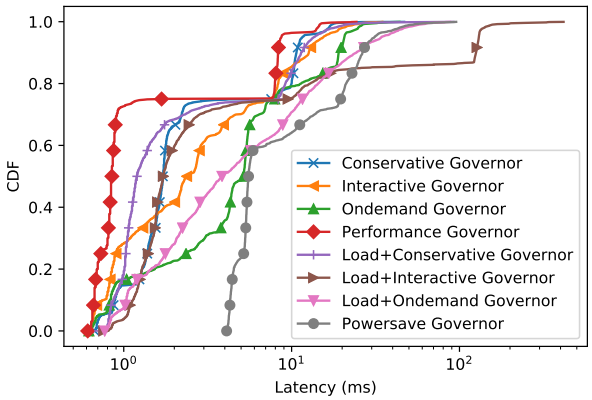
important thing to notice is that the change in the governor's behavior and the system as a whole, depends on whether the workload is CPU-bound or not. The added load, in general, keeps the Interactive and Ondemand governor policies tuned toward performance, keeping the CPU running at higher frequencies. The counter intuitive result is that if the workload is not CPU bound (as in Workload F), the added load actually improves median performance, albeit at the cost of increased variability. In the reverse case where the workload is CPU bound (Workload *G*), latency times become longer and more variable.

Performance by Workload Bottleneck. In general, the most CPU bound workloads tended to be scan heavy (Workloads E, *G*, *H*). The read-heavy workloads (B, C, D) are dominated by operations with negligible cost, and spend most of their time not in database code (sleeping or displaying video in our tests). Update-heavy workloads (A, F) tended to be IO-bound, leaving the CPU underutilized. The variation in performance between best-case (Performance governor) and worst-case (Powersave governor) is roughly one order of magnitude, increasing from 1ms to 10ms or from 10ms to 100ms.

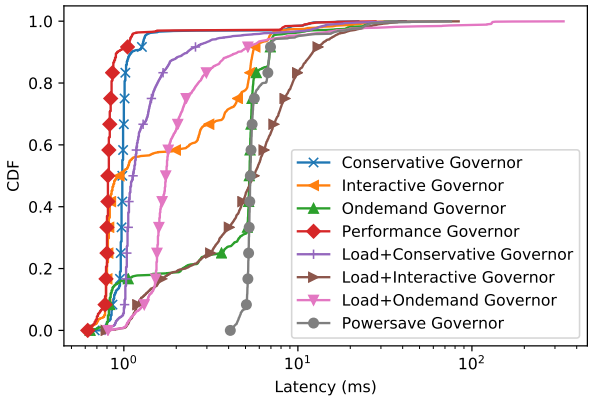
This effect is most pronounced for the faster operations like selects. Slower operations, which tend to be IO-bound, are still affected, but not to the same degree (e.g., compare Performance and Powersave governor for workload *G* in Figure 2). In general, adding system load (Youtube) causes CPU time and performance variability to move in opposite directions. CPU time gets better (core kept warm), but latency variation is significantly worse.

Ramp-Down vs Contention. The Ondemand and Interactive governors adjust performance continuously. Hence, it is possible for added load to actually decrease latencies. Concretely, higher loads have two effects: (1) The governors are more likely to keep the CPU running at higher frequencies, and (2) more contention from scarce resources makes performance slower and more variable. On the CPU-bound workloads, contention is the dominant factor, and performance suffers under load. Figure 3 illustrates the performance breakdown for YCSB's three read/write workloads. On the IO-heavy Workload A (Figure 3a), median performance for both Interactive (◀, ▶) and Ondemand (▲, ▼) governors improves under load, as contention has a comparatively small effect. However, at roughly the 85th percentile, performance degrades significantly. These operations are exclusively updates, and their reduced performance is most likely caused by increased bus or flash drive contention.

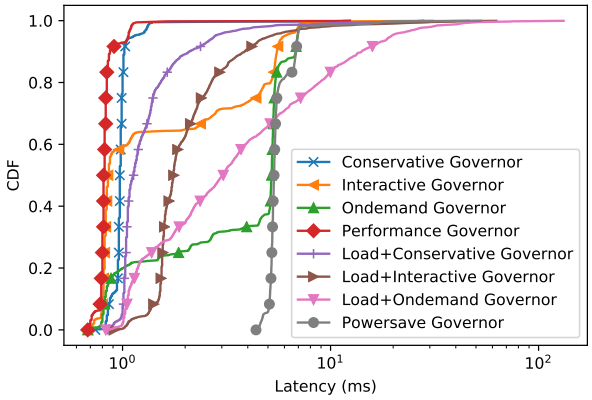
Android Defaults are Highly Variable. Note also the comparative behavior of the Ondemand and Interactive governors (default governors for the Nexus 5 and 6, respectively) on Workloads B



(a) Workload A (Write-Heavy — 50/50 read/write)



(b) Workload B (Read-Heavy — 95/5 read/write)

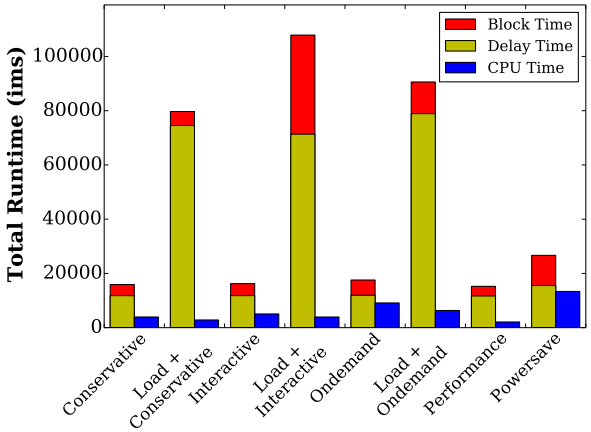


(c) Workload C (Pure Read)

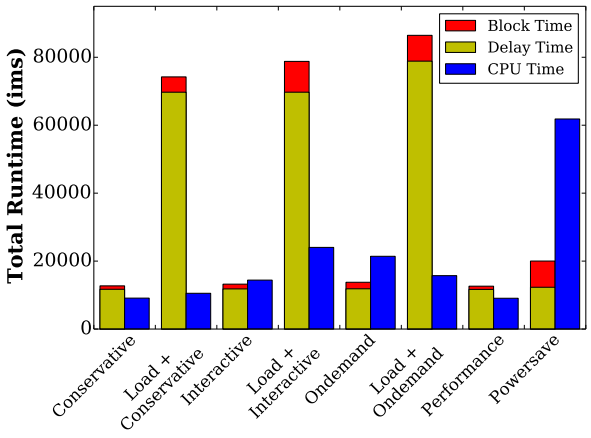
Figure 3: CDF of Governor Performance on Workloads A-C

and C (Figures 3b and 3c respectively) in response to added load. The Interactive governor's performance is significantly improved on the read-heavy workload B, but begins to suffer from variable performance on workload C. Conversely, Ondemand performance is relatively unaffected by load on workload B, but improves significantly in both performance and reduced variability on Workload C as governor-caused performance fluctuations are eliminated.

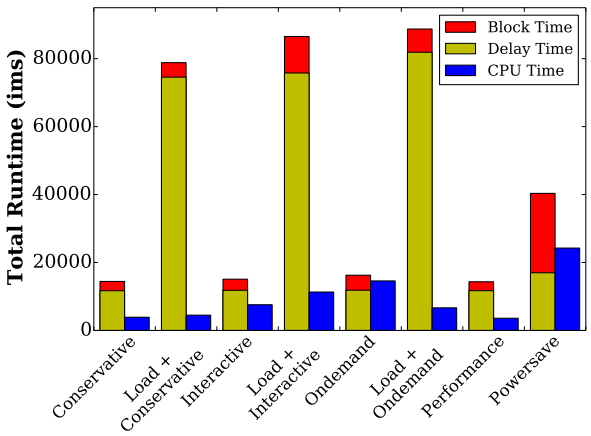
Conservative is Surprisingly Fast. Conservative, which as a policy is indented to keep CPU frequencies low, is consistently second



(a) Workload A



(b) Runtime on Workload E



(c) Runtime on Workload *H*

Figure 4: Per-Governor Runtimes for Workloads A, E, and *H*

fastest. It is possible that this is an artifact of the trial duration, and that decayed performance could appear in multi-hour trials.

Overall Throughput. Figure 4 shows performance for workloads A, E, and *H*. Blue bars indicate time spent running queries (CPU Time), while the stacked Yellow and Red bars show time spent

sleeping in between queries (Delay Time) or blocked on IO (Block Time), respectively. The Powersave governor is clearly slower, but especially bad on (CPU-intensive) scan-heavy workloads. Most notably, throughputs plummet under load. Specifically, the time that the benchmark app spends off-core, including both block time and delay time, is substantially worse. Worst-case latencies get worse under load for Interactive and Ondemand. Workload A is especially bad for interactive.

4 IS MY GOVERNOR WORKING?

As databases are commonly used by smartphone apps, their performance is often crucial to user experience. In many cases, the problem can be traced back to an inappropriate choice of governor. Another governor, with a different set of policies, could furnish better performance. Figuring out whether this is the case, however, is not straightforward. One governor is not necessarily better than another in all areas (with the obvious exception of Performance, which is by design an upper bound). Much more typically, better performance in one respect is counterbalanced by worse performance in another. Hence, we want to help developers answer the question of whether the governor is responsible for their app's performance issues Specifically, what should a developer look for, and how should they interpret their findings? We now present a decision process, summarized in Figure 5, which provides 3 sets of color-coded gradients of expected performance for each of 3 metrics: throughput (Figure 5a), median latency (Figure 5b), and 95th percentile latency (Figure 5c). In general, Blue means not a problem, Yellow indicates a potential problem, and Red indicates a serious limitation. The top 5 rows in each graph represent database performance metrics of each of the 5 governor choices under study when run on an otherwise unloaded system. Workloads are clustered by category: Write-Heavy (A, F) on the left, Read-Heavy (B, C, D) in the middle, and Scan-Heavy (E, *G*, *H*).

4.1 Governor Choice

The first metric the app developer should identify is what governor is already being used. Developers seeking absolute maximal or minimal database performance have likely already selected either the Performance or Powersave governor respectively. As the corresponding horizontal rows in the decision graphs show, these two choices unsurprisingly furnish upper and lower bounds to performance under all 3 metrics with an unloaded system.

For most developers, things will not be so simple: other governor choices offer trade-offs among different performance metrics. The Android system defaults on the Nexus platforms are the Ondemand and Interactive governors; one of these two choices will be the likely starting point for analysis.

4.2 Overall System Load

Given the likely selection of one of the mid-level governors, the second factor that developers to evaluate is the overall system load. Different types of apps can expect to be run under different system conditions. An interactive game, for example, can reasonably expect to enjoy uninterrupted complete foreground usage for several minutes. Others apps will have to contend with going out of focus or shunted into the background while other tasks consume resources.

For each of the 3 mid-level governors (Interactive, Ondemand, and Conservative), the 3 performance evaluation graphs each provide 2 alternative paths. Developers whose apps run on relatively unloaded systems should continue to reference the appropriate governor rows from the top of the decision graphs. If instead the app runs on a resource-loaded system, they should consult the 3 appropriately labeled lower rows. Considering the system load level on which an app's database runs is crucial, as the popular mid-level governors can offer conflicting performance trade-offs.

Decision Step Summary: Using the appropriate governor choice / system load level combination, select the appropriate row of the performance analysis graphs.

4.3 App Database Operation Mixture

The third metric developers should identify is the nature of database operations their apps request. The vertical columns of the graphs in Figure 5 represent the performance of workloads containing different combinations of operations. The last two columns in each graph, *G* and *H*, represent custom workloads designed to reflect the actual workload mixes we encountered in our previous study [14]. App developers lacking further specific information about their apps' usage patterns should focus on the data in these two columns. Apps known to be read-heavy in operation, however, should look at the information in columns B-C-D most closely (representing read-heavy workloads). Similarly, write-heavy apps should look at A and F, and scan-heavy ones at E.

Decision Step Summary: Use the type of database operation mixture to determine which column(s) of each of the 3 performance analysis graphs to consult.

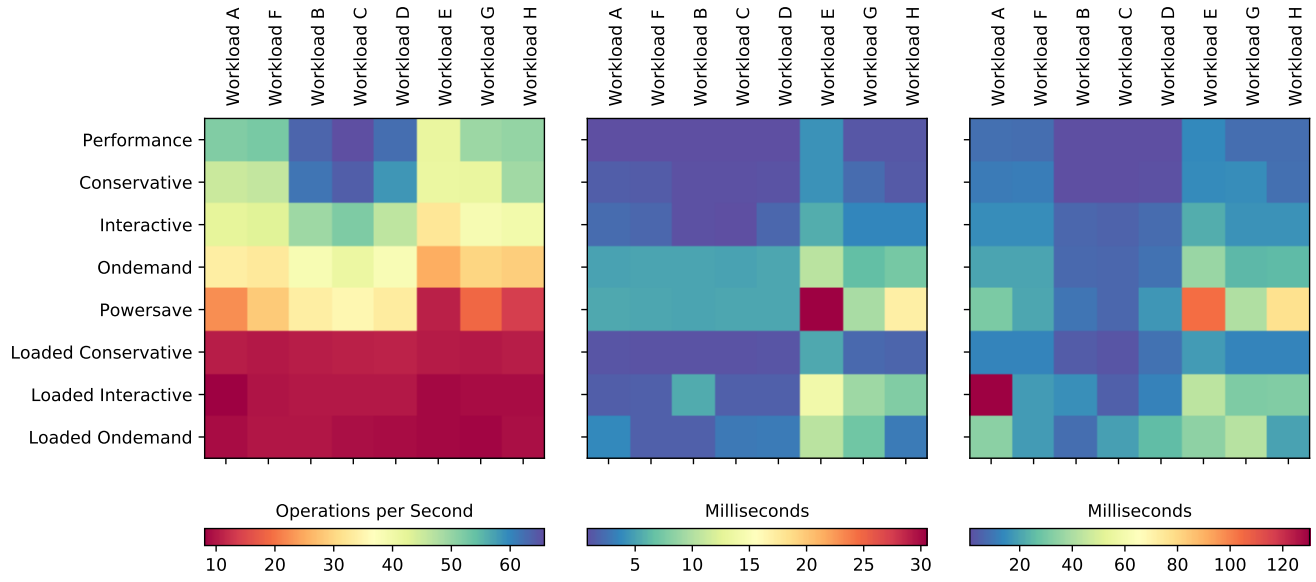
4.4 Database Performance Metrics

The final item developers need to consider is what measures are relevant for their apps. Each of the three performance graphs illustrate expected performance for three key measures: throughput, median latency, and 95th percentile latency. Using the rows and columns obtained in previous steps, developers can now obtain a customized set of performance expectations: Use cases falling into red or yellow regions may wish to consider workarounds like the use of futures for query responses to mitigate latency issues.

Decision Step Summary: The color gradient of the 3 boxes identified in previous steps indicates expected database performance bands in 3 key areas.

5 RELATED WORK

There have a been a number of performance studies focusing on mobile platforms and governors for managing their runtime performance characteristics [3, 4, 6, 7, 15]. Most of these studies focus on managing the performance and energy tradeoff and none look at the effect of the governor on embedded database performance. A few make the argument that for more effective over all system utilization considerations of the whole program stack must be made [13] and instead of managing applications individually, system wide services should be created for more wholistic management [11]. More recently, there has been interest in specialized studies focusing on performance and energy consumption of specific subsystems, like



(a) Expected throughput **(b) Expected median latency** **(c) Expected 95%ile latency**

Figure 5: Performance metrics of governors under varying database workloads and system conditions.

mobile web [2]. These studies do not, however, document the competing performance metric tradeoffs between governors. Nor do they explore the effect of system load on performance rankings of governor choices. We view our study and performance debugging methodology for embedded databases on mobile devices to be a first step at understanding the performance effect of the mobile platform on mobile databases and PocketData [14].

ACKNOWLEDGMENTS

This work was supported by NSF Award CNS-1629791 and relies on data gathered from PhoneLab [16]. The conclusions and opinions in this work are solely those of the authors and do not represent the views of the National Science Foundation.

REFERENCES

[1] Dominik Brodowski. 2018. (2018). https://android.googlesource.com/kernel/msm/+/android-7.1.0_r0.2/Documentation/cpu-freq/governors.txt
[2] Yi Cao, Javad Nejati, Muhammad Wajahat, Aruna Balasubramanian, and Anshul Gandhi. 2017. Deconstructing the Energy Consumption of the Mobile Page Load. *PMACS* 1, 1, Article 6 (June 2017), 25 pages. https://doi.org/10.1145/3084443
[3] Aaron Carroll and Gernot Heiser. 2010. An Analysis of Power Consumption in a Smartphone. In *USENIXATC*. 21–21.
[4] Xiang Chen, Yiran Chen, Mian Dong, and Charlie Zhang. 2014. Demystifying Energy Usage in Smartphones. In *Proceedings of the 51st Annual Design Automation Conference (DAC '14)*. ACM, New York, NY, USA, Article 70, 5 pages. https://doi.org/10.1145/2593069.2596676
[5] 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*. ACM, 143–154.
[6] Benedikt Dietrich and Samarjit Chakraborty. 2013. Power Management Using Game State Detection on Android Smartphones. In *MobiSys*. 493–494.
[7] Begum Egilmez, Gokhan Memik, Seda Ogrenci-Memik, and Oguz Ergin. 2015. User-specific Skin Temperature-aware DVFS for Smartphones. In *DATE*. 1217–1220.
[8] Google. 2018. Android Open Source Project. https://source.android.com/. (2018).
[9] Google. 2018. Youtube App. (2018). https://youtube.en.uptodown.com/android/
[10] Gtricks. 2018. (2018). https://www.gtricks.com/android/android-phone-overheating-cool-down-android-phone/
[11] Ahmed Hussein, Mathias Payer, Antony Hosking, and Christopher A. Vick. 2015. Impact of GC Design on Power and Performance for Android. In *SYSTOR*. Article 13, 12 pages.
[12] Business Insider. 2018. (2018). http://www.businessinsider.com/most-used-smartphone-apps-2017-8#2-youtube-9
[13] Melanie Kambadur and Martha A. Kim. 2014. An Experimental Survey of Energy Management Across the Stack. In *OOPSLA*. 329–344.
[14] Oliver Kennedy, Jerry Ajay, Geoffrey Challen, and Lukasz Ziarek. 2015. Pocket data: The need for TPC-MOBILE. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 8–25.
[15] Pietro Mercati, Andrea Bartolini, Francesco Paterna, Tajana Simunic Rosing, and Luca Benini. 2014. A Linux-governor Based Dynamic Reliability Manager for Android Mobile Devices. In *DATE*. Article 104, 4 pages.
[16] Anandatirtha Nandugudi, Anudipa Maiti, Taeyeon Ki, Muhammed Fatih Bulut, Murat Demirbas, Tevfik Kosar, Chunming Qiao, Steven Y. Ko, and Geoffrey Challen. 2013. PhoneLab: A Large Programmable Smartphone Testbed. In *SENSEMINE@SenSys*. ACM, 4:1–4:6.
[17] pcnexus. 2018. (2018). https://www.gtricks.com/android/android-phone-overheating-cool-down-android-phone/
[18] SQLite. 2017. Write-Ahead Logging. https://www.sqlite.org/draft/wal.html. (2017).
[19] Guru Prasad Srinivasa, Rizwana Begum, Scott Haseley, Mark Hempstead, and Geoffrey Challen. 2017. Separated By Birth: Hidden Differences Between Seemingly-Identical Smartphone CPUs. In *HotMobile*. ACM, 103–108.
[20] Shengqian Yang, Dacong Yan, and Atanas Rountev. 2013. Testing for Poor Responsiveness in Android Applications. In *MOBS*. 1–6.