Benchmarking Pocket-Scale Databases

Carl Nuessle[1], Oliver Kennedy[1], and Lukasz Ziarek[1]

[1] University at Buffalo
[2] {carlnues,okennedy,lziarek}@buffalo.edu

Abstract. Embedded database libraries provide developers with a common and convenient data persistence layer. They are a key component of major mobile operating systems, and are used extensively on interactive devices like smartphones. Database performance affects the response times and resource consumption of millions of smartphone apps and billions of smartphone users. Given their wide use and impact, it is critical that we understand how embedded databases operate in realistic mobile settings, and how they interact with mobile environments. We argue that traditional database benchmarking methods produce misleading results when applied to mobile devices, due to evaluating performance only at saturation. To rectify this, we present PocketData, a new benchmark for mobile device database evaluation that uses typical workloads to produce representative performance results. We explain the performance measurement methodology behind PocketData, and address specific challenges. We analyze the results obtained, and show how different classes of workload interact with database performance. Notably, our study of mobile databases at non-saturated levels uncovers significant latency and energy variation in database workloads resulting from CPU frequency scaling policies called governors — variation that we show is hidden by typical benchmark measurement techniques.

Keywords: PocketData · Mobile · SQLite · Android

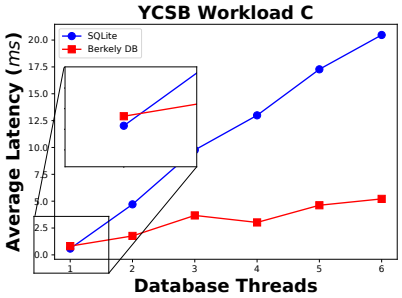
1 Introduction

General-purpose, embedded database libraries like SQLite and BerkeleyDB provide mobile app developers with full relational database functionality, contained entirely within their app's namespace. Embedded database libraries are used extensively in apps for smartphones, tablets, and other mobile computing devices, for example as the recommended approach for persisting structured data on iOS and Android. However, database libraries can often be a bottleneck [39], causing sluggish app performance and unnecessary battery drain. Hence, understanding the performance of database libraries and different database library configurations is critical, not just for library developers — but for any developer looking to optimize their app.

Unfortunately existing tools for measuring the performance of data management systems are presently targeted exclusively at server-class database systems [9, 1, 24, 12, 13], including distributed databases [21, 3] and key value stores [8,

Feature	Server-DB	Mobile-DB
Throughput	Relevant	Irrelevant
Latency	Relevant	Relevant
Startup Cost	Irrelevant	Relevant
Energy	Relevant	Crucial
Simult. Clients	10k+	1
Max Workload	100k+/s	400/s
HW Sharing	None or VM	Shared

(a) **Differences between server-class and pocket-class database workloads**



(b) **Database performance on YCSB workload A.**

Fig. 1: **Measuring mobile database performance using server-class benchmarks produces misleading results.**

2, 35]. Unsurprisingly, server-class database systems optimize for different criteria than embedded database libraries like SQLite or BerkeleyDB (Figure 1a). As a result, existing measurement tools can not be used directly to assess the performance of embedded database libraries. In this paper, we focus on one specific impedance mismatch: *Server-class database benchmarks use throughput as a proxy for overall database performance.* To determine performance, server-class database benchmarks measure throughput at saturation: The maximum query rate a database can sustain.

To understand why measurement at saturation is a problem, consider Figure 1b, which illustrates the results of one server-class benchmark (YCSB Workload C) applied to SQLite and BerkeleyDB, each using their default settings. Each point represents another thread worth of load being offered to the database. As more concurrent load is added, latency increases as contention overheads compound. The result is seemingly a clear victory for BerkeleyDB. However, this graph is not representative of actual smartphone usage. For example, our prior study [18] found that smartphone database instances need to cope with bursts of at most a few hundred queries, well below the throughput of a single thread. In this regime the systems are competitive, with SQLite being the faster of the two on some workloads.

More generally, by measuring performance at lower throughputs, results are more affected by noise from OS and hardware optimizations, background activity, and a host of other sources, many unique to mobile platforms. Although this noise has real implications for database performance in the wild, existing performance measurement techniques do not accurately capture it. In this paper, we identify several sources of measurement error that arise when measuring database performance at low throughputs, and show how they can produce misleading results. We introduce a new mobile benchmarking toolkit called PocketData, designed to work around these sources of error. To build PocketData, we extend the Android Operating System Platform (AOSP) [14] with new logging capabilities, control over relevant system properties, and a benchmark-runner

app. These extended capabilities make it easier to understand the precise causes of performance differences between systems or experimental trials. The result is a *toolkit for obtaining reliable, reproducible results when evaluating data management technologies on mobile platforms* like smartphones[3].

In our recent study of mobile database workloads [18], we made two key observations: (1) mobile workloads are dominated by key-value style queries, and (2) mobile database workloads are bursty. Following the first observation, we build on the Yahoo Cloud Services Benchmark (YCSB) [8], a popular key-value workload generator. To account for the latter observation, we extend the YCSB workload generator to operate at lower throughputs. We use the resulting workload generator to evaluate both performance and power consumption of SQLite on Android. One key finding of this evaluation was that for specific classes of workload, Android's default power management heuristics cause queries to take longer and/or consume more power. For example, we observe that *the default heuristics are often significantly worse than far simpler naive heuristics.* On nearly all workloads we tested, *running the CPU at half-speed significantly reduces power consumption, with minimal impact on performance.* Android's heuristics introduce higher latency and increase energy consumption due to excessive micromanagement of CPU frequencies.

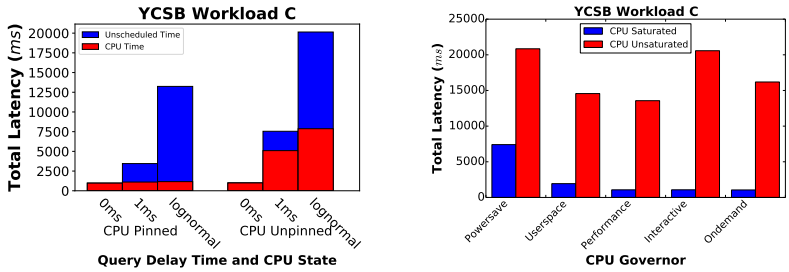
The specific contributions of this paper include: 1. We identify sources of error in database performance measurement at low-throughputs (Section 2). 2. We propose a database benchmarking framework called PocketData that makes it possible to mitigate these sources of error (Section 3). 3. We present results from an extensive benchmarking study of SQLite on mobile devices (Section 4). We cover related work and conclude in Section 5 and Section 6, respectively.

2 The Need for Mobile Benchmarking

Understanding the performance of data management systems is critical for tuning and system design. As a result, numerous benchmarks have emerged for *server-class data management systems* [9, 1, 24, 12, 13, 21, 3, 8, 2, 35]. In contrast, mobile data management [30, 28, 23] is a very different environment (Figure 1a). Here, we focus on one key difference: *mobile workloads operate significantly below saturation.* Measuring at saturation makes sense for typically multi-client server-class databases, which aim to maximize throughput. However, typical mobile data management happens at much lower rates [18], and on resource- and power-constrained hardware. As a result, metrics like latency and power consumption are far more important, while measuring performance at saturation hides meaningful performance quirks that can arise in practice.

The performance impact of frequency scaling is hidden at saturation. The most direct effect of measuring at below saturation is related to a feature called frequency scaling, which allows the operating system to adjust CPU performance in response to changing load. As load drops, lower CPU frequencies can significantly extend battery life. While frequency scaling does exist

[3] Available for download at `http://pocketdata.info`



(a) **Saturated v. Unsaturated** (b) **Latency by governor.**

Fig. 2: **Performance on for Workload C (read-only)**

on server-class database hardware, battery-powered mobile devices remain perpetually concerned with conserving energy. As such, they are far more aggressive with frequency scaling. During periods of low load, the OS can vary CPU frequencies to dozens of settings, or disable CPU cores altogether.

The particular policy heuristics to implement this adjustment are termed *governors*, such as the Ondemand governor used in most Linux distributions as well as earlier Android phones, or the Interactive governor used in more recent Android phones. Though their specific policies differ, both governors rely on only a single input datum: how busy the CPU is. By measuring the database at saturation, the CPU is kept completely, continuously busy, and virtually all governors react identically to this input: by running the CPU at maximum speed.

Figure 2a illustrates the impact of running at below saturation. To simulate operation at lower throughputs, we injected periodic thread sleeps into YCSB [8] Workload C. The figure shows three throughputs: At saturation (0ms delay), constant throughput below saturation (1ms delay), and bursty throughput below saturation (lognormal delay)[4]. As expected, in all cases, adding delays increased the time spent off-core (light-red). However, time spent doing useful work on the CPU core (dark-blue) also increases with delay. To confirm that this is a result of frequency scaling, the figure also shows the same experiment with the CPU pinned to its highest frequency; Here, on-core time is almost constant.

Making matters worse, the frequency scaling operation is expensive: No activity can be scheduled for several milliseconds while the core is scaled up or down. Hence, when the CPU is running at a low frequency, a database with a burst of work takes a double performance hit: first from having an initially slower CPU and second from waiting while the core scales up. Ironically, this means that a database running on a non-saturated CPU could significantly improve latencies by simply busy-waiting to keep the CPU pinned at a high frequency.

I/O performance is different at saturation. I/O on mobile devices is quite distinct from I/O on server-class devices. Most notably, mobile devices use exclusively flash media for persistent storage. Writes to flash-based storage are bursty, as the flash media's internal garbage collection identifies and reclaims overwrit-

[4] Here, we follow [18], which observes lognormal delay with a 6ms mean in typical use.

ten data blocks. On write-heavy workloads, this effect is far less pronounced below saturation, as the disk has a chance to "catch up".

CPU frequency scaling also plays a significant role in embedded database I/O behavior as well. Repeated idling, such as from lower loads or I/O-blocked operations are interpreted by the OS as a lack of work to be done and a signal to scale down cores to save power.

Governors are indistinguishable at saturation. Running benchmarks at full saturation, as a server-class study would do, obscures a broader performance factor. Consider Figure 2b, which shows the effect of frequency scaling on total latency when run with different CPU governor policies. The dark (blue) bars show database performance when the CPU is saturated; the lighter (red) bars show performance when the CPU is unsaturated. Each cluster shows the total latency for a workload when run under a particular CPU governor policy.

When running Workload C queries at saturation (dark-blue bars), database performance latency is identical across all governor choices, excepting only the Powersave governor which deliberately runs the CPU at a lower speed. Only when the workload is run below saturation (light-red bars) do significant differences between the governors begin to emerge. These differences can have a significant impact on real-world database performance and need to be addressed.

3 PocketData

Traditional database benchmarks [8, 36, 29] are designed for server-class databases, and rank databases by either the maximum number of queries processed per second (i.e., throughput) or equivalently the minimum time taken to process a batch of queries. In both cases, database performance is measured at saturation, which can produce misleading results on mobile. In this section, we first propose adjusting classical database benchmarks to run below saturation and then outline the PocketData workflow and runtime.

3.1 Benchmark Overview

The initial input to PocketData is a database and query workload, such as one generated by an existing server class benchmark. Specifically, we require an initial database configuration (e.g., as generated by TPC-H's `dbgen` utility), as well as a pre-generated workload: a sequence of queries (e.g., as generated by TPC-H's `qgen` utility). PocketData operates in three stages: First, a pre-processing stage prepares the query workload for later stages. Second, the benchmark database configuration is installed on the test device, and finally the prepared workload is evaluated.

Inter-Query Delays. The PocketData test harness simulates performance at levels below saturation by injecting delays in between queries. These delays are randomly generated by PocketData's workload preprocessor, which extends the pre-generated query workload with explicit instructions to sleep the

benchmark thread. The length and regularity of the inter-query delays is provided to as a parameter to the preprocessor. Throughout the remainder of the paper, we consider three values for this parameter: 1. A **log**normally distributed delay, mirroring typical app behavior [18]. 2. A fixed **1ms** inter-query delay, for comparison, and 3. **Zero** delay, or performance at saturation.

3.2 Benchmark Harness

The second and third stages are run by the benchmark harness, whicc a driver application and a rooted version of the standard Android platform with customized performance parameters. The application part of the benchmark connects to an embedded database through a modular driver. We developed drivers for: 1. Android OS's native SQLite integration, 2. BerkeleyDB through JDBC, and 3. H2 through JDBC. As it is used almost exclusively on the two major mobile platforms, our focus in this paper is on evaluating SQLite[5].

The benchmark harness takes three parameters: A CPU governor, a database configuration, and a workload annotated with delays. The selected governor is enabled by the benchmark as the phone boots up. After boot, the benchmark next initializes the database to the selected configuration, creating database files (if needed), creating tables, and pre-loading initial data. Once the database is initialized, the benchmark app exits and restarts.

After the benchmark app restarts it loads the pre-defined workload into memory. The choice to use a pre-defined, pre-loaded trace was made for two reasons. First, this ensures that overheads from workload generation remain constant across experiments; there is no cost for assembling the SQL query string representation. Second, having the same exact sequence of queries allows for completely repeatable experiments across different experimental configurations.

Metrics Collected. Log data was collected through ftrace. We instrumented the Android kernel, SQLite database engine, and driver application to log and timestamp the following events: 1. I/O operations like FSync, Read, and Write; 2. Context switches to and from the app's namespace; 3. Changes in CPU voltage scaling; and 4. Trace start and end times.

Logging context switches allows us to track points where the app was scheduled on-core, track background application use, and see when cores are idling. This is crucial, as unlike in server-class database measurement, we are intentionally operating the embedded database at well below saturation. The overhead of native-code platform events (1) and kernel-level events (2-3) are minimal. Trace start and end times, while injected from the app, are only 2 events and have minimal total impact.

3.3 The PocketData Benchmark

We base the PocketData measurement workload on insights drawn from our prior study [18], which found that smartphone queries typically follow key-value-style access and update patterns. Queries or updates operate on individual rows,

[5] Complete benchmark results are available at `http://www.pocketdata.info/`

or (rarely) the entire table. A quarter of apps observed by the study used exclusively key-value-style queries. Even the median app's workload was over 80% key-value style queries. Accordingly, we build PocketData by adapting the workloads from YCSB [8], an industry standard benchmark for key-value stores. We used an initial database of 500 records, approximately the median size of databases in our prior study [18] and a workload of 1800 operations per trial.

4 Benchmark Results

We organize this section by first discussing our application of the PocketData benchmark to our test environment. We then overview the results obtained from our study, and highlight areas identified for potential system performance improvement. Finally, we discuss measurement variance trends we observed, and identify two sources of this variance.

Reference Platforms. Our database benchmarking results were obtained from from two Android Nexus 6 devices, running Android OS 6.0.1, with 2GB RAM and a Quad-core 2.3 GHz CPU (quality bin 2 for both devices). One of the Nexus 6 devices was modified to permit energy measurements, which we collected using a Monsoon LVPM Power Meter[6]. To ensure measurement consistency, we modded the AOSP on the device to disable a feature that turns the screen on when it is plugged in or unplugged — the screen remained off throughout the benchmark. For one set of experiments, in order to analyze the source of variance in database latencies, we additionally modded the SQLite engine in AOSP to monitor time spent performing I/O operations.

4.1 Results Obtained and Analysis Method

Our key findings for the Nexus 6 are as follows:

- Below saturation, Android's default governors keep the CPU at approximately half-speed, even on CPU-intensive workloads, reducing performance.
- A governor that pins the CPU to half-speed outperforms both default governors on virtually all workloads below saturation.
- Below saturation and on a fixed workload, both of Android's default governors also under-perform with respect to power consumption.

Using the Monsoon meter, we measured the total energy consumed by the system, from launch to completion of the benchmark runner app while running a single workload. To account for a spike in power consumption as the runner app launches and exits, we count net energy use relative to a null workload that simply launches and exits the benchmark without running any queries.

Results by Workload. The multiple workloads within the PocketData benchmark yield finer insight into performance under different types of conditions. For conciseness, we focus our discussion on a workload subset that explores

[6] `http://www.msoon.com/LabEquipment/`

Workload	Description
YCSB-A	50% write, 50% read zipfian
YCSB-B	5% write, 95% read zipfian
YCSB-C	100% read zipfian
YCSB-E	5% append, 95% scan zipfian

Fig. 3: **The six YCSB and two PocketData workloads.**

these differences (A, B, C, E). As shown in Figure 3 this results in a gradient of read-heavy to write-heavy (C, B, A, respectively), as well as a more CPU-intensive scan-heavy workload. We specifically divide our discussion into three categories of workload: Read-heavy (B,C), Write-heavy (A), and Scan-heavy (E).

A second dimension of analysis is CPU load. As we discussed in the introduction, system performance can change dramatically when the CPU operates below saturation. Thus, we present results for two different CPU conditions: saturated (0ms delay) and unsaturated (lognormal delay).

Next, we ran each workload under each of 5 different CPU governor policies. 3 of them are non-default choices: **Performance** (run at the highest possible speed, 2.65 GHz), **Fixed-50** (The customizable Userspace governor set to run at a fixed midpoint frequency of 1.26 GHz), and **Powersave** (run the CPU at the lowest possible speed, 300 MHz). The last 2 choices, **Interactive** and **Ondemand**, are the current and previous Android defaults as discussed in Section 2.

The 4 workloads (A, B, C, and E), 2 CPU saturation settings, and 5 governor policies produce 40 measurement combinations. We ran each combination 3 times, and report the average and 90% confidence intervals. As we discuss below, certain workload combinations proved much more consistent in measurement than others. We observed measurement variance resulting from I/O blocking and the phone's power source. To investigate this aspect further, we re-ran several representative workloads, while measuring database file access time at the SQLite-kernel boundary. We re-ran each of these workloads under each of 3 different power source settings, 6 additional times each.

4.2 Read Heavy Workloads

Read-heavy database workloads are particularly important, as reads account for three-quarters of a typical database workload [18]. Energy consumption is also a key issue on mobile. CPU governors, in turn, heavily influence the behavior of both of these factors. We therefore focus on the performance-energy relationship of database operations under different governor settings. Our study results show that *system default governors result in sub-optimal latencies and energy costs for database workloads* in the bulk of representative read-heavy scenarios.

Workloads are CPU-Bound but respond quickly. Latencies from read operations are due nearly entirely to CPU time (plus explicit benchmark delays) as a consequence of pre-caching performed by the SQLite database library. Figure 2a, for read-only workload YCSB-C, illustrates this clearly: there is virtually no unscheduled time beyond the total time spent explicitly waiting (0s, 2s, and

12s, respectively). There was very little I/O activity under C, nearly all of it immediately at the start of the workload as the table is pre-fetched. Because of this pre-fetching, reads are serviced mostly from cache and there is little blocking.

Non-default governors offer better performance. Mobile platforms must always balance performance against energy. Figures 5 and 6 show the database latency and energy cost for each of the 5 governor choices for 2 read-heavy loads: C is read-only; B adds 5% writes. An ideal governor would be as close to the bottom left of the scatterplot as possible – that is, it should optimize both database latency and energy consumption.

Uninterrupted query timing essentially means the CPU will be running at saturation regardless of governor choice, and latencies tend to flatten. Thus, unsurprisingly, on uninterrupted, read-only workloads (Figure 5a), both default governors nearly match the performance governor's latency. However, as the vertical scale of 5a shows, running saturated read workloads with the Performance governor also significantly *saves* rather than costs energy versus all other choices. On this workload, there is no benefit to be gained by micro-managing system performance, and so the static governor significantly outperforms the dynamic defaults. The saturated 95% read workload B (Figure 6a) shows similar characteristics, albeit with an even more significant latency gap between the Performance and the default governors. Here, the limited I/O is interpreted by the system as a reduction in workload, and thus an opportunity to ramp down the CPU. However, the overhead of micromanagement again outweighs the benefits.

Unsaturated read-heavy workloads (Figures 5b and 6b) model bursty, interactive usage patterns. Here, we observe a performance-energy trade-off between the Performance and Fixed-50 governors. On both read-heavy workloads under the performance governor, the average query is processed approximately 0.5ms faster, while under the Fixed-50 governor power consumption is reduced by approximately a third. Notably, the Fixed-50 governor outperforms both default governors on both workloads and on both axes: Lower energy and lower latency.

Keeping the CPU hot can reduce energy costs. We observe that both default governors perform better on saturated workloads. When such saturation is possible, it can be advantageous, not just from a latency but also an energy standpoint, to keep the CPU busy. This makes batching read queries especially important, as doing so can significantly reduce power consumption. Alternatively, it may be possible for apps to reduce power consumption by busy-waiting the CPU during I/O operations when such operations are short and infrequent.

Frequency scaling has a non-monotone effect on energy consumption. On the read-only workload C (Figure 7a), energy cost is minimized with the CPU running at half (Fixed-50), rather than minimum speed (Powersave). Energy consumption scales super-linearly with frequency and there is an unavoidable fixed energy cost to simply keeping the core powered on (although recall that we keep the screen off during tests). Thus, the benefit of slowing the processor down is outweighed by the cost of keeping the core powered up longer.

Threads are very sensitive to governor performance differences. When running read-heavy workloads, unless the CPU is already at saturation, the CPU often runs well below maximum speed under the Interactive governor. Figure 4 shows that, with the addition of 1ms pauses, the CPU frequency is largely stuck near 500 MHz, well below the 2.65 GHz maximum (the minimum is 300 MHz). As the core is running at a lower frequency when a query arrives, the query takes a performance hit. As the query finishes the Interactive governor ramps up the CPU unnecessarily, wasting energy. This same effect appears in Figure 2b, which shows higher latencies for intermittent queries (light-red bars on graph) with either default governor (Interactive or Ondemand) than when the CPU is saturated (Performance).

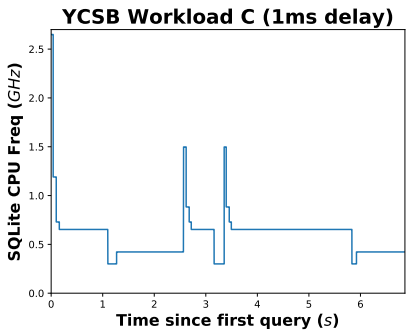


Fig. 4: **Benchmark App Thread's CPU Frequency.**

4.3 Write Heavy Workloads

Write-heavy workloads on the Android platform have higher latencies than their read-heavy counterparts, and latencies of the fixed-speed (non-default) governors scale inversely with energy consumption.

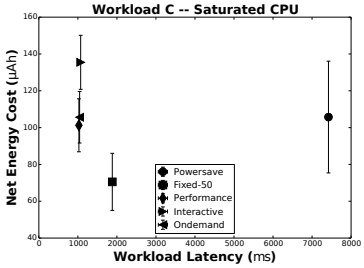
Non-default governors again improve performance. Figure 7 shows the latency/energy metrics for Workload A, which is 50% write operations. Looking at the results for the saturated workload (Figure 7a), we observe that the Fixed-50 governor significantly outperforms all other governors. The Powersave governor in this case has the lowest energy cost, but also has the worst performance. However, we note that, for write-heavy threads running on saturated CPUs (which is the case when a series of write operations arrive consecutively), the system defaults again under-perform significantly.

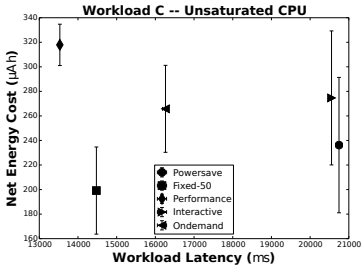
Write-heavy operations that occur intermittently exhibit a similar latency-energy metric to that of intermittent read-heavy threads. As before, Figure 7b shows the tradeoff: Performance and Powersave offer the opposing extremes of latency and energy, while the Fixed-50 governor compromises between extremes.

4.4 Scan Heavy Workloads

Scan-heavy workloads involve longer-running, CPU-bound queries, which keep the CPU loaded for longer intervals. As a result, we see less variation between saturated and unsaturated workloads.

Table Scans are CPU-Intensive. In workload E, reads are scan operations. The increase in latency for workload E is mostly due to sharply higher CPU time, with marginally greater non-scheduled non-benchmark delay time. This





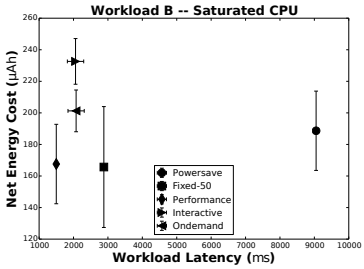
Governor	Latency	LatErr	LatErr%	Energy	EnErr	EnErr%
Powersave	7420.37	70.56	0.95	105.74	30.36	28.71
Fixed-50	1881.71	42.89	2.28	70.53	15.51	21.99
Performance	1026.28	34.70	3.38	101.26	14.39	14.21
Interactive	1071.52	5.70	0.53	135.47	14.68	10.83
Ondemand	1044.73	9.08	0.87	105.63	14.00	13.25

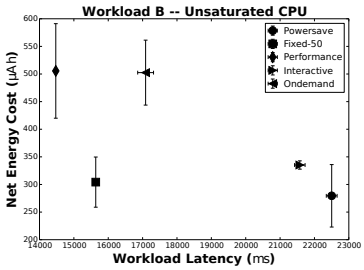
(a) **Uninterrupted query arrivals**

Governor	Latency	LatErr	LatErr%	Energy	EnErr	EnErr%
Powersave	20746.12	83.19	0.40	236.23	55.15	23.34
Fixed-50	14475.99	76.37	0.53	199.18	35.49	17.82
Performance	13531.38	34.55	0.26	317.89	16.84	5.30
Interactive	20552.94	29.13	0.14	274.67	54.59	19.87
Ondemand	16261.75	75.97	0.47	265.77	35.43	13.33

(b) **Intermittent query arrivals**

Fig. 5: **Workload C: Latency performance and energy costs.**





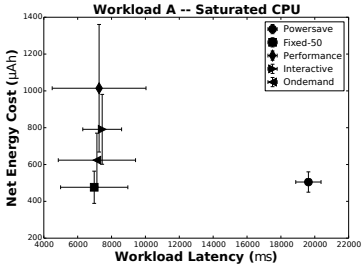
Governor	Latency	LatErr	LatErr%	Energy	EnErr	EnErr%
Powersave	9050.50	59.73	0.66	188.66	25.13	13.32
Fixed-50	2881.91	26.50	0.92	165.69	38.30	23.12
Performance	1499.00	43.99	2.93	167.59	25.17	15.02
Interactive	2058.75	237.60	11.54	232.62	14.47	6.22
Ondemand	2075.51	236.12	11.38	201.27	13.22	6.57

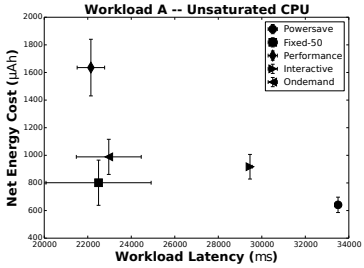
(a) **Uninterrupted query arrivals.**

Governor	Latency	LatErr	LatErr%	Energy	EnErr	EnErr%
Powersave	22501.64	157.16	0.70	279.47	56.54	20.23
Fixed-50	15636.47	33.89	0.22	304.26	45.42	14.93
Performance	14473.82	43.26	0.30	505.56	85.59	16.93
Interactive	21574.05	152.85	0.71	335.25	7.54	2.25
Ondemand	17087.27	230.54	1.35	502.52	58.80	11.70

(b) **Intermittent query arrivals.**

Fig. 6: **Workload B: Latency performance and energy costs.**





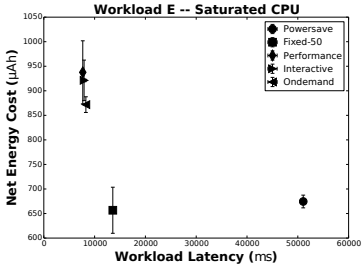
Governor	Latency	LatErr	LatErr%	Energy	EnErr	EnErr%
Powersave	19623.81	746.31	3.80	505.00	55.21	10.93
Fixed-50	6976.35	1985.89	28.47	476.50	87.50	18.36
Performance	7264.69	2769.70	38.13	1014.37	346.41	34.15
Interactive	7447.07	1146.02	15.39	791.25	189.75	23.98
Ondemand	7135.57	2281.21	31.97	623.73	146.73	23.52

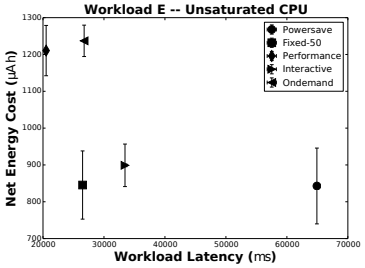
(a) **Uninterrupted query arrivals.**

Governor	Latency	LatErr	LatErr%	Energy	EnErr	EnErr%
Powersave	33503.64	148.43	0.44	641.33	55.67	8.68
Fixed-50	22496.20	2416.73	10.74	801.56	163.59	20.41
Performance	22143.90	631.65	2.85	1635.42	205.02	12.54
Interactive	29456.82	101.43	0.34	917.39	89.12	9.71
Ondemand	22966.05	1491.80	6.50	988.84	127.23	12.87

(b) **Intermittent query arrivals.**

Fig. 7: **Workload A: Latency performance and energy costs.**





Governor	Latency	LatErr	LatErr%	Energy	EnErr	EnErr%
Powersave	51067.97	193.85	0.38	674.52	12.96	1.92
Fixed-50	13582.61	125.16	0.92	656.67	46.93	7.15
Performance	7669.62	63.34	0.83	937.61	64.22	6.85
Interactive	7926.37	350.87	4.43	921.25	41.28	4.48
Ondemand	8281.21	74.40	0.90	871.84	15.98	1.83

(a) **Uninterrupted query arrivals.**

Governor	Latency	LatErr	LatErr%	Energy	EnErr	EnErr%
Powersave	64920.82	187.59	0.29	842.86	102.94	12.21
Fixed-50	26494.90	238.82	0.90	845.44	92.60	10.95
Performance	20516.58	111.90	0.55	1210.57	68.30	5.64
Interactive	33449.11	299.14	0.89	898.96	57.73	6.42
Ondemand	26748.52	33.52	0.13	1237.04	42.63	3.45

(b) **Intermittent query arrivals.**

Fig. 8: **Workload E: Latency performance and energy costs.**

penalty is due to additional actual computation rather than frequency scaling: Unlike with previous workloads, CPU time for E remains relatively unaffected by increased benchmark delay settings. For E, DB usages involving significant scans can still be serviced largely from cache, but they incur computation costs.

Non-default governors often improve performance. Workload E, comprised of 95% table scans, is shown in Figure 8. When database operations arrive uninterrupted, Figure 8a shows that the default Interactive governor offers the best latency. As in the simpler read-heavy workloads, the Fixed-50 governor (and not the Powersave governor) offers the lowest energy cost, with only a slight latency penalty.

Unsaturated scan-heavy threads generally follow the latency-energy trade-off pattern of previous workloads as well. Figure 8b shows that, as before, Performance and Powersave offer opposing extremes of latency and energy metrics. However, both are only negligibly better than the Fixed-50 governor, which also outperforms the default governors on both metrics.

4.5 Sources of Measurement Variance

Load	Reads	Writes	Syncs
A	4073	4986	2234
B	3675	572	244
C	3625	0	0
E	3719	1037	464

Fig. 9: **Number of SQLite I/O Operations**

Energy usage measurements showed a significant, though constant variation (reflected in the vertical error bars in graphs 5-8) across all test configurations. This is unsurprising, as overall energy usage is low, rendering measurement susceptible to noise from small variations in background system behavior. Note that the noise level stays constant, even as overall load increases as in workload E. Conversely, latency variance (horizontal error bars) is almost always low. On loads B, C and E, for both saturated and unsaturated CPUs, all had relatively

small margins: 11% at most. However, measurement error for workload A, and particularly for a saturated CPU, was anomalously high: from 15%-38% for all CPU policies except Powersave.

Workload A is write-heavy and involves a relatively large number of I/O operations compared to other loads (see Figure 9). To confirm that fluctuation in I/O time was indeed the source of the large latency error measurement for A, we instrumented SQLite to measure the time spent blocked on read, write, and fsync operations. We re-ran A with a saturated CPU 20 additional times. Figure 10 compares, for workload A under the default Interactive governor, the latencies of these runs. Total latency varies significantly, exhibiting a bimodal distribution, with one mode around 6s and a second mode at 11s. Looking deeper, the latency of each run is composed of I/O time (dark blue) and non-I/O time (light orange). While non-I/O time for each of the runs is quite consistent, I/O time also varies bimodally. Likely, this is due to flash storage overhead having to prepare for block erases and writes. The saturated CPUs particularly expose this: unsaturated CPUs allow flash erasure to take place during quiescent periods, rather than forcing the benchmark to block. Workloads B, C and E exhibit much less latency variance, as they lack significant write activity and do not need to erase flash. Workload A is dominated by write costs and thus suffers increased variance.

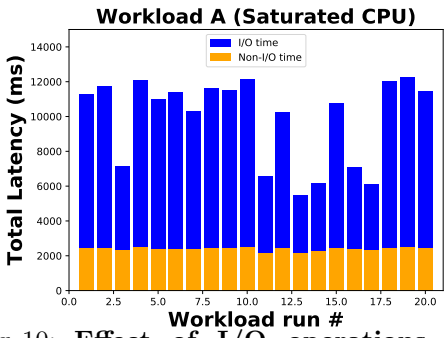


Fig. 10: **Effect of I/O operations on workload latency variance**

5 Related Work

Lightweight DBMSes. MySQL [40] got its start as a lightweight DBMS, while libraries like SQLite [30] and BerkeleyDB [28] both provide server-less database functionality within an application's memory space. TinyDB [23] is a lightweight DBMS intended for use in distributed IoT settings. In addition to aiming for a low memory footprint, it allows queries to be scheduled for distributed execution over a cluster of wireless sensor motes. While these approaches target application developers, other efforts like GestureDB [26] target users of mobile devices and optimize for different types of interaction modalities.

Benchmarking. A range of benchmarks do exist for server-class databases [36, 29, 4] and other data management platforms [8, 15, 20]. However, as we point out throughout this paper, assumptions typically made by these benchmarks produce invalid results when evaluating pocket-scale data management systems. There is however overlap on some non-traditional metrics like energy use in data

management platforms [27, 33, 32, 31]. Notably, configuration parameters optimized for these benchmarks typically involve significant changes to the hardware itself. Through features like frequency scaling on the CPU and RAM, mobile devices are capable of far more fine-grained reconfiguration on the fly, significantly changing the evaluation landscape.

Conversely, A number of other benchmarks target embedded devices. AndroStep [22] evaluates phone performance in general terms of CPU and energy usage. Energy is also a common specific area of study – Wilke et al. compare consumption by applications [38]. AndroBench [19] studies the performance of file systems, but uses SQLite as a representative workload for benchmarking filesystem performance. While these benchmarks use SQLite as a load generator, it is the filesystem being evaluated and not the database itself.

Profiling Studies. One profiling study by Wang and Rountev [37] explored sources of perceived latency in mobile interfaces. They found databases to be a common limiting factor. A study by Prasad et al. [34] looked at hardware performance profiles relative to CPU quality ratings assigned by the chip manufacturer. They found a wide distribution of thermal profiles and CPU performance for devices ostensibly marketed as being identical. Our previous study [18] used a user-study to explore characteristics of mobile database workloads, and forms the basis for PocketData as described in this paper.

There have a been a number of performance studies focusing on mobile platforms and governors for managing their runtime performance characteristics [6, 25, 10, 11, 7]. 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 [17] and instead of managing applications individually, system wide services should be created for more wholistic management [16]. More recently, there has been interest in specialized studies focusing on performance and energy consumption of specific subsystems, like mobile web [5]. 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.

6 Conclusions

The mobile platform presents unique characteristics for database benchmarking. The systems themselves are resource-limited, and the typical workloads differ markedly from those experienced by traditional server-class databases. Furthermore, mobile systems are structured differently, with power management and flash memory I/O contributing a significant amount of noise to measurement efforts. Measurement systems that fail to account for these differences will miss critical performance information. While we focused our study on SQLite, the

system default database, we designed our benchmark to be database-agnostic, and results from PocketData on other configurations can be found on our website `http://pocketdata.info`.

For a given database and workload, different governors yield different database performance and energy consumption metrics. A non-default governor selection can often improve markedly on either latency or energy performance – sometimes in both simultaneously. While the database is aware of the information necessary to make this choice, the kernel is typically not, suggesting opportunities for future improvement. In future work, we will explore how the kernel can be adapted to solicit this information and then incorporate it into a wiser governor selection.

References

1. Ahmed, M., Uddin, M.M., Azad, M.S., Haseeb, S.: Mysql performance analysis on a limited resource server: Fedora vs. ubuntu linux. In: SpringSim (2010)
2. Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., Paleczny, M.: Workload analysis of a large-scale key-value store. SIGMETRICS Perform. Eval. Rev. **40**(1), 53–64 (Jun 2012)
3. Baumgärtel, P., Endler, G., Lenz, R.: A benchmark for multidimensional statistical data. In: ADBIS (2013)
4. Bitton, D., DeWitt, D.J., Turbyfill, C.: Benchmarking database systems A systematic approach. In: VLDB. pp. 8–19. Morgan Kaufmann (1983)
5. Cao, Y., Nejati, J., Wajahat, M., Balasubramanian, A., Gandhi, A.: Deconstructing the energy consumption of the mobile page load. PMACS **1**(1), 6:1–6:25 (Jun 2017)
6. Carroll, A., Heiser, G.: An analysis of power consumption in a smartphone. In: USENIXATC. pp. 21–21 (2010)
7. Chen, X., Chen, Y., Dong, M., Zhang, C.: Demystifying energy usage in smartphones. In: DAC (2014)
8. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: SOCC (2010)
9. Curino, C.A., Difallah, D.E., Pavlo, A., Cudre-Mauroux, P.: Benchmarking oltp/web databases in the cloud: The oltp-bench framework. In: CloudDB (2012)
10. Dietrich, B., Chakraborty, S.: Power management using game state detection on android smartphones. In: MobiSys. pp. 493–494 (2013)
11. Egilmez, B., Memik, G., Ogrenci-Memik, S., Ergin, O.: User-specific skin temperature-aware dvfs for smartphones. In: DATE. pp. 1217–1220 (2015)
12. Erling, O., Averbuch, A., Larriba-Pey, J., Chafi, H., Gubichev, A., Prat, A., Pham, M.D., Boncz, P.: The ldbc social network benchmark: Interactive workload. In: SIGMOD (2015)
13. Frank, M., Poess, M., Rabl, T.: Efficient update data generation for dbms benchmarks. In: ICPE (2012)
14. Google: Android open source project. https://source.android.com/ (2018)
15. Gupta, A., Davis, K.C., Grommon-Litton, J.: Performance comparison of property map and bitmap indexing. In: DOLAP. pp. 65–71. ACM (2002)
16. Hussein, A., Payer, M., Hosking, A., Vick, C.A.: Impact of gc design on power and performance for android. In: SYSTOR. pp. 13:1–13:12 (2015)
17. Kambadur, M., Kim, M.A.: An experimental survey of energy management across the stack. In: OOPSLA. pp. 329–344 (2014)

18. Kennedy, O., Ajay, J.A., Challen, G., Ziarek, L.: Pocket Data: The need for TPC-MOBILE. In: TPC-TC (2015)
19. Kim, J., Kim, J.: Androbench: Benchmarking the storage performance of android-based mobile devices. In: ICFCE. Advances in Intelligent and Soft Computing, vol. 133, pp. 667–674. Springer (2011)
20. Klein, J., Gorton, I., Ernst, N.A., Donohoe, P., Pham, K., Matser, C.: Performance evaluation of nosql databases: A case study. In: PABS@ICPE. pp. 5–10. ACM (2015)
21. Kuhlenkamp, J., Klems, M., Röss, O.: Benchmarking scalability and elasticity of distributed database systems. pVLDB **7**(12), 1219–1230 (Aug 2014)
22. Lee, K.: Mobile benchmark tool (mobibench)
23. Madden, S.R., Franklin, M.J., Hellerstein, J.M., Hong, W.: TinyDB: An acquisitional query processing system for sensor networks. ACM TODS **30**(1), 122–173 (Mar 2005)
24. Malkowski, S., Jayasinghe, D., Hedwig, M., Park, J., Kanemasa, Y., Pu, C.: Empirical analysis of database server scalability using an n-tier benchmark with read-intensive workload. In: SAC (2010)
25. Mercati, P., Bartolini, A., Paterna, F., Rosing, T.S., Benini, L.: A linux-governor based dynamic reliability manager for android mobile devices. In: DATE. pp. 104:1–104:4 (2014)
26. Nandi, A., Jiang, L., Mandel, M.: Gestural query specification. PVLDB **7**(4), 289–300 (2013)
27. Niemann, R.: Towards the prediction of the performance and energy efficiency of distributed data management systems. In: ICPE Companion. pp. 23–28 (2016)
28. Olson, M.A., Bostic, K., Seltzer, M.I.: Berkeley DB. In: USENIX Annual Technical Conference, FREENIX Track. pp. 183–191. USENIX (1999)
29. O'Neil, P.E., O'Neil, E.J., Chen, X.: The star schema benchmark (ssb) (2007)
30. Owens, M., Allen, G.: SQLite. Springer (2010)
31. Poess, M., Nambiar, R.O.: Energy cost, the key challenge of today's data centers: a power consumption analysis of TPC-C results. PVLDB **1**(2), 1229–1240 (2008)
32. Poess, M., Nambiar, R.O., Vaid, K.: Optimizing benchmark configurations for energy efficiency. In: ICPE. pp. 217–226. ACM (2011)
33. Poess, M., Nambiar, R.O., Vaid, K., Stephens, J.M., Huppler, K., Haines, E.: Energy benchmarks: a detailed analysis. In: e-Energy. pp. 131–140. ACM (2010)
34. Srinivasa, G.P., Begum, R., Haseley, S., Hempstead, M., Challen, G.: Separated by birth: Hidden differences between seemingly-identical smartphone cpus. In: HotMobile. pp. 103–108. ACM (2017)
35. Tomás, G., Zeller, P., Balegas, V., Akkoorath, D., Bieniusa, A., Leitão, J.a., Preguiça, N.: Fmke: A real-world benchmark for key-value data stores. In: PaPoC (2017)
36. Transaction Processing Performance Council: TPC-H, TPC-C, and TPC-DS specifications. http://www.tpc.org/
37. Wang, Y., Rountev, A.: Profiling the responsiveness of android applications via automated resource amplification. In: MOBILESoft. pp. 48–58. ACM (2016)
38. Wilke, C., Piechnick, C., Richly, S., Püschel, G., Götz, S., Aßmann, U.: Comparing mobile applications' energy consumption. In: SAC. pp. 1177–1179. ACM (2013)
39. Yang, S., Yan, D., Rountev, A.: Testing for poor responsiveness in Android applications. In: Workshop on Engineering Mobile-Enabled Systems. pp. 1–6 (2013)
40. Yarger, R.J., Reese, G., King, T.: MySQL and mSQL - databases for moderate-sized organizations and websites. O'Reilly (1999)