The Journal of Systems and Software 146 (2018) 99–111

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Performance and programming effort trade-offs of android persistence frameworks

Zheng Jason Song*, Jing Pu, Junjie Cheng, Eli Tilevich

Software Innovations Lab, Virginia Tech, Blacksburg, VA 24061, United States

ARTICLE INFO

Article history:
Received 30 August 2017
Revised 26 June 2018
Accepted 21 August 2018
Available online 23 August 2018

Keywords:
Persistence framework
Mobile application
Performance
Programming effort

ABSTRACT

A fundamental building block of a mobile application is the ability to persist program data between different invocations. Referred to as *persistence*, this functionality is commonly implemented by means of persistence frameworks. Without a clear understanding of the energy consumption, execution time, and programming effort of popular Android persistence frameworks, mobile developers lack guidelines for selecting frameworks for their applications. To bridge this knowledge gap, we report on the results of a systematic study of the performance and programming effort trade-offs of eight Android persistence frameworks, and provide practical recommendations for mobile application developers.

© 2018 Elsevier Inc. All rights reserved.

1. Introduction

Any non-trivial application includes a functionality that preserves and retrieves user data, both during the application session and across sessions; this functionality is commonly referred to as *persistence*. In persistent applications, relational or non-relational database engines preserve user data, which is operated by programmers either by writing raw database operations, or via data persistence frameworks. By providing abstractions on top of raw database operations, data persistence frameworks help streamline the development process.

As mobile devices continue to replace desktops as the primary computing platform, Android is poised to win the mobile platform contest, taking the 82.8% share of the mobile market in 2015 (Smartphone OS market share, 2015) with more than 1.6 million applications developed thus far (Statista, 2015). Energy efficiency remains one of the key considerations when developing mobile applications (Jha, 2011; Kwon and Tilevich, 2013a; Li et al., 2014), as the energy demands of applications continue to exceed the devices' battery capacity. Consequently, in recent years researchers have focused their efforts on providing Android developers with insights that can be used to improve the energy efficiency of mobile applications. The research literature on the subject includes approaches ranging from general program analysis and modeling (Hao et al., 2013; Tiwari et al., 1996; Dong and Zhong, 2010; Cohen et al., 2012) to application-level analysis (Sahin et al., 2012; Hindle, 2012; Kwon and Tilevich, 2013b; Pinto et al., 2014).

Despite all the progress made in understanding the energy impact of programming patterns and constructs, a notable omission in the research literature on the topic is the energy consumption of persistence frameworks. Although an indispensable building block of mobile applications, these frameworks have never been systematically studied in this context, which can help programmers gain a comprehensive insight on the overall energy efficiency of modern mobile applications. Furthermore, to be able to make informed decisions when selecting a persistence framework for a mobile application, developers have to be mindful of the energy consumption, execution time, and programming effort trade-offs of major persistence frameworks.

To that end, this paper reports on the results of a comprehensive study we have conducted to measure and analyze the energy consumption, execution time, and programming effort trade-offs of popular Android persistence frameworks. For Android applications, persistence frameworks expose their APIs to the application developers as either object-relational mappings (ORM), object-oriented (OO) interfaces, or key-value interfaces, according to the underlying database engine. In this article, we consider the persistence libraries most widely used in Android applications (Mukherjee and Mondal, 2014). In particular, we study six widely used ORM persistence frameworks (Activeandroid by pardom, 0000; greenDAO, 0000; OrmLite, 0000; Sugar ORM, 0000; Sqlite database engine, 0000; DBFlow, 0000)), one OO persistence framework (Realm database engine, 0000), and one key-value database operation framework (Paper, 0000) as our experi-

* Corresponding author.
E-mail addresses: songz@vt.edu (Z.J. Song), pjing83@vt.edu (J. Pu), cjunjie@vt.edu (J. Cheng), tilevich@vt.edu (E. Tilevich).

https://doi.org/10.1016/j.jss.2018.08.038
0164-1212/© 2018 Elsevier Inc. All rights reserved.

mental targets. These frameworks operate on top of the popular SQLite, Realm, or NoSQL database engines.

In our experiments, we apply these eight persistence frameworks to different benchmarks, and then compare and contrast the resulting energy consumption, execution time, and programming effort (measured as lines of programmer-written code). Our experiments include a set of micro-benchmarks designed to measure the performance of individual database operations as well as the well-known DaCapo H2 database benchmark (Blackburn et al., 2006). To better understand the noticeable performance and programming effort disparities between persistence frameworks, we also introduce a numerical model that juxtaposes the performance and programming efforts of different persistence frameworks. By applying this model to the benchmarks, we generate several guidelines that can help developers choose the right persistence framework for a given application.

In other words, one key contribution of our study is informing Android developers about how they can choose a persistence framework that achieves the desired energy/execution time/programming effort balance. Depending on the amount of persistence functionality in an application, the choice of a persistence framework may dramatically impact the levels of energy consumption, execution time, and programming effort. By precisely measuring and thoroughly analyzing these characteristics of alternative Android persistence frameworks, this study aims at gaining a deeper understanding of the persistence's impact on the mobile software development ecosystem. The specific questions we want to answer are:

RQ1. How do popular Android persistence frameworks differ in terms of their respective features and capabilities?

RQ2. as it is realize What is the relationship between the persistence framework's features and capabilities and the resulting execution time, energy efficiency, and programming effort?

RQ3. How do the characteristics of an application's database functionality affect the performance of persistence frameworks?

RQ4. Which metrics should be measured to meaningfully assess the appropriateness of a persistence framework for a given mobile application scenario?

To answer **RQ1**, we analyze the documentation and implementation of the persistence frameworks under study to compare and contrast their features and capabilities. To answer **RQ2** and **RQ3**, we measure each of the energy, execution time, and programming effort metrics separately, compute their correlations, as well as analyze and interpret the results. To answer **RQ4**, we introduce a numerical model, apply it to the benchmarks used for the measurements, and generate several recommendation guidelines.

Based on our experimental results, the main contributions of this article are as follows:

1. To the best of our knowledge, this is the first study that empirically evaluates the energy, execution time, and programming effort trade-offs of popular Android persistence frameworks.
2. Our experiments consider multifaceted combinations of factors which may impact the energy consumption and execution time of persistence functionality in real-world applications, which include persistence operations involved, the volume of persisted data, and the number of transactions.
3. Based on our experimental results, we offer a series of guidelines for Android mobile developers to select the most appropriate persistence framework for their mobile applications. For example, ActiveAndroid or OrmLite fit well for applications processing relatively large data volumes in a read-write fashion. These guidelines can also help the framework developers to optimize their products for the mobile market.
4. We introduce a numerical model that can be applied to evaluate the fitness of a persistence framework for a given application scenario. Our model considers programming effort in addition to execution time and energy efficiency to provide insights relevant to software developers.

The rest of this paper is organized as follows. Section 2 summarizes the related prior work. Section 3 provides the background information for this research. Section 4 describes the design of our experimental study. Section 5 presents the study results and interprets our findings. Section 6 presents our numerical model and offers practical guidelines for Android developers. Section 7 discusses the threats to internal and external validity of our experimental results. Section 8 concludes this article.

This work extends our previous study of Android persistence frameworks, published in IEEE MASCOTS 2016 (Jing et al., 2016). This article is a revised and extended version of that paper. In particular, we describe the additional research we conducted, which now includes: (1) a comprehensive analysis of the studied frameworks' features, (2) the measurements and analysis for two additional persistence frameworks, (3) a study of the relationship between database-operations, execution time, and energy consumption, based on our measurements, and (4) a novel empirical model for selecting frameworks for a given set of development requirements.

2. Related work

This section discusses some prior approaches that have focused on understanding the execution time, energy efficiency, and programming effort factors as well as their interaction in mobile computing. Multiple prior studies have focused on understanding the energy consumption of different mobile apps and system calls, including approaches ranging from system level modeling of general systems (Tiwari et al., 1996) and mobile systems (Dong and Zhong, 2010), to application level analysis (Banerjee et al., 2014) and optimization (Kwon and Tilevich, 2013b). For example, Chowdhury et al. study the energy consumed by logging in Android apps (Chowdhury et al., 2017), as well as the energy consumed by HTTP/2 in mobile apps (Chowdhury et al., 2016). Liu et al. (2016) study the energy consumed by wake locks in popular Android apps. Many of these works make use of the Green Miner testbed (Hindle et al., 2014) to accurately measure the amount of consumed energy. As an alternative, the (Monsoon power monitor, 0000) is also used to measure the energy consumed by various Android APIs (Li et al., 2014; Linares-Vásquez et al., 2014). In our work, we have decided to use the Monsoon power meter, due to the tool's ease of deployment and operation.

Many studies also focus on the impact of software engineering practices on energy consumption. For example, (Sahin et al., 2012) study the relationship between design patterns and software energy consumption. Hindle et al. (2014) provide a methodology for measuring the impact of software changes on energy consumption. Hasan et al. (2016) provide a detailed profile of the energy consumed by common operations performed on the Java `List`, `Map`, and `Set` data structures to guide programmers in selecting the correct library classes for different application scenarios. Pinto et al. (2014) study how programmers treat the issue of energy consumption throughout the software engineering process.

The knowledge inferred from the aforementioned studies of energy consumption can be applied to optimize the energy usage of mobile apps, either by guiding the developer (Cohen et al., 2012; Pathak et al., 2012) or via automated optimization (Li et al., 2016). As discovered in Manotas et al. (2016), mobile app developers tend to be better tuned to the issues of energy consumption than developers in other domains, with a large portion of interviewed developers taking the issue of reducing energy consumptions into account during the development process.

Local databases are widely used for persisting data in Android apps (Lyu et al., 2017), and the corresponding database operation APIs are known to be as “energy-greedy”(Linares-Vásquez et al., 2014). Several persistence frameworks have been developed with the goal of alleviating the burden of writing SQL queries by hand. However, how these frameworks affect the energy consumption of Android apps has not been yet studied systematically. To bridge this knowledge gap, in this work, we study not only the performance of such frameworks, but also the programming effort they require, with the resulting knowledge base providing practical guidelines for mobile app developers, who need to decide which persistence framework should be used in a given application scenario.

3. Background

To set the context for our work, this section describes the persistence functionality, as it is commonly implemented by means of database engines and persistence frameworks.

The designers of the Android platform have recognized the importance of persistence by including the SQLite database module with the standard Android image as early as the release 1.5. Ever since this module has been used widely in Android applications. According to our analysis of the most popular 550 applications hosted on GooglePlay (25 most popular apps for each category, and 22 categories in all), over 400 of them (73%) involve interactions with the SQLite module.

The ORM (object-relational mapping) frameworks have been introduced and refined to facilitate the creation of database-oriented applications (Xia et al., 2009; Cvetković and Janković, 2010). The prior studies of the ORM frameworks have focused mainly on their *execution efficiency* and *energy efficiency*. Meanwhile, Vetro et al. (2013) show how various software development factors (e.g., design patterns, software architecture, information hiding, implementation of persistence layers, code obfuscation, refactoring, and data structure usage) can significantly influence the performance of a software system. In this article, we compare the performance of different persistence frameworks in a mobile execution environment with the goal of understanding the results from the perspective of mobile app developers.

Android persistence frameworks A persistence framework serves as a middleware layer that bridges the application logic with the database engine’s operations. The database engine maintains a schema in memory or on disk, and the framework provides a programming interface for the application to interact with the database engine.

As SQLite (the native database of Android) is a relational database engine, most Android persistence frameworks developed for SQLite are ORM/OO frameworks. One major function of such object-relational mapping (ORM) and object-oriented frameworks is to solve the *the object-relational impedance mismatch* (Subramanian et al., 1999) between the object-oriented programming model and the relational database operation model.

We evaluate 8 frameworks: Android SQLite, ActiveAndroid, greenDAO, OrmLite, Sugar ORM, DBFlow, Java Realm, and Paper, backed up by the SQLite, Realm, and NoSQL database engines, which are customized for mobile devices, with limited resources, including battery power, memory, and processor.

Persistence framework feature comparison Persistence frameworks differ in a variety of ways, including database engines, programming support (e.g., object and schema auto generation), programming abstractions (e.g., data access object (DAO) support, relationships, raw query interfaces, batch operations, complex updates, and aggregation operations), relational features support (e.g., key/index Structure, SQL join operations, etc.), and execution modes (e.g., transactions and caching). We focus on these features, as they may impact energy consumption, execution time, and programming effort. Table 1 compares these similarities and differences of the persistence frameworks used in our study.

1. *Database engine* Six of the studied persistence frameworks use SQLite (Newman, 2004), an ACID (Atomic, Consistent, Isolated, and Durable) and SQL standard-compliant relational database engine. Java Realm framework uses Realm (Realm database engine, 0000), an object-oriented database engine, whose design goal is to provide functionality equivalent to relational engines. Paper uses NoSQL (0000), a non-relational, schema-free, key-value data storage database. Therefore, as we here mainly compare features of relational database, Paper as a non-relational database engine, lacks many of these features.
2. *Object code generation* Some frameworks feature code generators, which relieve the developer from having to write by hand the classes that represent the relational schema in place.
3. *Schema generation* At the initialization stage, persistence frameworks employ different strategies to generate the database schema. Android SQLite requires raw SQL statements to create database tables, while OrmLite provides a special API call. greenDAO generates a special DAO class that includes the schema. The remaining frameworks (excluding Paper) automatically extract the table schema from the programmer defined entity classes.
4. *Data access method* DAO (Data Access Object) is a well-known abstraction strategy for database access that provides a unified object-oriented, entity-based persistence operation set (insert, update, delete, query, etc.). greenDAO, Sugar ORM, and OrmLite provide the DAO layer, while Android SQLite adopts a relational rather than DAO database manipulating interface. ActiveAndroid, DBFlow, and Java Realm provide a hybrid strategy—both DAO and SQL builder APIs.
5. *Relationship support* The three relationships between entities are one-to-one, one-to-many, and many-to-many. greenDAO and Java Realm support all three relationships. ActiveAndroid lacks support for Many-to-Many, while Sugar ORM and DBFlow only support One-To-Many. Android SQLite and OrmLite lack support for relationships, requiring the programmer to write explicit SQL join operations.
6. *Raw query interface support* Raw queries use naive SQL statements, thus deviating from pure object-orientation to execute complex database operations on multiple tables, nesting queries and aggregation functions. Android SQLite, greenDAO, OrmLite, and DBFlow—all provide this functionality.
7. *Batch operations* Batch operations commit several same-type database changes at once, thus improving performance. greenDAO, OrmLite, Sugar ORM, and DBFLow provide batch mechanisms for insert, update and delete. Java Realm provides batch inserts only, and the remaining two frameworks lack this functionality.
8. *Complex update support* Typically there are two kinds of database update operations: update columns to given values, or update columns based on arithmetic expressions. Android SQLite and ActiveAndroid can only use raw SQL manipulation interface to support expression updates. greenDAO, Sugar ORM, Java Realm, and DBFlow support complex updates via entity field modification. OrmLite is the only framework that provides both the value update and expression update abstractions.
9. *Aggregation support* Aggregating data in a relational database enables the statistical analysis over a set of records. Different frameworks selectively implement aggregation functionality. Android SQLite, OrmLite, and DBFlow support all of the aggregation functions via a raw SQL interface. Java Realm and

Table 1
Persistence framework feature comparison.

Features	Android SQLite	ActiveAndroid	greenDAO	OrmLite	Sugar ORM	Java Realm	DBFlow	Paper
Database Engine	SQLite	SQLite	SQLite	SQLite	SQLite	Realm	SQLite	NoSQL
Object Code Generation	✗	✗	√	✗	✗	✗	✗	N.A.
Schema Generation	manual	auto	auto	manual	auto	auto	auto	N.A
Data Access Method	✗	✗	√	√	✗	✗	√	N.A.
Relationship Support	✗	One-To-One,One-To-Many	One-To-One, One-To-Many, Many-to-Many	✗	One-To-Many	One-To-One, One-To-Many, Many-to-Many	One-To-Many	N.A.
Raw Query Interface Support	√	✗	√	√	✗	✗	√	N.A.
Batch Operations	✗	✗	Batch insert, batch update, batch delete	Batch insert, batch update, batch delete	Batch insert, batch update, batch delete	batch insert	batch insert, batch update, batch delete	N.A.
Complex Update Support	relational	relational	object	relational	object	object	object	N.A.
Aggregation Support	all	COUNT	COUNT	all	COUNT, FIRST, LAST	MAX, MIN, SUM, AVERAGE, FIRST, LAST	all	N.A.
Key/Index Structure	primary key, index, foreign key	integer single primary key, unique, index, foreign key	integer single primary key, unique, index	single primary key, index, foreign key	integer single primary key, unique	string or integer single primary key, index	primary key, index, foreign key, unique	N.A.
SQL JOIN Support	√	√	√	√	✗	✗	√	N.A.
Transaction Support	√	√	✗	√	✗	√	√	N.A.
Caching Support	✗	√	√	√	✗	✗	√	√

Sugar ORM provide an aggregation subset in the entity layer. ActiveAndroid and greenDAO support only the COUNT aggregation.

10. *Key/index structure* Key/Index structure identifies individual records, indicates table correlations, and increases the execution speed. Android SQLite and DBFlow fully support the database constraints—single or multiple primary keys (PK), index and foreign key (FK). ActiveAndroid supports integer single PK, unique, index, and FK. greenDAO supports integer single PK, unique, index. OrmLite supports single PK, index, and FK. Sugar ORM supports integer single PK and unique. Java Realm supports string or integer single PK, and index.
11. *SQL JOIN support* SQL JOIN clause combines data from two or more relational tables. Android SQLite and DBFlow only support raw JOIN SQL. ActiveAndroid incorporates JOIN in its object query interface. DAOs of greenDAO and OrmLite provide the JOIN operation. Sugar ORM and Java Realm lack this support.
12. *Transaction support* Transactions perform a sequence of operations as a single logical execution unit. All the studied engines with the exception of greenDAO, Sugar ORM, and Paper provide full transactional support.
13. *Cache support* OrmLite, ActiveAndroid, greenDAO, DBFlow and Paper support caching. They provide this advanced feature to maintain persisted entities in memory to speed-up future accesses, at the cost of extra processing required to initialize the cache pool.

4. Experiment design

In this section, we explain the main decisions we have made to design our experiments. In particular, we discuss the benchmarks, the measurement variables, and the experimental parameters.

4.1. Benchmark selection

DaCapo H2 (Blackburn et al., 2006) is a well-known Java database benchmark that interacts with the H2 Database Engine via JDBC. This benchmark manipulates a considerable volume of data to emulate bank transactions. The benchmark includes (1) a complex schema and non-trivial functionality, obtained from a real-world production environment. The database structure is complex (12 tables, with 120 table columns and 11 relationship between tables), while the database operations simulate the running of heavy-workload database-oriented applications; (2) complex database operations that require: batching, aggregations, and transactions.

Since DaCapo relies heavily on relational data structures and operations, we replace H2 with two relational database engines, SQLite or Realm, to adapt this benchmark for Android. In other words, we evaluate the performance of all persistence frameworks under the DaCapo benchmark except for Paper, which is a non-relational database engine.

However, using the DaCapo benchmark alone cannot provide performance evaluation of persistence frameworks under the low data volumes with simple schema conditions. To establish a baseline for our evaluation, we thus designed a set of micro benchmarks, referred to as *the Android ORM Benchmark*, which features a simple database schema with few data records. Specifically, this benchmark's database structure includes 2 tables comprising 11 table columns, and a varying small number of data records. Besides, this micro-benchmark comprises the fundamental database operation invocations “create table”, “insert”, “delete”, “select”, and “update”. As the database operations in many mobile applications tend to be rather simple, the micro-benchmark's results present valuable insights for application developers.

Note that database operations differ from database operation invocations. The invocations refer to calling the interfaces provided by the persistence framework (e.g., “insert”, “select”, “update” and “delete”). However, each invocation can result in multiple database operations (e.g., `android... SQLiteStatement.executeInsert()`).

4.2. Test suite implementation

Our experimental setup comprises a mobile app that uses the selected benchmarked frameworks to execute both DaCapo and the Android ORM Benchmark.[1] Through this app, experimenters can select benchmarks, parameterize the operation and data volume, as well as select ORM frameworks. The implementation of the test suite was directed by two graduate students, each of whom has more than three years of experience in developing commercial database-driven projects.

As stated above, the DaCapo database benchmark is designed for relational databases, so it would be non-trivial to reimplement it using Paper, which is based on a non-relational database engine (NoSQL). Therefore, the DaCapo benchmark is only applied to seven persistence frameworks, while the Android ORM Benchmark is applied to all eight persistence frameworks.

For each benchmark, the transaction logic (e.g., creating bank accounts for DaCapo) is implemented using various ORM frameworks, following the design guidelines of these frameworks. For example, greenDAO, Sugar ORM, and OrmLite provide object-oriented data access methods, so their benchmarks' data operations are implemented by using DAO. On the contrary, Android SQLite adopts a relational rather than DAO database manipulating interface, so its benchmark's data operations are implemented by using SQL builders. ActiveAndroid, DBFlow, and Java Realm provide a hybrid strategy—both DAO and SQL builder APIs. For these frameworks, if the benchmark's operation is impossible or non-trivial to express using DAO, the SQL builder API is used instead.

4.3. Parameters and variables

Next, we explain the variables used to evaluate the execution time, energy consumption, and programming effort of the studied persistence frameworks. We also describe how these variables are obtained.

- *Overall execution time* The overall execution time is the time elapsed from the point when a database transaction is triggered to the point when it completes.
- *Read/write database operation number* We focus on comparing the Read/Write numbers only for SQLite-based frameworks (ActiveAndroid, greenDAO, OrmLite, Sugar ORM, and DBFlow), as such frameworks use SQLite operation interfaces provided by the Android framework to operate on SQLite database. The write operations include executing SQL statements that are used to “insert”, “delete”, and “update”, while the read operations include only “select”. When performing the same combination of transactions, the differences in Read/Write number is the output of how different persistence frameworks interpret database operation invocations. The read/write ratio can also impact the energy consumption. The operation numbers are obtained by hooking into the SQLite operation interfaces provided by the Android System Library. For those interfaces provided for a certain type of database operation, we mark them as “Read” or “Write”; for those interfaces provided for general SQL execution, we search for certain keywords (e.g., insert, update, select, and delete), in the SQL strings, and further mark them as “Read” or “Write”.
- *Energy consumption* The energy consumption can be calculated using the real-time current of the Android device's battery. We use the Monsoon power monitor (0000) to monitor the current and voltage of the device's battery, as shown in Fig. 1. As the output voltage of a smartphone's battery remains stable, only the current and time are required to calculate the energy consumed. Eq. (1) is used to calculate the overall energy consumption, where $\bar{I}$ is the average current (mA), and t is the time window (ms). Micro-ampere-hour is a unit of electric charge, commonly used to measure the capacity of electric batteries.

$$E = \frac{\bar{I} * t}{3600s/hour} \tag{1}$$

  The equation shows that the energy consumption is proportional to the execution time, as well as to the current required by the device's hardware (e.g., CPU, memory, network, and hard disk). For the persistence frameworks, the differences of energy consumption reflect not only the execution time differences, but also the different CPU workload and hard disk R/W operations required to process database operations.
- *Uncommented line of codes (ULOC)* ULOC reflects the required effort, defined as the amount of code a programmer has to write to use a persistence framework. For its simplicity, ULOC was used to express programming effort in earlier studies (e.g., Lavazza et al., 2016). As all the test suites are implemented only in Java and SQL, the ULOC metric is reflective of the relative programming effort incurred by using a framework.

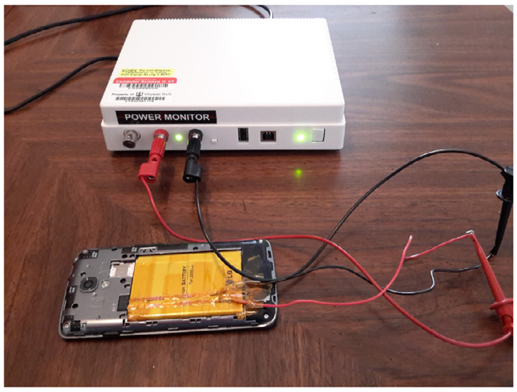


Fig. 1. Monsoon mobile device power monitor's main channel measurement connection.

Next, we introduce the input parameters for different benchmarks. For the DaCapo benchmark, we want to explore the performance boundary of different persistence frameworks under a heavy workload. Therefore, we vary the amount of total transactions to a large scale, and record the overall time taken and energy consumed.

For the micro benchmark, we study the “initialize”, “insert”, “select”, “update”, and “delete” invocations in turn. We change the number of transactions for the last four invocations, so for the “select”, “update” and “delete” invocations, the amount of data records also changes. Therefore, the input parameters for the micro benchmark is a set of two parameters: {NUMBER OF TRANSACTIONS, AMOUNT OF DATA RECORDS}.

4.4. Experimental hardware and tools

To measure the energy consumed by a device, its battery must be removed to connect to the power meter's hardware. Unfortunately, a common trend in the design of Android smartphones

[1] All the code used in our experiments can be downloaded from https://github.com/AmberPoo1/PEPBench.

makes removable batteries a rare exception. The most recently made device we have available for our experiments is an LG LS740 smartphone, with 1GB of RAM, 8GB of ROM and 1.2GHz quad-core Qualcomm Snapdragon 400 processor (Singh and Jain, 2014), running Android 4.4.2 KitKat operating system. Although the device was released in 2014 and runs at a lower CPU frequency than the devices in common use today, the Android hardware design, at least with respect to the performance parameters measured, has not experienced a major transformation, thus not compromising the relevance of our findings.

We execute all experiments as the only load on the device's OS. To minimize the interference from other major sources of energy consumption, we set the screen brightness to the minimal level and turn off the WiFi/GPS/Data modules. As the CPU frequency takes time to normalize once the device exits the sleep mode, we run each benchmark 5 times within the same environment, with the first two runs to warm up the system and wait until the background energy consumption rate stabilizes. The reported data is calculated as the average of the last 3 runs, with the differences of the three runs of the same benchmark being not larger than 5%.

To understand the Dalvik VM method invocations, we use Traceview, an Android profiler that makes it possible to explore the impact of programming abstractions on the overall performance. Unfortunately, only the Android ORM benchmark is suitable for this exploration, due to the Traceview scalability limitations.

5. Study results

In this section, we report and analyze our experimental results.

5.1. Experiments with the android ORM benchmark

In this group of experiments, we study how the types of operation (insert, update, select, and delete) and the variations on the number of transactions impact energy consumption and execution time with different frameworks using the micro benchmark[2] The experimental results for each type of persistence operation are presented in Figs. 2 and 3. The first row of Fig. 2(a) and (b) shows the energy consumption and execution time of the "insert" database invocation, and Fig. 2(c)(d), (e)(f), and (g)(h) show that of the "update", "select", and "delete" database invocations, respectively. Fig. 3(a)–(d) show the database read and write operations of these four invocations, respectively.

The results show that the persistence frameworks differ in terms of their respective energy consumption, execution time, read, and write measurements. Next, we compare the results by operation:

Insert We observe that DBFlow takes the longest time to perform the insert operation, while ActiveAndroid takes the second longest, with the remaining frameworks showing comparable performance levels. DBFlow performs the highest number of database operations, a measurement that explains its long execution time. Different from other frameworks, DBFlow requires that a database read operation be performed before a database write operation to ensure that the-record-to-insert has not been already inserted into the database. Besides, the runtime trace reveals that interactions with the cache triggered by inserts in ActiveAndroid are expensive, costing 62% of the overall execution time. By contrast, greenDAO exhibits the shortest execution time, due to its simple but efficient batch insert encapsulation, as shown in Table 1.

Update We observe that the cost of the Java Realm update is several orders of magnitude larger than that of other frameworks, especially as the number of transactions grows. Several reasons can explain the high performance costs of the update operation in Java Realm. As one can see in Table 1, Java Realm lacks support for batch updates. Besides, the update procedure invokes the underlying library method, `TableView.size()`, which operates on a memory-hosted list of entities and costs more than 98% of the overall execution time. The execution time of Sugar ORM is also high, due to it having the highest number of read and write operations. Sugar ORM needs to search for the target object before updating it. This search procedure is designed as the recursive `SugarRecord.find()` method, which costs 96% of the overall execution time.

Select and delete For the select and delete operations, we observe that Sugar ORM (1) exhibits the worst performance in terms of execution time and energy consumption; (2) performs the highest number of database operations, as it executes an extra query for each atomic operation. The inefficiency of the select and delete operations in Sugar ORM stems from the presence of these extra underlying operations. However, as discussed above, the bulk of the execution time is spent in the recursive *find* method. OrmLite, greenDAO, DBFlow, Paper, and Android SQLite show comparable performance levels when executing these two operations.

Table 2 sums up the rankings of each persistence framework w.r.t. different database operation invocations. We also measure the Uncommented Lines of Code (ULOC) for implementing all the basic database operation invocations for each persistence framework and include this metric in the table. From Table 2 and our analysis above, we can draw the following conclusions:

1. By adding up the rankings of different operations, we can rank these frameworks in terms of their overall performance: Android SQLite $>$ greenDAO $=$ DBFlow $>$ OrmLite $>$ Java Realm $>$ Paper $>$ Sugar ORM $>$ ActiveAndroid , where "$>$" means "having better performance than", and "$=$" means "having similar performance with".
2. Considering the programming effort of implementing all database operations using different frameworks, DBFlow and Paper require less programming effort than the other frameworks.
3. When considering the balance of programming effort and performance, DBFlow can be generally recommended for developing database-oriented mobile application with a standard database operation/schema complexity.
4. Sugar ORM would not be an optimal choice when the dominating operations in a mobile app are select or delete, DBFlow would not be optimal when the dominating operation is insert, while Java Realm would not be optimal when the dominating operation is update.

5.2. Experiments with the dacapo benchmark

In this group of experiments, we use the DaCapo benchmark to study how the energy consumption and execution time of each framework changes in relation to the number of executed bank transactions. The benchmark comes with a total of 41,971 records, so in our experiments we differ the number of bank transactions. In our measurements, we vary the number of bank transactions over the following values: 40, 120, 200, 280, 360, 440, 520, 600, 800, 1000, 1500. The total number of transactions is the sum of basic bank transactions, as listed in Table 3. Each transaction comprises a complex set of database operations. The key transactions in each run are "New Order", "Payment by Name", and "Payment by ID", which mainly execute the "query" and "update" operations. In our experiments, "New Order" itself takes 42.5% of the entire number of transactions.

[2] We use the terms *the micro benchmark* and *the Android ORM benchmark* interchangeably in the rest of the presentation.

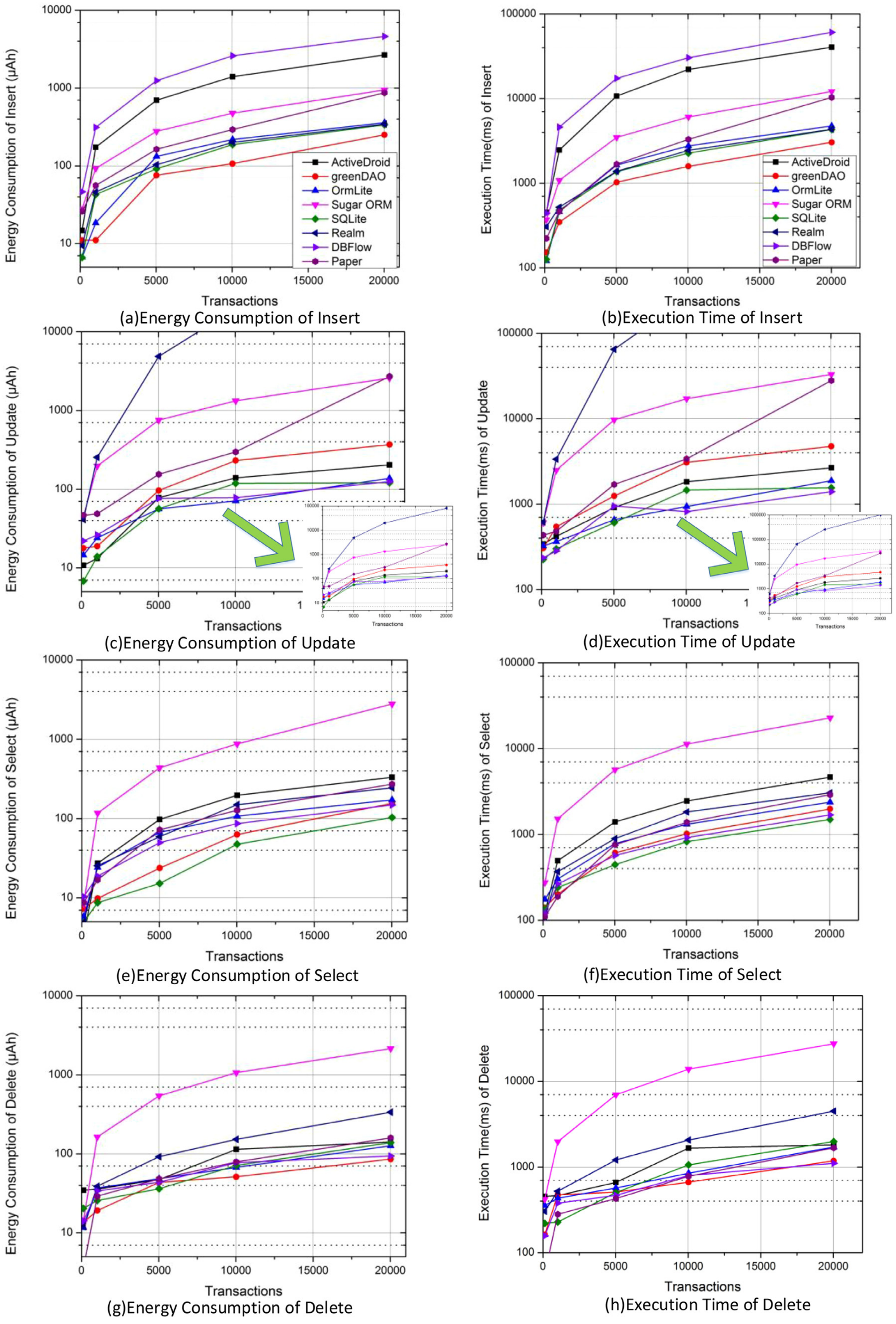


Fig. 2. Energy/execution time for android ORM benchmark with alternative persistence frameworks.

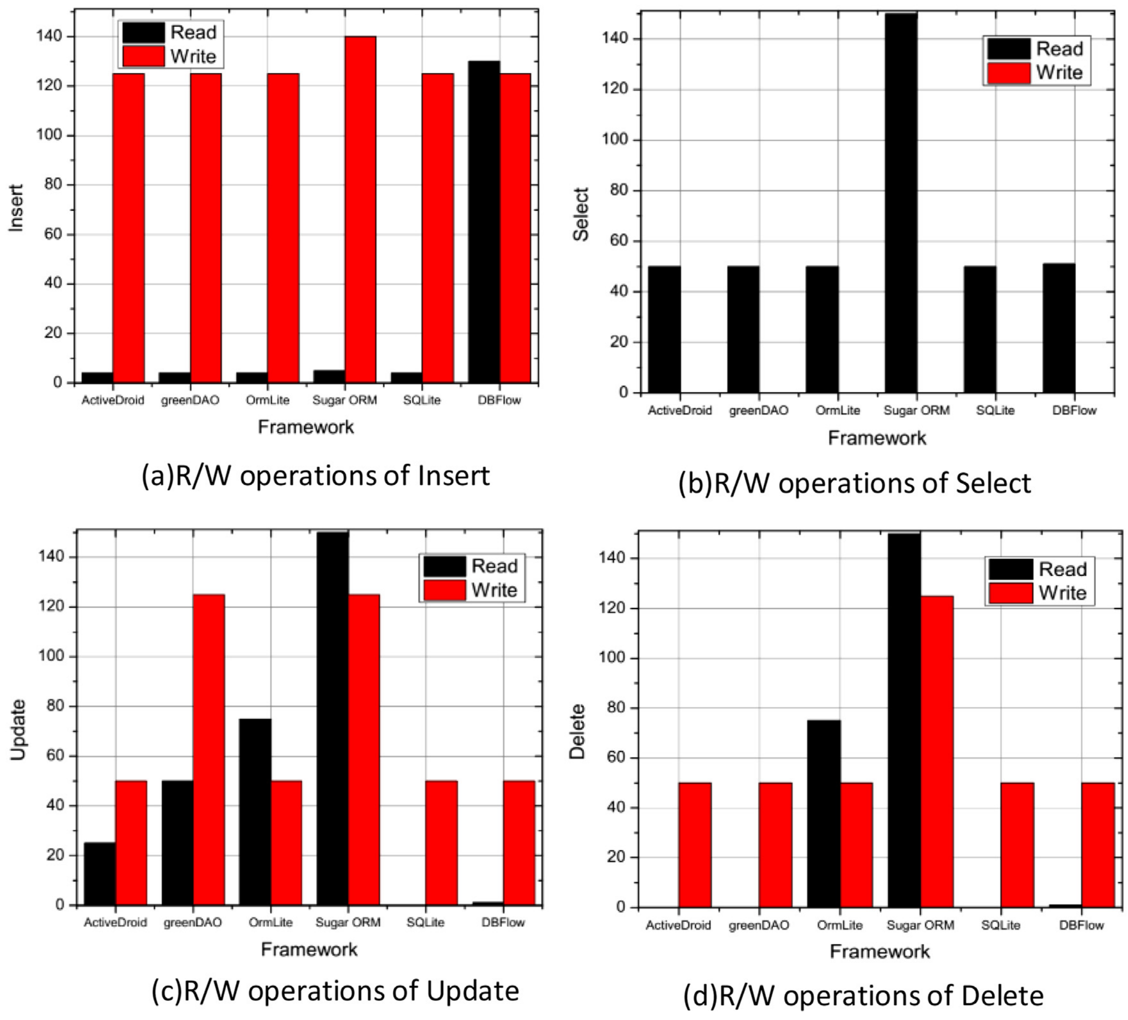


Fig. 3. Read/write operations for android ORM benchmarks with alternative persistence frameworks.

Table 2
Comparison of persistence frameworks in the android ORM experiment.

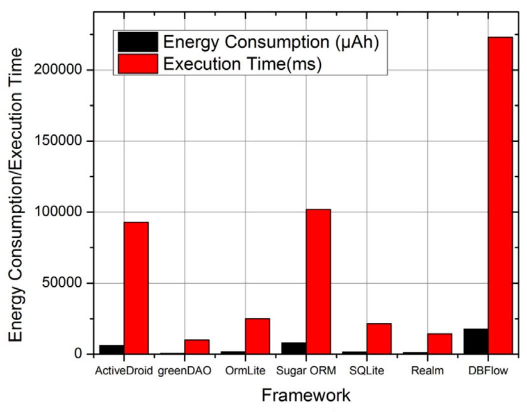
Compared Item	SQLite	DBFlow	greenDAO	ORMLite	Realm	Paper	Sugar	ActiveAndroid
ULOC	306	181	241	326	313	190	226	253
Initialization ranking	6	1	5	7	3	4	2	8
Insert ranking	2	8	1	4	3	5	6	7
Update ranking	1	2	5	3	8	7	6	4
Select ranking	1	2	3	4	5	6	8	7
Delete ranking	4	2	1	3	7	6	8	5
Summed up ranking	14	15	15	21	26	28	30	31

Table 3
Number of operations for each transaction type with 1500 overall transactions.

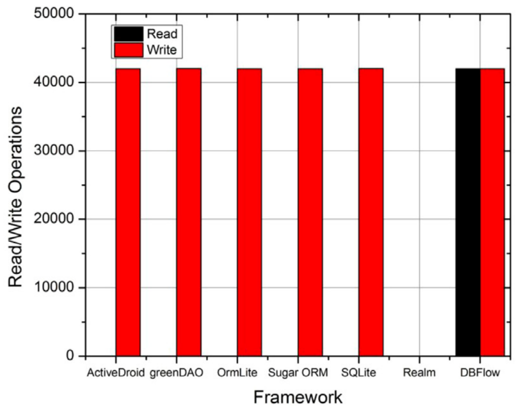
Transaction type	Operation amount
New order rollback	8
Order status by ID	26
Order status by name	42
Stock level	59
Delivery schedule	62
Payment by ID	245
Payment by name	391
New order	667

In Fig. 4, (a) and (b) show the energy/execution time and read/write operations of the DaCapo Initialization, respectively. Fig. 4(c) shows the energy consumption for each transaction number, and Fig. 4(d) shows the execution time for each transaction number. Fig. 4(e) and (f) show the read/write operation number, respectively. As the write operation number of ActiveDroid, greenDao, OrmLite, Sugar ORM and SQLite are very close (e.g, when the transaction amount is 1500, the number of write operation are 18209, 18200, 18205, 18211, 18212 respectively), we only present the average operation number of these five frameworks in Fig. 4(e). Similarly, we use the line in pink to present Sugar ORM/greenDao, and the line in green for SQLite/ActiveDroid in Fig. 4(f). Table 3 shows the number of operations performed by each transaction.

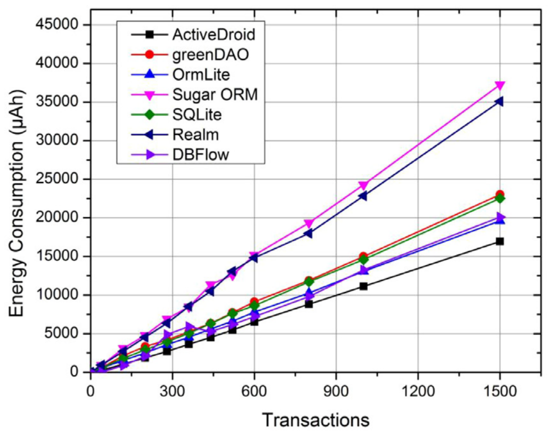
The dominant database operation in the initialization phase is insert, and (a) shows the performance levels consistent with those seen in the Android ORM benchmark for the same opera-



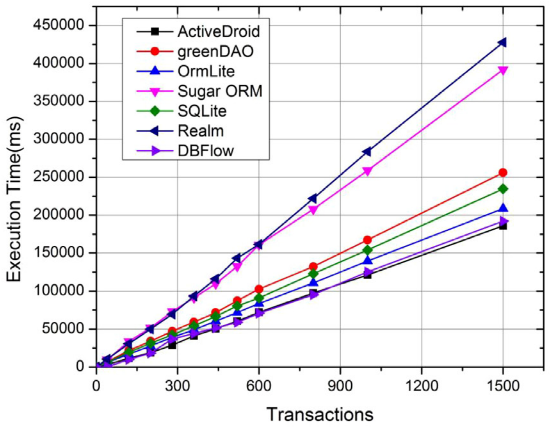
(a) Energy Consumption/Execution Time of DaCapo Initialization



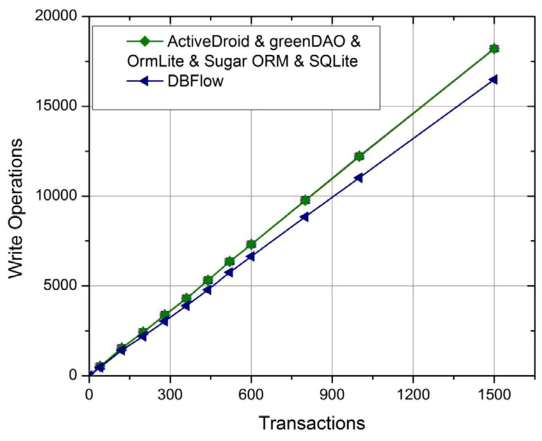
(b) R/W Operations of DaCapo Initialization



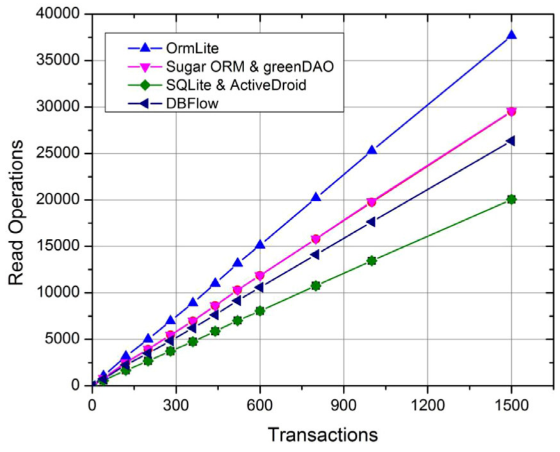
(c) Energy Consumption of DaCapo Transactions



(d) Execution Time of DaCapo Transactions



(e) Write Operations of DaCapo Transactions



(f) Read Operations of DaCapo Transactions

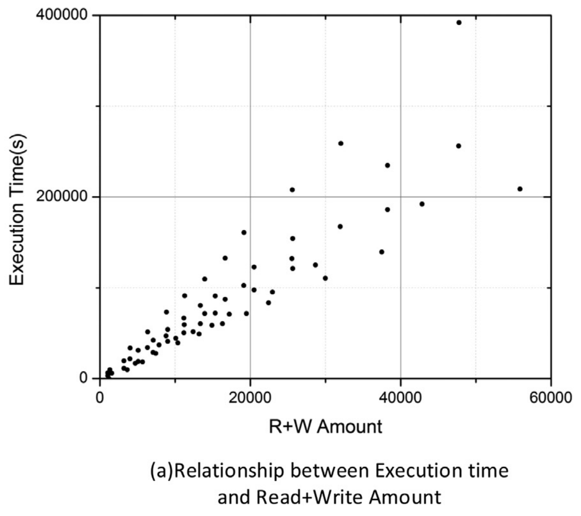
Fig. 4. Energy/execution time/read and write for DaCapo benchmark with alternative persistence frameworks. (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.)

tion: DBFlow, Sugar ORM and ActiveAndroid have the longest runtime. greenDAO performs better than Android SQLite, possibly due to greenDAO supporting batch insert (see Table 1 for details).

From Fig. 4, we observe that Java Realm and Sugar ORM have the longest execution time when executing the transactions whose major database operation is update (e.g., "New order", "New order rollback", "Payment by name", and "Payment by ID"). This conclusion is consistent with that derived from the Android ORM update experiments shown in Section 5.1. Android SQLite takes rather long to execute, as it involves database aggregation (e.g., *sum*, and the table queried had 30,060 records) and arithmetic operations (e.g. $field - 1$) in the select clause. Meanwhile, as ActiveAndroid only uses the raw SQL manipulation interface for complex update operations (Table 1), it exhibits the best performance, albeit at the cost of additional programming effort.

From Fig. 4(c)–(f) and Table 4 we conclude that:

1. ActiveAndroid offers the overall best performance for all DaCapo transactions. It shows the best performance for the most common transactions, at the cost of additional programming



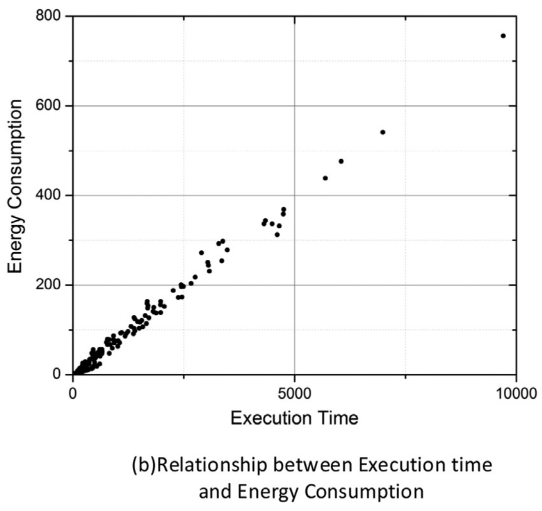


Fig. 5. Energy/execution time/RW amount relationships.

Table 4
LOC for DaCapo benchmark.

DaCapo	ULOC
greenDAO	2200
DBFlow	2407
Sugar ORM	2911
ActiveAndroid	2923
Android SQLite	3068
Java Realm	3071
OrmLite	3310

effort. Besides, its execution invokes the smallest number of database operations, due to its caching mechanism.

2. Sugar ORM and Java Realm have the longest execution time, in line with the Android ORM benchmark's results discussed in Section 5.1.
3. greenDAO's performance is in the middle, while requiring the lowest programming effort, taking 24.5% fewer uncommented lines of code to implement than the other frameworks.
4. DBFlow takes more time and energy to execute that does ActiveAndroid; it also requires a higher programming effort than greenDAO does. Nevertheless, it strikes a good balance between the required programming effort and the resulting execution efficiency.

5.3. Relationship of energy consumption, execution time and DB operations

We also discuss the relationship between energy consumption, execution time, and database operations. We combine all the previous collected read/write, execution time and energy consumption data from all the benchmarks. As shown in Fig. 5(a), the number of total database operations (Read and Write) has a dominating impact on both the execution time and energy consumption results: (1) as the number of data operations increases so usually do the execution time and energy consumption; (2) the execution time and energy consumption are also impacted by other factors (e.g., the complexity of data operations, the framework's implementation design choices, etc.).

Meanwhile, as shown in Fig 5(b), there is a significant positive relationship between the time consumption and the energy consumption, with r(142) = 0.99, p < 0.001. Hence, the longer a database task executes, the more energy it ends up consuming.

6. Numerical model

The experiments above show that for the same application scenario different frameworks exhibit different execution time, energy consumption, and programming effort. However, to derive practical benefits from these insights, mobile app developers need a way to quantify these trade-offs for any combination of an application scenario and a persistence framework. To that end, we propose a numerical model PEP (**P**erformance, **E**nergy consumption, and **P**rogramming **E**ffort) for mobile app developers to systematically evaluate the suitability of persistence frameworks.

6.1. Design of the PEP model

Our numerical model follows a commonly used technique for evaluating software products called *a utility index* (Christodoulou et al., 2009). Products with equivalent functionality possess multidimensional feature-sets, and a utility index is a score that quantifies the overall performance of each product, allowing the products to be compared with each other.

As our experiments show, the utility of persistence frameworks is closely related to application features (e.g., data schema complexity, operations involved, data records manipulated, database operations executed). Therefore, it is only meaningful to compare the persistence frameworks within the context of a certain application scenario. Here, we use p to denote an application with a set of features.

Let $\mathcal{O} = \{o = 1, 2, 3\ldots\}$ be a set of frameworks. Let $E_o(p), \forall o \in \mathcal{O}$ denote the energy consumption of different implementations of p using various frameworks o, while $T_o(p), \forall o \in \mathcal{O}$ denotes the execution time. As the energy consumption and the execution time of database operations correlate linearly in our experimental results, we use the Euclidean distance of a two dimensional vector to calculate the overall performance, which can be denoted as $P_o p = \sqrt{T_o(p)^2 + E_o(p)^2}, \forall o \in \mathcal{O}$.

The programming effort is represented by the ULOC, and here we use $L_o(p), \forall o \in \mathcal{O}$ to denote the programming effort of different implementations of the project p using different persistence frameworks.

We consider both the framework's performance and programming effort to compute the utility index $I_o(p)$:

$$I_o(p) = \frac{\min(P_o(p), \forall o \in \mathcal{O})}{P_o(p)} / \frac{L_o(p)}{\min(L_o(p), \forall o \in \mathcal{O})}, \forall o \in \mathcal{O} \quad (2)$$

Table 5
Applying numerical model on DaCapo and androidORM benchmark.

Benchmark	Parameters & Index	ActiveAndroid	greenDAO	OrmLite	Sugar	SQLite	Realm	DBFlow	Paper
DaCapo	Trans=40,τ=0.5	0.328	0.184	0.135	0.166	0.109	0.098	**0.956**	N.A.
DaCapo	Trans=40,τ=1	0.284	0.184	0.135	0.094	0.140	0.083	**0.914**	N.A.
DaCapo	Trans=40,τ=1.5	0.247	0.184	0110	0.082	0.118	0.070	**0.878**	N.A.
DaCapo	Trans=1500,τ=0.5	**0.867**	0.726	0.726	0.412	0.671	0.368	**0.924**	N.A.
DaCapo	Trans=1500,τ=1	0.752	0.726	0.592	0.358	0.568	0.312	**0.884**	N.A.
DaCapo	Trans=1500,τ=1.5	0.652	0.726	0.483	0.133	0.481	0.264	**0.845**	N.A
Android ORM	Trans=1025,τ=0.5	0.050	0.166	0.124	0.044	0.180	0.055	0.255	**0.976**
Android ORM	Trans=1025,τ=1	0.051	0.144	0.092	0.039	0.139	0.042	0.255	**0.952**
Android ORM	Trans=1025,τ=1.5	0.043	0.125	0.069	0.035	0.107	0.031	0.255	**0.929**
Android ORM	Trans=20025,τ=0.5	0.159	**0.740**	0.633	0.289	**0.769**	0.007	0.591	0.635
Android ORM	Trans=20025,τ=1	0.165	**0.641**	0.472	0.080	0.591	0.005	0.591	0.621
Android ORM	Trans=20025,τ=1.5	0.114	0.555	0.352	0.071	0.454	0.004	0.591	**0.606**

The equation's first part, $\frac{\min(P_o(p),\forall o\in\mathcal{O})}{P_o(p)}$, compares the performance of a mobile app implemented by means of the persistence framework o, and the implementation that has the best performance. The equation's second part, $\frac{L_o(p)}{\min(L_o(p),\forall o\in\mathcal{O})}$, compares the programming effort between an implementation $o(p)$ and the implementation that requires the minimal programming effort. When the utility index of a framework o-based implementation is close to 1, the implementation is likely to offer acceptable performance, with low programming effort.

Consider the following example that demonstrates how to calculate the utility index. If for an app p, Android SQLite might provide the best performance, while the greenDAO-based implementation consumes twice the energy and takes twice the execution time. Therefore, the performance index of $P_{greenDAO}(p)$ is 0.5. On the other hand, the greenDAO-based implementation might require the lowest programming effort, as measured by the ULOC metric. Therefore, the implementation complexity index of $L_{greenDAO}p$ is 1. Thus, the overall utility index is $0.5/1 = 0.5$.

Application developers apply dissimilar standards to judge the trade-offs between performance and programming effort. Some developers focus solely on performance, while others may prefer the shortest time-to-market. We introduce τ to express these preferences.

$$I_o(p) = \frac{\min(P_o(p), \forall o \in \mathcal{O})}{P_o(p)} / \left(\frac{L_o(p)}{\min(L_o(p), \forall o \in \mathcal{O})} \right)^{\tau} \qquad (3)$$

where $\tau > 0$. When $\tau > 1$, the larger τ is, the more weight is assigned to the programming effort target. Otherwise, when $\tau < 1$, the lower τ is, the more weight is assigned to the performance target.

6.2. Evaluating the benchmarks

To provide an insight into how the persistence frameworks evaluated in this article fit different application scenarios, we apply the PEP model to the Android ORM and DaCapo benchmarks. We consider typical low and high transaction volumes, respectively, for each benchmark. Specifically, for the Android ORM benchmark, we evaluate two sets of input, 1025 transactions and 20,025 transactions. For the DaCapo benchmark, we evaluate two sets of input, 40 transactions and 1500 transactions. For each input set, we assign τ to 0.5, 1, and 1.5, in turn, to show whether the developers are willing to invest extra effort to improve performance. Specifically, when $\tau = 0.5$, the developer's main concern is performance; when $\tau = 1$, the balance of performance and programming effort is desired; when $\tau = 1.5$, the developer wishes to minimize programming effort. Table 5 shows the calculated index values of the persistence frameworks for all cases.

From the results presented in Table 5, we can draw several conclusions: (1) For the DaCapo benchmark or similar mobile apps with heavy data processing functionality and complicated data structures, DBFlow represents the best performance/programming effort trade-off. When the number of data operations is very high, ActiveAndroid should also be considered, as it provides the best execution performance, especially when the programmer is not as concerned about minimizing the programming effort; (2) For the Android ORM benchmark or similar mobile apps with less complicated data structures, when the number of data operations is small, the top choice is Paper, as this framework reduces the programming effort while providing high execution efficiency. However, when the number of data operations surpasses a certain threshold (over 10K simple data operations), the execution performance of Paper experiences a sudden drop due to scalability issues. In such cases, greenDAO and Android SQLite would be the recommended options.

In the following discussion, we use hypothetical use cases to demonstrate how the generated guidelines can help mobile app developers pick the best framework for the application scenario in hand. Consider four cases: (1) a developer wants to persist the user's application-specific color scheme preferences; (2) a team of developers wants to develop a production-level contact book application; (3) an off-line map navigation application needs to store hundreds of MBs of data, comprising map fragments, points of interests, and navigation routines; (4) an MP3 player app needs to retrieve the artist's information based on some features of the MP3 being played. For use case 1, the main focus during the development procedure is to lower the programming effort, while the data structures and the number of data operations are simple and small. Therefore, for this use case, we would recommend using Paper. For use case 2, the main focus is to improve the responsiveness and efficiency, as the potential data volume can get quite large. Given that minimizing the programming effort is deprioritized, we would recommend using greenDAO or Android SQLite. For use case 3, complex data structures are required to be able to handle the potentially large data volumes, while maintaining quick responsiveness and high efficiency is expected of navigation apps. Therefore, we would recommend using ActiveAndroid. For use case 4, the main application's feature is playing MP3s, and the ability to retrieve the artist's data instantaneously is non-essential. To save the programming effort of this somewhat auxiliary feature, we would recommend using DBFlow.

7. Threats to validity

Next, we discuss the threats to the validity of our experimental results. Although in designing our experimental evaluation, we tried to perform as an objective assessment as possible, our design choices could have certainly affected the validity and applicability of our conclusions.

The key external threat to validity is our choice of the hardware devices, Android version, and profiling equipment. Specifically, we conduct our experiments with an LG mobile phone, with 1.2 GHz quad-core Qualcomm Snapdragon 400 processor, running Android 4.4.2 KitKat, profiled with the Monsoon Power Monitor. Even though these experimental parameters are representative of the Android computing ecosystem, changing any of these parameters could have affected some outcomes of our experiments.

The key internal threat to validity are our design choices for structuring the database and the persistence application functionality. Specifically, while our Android ORM benchmark set explores the object features of Android persistence frameworks, the original DaCapo (Blackburn et al., 2006) H2 benchmark manipulates relational database structures directly, without stress-testing the object-oriented persistence frameworks around it. To retarget DaCapo to focus on persistence frameworks rather than the JVM alone, we adapted the benchmark to make use of transparent persistence as a means of accessing its database-related functionality. Nevertheless, the relatively large scale of data volume, with the select and update operations bank transactions dominating the execution, this benchmark is representative of a large class of database application systems, but not all of them. Besides, we have not tested our PEP model on real-world applications. Hence, it is not confirmed yet how accurate the model would be for such applications.

8. Conclusions

In this paper, we present a systematic study of popular Android ORM/OO persistence frameworks. We first compare and contrast the frameworks to present an overview of their features and capabilities. Then we present our experimental design of two sets of benchmarks, used to explore the execution time, energy consumption, and programming effort of these frameworks in different application scenarios. We analyze our experimental results in the context of the analyzed frameworks' features and capabilities. Finally, we propose a numerical model to help guide mobile developers in their decision making process when choosing a persistence framework for a given application scenario. To the best of our knowledge, this research is the first step to better understand the trade-offs between the execution time, energy efficiency, and programming effort of Android persistence frameworks. As a future work direction, we plan to apply the PEP model presented above to real-world applications, in order to assess its accuracy and applicability.

Acknowledgment

This research is supported by the National Science Foundation through the Grants CCF-1649583 and CCF-1717065. The authors would like to thank the anonymous reviewers, whose insightful comments helped improve this article.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.jss.2018.08.038.

References

Activeandroid by pardom. Last accessed data: 19-June-2018 http://www.activeandroid.com.

Banerjee, A., Chong, L.K., Chattopadhyay, S., Roychoudhury, A., 2014. Detecting energy bugs and hotspots in mobile apps. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, pp. 588–598.

Blackburn, S.M., Garner, R., Hoffmann, C., Khang, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., et al., 2006. The DaCapo benchmarks: java benchmarking development and analysis. In: Proceedings of the ACM Sigplan Notices, 41. ACM, pp. 169–190.

Chowdhury, S., Di Nardo, S., Hindle, A., Jiang, Z.M.J., 2017. An exploratory study on assessing the energy impact of logging on android applications. Empir. Softw. Eng. 1–35.

Chowdhury, S.A., Sapra, V., Hindle, A., 2016. Client-side energy efficiency of HTTP/2 for web and mobile app developers. In: 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE, 1. IEEE, pp. 529–540.

Christodoulou, S.E., Ellinas, G., Michaelidou-Kamenou, A., 2009. Minimum moment method for resource leveling using entropy maximization. J. Constr. Eng. Manag. 136 (5), 518–527.

Cohen, M., Zhu, H.S., Senem, E.E., Liu, Y.D., 2012. Energy types. In: Proceedings of the ACM SIGPLAN Notices, 47. ACM, pp. 831–850.

Cvetković, S., Janković, D., 2010. A comparative study of the features and performance of ORM tools in a NET environment. In: Objects and Databases. Springer, pp. 147–158.

DBFlow A blazing fast, powerful, and very simple ORM Android database library that writes database code for you. Last accessed data: 19-June-2018 https://github.com/Raizlabs/DBFlow.

Dong, M., Zhong, L., 2013. Sesame: self-constructive system energy modeling for battery-powered mobile systems. In: Proceedings of the 9th international conference on Mobile systems, applications, and services. ACM, 2011.

greenDAO: the superfast android orm for sqlite Last accessed data: 19-June-2018 http://greenrobot.org/greendao/.

Hao, S., Li, D., Halfond, W.G., Govindan, R., 2013. Estimating mobile application energy consumption using program analysis. In: Proceedings of the 35th International Conference on Software Engineering (ICSE), 2013. IEEE, pp. 92–101.

Hasan, S., King, Z., Hafiz, M., Sayagh, M., Adams, B., Hindle, A., 2016. Energy profiles of Java collections classes. In: Proceedings of the 38th International Conference on Software Engineering (ICSE), 2016 IEEE/ACM. IEEE, pp. 225–236.

Hindle, A., 2012. Green mining: a methodology of relating software change to power consumption. In: Proceedings of the 9th IEEE Working Conference on Mining Software Repositories. IEEE Press, pp. 78–87.

Hindle, A., Wilson, A., Rasmussen, K., Barlow, E.J., Campbell, J.C., Romansky, S., 2014. Greenminer: a hardware based mining software repositories software energy consumption framework. In: Proceedings of the 11th Working Conference on Mining Software Repositories. ACM, pp. 12–21.

Jha, S., 2011. Poorly written apps can sap 30 to 40% of a phones juice. In: Proceedings of the CEO, Motorola Mobility, Bank of America Merrill Lynch 2011 Technology Conference.

Jing, P., Zheng, S., Eli, T., 2016. Understanding the energy, performance, and programming effort trade-offs of Android persistence frameworks. In: Proceedings of the MASCOTS'16. IEEE.

Kwon, Y.-W., Tilevich, E., 2013. The impact of distributed programming abstractions on application energy consumption. Inf. Softw. Technol. 55 (9), 1602–1613.

Kwon, Y.-W., Tilevich, E., 2013. Reducing the energy consumption of mobile applications behind the scenes. In: Proceedings of the 2013 IEEE International Conference on Software Maintenance. IEEE, pp. 170–179.

Lavazza, L., Morasca, S., Tosi, D., 2016. An empirical study on the effect of programming languages on productivity. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing. ACM, pp. 1434–1439.

Li, D., Hao, S., Gui, J., Halfond, W.G., 2014. An empirical study of the energy consumption of Android applications. In: Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), 2014. IEEE, pp. 121–130.

Li, D., Lyu, Y., Gui, J., Halfond, W.G., 2016. Automated energy optimization of HTTP requests for mobile applications. In: Proceedings of the 38th International Conference on Software Engineering (ICSE), 2016 IEEE/ACM. IEEE, pp. 249–260.

Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Oliveto, R., Di Penta, M., Poshyvanyk, D., 2014. Mining energy-greedy API usage patterns in Android apps: an empirical study. In: Proceedings of the 11th Working Conference on Mining Software Repositories. ACM, pp. 2–11.

Liu, Y., Xu, C., Cheung, S.-C., Terragni, V., 2016. Understanding and detecting wake lock misuses for Android applications. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, pp. 396–409.

Lyu, Y., Gui, J., Wan, M., Halfond, W.G., 2017. An empirical study of local database usage in Android applications. In: Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), 2017. IEEE, pp. 444–455.

Manotas, I., Bird, C., Zhang, R., Shepherd, D., Jaspan, C., Sadowski, C., Pollock, L., Clause, J., 2016. An empirical study of practitioners' perspectives on green software engineering. In: Proceedings of the 38th International Conference on Software Engineering (ICSE), 2016 IEEE/ACM. IEEE, pp. 237–248.

Monsoon power monitor. Last accessed data: 19-June-2018. Available: https://www.msoon.com/LabEquipment/PowerMonitor/.

Mukherjee, S., Mondal, I., 2014. Future practicability of android application development with new android libraries and frameworks. Int. J. Comput. Sci. Inf. Technol. 5 (4), 5575–5579.

Newman, C., 2004. SQLite (Developer'S library). Sams.

NoSQL database engine Last accessed data: 19-June-2018. Available: http://nosql-database.org/.

OrmLite – lightweight object relational mapping (ORM) Java package Last accessed data: 19-June-2018. Available: http://greenrobot.org/greendao/.

Paper Fast and simple data storage library for Android. Last accessed data: 19-June-2018. Available:https://github.com/pilgr/Paper.
Pathak, A., Hu, Y.C., Zhang, M., 2012. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In: Proceedings of the 7th ACM european conference on Computer Systems. ACM, pp. 29–42.
Pinto, G., Castor, F., Liu, Y.D., 2014. Mining questions about software energy consumption. In: Proceedings of the 11th Working Conference on Mining Software Repositories. ACM, pp. 22–31.
Realm database engine. Last accessed data: 19-June-2018. Available:https://realm.io/.
Sahin, C., Cayci, F., Gutiérrez, I.L.M., Clause, J., Kiamilev, F., Pollock, L., Winbladh, K., 2012. Initial explorations on design pattern energy usage. In: Proceedings of the First International Workshop on Green and Sustainable Software (GREENS), 2012. IEEE, pp. 55–61.
Singh, M.P., Jain, M.K., 2014. Evolution of processor architecture in mobile phones. Int. J. Comput. Appl. 90 (4), 34–39.
Sqlite database engine. Last accessed data: 19-June-2018. Available: https://www.sqlite.org/.
Sugar ORM – insanely easy way to work with Android databases, Last accessed data: 19-June-2018. Available: http://satyan.github.io/sugar/.
Smartphone OS market share, 2015, Last accessed data: 19-June-2018. http://www.idc.com/prodserv/smartphone-os-market-share.jsp.
Statista: mobile apps available in leading stores 2015, Last accessed data: 19-June-2018. http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/.
Subramanian, M., Krishnamurthy, V., Shores, R., 1999. Performance challenges in object-relational DBMSs. IEEE Data Eng. Bull. 22 (2), 27–31.
Tiwari, V., Malik, S., Wolfe, A., Lee, M.T.-C., 1996. Instruction level power analysis and optimization of software. In: Proceedings of the Technologies for Wireless Computing. Springer, pp. 139–154.
Vetro, A., Ardito, L., Procaccianti, G., Morisio, M., 2013. Definition, implementation and validation of energy code smells: an exploratory study on an embedded system. In: Proceedings of the Third International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies(Energy), 2013, pp. 34–39.
Xia, C., Yu, G., Tang, M., 2009. Efficient implement of ORM (object/relational mapping) use in J2EE framework: Hibernate. In: Proceedings of the International Conference on Computational Intelligence and Software Engineering, 2009. CiSE 2009. IEEE, pp. 1–3.

Zheng Song is a Ph.D student in the computer science department of VIrginia Tech. His research interests include mobile computing and software engineering.

Junjie Cheng is now a 4th year undergraduate student in the computer science department of VIrginia Tech.

Jing Pu holds a master's degree in computer science from Virginia Tech.

Eli Tilevich is an associate professor in the department of computer science of Virginia Tech. His research interests include systems end of software engineering; distributed systems and middleware; automated software transformation; mobile applications; energy efficient software; CS education; music informatics.