Understanding the Performance-Energy Tradeoffs of Object-Relational Mapping Frameworks

Alexandre Bonvoisin, Clément Quinton, Romain Rouvoy

Understanding the Performance-Energy Tradeoffs of Object-Relational Mapping Frameworks

Alexandre Bonvoisin, Clément Quinton, Romain Rouvoy
Univ. Lille, CNRS, Inria, UMR 9189 CRIStAL, F-59000 Lille, France
{alexandre.bonvoisin;clement.quinton;romain.rouvoy}@univ-lille.fr

***Abstract*—*Object-Relational Mapping* (ORM) frameworks are the cornerstone of online services. To reply to incoming requests, these services often rely on these frameworks as a convenient data access layer. However, such frameworks might also be the source of performance inefficiency when configured and used inappropriately.**

This paper, therefore, compares different configurations of state-of-the-art Java-based ORM frameworks to unveil their performance efficiency, traditionally evaluated through metrics such as execution time and memory usage. However, rising environmental concerns have brought energy consumption to the forefront of the conversation. Beyond performance-centric measurements, we shed light on the energy consumption of these building blocks and explore the trade-offs that conceal the expected quality of service and environmental concerns.

Our empirical results, obtained with an ORM-based version of the reference *Transaction Processing Performance Council benchmark C* (TPC-C) benchmark, highlight that the adoption of an ORM should be carefully configured by developers to leverage the resources offered by underlying databases.

***Index Terms*—ORM Frameworks, Energy consumption, Performance**

I. Introduction

The energy consumption of software systems has gained prominence as a significant environmental and societal concern. Over the past decade, multiple studies have underscored the substantial influence of software on energy consumption and emphasized the importance of green software design to improve the energy efficiency of software systems. For instance, previous works in this field have studied and compared the energy consumed by common operations on Java collections [1] or the use of different Java I/O libraries [2]. However, while various tools and methods have been developed to comprehend, measure, and predict software energy consumption [3], the complexity of modern software environments and the composition of software layers make this sustainability objective particularly challenging [2].

Especially, little effort has been invested in the study of the energy efficiency of the data access layer. Yet, there exist various means to access and manage data, ranging from low-level and hand-crafted approaches to more abstract and user-friendly frameworks delivering some added features. This is particularly the case in Java, which provides—on the one hand—the *Java DataBase Connectivity* (JDBC) API, a standard library to communicate with any database from a Java program. Using JDBC, developers can retrieve data sets by writing queries and running procedures using the *Structured Query Language* (SQL), hence mixing SQL code with their business logic. On the other hand, *Jakarta Persistence API* (JPA) is another Java standard that maps data stored in relational databases with Java objects, in a so-called *Object-Relational Mapping* (ORM). For example, EclipseLink[1] and Hibernate[2] are the most popular JPA implementations [4], [5]. These ORM frameworks are widely appreciated for developing applications that interact with database management systems [6]. Interestingly, they provide automated support for managing an object-oriented model of data stored in a relational one. While such frameworks tend to ease the development of applications manipulating persistent data, it remains unclear if they succeed in delivering energy-efficient solutions to query and retrieve data at large. In this paper, we thus assess the performance and related energy consumption of the plain JDBC and ORM-based approaches. Precisely, the purpose of the study is to answer the following research questions:

- **RQ1:** How much energy overhead is introduced by ORM strategies compared to plain JDBC?
- **RQ2:** How does ORM framework performance relate to energy efficiency?
- **RQ3:** Does the configuration of an ORM framework influence its energy consumption?

Concretely, we leverage the widely accepted *Transaction Processing Performance Council benchmark C* (TPC-C) to compare the data access strategies implemented by the popular EclipseLink and Hibernate ORM frameworks. We take advantage of extensive performance and energy-related measurements to study diverse ORM features affecting the energy efficiency of Java applications. The key contributions of this paper can therefore be summarized as:

1) Providing a better understanding of the energy and performance variations of different ORM strategies under diverse workloads;
2) Identifying controllable factors that contribute to the variation in ORM frameworks' energy consumption;

We believe that this empirical knowledge can be effectively used by Java developers to reduce the energy footprint of their applications, while eventually improving their performances.

The remainder of this paper is organized as follows. Section II introduces the data access strategies commonly implemented by Java applications. Section III details the

[1]https://eclipse.dev/eclipselink/
[2]https://hibernate.org/

experimental protocol we adopted to measure the energy efficiency of ORM frameworks. Section IV dives into the results of our experiments to answer the above research questions. Finally, Section VI discusses related work, while Section VII concludes the paper.

II. Data Access Strategies

In this section, we introduce the 5 data access strategies we assessed. In particular, we cover both raw and ORM strategies, the latter being built upon JPA 3.1 features.[3] JPA is a Java specification that eases the management of relational data in applications. Within JPA, entities are Java classes representing persistent data structures stored in relational databases. These entities are typically annotated to define how they map to corresponding database tables, outlining the relationships between Java objects and database columns (cf. Listing 1). Each instance of an entity corresponds to a record or row in the associated database table. Furthermore, entities can be annotated to specify additional metadata, including the definition of plain-text queries associated with unique names, later used to identify a query to be executed.

```
@SqlResultSetMapping(
   name = "Customer.getEmailSQLMapping",
   columns = @ColumnResult(
             name = "email", type = String.class))
@NamedNativeQuery(name = "Customer.getEmailSQL",
   query = "SELECT email FROM customer_table "
         + "WHERE unique_id = ?",
   resultSetMapping =
       "Customer.getEmailSQLMapping")
@NamedQuery(name = "Customer.getEmailJPQL",
   query = "SELECT c.email FROM Customer c "
         + "WHERE c.id = ?1")
@Entity @Table(name = "customer_table")
public class Customer implements Serializable {
   @Column(name = "unique_id")
   @Id private Integer id; // Primary key
   @Column(name = "email")
   private String email;
}
```

Listing 1. Entity class declarations

Beyond mapping, JPA also introduces two caching features. The first-level cache *a.k.a.* "Persistence Context" is bound to a database transaction and cannot be disabled, as its goal is also to monitor changes on "managed" entities. When an entity is retrieved from the database, it is stored in this cache, ensuring data consistency by automatically updating changes to the object and reducing the number of database queries during the transaction. As for the second cache level, it is optional. If used, it extends the persistence context and developers must take care of the data cached to prevent data staleness. As this cache requires an external caching library, we disabled it to solely assess ORM capabilities without introducing any bias related to the choice of the caching library.

When conforming to JPA features, one can seamlessly transition between various vendors without modifying the application source code. This migration process merely involves substituting the vendor-specific package in the project path, ensuring a fair comparison among providers. To illustrate how each strategy handles queries, we provide for each of the 5 strategies the related code sample to execute the same query, *i.e.*, *select the email for a given customer*.

[3]https://jakarta.ee/specifications/persistence/3.1/

A. Plain JDBC Access Strategy

When they are not using an ORM, developers leverage a JDBC driver to interact with the *Database Management System* (DBMS). As illustrated in Listing 2, they explicitly manage database connections, craft SQL queries, and handle query results. This approach provides fine-grained control over the DBMS but demands specific skills for developers, including writing and maintaining custom SQL queries and dealing with low-level database intricacies. In the remainder of the paper, we will refer to this implementation as the plain JDBC one.

```
String query = "SELECT email FROM customer_table "
             + "WHERE unique_id = ?";
ResultSet rs = jdbcCon.prepareStatement(query)
                      .setInt(1, id).executeQuery();
String email = rs.next()?rs.getString("email"):"";
```

Listing 2. Plain JDBC access strategy

B. ORM Access Strategies

When adopting an ORM framework to interact with the DBMS, 4 distinct strategies can be considered. Each of these data access strategies comes with its benefits and trade-offs, and the choice of a strategy depends on various factors, such as performance requirements, compatibility with DBMS features, and code maintainability.

1) Native Access Strategy: .This approach refers to the use of raw SQL queries using the JPA's API (cf. Listing 3). As the specification's terminology refers to raw SQL queries as "native queries", we named this strategy as the "native" one. In contrast to the plain JDBC strategy, which imposes manual data mapping, the JPA standard encourages that the mapping between query result sets and Java objects are declared through the framework configuration (cf. Java annotations in Listing 1).

```
EntityManager em = // ...
String email = (String)
    em.createNamedQuery("Customer.getEmailSQL")
              .setParameter(1,id).getSingleResult();
```

Listing 3. Native access strategy

2) JPQL API Access Strategy: This data access strategy uses the *Jakarta Persistence Query Language* (JPQL), a query language with an SQL-like syntax. This language provides an abstraction layer over raw SQL queries that enables developers to define queries independently of the SQL dialect used by the targeted data store. In JPQL, developers manipulate Java fields and class names, which are mapped to database tables—instead of explicitly specifying column and table names within the queries (cf. Listing 1). This approach simplifies the query creation process and enhances portability, as the queries can be adapted to different databases without any modification (cf. Listing 4).

```
EntityManager em = // ...
String email = em.createNamedQuery(
       "Customer.getEmailJPQL", String.class
     ).setParameter(1,id).getSingleResult();
```

Listing 4. JPQL access strategy

3) Criteria API Strategy: This strategy relies on the JPA Criteria API, which enables developers to construct queries using a set of Java objects and methods calls, rather than composing queries as plain text like raw SQL or JPQL (cf. Listing 5).

```
EntityManager em = // ...
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<String> query =
                    cb.createQuery(String.class);
Root<Customer> root = query.from(Customer.class);
ParameterExpression<Integer> idParameter =
                     cb.parameter(Integer.class);
Predicate predicate = cb.equal(idParameter,
                         root.get(Customer_.id));
String email = em.createQuery(query
                .select(root.get(Customer_.email))
                .where(predicate)
               ).setParameter(idParameter,idValue)
               .getSingleResult();
```

Listing 5. Criteria access strategy

4) Managed Entity Strategy: As introduced previously, managed entities are objects mapped to database records, which are actively monitored by the ORM framework to ensure that any changes are automatically reflected in the underlying database (cf. Listing 6).

```
EntityManager em = // ...
Customer c = em.find(Customer.class,id);
String email = c.getEmail();
```

Listing 6. Entity access strategy

III. Experimental Methodology

This section describes our hardware settings, the benchmarks we used, and our data collection process. All experimental materials and performance data monitored during our experiments are publicly accessible.[4]

A. Hardware Settings

To enforce reproducibility, we conducted all our experiments using a consistent technical setup. The hardware infrastructure comprises two identical Dell PowerEdge R640 servers, each equipped with a single Intel Xeon Gold 5220 @2.20GHz processor (18 cores/36 threads), 96 GB of DRAM, 480 GB SSD, and two 25Gb/s network cards.

B. Database Benchmark

To assess the energy efficiency of strategies described in Section II, we opted for the *Transaction Processing Performance Council benchmark C* (TPC-C)[5] [7] defined in the BenchBase (formerly OLTPBench) framework,[6] an open-source database benchmarking tool [8]. While this well-known benchmark was initially designed to assess database capabilities through an intensive workload simulating business activities between customers and a wholesale supplier, many aspects of its specification are well-suited for evaluating data access layer strategies.

1) Database Schema: The schema consists of 9 tables representing various entities involved in wholesale supplier operations. For instance, the Warehouse table stores unique identifiers and addresses of warehouses, while the Customer table holds details about individual customers, including identifiers, names, and number of payments. The number of warehouses is a key parameter to set up and run the benchmark, as it serves as a *scaling factor*, for adjusting both the number of records to be inserted into the tables and the number of concurrent users to be simulated.

2) Input Workload: In the TPC-C benchmark specification, a transaction is a sequence of SQL queries designed to simulate specific business operations. There are 5 types of transactions of various complexity levels, each with a frequency that determines the proportion of transactions of that type in the workload. We used the same frequencies as defined in [7]. The queries defined within the transactions are pivotal to this study, as they provide a comparable basis to the queries generated by the different ORM strategies.

C. ORM Frameworks

As introduced previously, we employed the TPC-C implementation from the BenchBase framework as the basis for implementing our data access strategies. Apart from the modification of the data access layer, we also edited the framework's implementation related to the transaction retry feature. Rather than making additional attempts to execute a failed transaction again, our implementations categorize such attempts as failures from the outset. This adjustment was implemented in all benchmark variants to ensure that the measurements collected remain unaffected by potential variations introduced by retry attempts. As ORM strategies are implemented according to the last 3.1 version of JPA, the considered JPA provider at build time are EclipseLink (version 4.0.0), the reference implementation, and Hibernate (version 6.1.3.$Final$), the most popular alternative in the community.

D. Data Collection Process

To collect data on different metrics for each access strategy, we implemented a thorough measurement protocol.

1) Collected Metrics: To assess the efficiency of data access layer strategies, we focused on two key performance metrics: power consumption, which reflects resource usage, and goodput, which highlights operational efficiency.

a) Power Consumption: We measured power usage for both the CPU package (PKG) and the *Dynamic Random Access Memory* (DRAM). This measurement was carried out using the *Running Average Power Limit* (RAPL) interface [9], a feature available with recent Intel processors known for its

[4]https://zenodo.org/doi/10.5281/zenodo.10061193
[5]https://www.tpc.org/tpcc/
[6]https://db.cs.cmu.edu/projects/benchbase/

high accuracy and reliability in power monitoring [10], [11]. The energy efficiency of an assessed strategy is determined by the system's power consumption during the workload, after subtracting the idle power consumption, as described in [12].

b) Goodput: The goodput is a metric reported by the benchmark that indicates the efficiency of a database system by measuring the number of successful transactions completed per second. A higher goodput indicates increased effectiveness, as it directly correlates with the system's ability to execute successful operations at a faster rate.

2) Software Setup: We configured both servers with a minimal version of Ubuntu 20.04, including only essential components required for system operation and remote access through *Secure Shell* (SSH). On one server, we installed Git, XMLStarlet, and Docker to facilitate the build and execution of Java applications implementing the strategies covered by this study. These implementations are compiled as Java 17 applications using a Maven container set up for reproducible builds. They are then packaged into a Docker image based on `eclipse-temurin:17.0.8_7-jre-alpine`, ensuring a lightweight *Java Runtime Environment* (JRE). While Docker introduces an additional layer that might lead to increased energy consumption, it has a consistent and minimal impact on energy variation, resulting in a constant, negligible overhead [13]. The other server was exclusively equipped with Docker to run an on-demand, volatile `postgres:14.5` database mounted with a docker volume cleaned before each measurement. The database is started and populated with an initial dataset that will be processed by the workload transactions. As the benchmark's default behavior is to randomly generate these records, we set the random seed to 0, a commonly used value in Java programs. This adjustment guarantees consistent initial data for all measurements. It is also important to note that our measurements were conducted with a single Docker container process running on the servers. Finally, all running containers are stopped to put both systems in an idle state.

3) Metrics Collection: A single measurement run is a multi-step process conducted as follows. First, we measure the power consumption of both servers while they are at rest. These measurements are taken over 5 seconds, as we observed that if the CPU is not actively in use, power consumption quickly stabilizes within this period. Throughout this time frame, both systems remain in an idle state, with no active applications or processes running. Second, we start the remote database and wait 30 seconds for it to set up and be ready to accept requests. We then initiate the execution of the assessed software variant, during which we gather data on the energy consumption and the achieved goodput throughout the workload duration, set to 60 seconds in the BENCHBASE default configuration file. Finally, we dismantle the database and perform an additional 5-seconds measurement to assess the power consumption of both servers while back at rest.

4) Workload Parameters: We evaluated each data access strategy under changing workloads, setting the target rate from 10 to an unlimited number of *Transaction Per Second* (TPS), with increments of 10, 100, 1000, and 10.000 TPS. Given the stochastic nature of the TPC-C workload, we repeated the measurement runs 40 times for each combination of rate goal and data access strategy and calculated the median of the results, adding up to 4200 runs.

In the default configuration of the BENCHBASE framework, the scaling factor presented in Section III-B1 is set such that it simulates a workload with too many clients in comparison to the capabilities of the machine. To align with the benchmark specification recommending 10 clients per unit of the scaling factor, we set the number of concurrent clients to 50. By using a scaling factor of 5 and allowing 50 clients to concurrently query the database, we aim to create a workload that is close enough to real-world scenarios. To prevent inconsistencies and ensure that transactions do not interfere with each other, transactions were performed at the serializable isolation level, also defined in the BENCHBASE default configuration file.

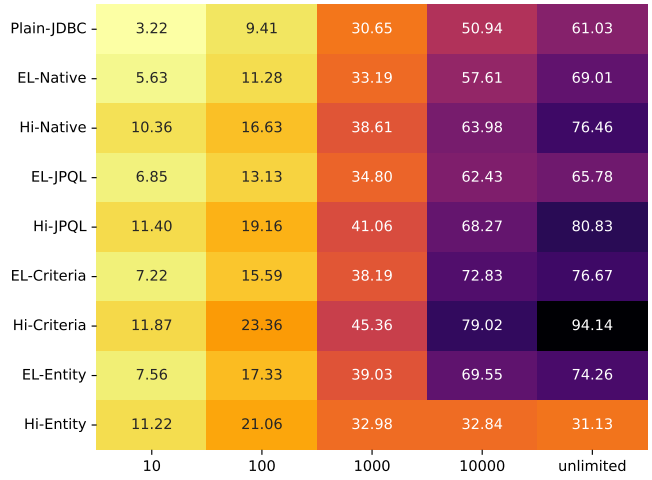
IV. EMPIRICAL RESULTS

We consider the plain JDBC strategy as the baseline to evaluate the overhead induced by other strategies. By assessing these different ORM-based strategies, we aim to gain better insights into the influence of such strategies and the trade-off they offer between performance and energy efficiency.

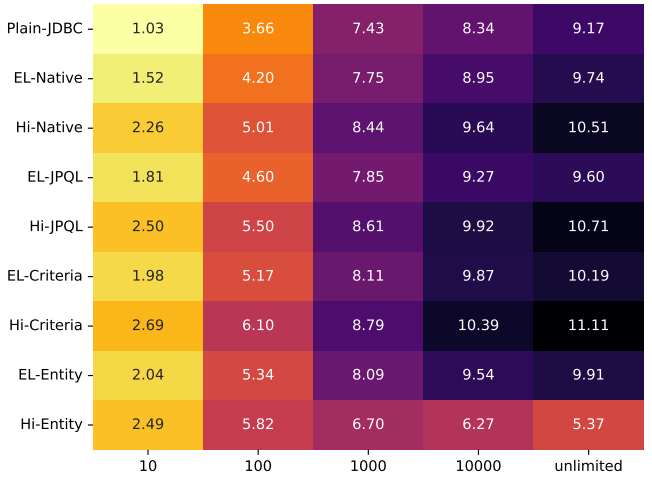
A. RQ1: How much energy overhead is introduced by ORM strategies compared to plain JDBC?

To answer our first research question, we thus compared the energy consumption of the different ORM access strategies with the plain JDBC one when executing a certain workload, as explained in Section III-D. Figures 1(a) and 1(b) depict, for each implementation, the median power usage depending on the target rate for CPU and DRAM, respectively (EL stands for ECLIPSELINK and Hi for HIBERNATE). Both figures highlight consistent trends as the target rate increases, *i.e.*, the power usage increases accordingly. Yet, one can observe that HIBERNATE ENTITY demonstrates low energy consumption at higher rates. This can be explained by its poor performance. As shown by Figure 2, the number of completed transactions HIBERNATE ENTITY can process is plateauing once the target rate reaches 1000 transactions per second. HIBERNATE ENTITY thus faces performance issues related to completed transactions, arising because of the increased concurrency resulting from the higher target rate. Consequently, HIBERNATE ENTITY transitions into a degraded mode, causing the built-in workload injector to reduce the number of initiated transactions which, in turn, reduces energy consumption.

HIBERNATE ENTITY left apart, we observe that the plain JDBC strategy exhibits the lowest energy consumption, while the use of ORM strategies results in a significant power usage increase, especially regarding CPU consumption. For instance, at the target rate 100, ORM strategies overhead varies from 16% to 59% for the ECLIPSELINK NATIVE and HIBERNATE CRITERIA strategies, respectively. As ORM strategies introduce an additional abstraction layer compared to the plain JDBC one (see Section II-B), such trends were predictable.



(a) CPU median power usage (Watts)



(b) DRAM median power usage (Watts)

Fig. 1. CPU and DRAM median power usage (Watts) across configurations and a varying target rate.

For a comprehensive and fair analysis, the benchmark must execute an identical input workload. As some ORM strategies generate the executed SQL queries, we ensured that strategy implementations adhered closely to the business logic defined in the benchmark while complying with the concepts outlined by each ORM strategy. Among all studied ORM strategies, only the *Native* ones guarantee the execution of SQL queries that match those defined in the plain JDBC strategy. When focusing on *Native* strategies, one can observe that, although the overhead is less significant, it is non-negligible. For instance, at the target rate of $10,000$, there is an 11% and 20% increase for the native ECLIPSELINK and HIBERNATE ORM regarding CPU usage, respectively. While less important, the energy consumption increase regarding DRAM usage for the same target rate even so reaches 6% and 13% for the native ECLIPSELINK and HIBERNATE ORM strategies, respectively.

> **RQ1**: Our empirical study shows that the plain JDBC strategy consistently reports being the most energy-efficient compared to the ORM ones. The overhead introduced by ORM strategies is imposed by the additional abstraction layers offered by such frameworks. Among ORM strategies and apart from the specific case of HIBERNATE ENTITY which performs poorly at high rates, *Criteria* is the most consuming strategy for both HIBERNATE and ECLIPSELINK frameworks.

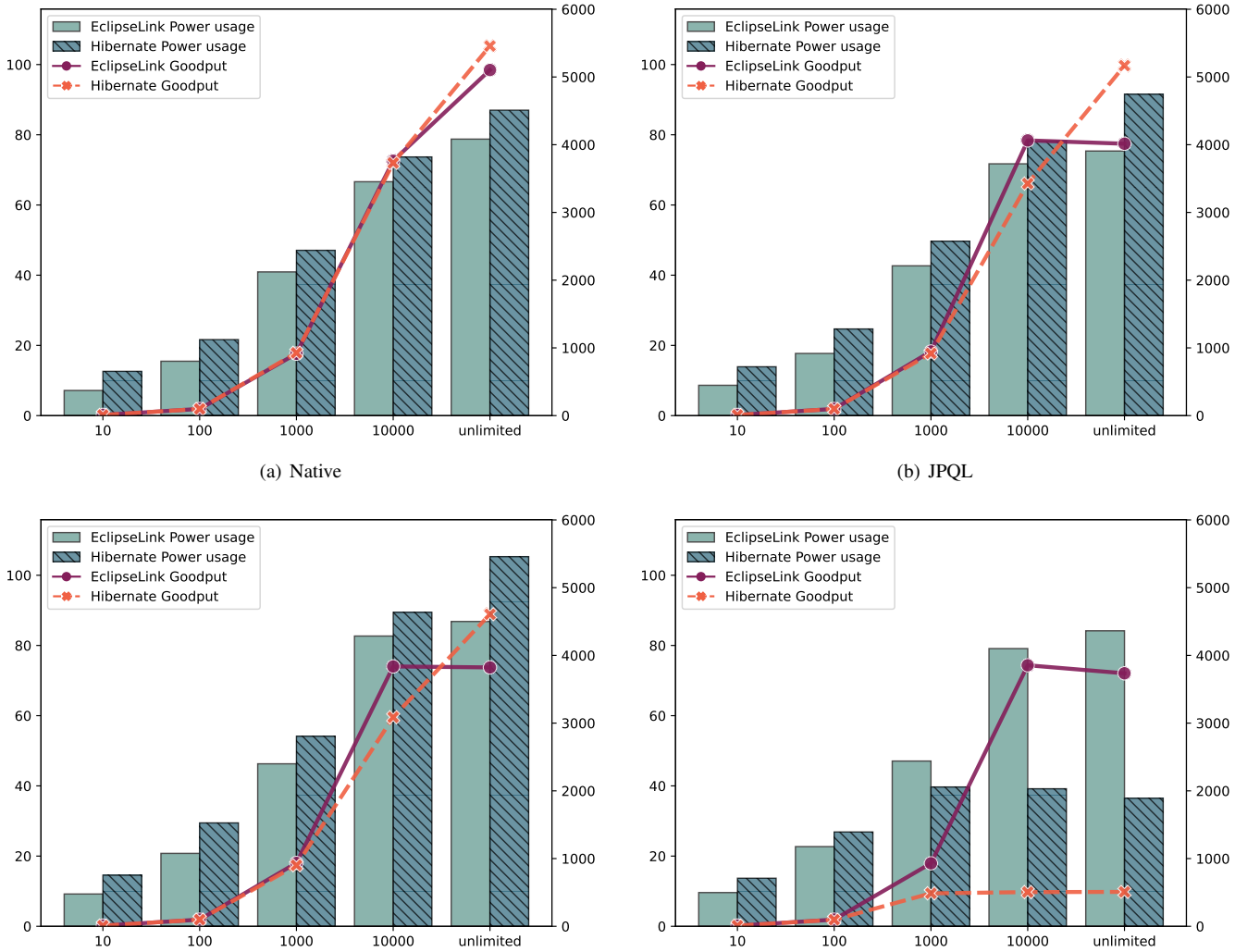
B. RQ2: How does ORM framework performance relate to energy efficiency?

To answer our second research question, we compared the goodput of ECLIPSELINK and HIBERNATE given their energy consumption when executing a varying workload. Figure 2 provides an overview of this comparison. Several observations can be made. When employing the *Native* strategy, ECLIPSELINK and HIBERNATE demonstrate similar performances as they both execute identical raw SQL queries, resulting in comparable sets of completed transactions. However, ECLIPSELINK seems to offer the best energy efficiency, consuming 45% less energy at a rate of 10 and consistently showing an average reduction of 22% across all rates.

Regarding the other three strategies, ECLIPSELINK always exhibits a recurrent performance pattern, performing consistently across varying rates and reaching equivalent or higher goodput than HIBERNATE, until reaching $10,000$ transactions per second. Once this rate is overtaken, ECLIPSELINK goodput is then plateauing.

In contrast, the HIBERNATE goodput remains consistent across the *Native*, *JPQL*, and *Criteria* strategies, even at unlimited rates. Regarding these 3 strategies, ECLIPSELINK is always better performing while consuming less energy than HIBERNATE at rates lower or equivalent to $10,000$, as highlighted by Figures 2(b) and 2(c).

We also observe that HIBERNATE experiences a significant drop in performance when managing transactions relying on the *Entity* strategy (see Figure 2(d)). When further investigating these results, we found out that both frameworks handle queries differently. In particular, when fetching an entity that makes references to other ones, HIBERNATE does not check whether referenced entities are already present in the first-level cache described in Section II. Instead, it directly uses join operations to fetch associated data from multiple tables. Listing 7 illustrates such operations, which instantiate all entities corresponding to transitive relationships—*i.e.*, the full graph of relationships. Consequently, HIBERNATE might redundantly retrieve data that is already stored in the `Persistence Context`. ECLIPSELINK behaves differently as it *(i)* sends queries only when referenced entities are not present in the first-level cache, and *(ii)* employs independent queries to fetch individual foreign entities rather than using joins, as shown by Listing 8, hence achieving better performances at scale.



(a) Native

(b) JPQL

(c) Criteria

(d) Entity

Fig. 2. Goodput (Tx/sec, right-Y axis) and global power usage (Watts, left-Y axis) median variations across varying ORM strategies.

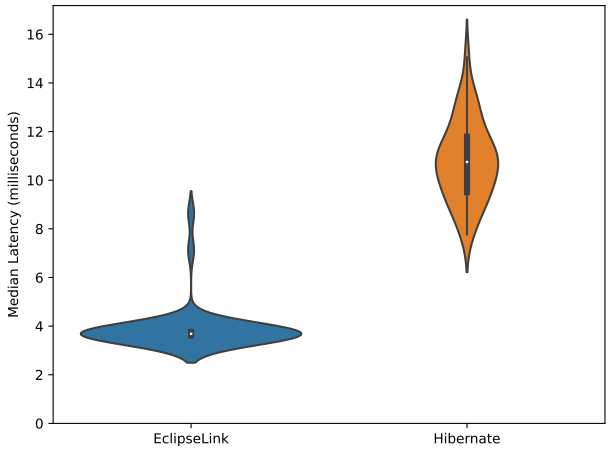
```
-- Some columns have been omitted for clarity
SELECT w1_0.w_id, w1_0.w_tax, ...
FROM warehouse w1_0 WHERE w1_0.w_id=?

SELECT d1_0.d_id, d1_0.d_next_o_id, d1_0.d_tax,
    d1_0.d_w_id, w1_0.w_id, ...
FROM district d1_0
JOIN warehouse w1_0 ON w1_0.w_id=d1_0.d_w_id
WHERE (d1_0.d_id,d1_0.d_w_id) IN((?,?))
FOR NO KEY UPDATE
-- Above, referenced entities already fetched
SELECT c1_0.c_id, c1_0.c_d_id,c 1_0.c_w_id,
    c1_0.c_credit, c1_0.c_discount, c1_0.c_last,
    d1_0.d_id, d1_0.d_w_id, w1_0.w_id, ...
FROM customer c1_0
JOIN district d1_0 ON d1_0.d_id=c1_0.c_d_id AND
    d1_0.d_w_id=c1_0.c_w_id LEFT JOIN warehouse
    w1_0 ON w1_0.w_id=d1_0.d_w_id
WHERE (c1_0.c_id,c1_0.c_d_id,c1_0.c_w_id)
    IN((?,?,?))
```

Listing 7. Transaction from HIBERNATE's managed entity strategy

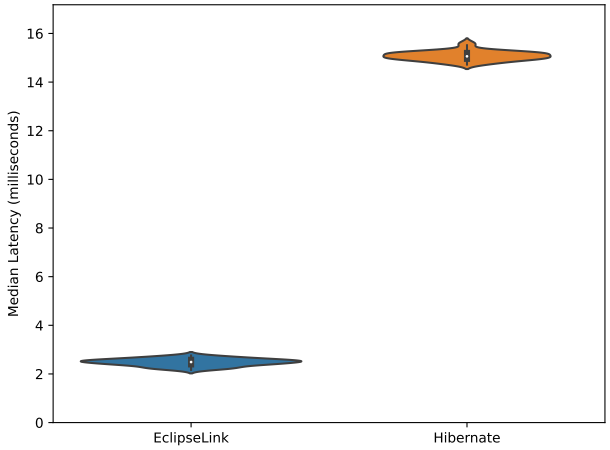
```
-- Some columns have been omitted for clarity
SELECT w_id, w_tax, ...
FROM warehouse WHERE (w_id = ?)

SELECT d_id, d_next_o_id, d_tax, ...
FROM district WHERE ((d_id = ?) AND (d_w_id = ?))
FOR UPDATE

SELECT c_id, c_discount, c_last, c_credit, ...
FROM customer WHERE (((c_id = ?) AND (c_w_id = ?))
AND (c_d_id = ?))
```

Listing 8. Transaction from ECLIPSELINK's managed entity strategy

Using the `JOIN` clause for data retrieval from multiple tables can increase transaction complexity and disrupt concurrent operations in the DBMS. This increased workload leads to longer execution times, heightened latency (2.5 times more at rate $1,000$ and up to 5 times more at rate $10,000$, as depicted in Figure 3), and more data locks due to the serialization isolation level, resulting in increased transaction failures.



(a) Target rate $1,000$ Tx/sec



(b) Target rate $10,000$ Tx/sec

Fig. 3. Distribution of transactions median latency across measurements for the Entity strategy.

To mitigate the negative effects of transaction failures, the built-in workload injector mechanism reduces the number of transactions to be processed, finally resulting in a reduction in goodput and power consumption as well.

RQ2: We analyzed the goodput of different strategies under varying workloads, in view of their power usage. Our empirical results indicate that ECLIPSELINK consumes less energy than HIBERNATE to complete the same workload for rates up to $1,000$. At higher rates, HIBERNATE exhibits a better goodput for all strategies except for the *Entity* one, which experiences a significant drop in performances. This is due to generated queries that play a critical role in understanding the performances of ORM frameworks. Customized SQL queries tailored to specific tasks exhibit the best performance, but this approach does not benefit from the abstraction layers provided by ORM frameworks. Overall, ECLIPSELINK stands out for generating well-performing queries.

C. RQ3: Does the configuration of an ORM framework influence its energy consumption?

To answer our third research question, we introduced modifications to the default ORM configuration used in previous experiments to assess whether these changes may improve energy efficiency. In particular, we focused on 3 different features of different scopes, namely *lazy fetching*, *batch processing*, and *partial update*. We chose to apply these changes to the *Entity* strategy for two main reasons. First, it offers the highest level of abstraction from a Java perspective. Second, we aimed to investigate whether these changes could address some of the HIBERNATE performance issues highlighted by RQ2. While the first feature is a JPA feature that can be enabled seamlessly on ECLIPSELINK and HIBERNATE, the remaining two demand vendor-specific adjustments.

a) Lazy Fetching: ECLIPSELINK employs two distinct lazy variants to enable lazy fetching. Despite the compliance of the JPA's lazy annotation, bytecode instrumentation (also known as weaving) is required for actual lazy data retrieval. This instrumentation can be achieved dynamically by using a *javaagent* property when launching the program, or statically by enhancing the compiled classes through static class rewriting. In contrast, HIBERNATE enables lazy loading by default, as it relies on entity proxies. When a lazily-fetched attribute is accessed, HIBERNATE retrieves all the fields of an entity to transition from a proxy instance to a concrete entity. Additionally, HIBERNATE uses bytecode instrumentation to inject vendor-specific features, allowing fetching of a single lazy field, rather than loading all the fields, as is implemented by the proxy-based approach.

b) Batch Processing: One limitation of JPA is the absence of a defined batch processing mechanism, especially when working with entities. The specification does not provide a standard approach to consolidate multiple update queries into a single operation treated as a batch. For instance, if one needs to update 100 records, 100 individual queries would have to be executed without batch processing, instead of one single batch operation grouping all of these updates. Fortunately, both ECLIPSELINK and HIBERNATE offer support to address this gap, enabling a fair comparison between them.

c) Partial Update: This setting is a specific feature of HIBERNATE. By default, when the framework generates an update query, it sends an update for all fields, even if there are no changes in multiple fields. We considered enabling this feature in HIBERNATE to align with the default behavior of ECLIPSELINK.

For each ORM framework, we thus derived configurations activating each[7] of the 3 features independently or in combination, resulting in 14 different configurations. For each configuration, we measured its power usage and goodput metrics and computed an overall efficiency score P_{conf}, such that $P_{\text{conf}} = \frac{power}{goodput}$, which captures the energy consumption per successful transaction.

[7]When applicable: the partial update feature cannot be activated on ECLIPSELINK as it is a default one.

Fig. 4. Hierarchical clustering heatmap for the Entity strategy with different enabled features (rate $10,000$).
la = lazy fetching {→ for EL : s = with static weaving, d = with dynamic weaving}; ba = batch processing; up = partial update.

Then, we compared each pair of configurations in terms of their respective P_{conf}. Figure 4 presents the result of such pairwise comparison at a rate of $10,000$. Each cell defines the euclidean distance between the P_{conf} of the two associated

configurations, computed as $|P_{\text{conf1}} - P_{\text{conf2}}|$. The closer to 0 the value is, the more similar P_{conf} is exhibited by the two compared configurations, hence reporting a little influence of the differing features. For instance, ECLIPSELINK default configuration (EL) P_{conf} is very comparable to the HIBERNATE configuration with all 3 lazy fetching, batch processing and partial update features activated (Hi-la-ba-up), as it exhibits a distance of 0.0087. Activating such features thus proves particularly efficient to address the performance drop observed for HIBERNATE in RQ2 and highlighted in Figure 2(d).

In addition to individual comparisons, Figure 4 also outlines group tendencies—*i.e.*, clusters of configurations with similar P_{conf} values—as a dendrogram. To yield these clusters, the data underwent hierarchical clustering using the single linkage method, which relies on the nearest neighbor criterion. The effectiveness of this clustering process was evaluated using the cophenetic correlation coefficient. The resulting high value of 0.96 indicates a robust preservation of the pairwise distances between the original data points. These clusters are organized into a tree-based structure defined on top of the table. For instance, the left branch of the root node relates to 4 columns gathering 4 HIBERNATE variants, while the right branch groups the other 10 ORM variants together. More precisely, the left cluster corresponds to all HIBERNATE configurations where the lazy fetching feature is not activated.

The heatmap shows that these 4 HIBERNATE variants are the worst-performing ones. When compared with all ECLIPSELINK variants (the 6 central rows), they exhibit the longest distances, ranging from 0.0522 (HIBERNATE with batch processing and partial update [Hi-ba-up] *vs. e.g.*, ECLIPSELINK default configuration [EL]) to 0.070 (HIBERNATE with partial update [Hi-up] *vs.* ECLIPSELINK with dynamic lazy fetching and batch processing [EL-la-d-ba]). Such distance values highlight the best and worst performing configurations—*i.e.*, [EL-la-d-ba] and [Hi-up], respectively. Within the HIBERNATE configurations (bottom-right cluster), the best-performing one is the [Hi-la-up]. This variant with lazy fetching and partial update shows the longest distance of 0.527 from [Hi-up] and the shortest distance of 0.0173 from [EL-la-d-ba].

> **RQ3**: To answer RQ3, we studied the impact of 3 different features of different scopes, namely *lazy fetching*, *batch processing*, and *partial update*. Changing the configuration of an ORM framework can influence its power consumption and performance. Specifically, enabling the *lazy fetching* feature for HIBERNATE appears to be the most impactful factor. Additional performance and energy gains can be obtained by activating batch processing and partial update features.

V. DISCUSSION AND THREATS TO VALIDITY

Discussion. Our empirical study assessed ECLIPSELINK as the ORM reporting the best-performing queries and being the most energy-efficient, but still less energy-efficient than the plain JDBC approach. From a software engineering perspective, transitioning from an ORM approach to the plain JDBC one involves multiple considerations. While this shift may yield performance advantages, it could also result in extended development time or raise challenges in maintenance. Practitioners must carefully weigh the trade-offs between the convenience offered by an ORM and the potential performance and energy benefits associated with plain JDBC. As shown in Section IV, some JPQL or Entity configurations may offer a good trade-off to reduce energy consumption while improving maintenance.

Threats to validity. Some factors must be considered as they could affect the validity of our study.

Internal factors. Our primary objective was to understand if the varying degrees of abstraction maintained comparable performance when considering the specific features and optimizations associated with each ORM. We conducted our experiments with the PostgreSQL 14.5 release DBMS. Since ORM frameworks generate queries tailored to the SQL dialect expected by the target DBMS, results may differ when reproducing our experiments on another release of PostgreSQL or a whole different DBMS, such as MySQL. Regarding such assessments, the performances of the database itself could also be considered, as well as query-level analysis considering other operations (*e.g.*, table joins, write, etc.). Other potential threats to the validity of this study lie in the choice of the JRE [14], and its potential effects on a 'warm-up' phase [15], as well as the stochastic nature of the benchmark that may prevent from reproducing exact measurements.

External factors. To assess the energy consumption of ORM approaches, we adopted the well-known TPC-C, widely used to assess DBMS capabilities of online transaction processing systems through an intensive workload. However, ORM approaches may exhibit different performances when assessed against other benchmarks. While we could consider additional TPC benchmarks, like TPC-H,[8] ORM frameworks may, unfortunately, fail to deliver native support for some of the ad-hoc queries that involve complex joins, aggregations, and filtering operations, especially when it comes to subqueries.

Since the most popular and widely-used ORM frameworks, only ECLIPSELINK and HIBERNATE were considered in this study. Results may differ when assessing other ORM frameworks, such as EBEAN,[9] or non-ORM data access technologies, such as JDBI,[10] JOOQ[11] or MYBATIS.[12]

VI. RELATED WORK

In this section, we review and discuss the state of the art related to energy efficiency and performance in the Java ecosystem and, then, we focus on data access strategies-related approaches.

[8]https://www.tpc.org/tpch/
[9]https://ebean.io/
[10]https://jdbi.org/
[11]https://www.jooq.org/
[12]https://mybatis.org/mybatis-3/

Java Ecosystem. In the past few years, many studies have contributed valuable insights to the field of software energy efficiency, especially in the Java ecosystem. For example, Kumar *et al.* [16] conducted a comprehensive evaluation of Java programming constructs, including data types, operators, and control statements, to understand their energy efficiency implications. Other efforts delved into the energetic profiles of Java Collections and I/O libraries [1], [17], highlighting the need to select the proper implementation. Pinto *et al.* [18] examined the thread-safe Java collections and found significant variations in the energy footprint of analogous operations among various implementations of the same Java collection. Building upon this work, Oliveira *et al.* proposed a tool [19] designed to assist developers in selecting energy-efficient collection implementations to build more sustainable software.

Other research efforts have shown that minor adjustments can yield significant energy savings, such as Rocha *et al.* [20] who classified the energy consumption of 22 file read and write methods in Java native I/O classes, showing that small changes may improve the energy consumption by 2. Building upon this work, Ournani *et al.* [2] assessed additional native and third-party I/O libraries.

Another research line focused on programming practices, especially on design patterns aiming at enhancing productivity and maintainability. For instance, Nourredine *et al.* [21] studied the energy consumption of Observer and Decorator pattern implementations, while Connolly Bree *et al.* [22] investigated the Visitor one. Ournani *et al.* [23], [24] assessed that structural design pattern had no negative impact on the energy consumption of Java-based software.

While all these approaches provide valuable insights into software performance and energy consumption, none of them explored the concepts assessed in this paper within the realm of Java.

ORM Ecosystem. Procaccianti *et al.* also studied the energy efficiency of ORM frameworks [25]. Contrary to our study, they did not address JPA implementations, but rather compared raw SQL queries with PROPEL (a PHP ORM) and TINY QUERIES. Also, they did not investigate the potential improvement brought by changing ORM configurations. Verdecchia *et al.* [26] conducted a study on the performance and energy implications of refactoring code smells in Java ORM-based applications. The research revealed that addressing code smells through refactoring not only enhanced energy efficiency but also potentially improved the performance of the system. Surprisingly, when all code smells were refactored, they observed a performance overhead of approximately 6.8%, while the energy consumption was reduced by 10.7%. This finding underscores the complex relationship between performance and energy consumption, highlighting that these two aspects do not always correlate directly with each other.

Chen *et al.* investigated various performance issues related to software developed using ORM frameworks. Specifically, they conducted two distinct studies on the detection of performance anti-patterns related to JPA and the performance impact of redundant data access. In [27], they review code patterns that can lead to performance issues. They propose a framework to assist developers in automatically flagging performance anti-patterns in the source code of their ORM framework. In addition, they focus in [28] on the redundant data problems between the needed data in the code and the SQL-requested data, problems are usually caused by requesting/updating too much data than needed.

While both of these approaches offer valuable insights into the performance of ORM strategies, they do not take into account the power usage dimension. In contrast, our study provides a comprehensive analysis that includes both performance and power usage, with a particular focus on understanding performance in the context of its impact on energy consumption.

VII. CONCLUSION

In this paper, we investigated the influence of ORM configurations on both performance and energy consumption and the potential trade-offs regarding these two metrics. Using the widely accepted TPC-C benchmark, we assessed the energy overhead introduced by ORM frameworks, compared to plain JDBC. Across varying workloads, we identified ECLIPSELINK as the most energy-efficient ORM framework. This framework exhibits an 11% overhead in CPU usage compared to the plain JDBC approach at a high transaction rate, an overhead nonetheless mitigated by the benefits ORM frameworks provide, particularly in terms of data consistency between the middleware and the database. We also showed that energy efficiency might come at the price of performance and that developers may face potential trade-offs when configuring their ORM framework. In particular, we showed that HIBERNATE can be fine-tuned to get closer to ECLIPSELINK performances, providing insights into the varying capabilities of ORM tools when it comes to energy-aware data access.

In future work, we plan to study the energy efficiency of web frameworks in the context of cloud-native applications. Such frameworks provide a layered way to build web applications, and we aim to assess the energy efficiency of their various layers. For instance, we consider evaluating presentation layer libraries to provide additional guidelines for Java developers who aim to minimize the energy footprint of their web applications without sacrificing performance, aligning software engineering practices with environmental sustainability goals.

ACKNOWLEDGMENT

This work received support from the French government through the *Agence Nationale de la Recherche* (ANR) under the France 2030 program, including partial funding from the CARECLOUD (ANR-23-PECL-0003), DISTILLER (ANR-21-CE25-0022), and KOALA (ANR-19-CE25-0003-01) projects. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations.[13]

[13]See https://www.grid5000.fr

REFERENCES

[1] S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, and A. Hindle, "Energy profiles of java collections classes," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 225–236.

[2] Z. Ournani, R. Rouvoy, P. Rust, and J. Penhoat, "Evaluating The Energy Consumption of Java I/O APIs," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2021, pp. 1–11.

[3] F. Rieger and C. Bockisch, "Survey of approaches for assessing software energy consumption," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Comprehension of Complex Systems*, ser. CoCoS 2017. ACM, 2017, p. 19–24. [Online]. Available: https://doi.org/10.1145/3141842.3141846

[4] T.-H. Chen, W. Shang, A. E. Hassan, M. Nasser, and P. Flora, "Detecting problems in the database access code of large scale systems - an industrial experience report," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, 2016, pp. 71–80.

[5] S. P. R. Katamreddy and S. S. Upadhyayula, *Working with JDBC*. Apress, 2023, pp. 101–118. [Online]. Available: https://doi.org/10.1007/978-1-4842-8792-7_5

[6] C. Yan, A. Cheung, J. Yang, and S. Lu, "Understanding database performance inefficiencies in real-world web applications," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, ser. CIKM '17. ACM, 2017, p. 1299–1308. [Online]. Available: https://doi.org/10.1145/3132847.3132954

[7] P. Tözün, I. Pandis, C. Kaynak, D. Jevdjic, and A. Ailamaki, "From a to e: Analyzing tpc's oltp benchmarks: The obsolete, the ubiquitous, the unexplored," in *Proceedings of the 16th International Conference on Extending Database Technology*, ser. EDBT '13. ACM, 2013, p. 17–28. [Online]. Available: https://doi.org/10.1145/2452376.2452380

[8] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudré-Mauroux, "Oltp-bench: An extensible testbed for benchmarking relational databases," *PVLDB*, vol. 7, no. 4, pp. 277–288, 2013. [Online]. Available: http://www.vldb.org/pvldb/vol7/p277-difallah.pdf

[9] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "Rapl: Memory power estimation and capping," in *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, ser. ISLPED '10. ACM, 2010, p. 189–194. [Online]. Available: https://doi.org/10.1145/1840845.1840883

[10] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen, and Z. Ou, "Rapl in action: Experiences in using rapl for power measurements," *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 3, no. 2, mar 2018. [Online]. Available: https://doi.org/10.1145/3177754

[11] S. Desrochers, C. Paradis, and V. M. Weaver, "A validation of dram rapl power measurements," in *Proceedings of the Second International Symposium on Memory Systems*, ser. MEMSYS '16. ACM, 2016, p. 455–470. [Online]. Available: https://doi.org/10.1145/2989081.2989088

[12] E. Guégain, C. Quinton, and R. Rouvoy, "On reducing the energy consumption of software product lines," in *Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume A*, ser. SPLC '21. ACM, 2021, p. 89–99. [Online]. Available: https://doi.org/10.1145/3461001.3471142

[13] E. A. Santos, C. McLean, C. Solinas, and A. Hindle, "How does docker affect energy consumption? evaluating workloads in and out of docker containers," *Journal of Systems and Software*, vol. 146, pp. 14–25, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121218301456

[14] Z. Ournani, M. C. Belgaid, R. Rouvoy, P. Rust, and J. Penhoat, "Evaluating the impact of java virtual machines on energy consumption," in *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ser. ESEM '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3475716.3475774

[15] E. Barrett, C. F. Bolz-Tereick, R. Killick, S. Mount, and L. Tratt, "Virtual machine warmup blows hot and cold," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, oct 2017. [Online]. Available: https://doi.org/10.1145/3133876

[16] M. Kumar, Y. Li, and W. Shi, "Energy consumption in java: An early experience," in *2017 Eighth International Green and Sustainable Computing Conference (IGSC)*, 2017, pp. 1–8.

[17] R. Pereira, M. Couto, J. a. Saraiva, J. Cunha, and J. a. P. Fernandes, "The influence of the java collection framework on overall energy consumption," in *Proceedings of the 5th International Workshop on Green and Sustainable Software*, ser. GREENS '16. ACM, 2016, p. 15–21. [Online]. Available: https://doi.org/10.1145/2896967.2896968

[18] G. Pinto, K. Liu, F. Castor, and Y. D. Liu, "A comprehensive study on the energy efficiency of java's thread-safe collections," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 20–31.

[19] W. Oliveira, R. Oliveira, F. Castor, B. Fernandes, and G. Pinto, "Recommending energy-efficient java collections," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 160–170.

[20] G. Rocha, F. Castor, and G. Pinto, "Comprehending Energy Behaviors of Java I/O APIs," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2019, pp. 1–12.

[21] A. Noureddine and A. Rajan, "Optimising energy consumption of design patterns," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, 2015, pp. 623–626.

[22] D. Connolly Bree and M. Ó Cinnéide, "Energy efficiency of the visitor pattern: contrasting java and c++ implementations," *Empirical Software Engineering*, vol. 28, no. 6, p. 145, Oct 2023. [Online]. Available: https://doi.org/10.1007/s10664-023-10387-8

[23] Z. Ournani, R. Rouvoy, P. Rust, and J. Penhoat, "Tales from the code #1: The effective impact of code refactorings on software energy consumption," in *ICSOFT*. SCITEPRESS, 2021, pp. 34–46.

[24] ——, "Tales from the code #2: A detailed assessment of code refactoring's impact on energy consumption," in *ICSOFT (Selected Papers)*, ser. Communications in Computer and Information Science, vol. 1622. Springer, 2021, pp. 94–116.

[25] G. Procaccianti, P. Lago, and W. Diesveld, "Energy efficiency of orm approaches: An empirical evaluation," in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '16. ACM, 2016. [Online]. Available: https://doi.org/10.1145/2961111.2962586

[26] R. Verdecchia, R. A. Saez, G. Procaccianti, and P. Lago, "Empirical evaluation of the energy impact of refactoring code smells," in *ICT4S2018. 5th International Conference on Information and Communication Technology for Sustainability*, ser. EPiC Series in Computing, B. Penzenstadler, S. Easterbrook, C. Venters, and S. I. Ahmed, Eds., vol. 52. EasyChair, 2018, pp. 365–383. [Online]. Available: https://easychair.org/publications/paper/MxpT

[27] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Detecting performance anti-patterns for applications developed using object-relational mapping," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. ACM, 2014, p. 1001–1012. [Online]. Available: https://doi.org/10.1145/2568225.2568259

[28] ——, "Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks," *IEEE Transactions on Software Engineering*, vol. 42, no. 12, pp. 1148–1161, 2016.