Detecting Performance Anti-patterns for Applications Developed using Object-Relational Mapping

Tse-Hsun Chen[1], Weiyi Shang[1], Zhen Ming Jiang[2], Ahmed E. Hassan[1]
Mohamed Nasser[3], Parminder Flora[3]
Queen's University[1], York University[2], BlackBerry[3], Ontario, Canada
{tsehsun, swy, ahmed}@cs.queensu.ca[1], zmjiang@cse.yorku.ca[2]

ABSTRACT

Object-Relational Mapping (ORM) provides developers a conceptual abstraction for mapping the application code to the underlying databases. ORM is widely used in industry due to its convenience; permitting developers to focus on developing the business logic without worrying too much about the database access details. However, developers often write ORM code without considering the impact of such code on database performance, leading to cause transactions with timeouts or hangs in large-scale systems. Unfortunately, there is little support to help developers automatically detect suboptimal database accesses.

In this paper, we propose an automated framework to detect ORM performance anti-patterns. Our framework automatically flags performance anti-patterns in the source code. Furthermore, as there could be hundreds or even thousands of instances of anti-patterns, our framework provides support to prioritize performance bug fixes based on a statistically rigorous performance assessment. We have successfully evaluated our framework on two open source and one large-scale industrial systems. Our case studies show that our framework can detect new and known real-world performance bugs and that fixing the detected performance anti-patterns can improve the system response time by up to 98%.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; C.4 [**Performance Of Systems**]: [Performance Anti-patterns]

General Terms

Performance

Keywords

Performance, Performance Anti-pattern, Static, Dynamic Analysis, Performance Evaluation

ICSE '14, May 31 – June 7, 2014, Hyderabad, India

1. INTRODUCTION

Object-Relational Mapping (ORM) provides developers a conceptual abstraction for mapping database records to objects in object-oriented languages. Through such mapped objects, developers can access database records without worrying about the database access and query details. For example, developers can call `user.setName("Peter")` to update a user's name in a database table. As a result, ORM gives developers a clean and conceptual abstraction for mapping the application code to the database. Such abstraction significantly reduces the amount of code that developers need to write [1, 2].

ORM is widely used in practice as it significantly reduces the overhead of bridging business logic and database [3]. Since Java 5, Java has included a standard persistence API (called JPA) that supports ORM. There are many different implementations of JPA available, such as Hibernate [4] and Apache OpenJPA [5]. ORM's popularity is not only limited to Java, but also other programming languages such as C#, Ruby, and VB.Net provide ORM abstraction [3, 6].

Despite ORM's advantages, it may introduce potential performance problems. Developers may not be aware which source code snippets would result in a database access nor whether such access is inefficient. Thus, developers would not proactively optimize the database access performance. For example, code that is efficient in memory (e.g., loops) may cause database performance problems when using ORM due to data retrieval overheads. In addition, developers usually only test their code on a small scale (e.g., unit tests), while performance problems would often surface at larger scales and may result in transaction timeouts or even hangs. Therefore, detecting and understanding the impact of such potential performance overhead is important for developers [7].

In this paper, we propose an automated framework, which detects the ORM performance anti-patterns. Our framework can detect hundreds of instances of ORM performance anti-patterns based on static code analysis. Furthermore, to cope with the sheer number of the detected anti-patterns, our framework provides suggestions to prioritize bug fixes based on a statistically rigorous performance assessment (improvement in response time if the detected anti-patterns are addressed).

The main contributions of this paper are:

- This is the first work that proposes a systematic and automated framework to detect and assess performance anti-patterns for applications developed using ORM.
- Our framework provides a practical and statistically

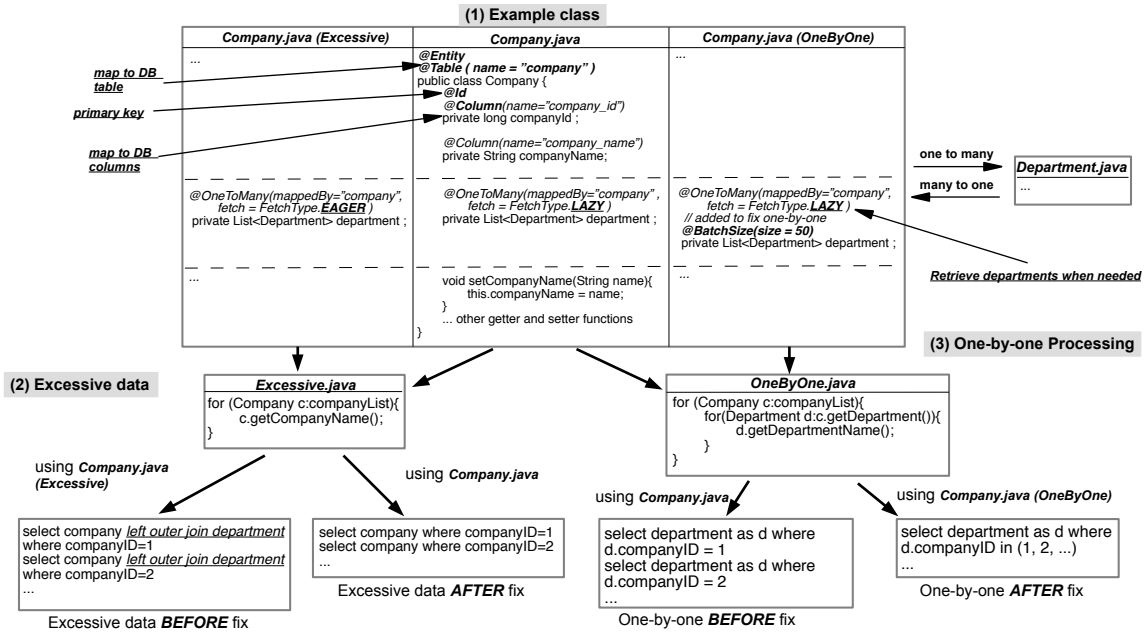

Figure 1: A motivating example. (1) shows the original class files and the ORM configurations; (2) shows the modified `Company.java`, the excessive data application code, and the resulting SQLs; and (3) shows the modified `Company.java`, the one-by-one processing application code, and the resulting SQLs.

rigorous approach to prioritize the detected performance anti-patterns based on the expected performance gains (i.e., improvement in response time). The prioritization of detected anti-patterns is novel relative to prior anti-pattern detection efforts ([8, 9, 10, 11, 12]).

- Through case studies on two open source systems (PetClinic [13] and BroafLeaf commence [14]) and one large-scale enterprise system (EA), we show that our framework can find existing and new performance bugs. Our framework, which has received positive feedbacks from the EA developers, is currently being integrated into the software development process to regularly scan the EA code base.

Paper Organization The rest of this paper is organized as follows. Section 2 provides motivating examples of the performance impact of anti-patterns. Section 3 describes our framework for detecting and prioritizing performance anti-patterns. Section 4 discusses the results of our case study. Section 5 provides some discussions on the detected anti-patterns and presents some future work. Section 6 describes the threats to validity. Section 7 surveys the related work. Finally, Section 8 concludes the paper.

2. MOTIVATING EXAMPLES

In this section, we present realistic examples to show how such ORM performance anti-patterns may affect the performance of a system. We develop a simple Java program as an illustration (Figure 1 (1)). The program manages a relationship between two classes (Company and Department), and provides a set of getter and setter functions to access and change the corresponding DB columns (e.g, setCompanyName).

In this example, there is a one-to-many relationship between Company and Department, i.e., one company can have multiple departments. This relationship is represented using annotation **@OneToMany** on the instance variable department in `Company.java` and **@ManyToOne** on the instance variable company in `Department.java` (details not shown in Figure 1 but very similar to `Company.java`). **@Column** shows which database column the instance variable is mapped to (e.g., companyId maps to column *company_id*), and **@Entity** shows that the class is a database entity class which maps to the database table specified in **@Table** (e.g., Company class maps to *company* table).

Fetch type in ORM, which can be either *LAZY* or *EAGER*, determines how the data of a java object is retrieved from the database. Using `Company.java` in Figure 1 as an example, a fetch type of *EAGER* means that department objects are always fetched from the database when a company object is initialized (i.e., calling *new* Company()) even when department objects might never be needed. On the other hand, a fetch type of *LAZY* means that department objects are retrieved only when their information is used (e.g., when calling department.getDepartmentName()).

In the following subsections, we discuss how the performance anti-patterns may affect system performance, and show the performance difference before and after fixing the anti-patterns. We focus on the following two performance anti-patterns that are commonly seen in real-world code [15, 16] and that can possibly cause serious performance problems: (1) *Excessive data*, which retrieves unused/unnecessary data from the database; and (2) *One-by-One Processing*, which repeatedly performs similar database operations in loops [17]. In this paper, we focus our study on JPA, which is a very popular ORM standard API for Java.

2.1 Excessive Data

Developers may set the relationship between two database entity classes to *EAGER* if the two classes are always associated together (e.g., accessing class Company will always lead to an access to class Department). In such cases, set-

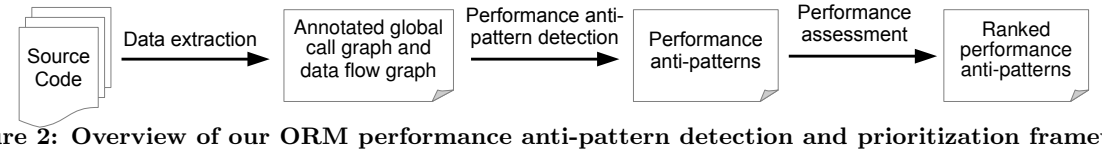

Figure 2: Overview of our ORM performance anti-pattern detection and prioritization framework.

ting the relationship to *EAGER* may improve performance by retrieving two database entity objects from the database in one SQL statement using a *join*. However, such an *EAGER* fetch is not optimal if the eagerly retrieved data is never used.

Figure 1 (2) shows an example of the excessive data anti-pattern. `Excessive.java` shows an application program that accesses the name of all the companies. If we execute `Excessive.java` using the ORM configurations in `Company.java (Excessive)`, it would generate SQL statements to retrieve both company objects as well as department objects from the database due to the *EAGER* setting on the department instance variable in `Company.java`, even though we do not access Department objects in `Excessive.java`. One way to fix this performance anti-pattern is to change the fetch type from *EAGER* to *LAZY* (the original setting in `Company.java`).

To demonstrate the performance impact of excessive data, we run the program shown in Figure 1 (2). We populate the database with 300 records in the Company table and 10 records of Department for each record in the Company table. The response time before fixing `Excessive.java` is 1.68 seconds; fixing the performance anti-pattern reduces the response time to 0.48 seconds (a 71% improvement).

2.2 One-by-one Processing

Figure 1 (3) shows an example of the one-by-one processing anti-pattern. The one-by-one processing anti-pattern is a special case of the *Empty Semi Trucks* anti-pattern [17], which occurs when a large number of requests is needed to perform a task. In this paper, we study the effect of such anti-pattern in the ORM context. `OneByOne.java` shows an application program that iterates through all the companies (companyList), and finds the names of all the departments in each company. If we execute `OneByOne.java` using the same ORM configurations in `Company.java`, it would generate one *select department* statement for *each* company object.

One way to solve this particular issue is to change the ORM configuration for retrieving department objects to the configuration in `Company.java (OneByOne)`. After adding a batch size (e.g., **@BatchSize**(size=50)) to the instance variable department, ORM will select 50 department objects in each batch. The fix reduces the number of SQL statements significantly, and could help improve the database performance. Note that the fix for one-by-one processing may vary in different situations (e.g., doing database updates in loops) and can be sometimes difficult. Section 6 has more discussions regarding this issue.

To demonstrate the performance impact of one-by-one processing, we run the program shown in Figure 1 (3) with the same populated database as in Section 2.1. The response time before fixing the anti-pattern is 1.68 seconds, and the response time after the fix is 1.39 seconds (a 17% improvement).

Our motivating examples show that there is a significant performance improvement even in simple sequential database read operations after fixing both performance anti-patterns.

3. OUR FRAMEWORK

This section describes our framework for detecting and prioritizing ORM performance anti-patterns. As shown in Figure 2, our framework consists of three phases: Data Extraction Phase, Performance Anti-pattern Detection phase and Performance Assessment phase. We first extract all the code paths, which involve database accesses. Then, we detect the performance anti-patterns among these database-accessing code paths. Finally, we perform a statistically rigorous performance assessment so that developers can prioritize the performance bug fixes among hundreds of anti-patterns. In the rest of this section, we explain these three phases in detail. We use the example in Section 2 to illustrate our approach.

3.1 Data Extraction

In this phase, we identify all the code paths, which involve database accesses. For each source code file, we extract the local call graphs, database-accessing functions, and ORM configurations (Section 3.1.1). Then, we combine the information from all the files and identify code paths (i.e., scenarios), which involve database accesses (Section 3.1.2).

3.1.1 Extracting Database-accessing Functions, Local Call Graphs, and ORM Configurations

We first extract the call graphs and data flows in each file, then we identify all ORM-related information such as annotations and configurations. As shown in Section 2, ORM uses annotations to annotate a Java class as a database entity class or to setup different data retrieving strategies and configurations. We store the information about the ORM configuration of each database entity class and database instance variable, and identify functions that could result in database accesses. Using Figure 1 as an example, we store the relationship between Company and Department as one-to-many. We also mark the getter and setter functions (e.g., setDepartmentName), which access or modify database instance variables as database-accessing functions.

3.1.2 Identifying Database-accessing Code Paths

We identify the code paths which involve database accesses. We accomplish this by:

1) Constructing Global Call and Data Flow Graphs: We build a global call graph for all the functions and we keep track of each object's data usage during its lifetime. Since functions in object-oriented languages are usually invoked through polymorphism, we connect the function call graphs using the corresponding functions in the subclasses for abstract classes or the implemented class for interfaces.

2) Marking Database-accessing Code Paths and Data Flows Using Taint Analysis: We use taint analysis, commonly used in computer security [18], to identify code paths and data flows, which involve database accesses. Taint analysis keeps track of a possibly malicious variable (e.g., variable V), and if V is used in an expression to manipulate another variable M, M is also considered suspicious. Similarly, if a function in a call path contains a database-accessing

function that is determined by the first step, we mark all the functions in the code path as database-accessing functions. For example, in a call path, function A calls function B and function B calls function C. If function C is a database-accessing function, we consider the code path of A $\rightarrow$ B $\rightarrow$ C as a database-accessing code path. We perform taint analysis statically by computing the node reachability across functions in the global function call graph.

3.2 Performance Anti-pattern Detection

With identified database-accessing code paths and data flows, we use a rule-based approach to detect ORM performance anti-patterns. Different anti-patterns are encoded using different rules. In this paper, we focus on detecting two of the most pervasive ORM anti-patterns, but new anti-patterns can be integrated to the framework by adding new detection rules. Section 5 has more discussions about extending our framework.

Detecting Excessive Data Anti-patterns: When a database entity object is initialized, a database call is invoked to retrieve the required information. Therefore, we use point-to analysis to first identify the variables that point to the database entity object. Then, we analyze the data flow of the object to detect excessive data. If a database entity object does not use any of the *eagerly* retrieved information during its lifetime (identified by traversing the data flow graph), we report the object usage as an instance of performance anti-pattern.

Detecting One-by-one Processing Anti-patterns: To detect one-by-one processing anti-patterns, the framework first identifies the functions that are directly called within both single and nested loops (we consider all kinds of loops, such as *for*, *while*, *foreach*, and *do while*). If a directly-called function is a database-accessing function, the framework simply reports it as an one-by-one processing anti-pattern. The framework also traverses all the child nodes in the function call graphs of the functions that are called in loops. We analyze the child nodes across multiple functions, and determine whether there is a node in the call graph that may access the database. If so, the framework reports the call path that contains the node as an instance of one-by-one processing anti-pattern.

Due to ORM configurations, some database-accessing functions may not always access the database. In such cases, we do not report them as anti-patterns. For example, if Department is eagerly retrieved by Company, retrieving Company will automatically retrieve Department. Therefore, accessing Department through Company in a loop will not result in additional database accesses, and will not be reported by our framework. If a database-accessing function in a loop is already being optimized using some ORM configurations (i.e., fetch plan is either ***Batch***, ***SubSelect***, or ***Join***) [19], we do not identify it as a performance anti-pattern.

3.3 Performance Assessment

Previous performance anti-pattern detection studies generally treat all the detected anti-patterns the same and do not provide methodologies for prioritizing the anti-patterns [8, 9, 10, 11, 12]. However, as shown in the case studies (Section 4), there could be hundreds or thousands of performance anti-pattern instances. Hence, it is not feasible for developers to address them all in a timely manner. Moreover, some instances of anti-patterns may not be worth fixing due to the high cost of the fix and the small performance improvement. For example, if a one-by-one processing anti-pattern always processes a table with just one row of data, there is little improvement in fixing this particular anti-pattern.

In this phase, we assess the performance impact of the detected anti-patterns through statistically rigorous performance evaluations. The assessment result may not be the actual performance improvement after fixing the anti-patterns, but may be used to prioritization the fixing efforts. We repeatedly measure and compare the system performance before and after fixing the anti-patterns by exercising the readily available test suites. Anti-patterns, which are expected to have big performance improvements (e.g., 200% performance improvement) after the fixes, will result in higher priorities. The rest of this subsection explains the three internal parts in performance assessment phase: (1) exercising performance anti-patterns and calculating test coverage, (2) assessing the performance improvement after fixing the anti-patterns, and (3) statistically rigorous performance evaluations.

Part 1 - Exercising performance anti-patterns and calculating test coverage

One way to measure the impact of performance anti-patterns is to evaluate each anti-pattern individually. However, since system performance is highly associated with run-time context and input workloads [12, 20, 21], we need to assess the impact of performance anti-patterns using realistic scenarios and workloads.

Since performance anti-patterns are detected in various software components, it is difficult to generate workflows to exercise all the performance anti-patterns manually. Therefore, we use the readily available performance test suites to exercise the performance anti-patterns. For the systems that do not have performance test suites, we use integration test suites as an alternate choice. Although integration test suites may not be designed for testing performance critical parts of a system, they are designed to test various *features* in a system (i.e., use case tests), which may give a better test coverage. In short, performance and integration test suites group source code files that have been unit tested into larger units as suites (e.g., features or business logic), which better simulate system workflows and user behaviours [22].

Since we do not have control over which instances of performance anti-patterns are exercised by executing the test suites, it is important to know how many performance anti-patterns are covered by the tests. We use a profiler to profile the code execution paths of all the tests, and use the execution path to calculate how many instances of performance anti-patterns are covered in the tests.

Part 2 - Assessing performance improvement after fixing anti-patterns

In this part, we describe our methodology to assess the performance improvement of fixing the excessive data and one-by-one processing anti-patterns.

Assessing the excessive data anti-patterns: As mentioned in Section 2, the eager-fetch setting may be necessary in some situations to improve performance, but may cause performance degradation when the retrieved data is not used. We fix excessive data anti-patterns by fixing the source code (i.e., change the fetch type of database entity objects from *EAGER to* LAZY where appropriate). Finally, we measure and compare the system performance before and after fixing

the excessive data anti-patterns.

Assessing the one-by-one processing anti-patterns: Fixing one-by-one processing can be much more complicated than what we have shown in Section 2. One common fix to one-by-one processing is using batches. However, for example, performing a batch insert to the *Department* table in ORM may require writing specific ORM SQL queries, such as "*insert into Department (name) values ('department1'), ('department2') ...* ", to replace ordinary ORM code. As a result, manually fixing the anti-patterns of one-by-one processing requires a deep knowledge about a system's structure, design and APIs. Due to the complexity, it is very difficult to fix all the detected one-by-one processing anti-patterns automatically. Therefore, we follow a similar methodology by Jovic *et al.* [23] to assess the anticipated system performance after fixing the one-by-one processing anti-patterns.

As shown in Section 2, the slow performance in the one-by-one processing anti-patterns is mainly attributed to the inefficiency in the generated SQL statements. The one-by-one processing anti-patterns in ORM generate repetitive SQL statements with minor differences. The repetitive SQL statements can be optimized using batches, which reduce a large amount of query preparation and transmission overheads. Therefore, the performance measure for executing the optimized SQL statements could be a good estimate for the anticipated system performance after fixing the one-by-one processing anti-patterns. To obtain the generated SQL statements, we use a SQL logging library called *log4jdbc* [24] to log the SQL statements and SQL parameter values. *log4jdbc* simply acts as an intermediate layer between JDBC to the database, which relays the SQL statements to the database and outputs the SQL statements to log files.

We detect the repetitive SQL statements generated by ORM, and execute the SQL statements in batches using Java Database Connectivity (JDBC), which assess the performance impact after fixing the performance anti-pattern. Note that since JDBC does not support batch operations for *select*, we exclude *select* in our assessment. We execute the original non-optimized and optimized SQL statements separately and compare the performance differences in terms of response time.

Part 3 - Statistically rigorous performance evaluation

Performance measurements suffer from instability, and may lead to incorrect results if not handled correctly [25, 26, 27]. Thus, a rigorous study is required when doing performance measurements [25]. Therefore, our framework does repeated measurement and computes effect sizes of the performance improvement (i.e., quantifies the increase in response time statistically) to overcome the instability problem.

Repeated measurements: Georges *et al.* [26] recommend computing a confidence interval for repeated performance measurements when doing performance evaluation, since performance measurements without providing measures of variation may be misleading and incorrect [25]. We repeat the performance tests > 30 times (as suggested by Georges *et al.*), and record the response time before and after doing fixes. We use *Student's t-test* to examine if the improvement is statistically significant different (i.e., p-value < 0.05). A p-value < 0.05 means that the difference between two distributions is likely not by chance. Then, we compute the mean response time and report the 95% confidence interval. A *t-test* assumes that the population distribution is normally distributed. According to the central limit theorem our performance measures will be approximately normally distributed if the sample size is large enough [26, 28].

Effect size for measuring the performance impact: We conduct a rigorous performance improvement experiment by using *effect sizes* [29, 30]. Unlike *t-test*, which only tells us if the differences of the mean between two populations are statistically significant, effect sizes quantify the difference between two populations. Researchers have shown that reporting only the statistical significance may lead to erroneous results [30] (i.e., if the sample size is very large, p-value can be small even if the difference is trivial).

We use *Cohen's d* to quantify the effects [30]. *Cohen's d* measures the effect size statistically, and has been used in prior engineering studies [30, 31]. *Cohen's d* is defined as:

$$\textit{Cohen's } d = \frac{\bar{x}_1 - \bar{x}_2}{s}, \tag{1}$$

where $\bar{x}_1$ and $\bar{x}_2$ are the mean of two populations, and s is the pooled standard deviation [32].

The strength of the effects and the corresponding range of *Cohen's d* values are[33]:

$$\text{effect size} = \begin{cases} \text{trivial} & \text{if } \textit{Cohen's } d \leq 0.2 \\ \text{small} & \text{if } 0.2 < \textit{Cohen's } d \leq 0.5 \\ \text{medium} & \text{if } 0.5 < \textit{Cohen's } d \leq 0.8 \\ \text{large} & \text{if } 0.8 < \textit{Cohen's } d \end{cases}$$

We output the performance anti-patterns, ranked by their effect sizes, in an HTML report. This report has two parts: (1) the database entity objects that have excessive data; (2) the database-assessing functions and the code path of the detected one-by-one processing anti-patterns.

4. CASE STUDY

In this section, we apply our framework on two open source systems (PetClinic and BroafLeaf commence) and one large-scale closed-source enterprise system (EA). We seek to answer the following two research questions:

RQ1: What is the performance impact of the detected anti-patterns?

RQ2: How do performance anti-patterns affect system performance at different input scales?

Each research question is organized into three sections: Motivation, Approach and the Results. Table 1 shows the statistics of the studied systems.

All of our studied systems use JPA for ORM and follow the "Model-View-Controller" design pattern [34]. Pet Clinic is a sample open source system developed by the Spring Framework [35], which aims to provide a simple yet realistic design of a web application. Broadleaf is a large open-source e-commerce system that is used in many commercial companies worldwide for building online transaction platforms. EA supports a large number of users concurrently and is used by millions of users worldwide on a daily basis. We sought to use open source systems, in addition to the commercial system, so others can verify our findings and replicate our experiments on the open source systems, as we are not able to provide access to the EA.

Table 2: Performance assessment result for excessive data and one-by-one processing. Tests with p-value < 0.05 have statistically significant performance improvement (marked in bold). Numbers in the parentheses are the percentage reduction in response time.

(a) Excessive Data

System	Test Case Description	No. Excessive Data Covered	μ Before (sec)	μ After (sec)	Statistical Significance (p-value)	Effect size
Pet Clinic	**Browsing**	**2**	**130.09±2.04**	**2.04±0.05 (-98%)**	**<<0.001**	**42.41 (large)**
Broadleaf	Customer Phone	36	2.17±0.21	2.17±0.36 (0%)	0.99	0.00 (trivial)
	Offer Service	30	2.38±0.50	2.1±0.19 (-25%)	0.37	0.23 (small)
	Shopping Cart	**29**	**22.90±1.74**	**20.14±0.48 (-12%)**	**<0.05**	**0.78 (medium)**
	Checkout	**69**	**30.75±0.27**	**25.75±0.29 (-16%)**	**<<0.001**	**6.45 (large)**
	Customer Addr.	39	3.85±0.17	3.68±0.31 (-4%)	0.35	0.23 (small)
	Customer	28	2.08±0.39	1.95±0.33 (-6%)	0.62	0.12 (trivial)
	Order	**57**	**30.60±0.35**	**25.62±0.31 (-16%)**	**<<0.001**	**5.39 (large)**
	Offer	24	1.16±0.39	0.92±0.28 (-21%)	0.30	0.21 (small)
	Payment Info	33	2.18±0.30	2.20±0.42 (+1%)	0.93	0.02 (trivial)
EA	**Multiple Features**	**> 10**	—	**improved by 5%**	**< 0.05**	**0.68 (medium)**

(b) One-by-one Processing

System	Test Case Description	No. One-by-one Processing Covered	μ Before (sec)	μ After (sec)	Statistical Significance (p-value)	Effect size
Broadleaf	Customer Phone	61	1.00±0.30	1.09±0.33(-0.09%)	0.68	0.10 (trivial)
	Offer Service	67	1.23±0.47	1.08±0.34(-12%)	0.60	0.13 (trivial)
	Shopping Cart	**52**	**21.46±1.64**	**14.58±0.42(-32%)**	**<<0.001**	**2.07(large)**
	Checkout	**109**	**13.25±0.30**	**10.63±0.66(-20%)**	**<<0.001**	**1.86 (large)**
	Customer Addr.	**61**	**1.49±0.33**	**1.08±0.10(-27%)**	**<0.05**	**0.59 (medium)**
	Customer	61	1.11±0.42	0.95±0.32(-15%)	0.54	0.15 (trivial)
	Order	**104**	**13.12±0.27**	**10.20±0.18(-22%)**	**<<0.001**	**4.54 (large)**
	Offer	62	1.23±0.56	0.95±0.25(-22%)	0.37	0.23 (small)
	Payment Info	61	1.08±0.31	1.16±0.38(+8%)	0.74	0.08 (trivial)
EA	**Multiple Features**	> 10	—	**improved by 69%**	**<<0.001**	**55.3 (large)**

Table 1: Statistics of the studied systems and number of detected anti-pattern instances.

	Total lines of code (K)	No. of files	No. of 1-by-1 processing	No. of excessive data
Pet Clinic	3.3K	51	0	10
Broadleaf 3.0	206K	1,795	228	483
EA	> 300K	> 3,000	>10	>10

RQ1: What is the performance impact of the detected anti-patterns?

Motivation: System performance is highly associated with run-time context and input workloads [12, 20, 21]. Rarely or never executed anti-patterns would have less performance impact compared to frequently-executed ones. Therefore, we use test suites to assess the performance impact instead of executing the anti-patterns individually. In this research question, we want to detect and prioritize performance anti-patterns by exercising different features of a system using our proposed framework (Section 3).

Approach: Since performance problems are usually revealed under large data sizes [20, 36], we manually increase the data sizes in these test suites and write a data loader for loading data into the database. Our framework repeatedly exercises the test suites and computes the effect sizes of the performance impact. Moreover, we measure the performance impact before and after fixing all the anti-patterns in each test suite, separately. We use the percentage reduction in response time, statistical significance of such reduction, and effect size to measure performance impact (i.e., whether there is an actual difference and how large the effect is).

Results:

Anti-pattern detection results: Table 1 shows the anti-pattern detection result. For Pet Clinic, our framework does not find any one-by-one processing anti-patterns. However, our framework does find 10 instances of excessive data anti-pattern. In particular, one of these 10 instances is also discussed online by developers and it is shown to cause serious performance degradation [15]. By using our framework, this performance problem can be detected in the early stages of testing. In Broadleaf, our framework detected a total of 308 instances of one-by-one processing anti-pattern and 483 instances of excessive data anti-pattern. Since a large number of anti-pattern instances is detected, we only emailed the top 10 instances of high impact performance anti-patterns to the developers. We are currently waiting for their reply. Due to non-disclosure agreement (NDA), we cannot present the exact numbers of detected anti-patterns in EA. However, we can confirm that our framework is able to detect many of the existing and new performance problems in EA.

Performance benefits of removing excessive data: Table 2a shows the performance impact of excessive data in each test suite. Each test suite may cover multiple and overlapping test cases. Overall, excessive data anti-patterns have a statistically significant performance impact and the effects are at least from medium to large ($0.68 - 42.41$) in 5 out of 11 test suites.

The only test suite in Pet Clinic, which covers two instances of excessive data anti-pattern, is related to browsing information about different pet owners, pets, and pets' visits to the clinic. The performance impact of the excessive data anti-patterns is very large (*Cohen's d* is 42.41), and fixing the anti-patterns can improve the response time by 98%.

We execute all Broadleaf test suites that contain instances of excessive data anti-pattern. The anti-patterns cause a significant performance impact in 33% of the test suites. Ex-

cessive data in three test suites has a medium to large statistical significant performance impact (effect sizes is 0.78–6.45, and response time is improved by 12–16% after fixing the anti-patterns). However, the impact of excessive data anti-patterns in six test suites are trivial and are not statistical significant. It is interesting to see that these test suites have a small (≈1–3 seconds) and similar response time before and after fixing the performance anti-patterns, which implies that these anti-pattern instances have a very low performance impact.

We find that fixing excessive data anti-patterns in EA gives a statistically significant improvement and can improve the response time by 5% (a medium effect size of 0.68).

We manually investigate how excessive data anti-patterns affect the performance in all three systems, because the performance improvements vary considerably among the studied systems. We find that the differences in database schema may introduce a significant performance impact on excessive data anti-pattern. In Pet Clinic, we discover that when viewing the names of each owner's pet (i.e., retrieve `pet` objects from the database), the system eagerly retrieves every single visit of a pet (i.e., *one-to-many* relationship). As a result, many *selects* are executed to retrieve information about all pet's visits, which is not needed for displaying owner and pets' name. On the other hand, most instances of the excessive data anti-pattern in Broadleaf are *one-to-one* (e.g., one product has only one stack keeping unit) or *many-to-one* (e.g., multiple payments are done by one customer), which have a lower performance impact.

Performance benefits of removing one-by-one processing. Table 2b shows the performance impact of one-by-one processing in each test suite, and the assessed performance impact. Table 2b does not show Pet Clinic since our framework does not detect any instance of one-by-one processing.

In Broadleaf, one-by-one processing anti-patterns have a statistically significant performance impact in 4 out of the 9 test suites, and the effect sizes are at least medium (0.59–4.54). The assessed response time reduction is from 20–32%.

One-by-one processing anti-patterns have a non-statistical significant impact in other test suites. Similar to our findings in excessive data anti-patterns, these test suites have one common behaviour: very short response time. The results indicate that when the response time of a program is small, adding batches will not give much improvement. This also shows that not all anti-patterns are worth fixing.

Although we cannot show the mean response time and confidence interval in EA, the assessed response time reduction is high (69%) and the effect size is large (55.3). By using batch operations, the performance of EA was improved significantly.

Our performance assessment results show that performance anti-patterns have a statistically significant performance impact in 5/11 (excessive data) and 5/10 (one-by-one processing) test suites with effect sizes varying from medium to large. We find that fixing the performance anti-patterns may improve the response time by up to 98% (and on average by 35%).

Table 3: Performance assessment result for different scales of data sizes. We do not show the effect size for the tests where the performance improvements are not statistically significant (i.e., p-value >= 0.05).

(a) Excessive Data

System	Test Case Description	Effect Sizes for Different Input Sizes		
		small	medium	large
Pet Clinic	Browsing	16.91	27.94	42.41
Broadleaf	Shopping Cart	–	0.52	0.78
	Checkout	1.12	1.17	6.44
	Order	0.92	2.36	5.39
EA	Multiple Features	–	0.76	0.68

(b) One-by-one Processing

System	Test Case Description	Effect Sizes for Different Input Sizes		
		small	medium	large
Broadleaf	Shopping Cart	0.88	1.77	2.08
	Checkout	–	0.55	1.86
	Customer Addr.	–	0.51	0.59
	Order	–	1.46	4.54
EA	Multiple Features	16.2	21.8	55.3

RQ2: How do performance anti-patterns affect system performance at different input scales?

Motivation: In RQ1, we manually change the data sizes to large to study the impact of performance anti-patterns. However, populating large volumes of data into the database requires a long time, and a database expert is needed to ensure that the generated data satisfies all the database schema requirements [37]. In this research question, we study whether we still have the same prioritization ranking of the performance anti-patterns using smaller data sizes.

Approach: We focus only on the test suites which yield anti-patterns with statistical significant performance impact in RQ1. We manually modify the test suites to change the data sizes to medium and small as opposed to big data sizes in RQ1. We reduce the data sizes by a factor of 2 at each scale (e.g., medium data size is 50% of large data size and small data size is 50% of the medium data size). Finally, we re-run the performance tests to study how the performance impact and effect size change at smaller scales.

Results:

Excessive data. Table 3a shows the performance impact assessment result of excessive data at different scales. In Pet Clinic, the performance impact is statistically significant and the effect is large at all different scales. When the data size is small, the effect size is the smallest (16.91) among the three different data sizes. When the data size is medium the effect size becomes 27.94, which is significantly smaller than the effect size calculated using the large data size (42.41). In addition, the effect size increases along with the input size, which implies that identified performance anti-patterns in Pet Clinic may cause scalability problems.

In Broadleaf, excessive data anti-patterns in almost all test suites have a statistically significant performance impact in all three scales (effect size 0.52–42.41). The result implies that these anti-patterns can still be revealed using a smaller scale of input data.

The excessive data anti-pattern in EA has a statistically significant impact when the data sizes are medium and small, and the effect sizes are both medium (0.76 and 0.68). However, we find hat the effect size is slightly smaller from the large data size compared to that of the medium data size. The reason for having a smaller effect size in larger data size

is that some excessively retrieved objects have a *many-to-one* relations (e.g., multiple departments may belong to the same company), which only need to be retrieved once and cached. *Many-to-one* excessive data anti-pattern may cause large performance impact if the eagerly retrieved data is large (large transmission overheads [38]). Thus, our framework reports all the detected anti-pattern instances and annotates the relationship (e.g., *many-to-one* or *one-to-many*), so developers can determine the necessary action to them.

One-by-one processing. Table 3b shows the performance impact assessment of one-by-one processing at different scales. In general, one-by-one processing anti-patterns in Broadleaf have a higher performance impact (i.e., larger effect sizes) when the data sizes increase, because the data sizes directly affect the number of iterations in loops. However in most test suites, these one-by-one processing anti-patterns still have a statistically significant impact at smaller scales. We find that only three test suites do not have a significant performance impact when the data size is small, but all test suites have a significant performance impact when the data size is medium (effect size 0.51–1.77). We find a similar trend in EA, where the effect size increases as data size increases. We can still identify performance problems in these test suites using small to medium data sizes.

We find that the priority of the performance anti-patterns at different scales is consistent, i.e., the rank of the effect sizes in different test suites is consistent across different input data scales. For example, the rank of the effect sizes for test suites using medium and large data sizes is the same except for the ranks of the Shopping cart test and the Order test, which are swapped. As a result, if the generation of large dataset takes too long or takes too much effort, we are still very likely to reproduce the same set of severe performance problems using a smaller dataset.

We find that the prioritization of performance anti-patterns when exercised on medium scale data size is very similar to the large data size. This results show that developers may not need to deal with all the difficulties of populating large data into the database to reveal these performance problems.

5. DISCUSSION

In this section, we discuss the accuracy of our performance assessment, the distribution of our performance anti-patterns across the studied systems, extensions of our framework, and initial developer feedbacks.

The accuracy of our one-by-one processing performance assessment methodology. In our performance assessment methodology for one-by-one processing, we measure the response time of the original non-optimized and optimized SQL statements instead of directly fixing the code. To study the accuracy of this methodology, we develop a simple program with a known one-by-one processing anti-pattern (example in Section 2) for evaluation.

We create a database with 100,000 department names in all companies and verify using the example in Figure 1 (3). We fix the one-by-one processing pattern in the code by writing SQL statements using JPA-specific query language. The original code takes about 24.67 ± 0.84 seconds. Fixing the code results in a mean response time of 5.36 ± 0.06 seconds, and the assessment shows a mean response time of 10.42 ± 0.09 seconds. The experiment shows that our assessment methodology may be an over-estimate but can achieve a comparable performance improvement (78.3% v.s. 57.8%) to assess the impact of the anti-patterns. In the future, we plan to investigate automated performance refactoring approach for fixing the one-by-one processing anti-patterns.

Table 4: Skewness of performance anti-pattern instances found in the studied systems.

System	Excessive Data	One-by-one Processing	All Anti-patterns
Pet Clinic	0.9	—	0.9
Broadleaf	4.5	7.0	5.2
EA	10.4	7.0	9.0

Distribution of the detected performance anti-pattern instances. In Section 4, we studied how the impact of the performance anti-patterns changes with different scales of input data. However, we are not sure how these instances of anti-patterns are distributed across different software packages. The distribution of the anti-pattern instances may affect the allocation of QA resources. It would be challenging to manually review and verify these performance anti-pattern instances if they were found evenly in all software packages, since more general knowledge about the whole system would be needed. On the other hand, if the anti-pattern instances are mostly found within a few packages, database experts can put more QA resources on verifying them in these packages.

We analyze the anti-pattern detection results that we obtained from the case study, and count the number of anti-pattern instances in each software package. We plot the Cumulative Density Function (CDF) of number of total performance anti-pattern instances (both excessive data and one-by-one processing) in all the software packages. CDF can be used to explore the distribution of data, and provide observation such as "70% of the packages have at most one performance anti-pattern". More formally, CDF shows the probability that the value of a random variable will be less than or equal to a given value in a probability distribution.

We also compute the skewness [39] on the number of anti-pattern instances in all packages. If the skewness is larger than 1, the distribution is highly skewed; if the skewness value is between 0.5 and 1, the distribution is moderately skewed; and if the skewness value is less than 0.5, the distribution approximately symmetric [39].

Table 4 shows the skewness of the performance anti-pattern instances. We find that the anti-patterns are highly skewed in all studied systems (0.9–10.4). The skewness for excessive data anti-patterns is larger than 0.5 in Pet Clinic and larger than 1 in the other systems – implying that excessive data anti-patterns are found mostly in a few packages; and so are one-by-one processing anti-patterns (skewness 7.0 in Broadleaf and EA). The skewness of the total number of anti-pattern instances (combining excessive data and one-by-one processing) is also high (0.9–9.0). Since we want to compare the distribution of the number of anti-pattern instances in packages, we normalize the anti-pattern instance counts in Figure 3. Figure 3 (EA is excluded due to NDA) shows that most software packages do not have instances of performance anti-patterns (e.g., more than 60% of the packages in Broadleaf have almost zero performance anti-pattern instance), and only about 15–35% of the packages in Broadleaf have anti-pattern instances. As a result, developers and database experts can focus on reviewing the code

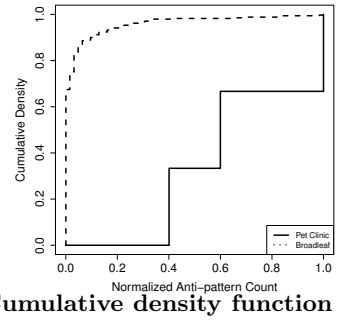

Figure 3: Cumulative density function (CDF) plots of number of performance anti-pattern instances found in packages. The x-axis shows the normalized number of performance anti-patterns in a package, and the y-axis shows the cumulative density.

Table 5: Names of the top 3 packages containing most performance anti-patterns.

System	Package Name	No. Excessive Data	No. One-by-one
Pet Clinic	service	5	—
	web	3	—
	model	2	—
Broadleaf	core.order.domain	34	28
	core.order.service	20	41
	core.offer.service.processor	10	41

and verifying the anti-pattern instances in a small number of packages, without needing whole-system knowledge.

We want to understand which packages tend to have more performance anti-pattern instances. We rank the packages by the number of performance anti-pattern instances (both excessive data and one-by-one processing together) contained in each package, and list the names in Table 5. We exclude EA in this study due to NDA. We find that packages related to handling events (e.g., the service and service.processor packages), data model (e.g., the model and domain packages), and GUI (the web package) contain more performance anti-pattern instances in these systems. In the future, we plan to study the root cause of the performance anti-patterns in these packages and how to avoid them.

Framework extension. In this paper, we proposed a rule-based approach, which detects and prioritizes two of the most pervasive ORM performance anti-patterns. Similar as any other pattern detection work, our framework cannot detect unseen performance anti-patterns. However, our framework could easily be extended by encoding other performance anti-patterns. In the future, we can plan to add new detection rules and apply the rules on the global call and data flow graphs to find new performance anti-patterns.

Initial Developer Feedback. We have received positive feedback from developers and performance testers in EA. The framework is able to help them narrow down the performance problems and find potential bottlenecks. We are currently working closely with the developers in EA for integrating the framework in their daily development processes.

6. THREATS TO VALIDITY

In this section, we discuss the threats to validity.

External Validity

We have only evaluated our framework on three systems. Some of the findings (e.g., which system components may have more performance anti-patterns) might not be generalizable to other systems. Although the studied systems vary in sizes and domains, other similar systems may have completely different results. Future work should apply our framework to more systems and even different programming languages (e.g., C#).

Construct Validity

Detection approach. We use static analysis for detecting performance anti-patterns. However, static analyses generally suffer from the problem of false positives, and our framework is no exception. For example, a detected performance anti-pattern may be seldom or never executed due to reasons like unrealistic scenarios or small input sizes. Therefore, we provide a performance assessment methodology to verify the impact and prioritize the fixing of the detected performance anti-patterns.

Experimental setup. We exercise the performance anti-patterns using either performance or integration test suites, so we do not have control over which performance anti-patterns will be exercised. As a result, our performance impact assessment study only applies to the exercised performance anti-patterns. However, our performance impact assessment methodology is general, and can be used to discover the impact of performance anti-patterns in different software components and to prioritize QA effort.

We manually change the data sizes to study how the impact of performance anti-patterns changes in different scales. Changing the data loader in the code and loading data in the database require a deep understanding of the system's structure and design of each test suite. Although we studied the code in each test suite and database schemas carefully, it is still likely that we did not change the inputs that are directly associated with the performance anti-patterns, or that the data does not generate representative workloads. However, case studies show that we can still flag a similar set of high impacting performance anti-patterns using different sizes of database.

Fixing performance anti-patterns and performance assessment. Fixing some performance anti-patterns may require API breaks and redesign of the system. Therefore, fixing them may not be an easy task. For example, to achieve maximal performance improvement, sometimes it is necessary to write SQL statements in ORM [40]. If the anti-pattern is generating many small database messages, which cause transmitting overheads and inefficient bandwidth usage, the solution is to apply batching [41]. In addition, different implementations of ORM support different ways to optimize performance. As a result, we provide a performance assessment methodology for assessing the performance impact. We use a similar methodology as Jovic *et al.* [23] to measure the performance impact of one-by-one processing, and we manually fix the excessive data anti-patterns in the code. Although our performance assessment methodology may not give the exact performance improvement and there may be other ways to fix the performance anti-patterns, we can still use the assessment result to prioritize the manual verification and performance optimization effort. We can further reduce the overheads of running the performance assessment approach using the result of our static analysis, and focus only on the parts of the system that are prone to performance anti-patterns.

It is possible that the performance fixes may have contradicting result in different use cases. For example, in some

cases using a fetch type of *EAGER* may yield a better performance, but may yield performance degradation in other cases. However, since ORM provides a programming interface to change the configuration and fetch plans dynamically, the problem can be solved by developers easily.

7. RELATED WORK

In this section, we discuss the following two areas of related research: (1) design-level performance anti-patterns and (2) detecting performance bugs.

Design-level performance anti-patterns. Most prior studies focus on discussing and detecting design-level performance anti-patterns. Smith and Williams [42] discuss design-level anti-patterns that may cause performance impacts, and provide solutions on how to refactor the design to eliminate the performance problem. In their followup work, Smith and Williams [41] document the problem and possible solution of another three performance anti-patterns. One of the anti-patterns discussed in their paper, called *Empty Semi Trucks*, is related to the one-by-one processing anti-pattern discussed in this paper. *Empty Semi Trucks* occurs when a large number of excessive requests is performed for a given task, such as retrieving information from the database. Our paper, on the other hand, focuses on detecting a variant of *Empty Semi Trucks* using static analysis, and we propose a framework that can automatically assess the performance impact of the detected anti-patterns. Nijjar *et al.* [43] extract a formal data models from the ORM specification of system, and develop heuristics to discover anti-patterns in the data model. Their framework can then automatically propose solutions to correct the data models. Cortellessa *et al.* [44] discuss various approaches of detecting design-level performance anti-patterns using software modelling and design-change rules.

Most prior studies on performance anti-patterns aim to improve the architecture and class design; where our work focuses on how developers write ORM code. To the best of our knowledge, our paper is the first work that aims to detect software performance anti-patterns using static analysis and provide a performance assessment automatically for systems that are developed using ORM.

Performance bug detection. Prior studies propose various approaches to detect different performance bugs through run-time indicators of such bugs.

Parsons *et al.* [10] present an approach for automatically detecting performance issues in enterprise applications built using component-based frameworks. Parsons *et al.* detect performance issues by reconstructing the run-time design of the system using monitoring and data mining approaches. Chis *et al.* [9] provide a tool to detect memory anti-patterns in Java heap dumps using a catalogue. Nistor *et al.* [8] propose a performance bug detection tool which detects performance problems by finding similar memory-access patterns during system execution. Nistor *et al.* report code loops with repetitive computation and partially similar memory-access patterns. Tamayo *et al.* [45] construct the program dependency graph using dynamic information flow, and combine the information with the corresponding database operations to identify performance bottlenecks. Their tool may also be used to identify problems related to batching, SQL synchronization, repetitive SQL queries, and extra operations.

Xu *et al.* [11] introduce copy profiling, an approach that summarizes runtime activity in terms of chains of data copies, which are indicators of Java runtime bloat (i.e., many temporary objects executing relatively simple operations). Xu *et al.* [46] introduce a run-time analysis to identify low-utility data structures where the involve costs that are out of line with the gained benefits. Xiao *et al.* [12] use different workloads to identify and predict workload-dependent performance bottlenecks (i.e., performance bugs) in GUI applications. Grechanik *et al.* [47] develop an approach for detecting database deadlocks with high degrees of automation. In another work, Grechanik *et al.* [48] combine dynamic analysis and static code analysis to prevent database deadlocks.

Most of the aforementioned studies typically detect performance bugs during the execution of the software, which may not cover all the code paths depending on the input workflow. In addition, generating workflows that exercise different components of a system requires good domain knowledge and can be very time consuming. In this paper, we provide a framework to statically identify performance anti-patterns by analyzing the source code of the system. The advantage of using static code analysis is that we can identify parts of the code that are not executed by the input workflow but may still contain performance bugs. In addition, we focus on ORM performance anti-patterns, which may not be captured by detecting performance anti-pattern using memory access (e.g., [8, 9, 10]). Moreover, we provide a statistical rigorous performance assessment approach that can be used to evaluate the impact of the detected performance anti-patterns, which is generally missing in prior studies.

8. CONCLUSION

Object-Relational Mapping (ORM) provides a conceptual abstraction between the application code and the database. ORM significantly simplifies the software development by automatically translating object accesses and manipulations to database queries. Developers can focus on business logic instead of worrying about non-trivial database access details. However, such mappings might lead to performance anti-patterns causing transactions timeout or hangs in large-scale software systems.

In this paper, we propose a framework, which can detect and prioritize instances of ORM performance anti-patterns. We applied our framework on three software systems: two open-source and one large enterprise systems. Case studies show that our framework can detect hundreds or thousands instances of performance anti-patterns, while also effectively prioritizing the fixing of these anti-pattern instances using a statistically rigorous approach. Our static analysis result can further be used to guide dynamic performance assessment of these performance anti-patterns, and reduce the overheads of profiling and analyzing the entire system. We find that fixing these instances of performance anti-patterns can improve the systems' response time by up to 98% (and on average by 35%).

Acknowledgements

We are grateful to BlackBerry for providing data that made this study possible. The findings and opinions in this paper belong solely to the authors, and are not necessarily those of BlackBerry. Moreover, our results do not in any way reflect the quality of BlackBerry software products.

9. REFERENCES

[1] Douglas Barry and Torsten Stanienda. Solving the java object storage problem. *Computer*, 31(11):33–40, November 1998.

[2] Neal Leavitt. Whatever happened to object-oriented databases? *Computer*, 33(8):16–19, August 2000.

[3] R. Johnson. J2ee development frameworks. *Computer*, 38(1):107–110, 2005.

[4] JBoss Community. Hibernate. http://www.hibernate.org/, 2013.

[5] Apache Software Foundation. Apache openjpa. http://openjpa.apache.org/, 2013.

[6] L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly Media, 2008.

[7] Weiyi Shang, Zhen Ming Jiang, Hadi Hemmati, Bram Adams, Ahmed E. Hassan, and Patrick Martin. Assisting developers of big data analytics applications when deploying on hadoop clouds. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 402–411, 2013.

[8] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: detecting performance problems via similar memory-access patterns. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 562–571, 2013.

[9] Adriana E. Chis. Automatic detection of memory anti-patterns. In *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA Companion '08, pages 925–926, 2008.

[10] Trevor Parsons and John Murphy. A framework for automatically detecting and assessing performance antipatterns in component based systems using run-time analysis. In *The 9th International Workshop on Component Oriented Programming*, WCOP '04, 2004.

[11] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, and Gary Sevitsky. Software bloat analysis: finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, pages 421–426, 2010.

[12] Xusheng Xiao, Shi Han, Dongmei Zhang, and Tao Xie. Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 90–100, 2013.

[13] Spring PetClinic. Petclinic. https://github.com/SpringSource/spring-petclinic/, 2013.

[14] Broadleaf Commerce. Broadleaf commerce. http://www.broadleafcommerce.org/, 2013.

[15] Julie Dubois. Improving the performance of the spring-petclinic sample application. http://blog.ippon.fr/2013/03/14/improving-the-performance-of-the-spring-petclinic-sample-application-part-4-of-5/, 2013.

[16] Patrycja Wegrzynowicz. Performance anti-patterns in hibernate. http://www.devoxx.com/display/DV11/Performance+Anti-Patterns+in+Hibernate, 2013.

[17] C.U. Smith and L.G. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. The Addison-Wesley object technology series. ADDISON WESLEY Publishing Company Incorporated, 2001.

[18] D. Gollmann. *Computer Security*. Wiley, 2011.

[19] JBoss Community. Hibernate. http://docs.jboss.org/hibernate/orm/4.2/manual/en-US/html/ch20.html#performance-fetching, 2014.

[20] Simon F. Goldsmith, Alex S. Aiken, and Daniel S. Wilkerson. Measuring empirical computational complexity. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 395–404, 2007.

[21] Dmitrijs Zaparanuks and Matthias Hauswirth. Algorithmic profiling. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 67–76, 2012.

[22] R.V. Binder. *Testing Object-oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 2000.

[23] Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. Catch me if you can: performance bug detection in the wild. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 155–170, 2011.

[24] Ryan Bloom. log4jdbc. https://code.google.com/p/log4jdbc/, 2013.

[25] Tomas Kalibera and Richard Jones. marking in reasonable timerigorous benchmarking in reasonable time. In *Proceedings of the 2013 international symposium on International symposium on memory management*, ISMM '13, pages 63–74, 2013.

[26] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 57–76, 2007.

[27] Matthew Arnold, Michael Hind, and Barbara G. Ryder. Online feedback-directed optimization of java. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '02, pages 111–129, 2002.

[28] D.S. Moore, G.P. MacCabe, and B.A. Craig. *Introduction to the Practice of Statistics*. W.H. Freeman and Company, 2009.

[29] Shinichi Nakagawa and Innes C. Cuthill. Effect size, confidence interval and statistical significance: a practical guide for biologists. *Biological Reviews*, 82:591–605, 2007.

[30] Vigdis By Kampenes, Tore Dybå, Jo E. Hannay, and Dag I. K. Sjøberg. Systematic review: A systematic review of effect size in software engineering experiments. *Inf. Softw. Technol.*, 49(11-12):1073–1086, November 2007.

[31] B.A. Kitchenham, S.L. Pfleeger, L.M. Pickard, P.W. Jones, D.C. Hoaglin, K. El Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Trans. Softw. Eng.*, 28(8):721–734, 2002.

[32] J. Hartung, G. Knapp, and B.K. Sinha. *Statistical Meta-Analysis with Applications*. Wiley, 2011.

[33] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. L. Erlbaum Associates, 1988.

[34] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, August 1988.

[35] SpringSource. Spring framework. www.springsource.org/, 2013.

[36] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, 2012.

[37] Gregory M. Kapfhammer, Phil McMinn, and Chris J. Wright. Search-based testing of relational schema integrity constraints across multiple database management systems. In *Proceedings of the 6th International Conference on Software Testing Verification and Validation*, ICST '13, 2013.

[38] P. Zaitsev, V. Tkachenko, J.D. Zawodny, A. Lentz, and D.J. Balling. *High Performance MySQL: Optimization, Backups, Replication, and More*. O'Reilly Media, 2008.

[39] G. Bulmer. *Principles of Statistics*. Dover Books on Mathematics Series. Dover Publications, 1979.

[40] J. Linwood and D. Minter. *Beginning Hibernate*. Apresspod Series. Apress, 2010.

[41] C. U. Smith and L.G. Williams. More new software performance antipatterns: Even more ways to shoot yourself in the foot. In *Proceedings of the 2003 Computer Measurement Group Conference*, CMG 2003, 2003.

[42] Connie U. Smith and Lloyd G. Williams. Software performance antipatterns. In *Proceedings of the 2Nd International Workshop on Software and Performance*, WOSP '00, pages 127–136, 2000.

[43] Jaideep Nijjar and Tevfik Bultan. Data model property inference and repair. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA '13, pages 202–212, 2013.

[44] Vittorio Cortellessa, Antinisca Di Marco, and Catia Trubiani. Software performance antipatterns: Modeling and analysis. In *Proceedings of the 12th International Conference on Formal Methods for the Design of Computer, Communication, and Software Systems: Formal Methods for Model-driven Engineering*, SFM'12, pages 290–335, 2012.

[45] Juan M. Tamayo, Alex Aiken, Nathan Bronson, and Mooly Sagiv. Understanding the behavior of database operations under program control. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 983–996, 2012.

[46] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. Finding low-utility data structures. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 174–186, 2010.

[47] M. Grechanik, B.M.M. Hossain, and U.A. Buy. Testing database-centric applications for causes of database deadlocks. In *Proceedings of the 6th International Conference on Software Testing Verification and Validation*, ICST '13, pages 174–183, 2013.

[48] Mark Grechanik, B. M. Mainul Hossain, Ugo Buy, and Haisheng Wang. Preventing database deadlocks in applications. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 356–366, 2013.