Optimizing the Performance-Related Configurations of Object-Relational Mapping Frameworks Using a Multi-Objective Genetic Algorithm

Ravjot Singh†, Cor-Paul Bezemer†, Weiyi Shang‡, Ahmed E. Hassan†
Software Analysis and Intelligence Lab (SAIL), Queen's University, Canada†
Department of Computer Science and Software Engineering, Concordia University, Canada‡
{rsingh, bezemer, ahmed}@cs.queensu.ca†, shang@encs.concordia.ca‡

ABSTRACT

Object-relational mapping (ORM) frameworks map low-level database operations onto a high-level programming API that can be accessed from within object-oriented source code. ORM frameworks often provide configuration options to optimize the performance of such database operations. However, determining the set of optimal configuration options is a challenging task.

Through an exploratory study on two open source applications (Spring PetClinic and ZK), we find that the difference in execution time between two configurations can be large. In addition, both applications are not shipped with an ORM configuration that is related to performance: instead, they use the default values provided by the ORM framework. We show that in 89% of the 9 analyzed test cases for PetClinic and in 96% of the 54 analyzed test cases for ZK, the default configuration values supplied by the ORM framework performed significantly slower than the optimal configuration for that test case. Based on these observations, this paper proposes an approach for automatically finding an optimal ORM configuration using a multi-objective genetic algorithm. We evaluate our approach by conducting a case study of Spring PetClinic and ZK. We find that our approach finds near-optimal configurations in 360-450 seconds for PetClinic and in 9-12 hours for ZK. These execution times allow our approach to be executed to find an optimal configuration before each new release of an application.

Keywords

object-relational mapping performance, performance configuration optimization

1. INTRODUCTION

As software becomes more complex and operates in different settings, it requires more flexibility in its underlying libraries and used frameworks. Hence, software libraries and frameworks tend to provide a high level of configurability. On the one hand, configurability offers a great deal of flexibility. However, this flexibility comes at a cost as configuration errors can have a disastrous impact [28,31]. The potentially large impact of misconfiguration has lead to a large body of research in the area of (mis)configurability [11,15,30,31,33,34]. Yin et al. [31] show that 27% of all reported customer issues are related to misconfiguration. Around half of these configuration issues are caused by misinterpretation of configuration options. Moreover, up to 20% of the reported cases of misconfiguration caused severe performance degradation [31]. Considering the high potential cost per issue, such as the estimated cost of 1.6 billion US dollar for a 1-second slowdown for Amazon [9], the cost of performance misconfiguration can rise to billions each year [32].

Object-relational mapping (ORM) is a technique that was introduced to provide a mapping between the higher level object-oriented model and the lower level relational model of a database management system. ORM frameworks offer a variety of configuration options that allow the user to configure how this mapping is performed statically and at run-time. Since ORM frameworks create a layer between the database and source code, their configuration can impact the performance of database operations. Chen et al. [3] show that ORM configurations can suffer from performance anti-patterns, indicating that ORM misconfiguration is a common problem.

In this paper, we first show through an exploratory study that ORM misconfigurations can indeed have a significant impact on the performance of an application. Additionally, we show that the optimal configuration performs significantly better than the configuration that is currently used by our subject applications for the analyzed workload.

The observations of our exploratory study motivate the second part of this paper, in which we propose an approach for automatically optimizing the performance-related configurations of ORM frameworks using a multi-objective genetic algorithm. We evaluate our approach through a case study of two open source applications (Spring PetClinic and ZK). Our case study shows that our approach is capable of finding configurations that are in the top 25% best-performing configurations for each studied workload. In short, we make the following contributions:

1. The observations that, in our studied subject applications:

   (a) ORM configurations have a large impact on the performance of an application

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ICPE'16, March 12-18, 2016, Delft, Netherlands
©2016 ACM. ISBN 978-1-4503-4080-9/16/03 ...$15.00.
DOI: http://dx.doi.org/10.1145/2851553.2851576.

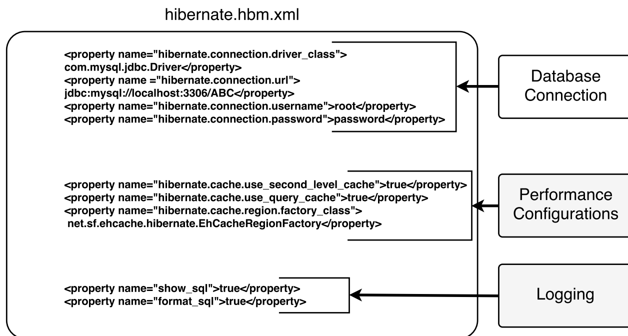


Figure 1: ORM configuration code for Hibernate

   (b) The default configuration as supplied by the ORM framework performs in many cases significantly worse than the optimal configuration

2. An approach for automatically finding a near-optimal ORM configuration using a multi-objective genetic algorithm

The rest of this paper is organized as follows. Section 2 provides background information about ORM configuration. Section 3 presents the setup and results of our exploratory study. Section 4 gives an introduction to genetic algorithms. Section 5 presents our approach for automatically finding an optimal ORM configuration for an application. Section 6 and 7 describe the setup and results of the case study that we performed to evaluate our approach. Section 8 describes the threats to the validity of our work. Section 9 gives an overview of related work. Finally, Section 10 concludes the paper.

2. ORM PERFORMANCE-RELATED CONFIGURATION

ORM frameworks allow developers to perform database operations without writing boilerplate code for maintaining the connection or executing queries. One of the most popular [13] ORM frameworks is Hibernate[1] for Java. Figure 1 shows an example of the configuration code for the Hibernate framework. The configuration file *hibernate.hbm.xml* contains system-wide configuration options.

In Figure 1, the performance-related configuration specifies that caching should be used. Enabling the cache does not necessarily lead to an improvement in performance. In fact, using a cache in a situation in which data is infrequently used will lead to many cache misses, which may lead to a non-optimal performance.

Another example is the *hibernate.max_fetch_depth* configuration option. Hibernate maps database objects onto object-oriented programming objects. Data normalization is a concept heavily used in databases to keep tables small, which makes searching through them faster. An example of a simple database is depicted by Figure 2. To retrieve all information for a person, we have to perform three separate queries or one query with multiple joins. The *hibernate.max_fetch_depth* configuration option defines how Hibernate approaches such cross table queries. When set to 0, the use of joins for such queries is disabled and Hibernate will use three separate queries. When configured to a value higher than 0, Hibernate will use that value as the maximum depth of joins to perform in such a query. Using a single query with multiple joins is generally faster than using separate queries when all information returned by the query is used. However, in many situations only a part of the data is used and the overhead caused by retrieving the unnecessary data results in non-optimal performance.

The former two examples show that ORM performance-related configuration is specific to the workload and application and requires domain knowledge, which makes finding the optimal ORM configuration a challenging task. In the next section, we perform an exploratory study on the performance impact of ORM misconfiguration.

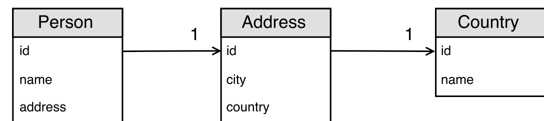


Figure 2: Database example

3. EXPLORATORY STUDY

To motivate our work, we perform an exploratory study to get an indication of how ORM configuration impacts performance.

How Does ORM Configuration Impact Performance?

By revealing the performance impact of changing the ORM configuration, we can demonstrate the importance of searching for an optimal configuration. For example, if the potential performance improvement after changing the configuration is small, it may not be worth the risk of making changes to the configuration of a stable software system.

We conducted an exploratory study with two subject applications (Spring PetClinic[2] and ZK[3]), that both use Hibernate. We selected eleven Hibernate configuration options that can influence the performance of the subject applications (see Table 3). We encode a configuration of these eleven options as a string of eleven bits. We converted the non-boolean options to boolean options by encoding their lowest allowed value as 0 and the highest allowed value as 1. This limits the total number of possible configurations to $2^{11} = 2,048$, which allows us to perform an exhaustive analysis on the solution space.

To evaluate the performance impact of changing the configuration, we need to evaluate the performance of an application when it is using that configuration. Because we do not have access to performance tests or workload generators for most systems, we use their unit tests, in particular, the ones that use ORM. To extract these tests we statically analyze the source code of a subject application and identify functions that use ORM using a list of keywords. Then we use reflection to find the unit tests that execute those functions. This process is described in more detail in Section 5.2.

We populated the database used by PetClinic and ZK with one million records that were randomly generated based on the database schema requirements. We selected 9 test cases for PetClinic and 54 for ZK. For each of the 2,048 possible

[1] http://hibernate.org/

[2] http://docs.spring.io/docs/petclinic.html

[3] http://zkoss.org/

Table 1: Example of the relative difference calculation

Selection	Test case	Avg. time c_1	Avg. time c_2	Diff.
#1	t_1	10	20	100%
#2	t_1	11	15	36%
#3	t_1	8	15	88%

configurations, we run the selected test cases 30 times to minimize variability in the performance measurements and we collect the execution time of each run for each test case.

To investigate the impact of ORM configurations on performance, we calculate the difference in execution time of the test cases when they are executed using two randomly selected configurations. We perform the following process 1,000 times for each selected test case in the subject application:

1. Randomly select two ORM configurations: c_1 and c_2
2. Take the average of the execution time of 30 runs for the test case using configuration c_1
3. Repeat step 2 using configuration c_2
4. Calculate the relative difference in execution times across both configurations

Based on the collected data, we make the following observation:

> *Observation 1: Changing the ORM configuration can have a large impact on the execution time of an application.*

The impact of performance-related ORM configuration options is large. Table 1 shows example output of the process we used to calculate the relative difference for one test case t_1 and three random selections of c_1 and c_2. Figure 3 shows the distribution of the standard deviation of the relative difference in execution time for two randomly selected ORM configurations across all selected test cases. For PetClinic, the median standard deviation is around 20%, while the median standard deviation for ZK is 140%.

How Does the Default ORM Configuration Perform?

Our subject applications do not include an ORM configuration. Instead, they use the default values provided by the ORM framework. We investigate the performance of this default configuration as compared to the optimal configuration.

We compared the execution times for each test case using the default configuration with the execution times using the optimal configuration for that test case using a *t-test* ($p < 0.05$). For the test cases that had a significant difference in execution time, we also calculated the effect size using *Cohen's d* [1]. Based on earlier work [14], we use the following classification for d:

$$Effect\ Size: d = \begin{cases} < 0.16 & Trivial \\ 0.16 - 0.6 & Small \\ 0.6 - 1.4 & Medium \\ > 1.4 & Large \end{cases}$$

Table 2 summarizes the results of this comparison. We observe that for 89%-96% of the test cases, their optimal configuration is significantly better than the default configuration as supplied by the ORM framework. In 88%-100% of those cases, the difference was classified to have at least a medium effect size.

Table 2: Default vs. optimal configuration

	PetClinic	ZK
# test cases using ORM	9	54
% test cases significantly slower than optimal	89%	96%
Effect size (only for cases that are significantly slower):		
d = small	12%	0%
d = medium	25%	71%
d = large	63%	29%

Table 3: Performance-related configuration in Hibernate [19]

Option	Value
order_updates	TRUE|FALSE
jdbc.batch_size	LOW|HIGH
order_inserts	TRUE|FALSE
connection.release_mode	TRUE|FALSE
default_batch_fetch_size	LOW|HIGH
jdbc.batch_versioned_data	TRUE|FALSE
max_fetch_depth	LOW|HIGH
id.new_generator_mappings	TRUE|FALSE
jdbc.fetch_size	TRUE|FALSE
bytecode.use_reflection_optimizer	TRUE|FALSE
cache.use_second_level_cache	TRUE|FALSE

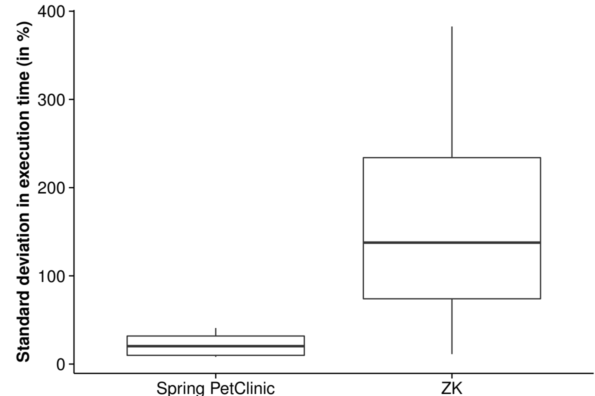


Figure 3: Standard deviation of relative difference in execution time (in %) for two randomly selected ORM configurations

> *Observation 2: For both subject applications, the default configuration as supplied by the ORM framework performs significantly worse than the test-specific optimal configuration for each test case.*

Our two aforementioned observations motivate our work for finding a method that automates the configuration process. In the remainder of this paper, we will present and evaluate our approach for automatically optimizing the performance-related ORM configurations using a multi-objective genetic algorithm. First, we will give a brief introduction to genetic algorithms.

4. GENETIC ALGORITHMS

A genetic algorithm is a search-based heuristic that searches for an optimized solution in a population of solutions based

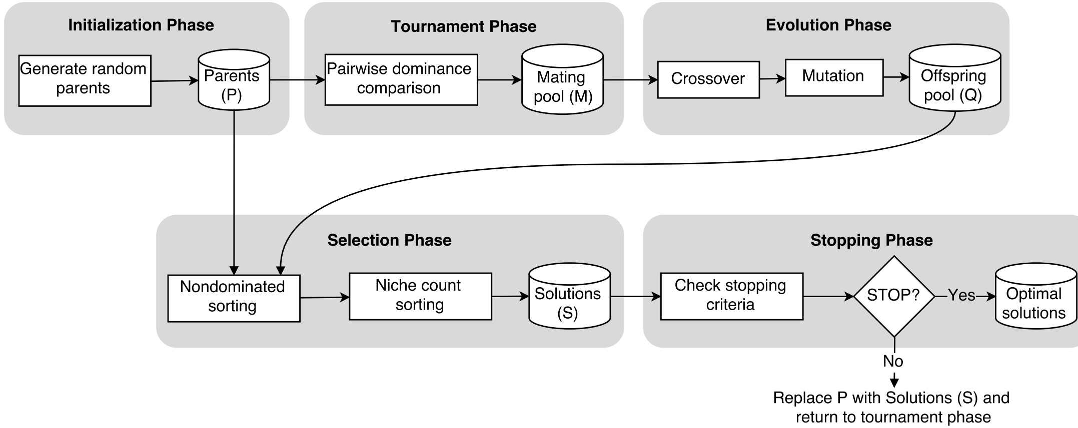


Figure 4: Phases in NSGA-II

on one or more objectives [4]. Since these objectives may be conflicting, there may be multiple optimal solutions that all have a slight bias towards one or more objectives. Genetic algorithms follow a process that closely resembles natural evolution, in particular, the 'survival of the fittest' principle. During the search, solutions that have attributes that appear to have a positive impact on one of the objectives, are selected to steer the evolution of the population. Attributes are characteristics that solutions can inherit from their parent solutions. The goal of a genetic algorithm is to improve the population during every iteration (*generation*). In our approach, we will use the non-dominated sorted genetic algorithm (NSGA-II) [5], a *multi-objective genetic algorithm* (MOGA) which aims to quickly find a set of optimal solutions. Figure 4 depicts the phases of NSGA-II, which we will explain in the remainder of this section. First, we will explain the concept of dominance and we will describe our running example that we use to demonstrate the algorithm.

4.1 Dominance

To compare two solutions that have multiple objectives, MOGAs usually rely on the concept of *dominance* [4]. A solution s_1 is said to dominate solution s_2, i.e. $s_1 \preceq s_2$, when 1) s_1 is no worse than s_2 in all objectives and 2) s_1 is strictly better than s_2 in at least one objective. By using dominance, we can compare multi-objective solutions without purposely creating a bias towards an objective.

4.2 Running Example

In a genetic algorithm, members of the population are represented as binary strings. For every member, a bit i indicates whether that member possesses the boolean attribute i. In our work, we encode configuration options as boolean attributes. In this section, we will use a running example of four boolean configuration options: {*order_updates, order_inserts, jdbc.batch_size, cache.use_second_level_cache*}. As a result, the member 1110 in the population of all combinations of these four configuration options represents the configuration in which all options except the cache are enabled. For simplicity, we will only use *execution time* as the objective in our running example. The member (1110, 50) represents that the execution time of a workload using configuration 1110 is 50 seconds. In the remainder of this section, we will explain the phases of NSGA-II using our running example.

4.3 Initialization Phase

During the initialization phase, the algorithm is started by randomly selecting a set P of members, i.e., binary strings, that will act as the initial parents in the evolution. $|P|$ represents the number of members $\in P$. The randomly selected parents in our random example are:

$$P = \{(0000, 100), (1110, 50), (1101, 100), (0111, 60)\}$$

4.4 Tournament Phase

At the start of each generation, a tournament will be held to select the best performing parents. In the tournament phase, we:

1. Randomly select two parents $P_1, P_2 \in P$ that have not been in the tournament for this generation
2. Perform a pairwise dominance comparison of P_1 and P_2 and add the dominant parent to the mating pool M
3. Repeat steps 1 and 2 until all parents $\in P$ have been in the tournament

After the tournament phase, M contains $|P|/2$ members that will be used during the evolution phase.

In our running example we compare (0000, 100) with (1110, 50) and (1101, 100) with (0111, 60). Because we have only one objective, the configurations that have a shorter execution time win the comparison. As a result,

$$M = \{(1110, 50), (0111, 60)\}$$

4.5 Evolution Phase

During the evolution phase, randomly-selected parents from the mating pool produce offspring. Evolution happens as follows:

1. Randomly select from M two parents P_1 and P_2 that have not yet evolved in this generation
2. Create an offspring that randomly inherits attributes, i.e., bits, from P_1 and P_2 (*crossover*)
3. Flip a random number of bits of the offspring (*mutation*)

4. Store the offspring in the offspring pool Q

In our running example we create the offspring 0110 by inheriting the first three bits from (0111, 60) and the last bit from (1110, 50) (crossover). Then, we mutate the offspring into the mutation 1111 by flipping the first and last bit. We evaluate the execution time of the workload using the mutated offspring and find that it is 40 seconds. Hence,

$$Q = \{(1111, 40)\}$$
$$P \cup Q = \{(0000, 100), (1110, 50), (1101, 100), (0111, 60), (1111, 40)\}$$

4.6 Selection Phase

During the selection phase, members with the most positive impact on the objectives are selected. First, P and Q are combined into population X by taking their union. To select the optimal solutions, non-dominated and crowding-distance sorting are used.

During non-dominated sorting, X is ranked based on dominance: for every two members $x_i, x_j \in X$, the dominating member is assigned a higher rank than the other member, with 1 being the highest rank. We indicate the subpopulation that shares the same rank k with F_k. As a result, F_1 contains the optimal solutions found thus far, since rank 1 contains all $x_i \in X$ for which there is no $x_j \in X$ that dominates them.

Applying non-dominated sorting to X in our running example results in:

$$F_1 = (1111, 40)$$
$$F_2 = (1110, 50)$$
$$F_3 = (0111, 60)$$
$$F_4 = \{(0000, 100), (1101, 100)\}$$

To continue the evolution, we want to select $|P|$ members from the highest ranks, starting at F_1. If a rank has more members than we want to select from it, we calculate the niche count [5] between all pairs of members of the subpopulation in that rank. We calculate the niche count by counting the number of 'niche' attributes a member has compared to the other member, i.e., the number of attributes it can contribute to the population. For every two members $x_i, x_j \in X$, the member with the highest niche count has a higher niche rank, with 1 being the highest rank. We select the required number of members from the highest niche ranks. When members share a niche rank, we select random members from that rank.

(1101, 100) has a higher niche count than (0000, 100), because 1101 can contribute the first, second and fourth attribute to the population, while 0000 can contribute none. Hence, applying niche count sorting to F_4 results in:

$$F_{4,1} = (1101, 100)$$
$$F_{4,2} = (0000, 100)$$

Therefore, the set of solutions we select is:

$$Solutions = \{(1111, 40), (1110, 50), (0111, 60), (1101, 100)\}$$

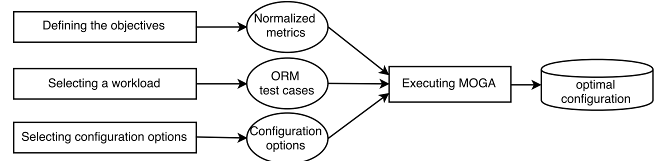


Figure 5: Finding the optimal ORM configuration

4.7 Stopping Phase

The algorithm stops when the following constraint is met: the output of the selection phase consists of members from F_1 only. If this constraint has not been satisfied, a new generation is started by using the output of the selection phase as P for the new generation. Alternatively, a generation threshold T can be specified that stops the algorithm after T generations.

5. AUTOMATIC PERFORMANCE OPTIMIZATION

In this section, we present our approach for automatically finding an optimal ORM configuration for a given application and workload. Our approach finds cross-test optimal configurations, i.e., configurations that optimize the performance for all test cases and objectives. Our approach requires three inputs: 1) the objectives that must be optimized, 2) the workload and 3) the configuration options. Figure 5 depicts the steps of our approach, which will be explained in this section.

5.1 Defining the Objectives

First, we must define the objectives that we want to optimize. Because the performance of a system has multiple, often conflicting aspects, we use a multi-objective approach. For example, optimizing both CPU usage and memory usage is a trade-off between CPU and memory. By following a multi-objective approach, we can identify the configuration that finds the optimal trade-off between conflicting objectives.

Our approach uses normalized performance metrics as objectives to avoid a bias towards objectives that have a large range of values. For example, execution time has a larger range than CPU usage, which is expressed in percentage. When optimizing execution time and CPU usage together, the algorithm may prefer the large absolute improvements a configuration can make for the execution time over the seemingly smaller improvements that can be made for CPU. In addition, normalization makes our results easier to interpret. We normalize a metric by calculating the change in percentage using a configuration compared with the performance using the current application configuration. We demonstrate the normalization process using CPU usage in n test cases for a configuration c as example below.

1. Monitor the average CPU usage for each test case i using the default configuration ($CPU_{i,current}$)
2. Monitor the average CPU usage for each test case i using c ($CPU_{i,c}$)
3. Calculate the % change for each test case i between $CPU_{i,c}$ and $CPU_{i,current}$:

$$\Delta CPU_{i,c} = \frac{CPU_{i,c} - CPU_{i,current}}{CPU_{i,current}} * 100$$

Table 4: Example CPU usage normalization

Testcase	$CPU_{i,c}$	$CPU_{i,current}$	$\Delta CPU_{i,c}$
t_1	90	100	-10
t_2	110	100	10
t_3	70	100	-30

4. Calculate the % change for all n test cases combined:

$$\Delta CPU_c = aggr(\{CPU_{t,c} | t \in \{1,..,n\}\})$$

In the last step of the process, *aggr()* is an aggregation such as *median()*, which can be chosen based on the desired objective. As ΔCPU_c indicates the relative increase in CPU usage using configuration c as compared to the default configuration, our objective is to find the configuration c that has the smallest value for ΔCPU_c. In $CPU_{t,c}$ t is the subset of all test cases for an application which use ORM.

Table 4 shows an example of 3 test cases t_1, t_2 and t_3. The $\Delta CPU_{i,c}$ column shows that configuration c improves the CPU usage for t_1 and t_3 by 10 and 30 percent compared to the current configuration. The choice for the aggregation function influences ΔCPU_c. For example, for *median*(), $\Delta CPU_c = 10$ and for *mean*(), $\Delta CPU_c = -10$.

5.2 Selecting a Workload

To get the performance impact of a configuration compared with the default configuration, we must run a workload for the application that we want to optimize using that configuration. We propose to use a subset of the unit test suite that uses ORM. For example, we can select these test cases for Java projects as follows:

1. Define a list of keywords that identify ORM usage in functions
2. Search the subject application for functions that use ORM using the keywords
3. Collect the call hierarchy using reflection and the Eclipse JDT[4]
4. Select the unit test cases in the call hierarchy of the functions that use ORM

5.3 Selecting Configuration Options

We must define the set of configuration options that we want to use during the optimization process. This set can be selected from the documentation of the ORM framework. Every boolean configuration option can be encoded using one bit. Non-boolean options can be encoded by reserving a group of bits large enough to express the possible values. For simplicity, we use only the minimum and maximum allowed values of such options in this paper.

5.4 Executing the MOGA

After defining the objectives and selecting the workload and configuration options, we can start the MOGA as explained in Section 4. During its execution, the algorithm will generate new configurations. Every time a new configuration is generated, it is evaluated using the workload and the normalized metrics for that configuration are stored in a temporary local database, so that a configuration does not have to be reevaluated when it is encountered again during the evolution.

When the algorithm terminates, a set of optimal configurations found thus far is returned. We order these configurations based on the number of options that are changed compared to the default configuration and select a random configuration with the lowest number of changed options. Because the execution of the MOGA is fully automated, it can be integrated in a continuous integration environment to find the optimal ORM configuration for every new release. By proposing an improved configuration that requires few changes to the default configuration, we can reduce the risk of updating the configuration.

6. CASE STUDY SETUP

We evaluate our approach through a case study with two subject applications. In this section, we present the subject applications and the setup of our case study.

6.1 Subject Applications

To evaluate our approach, we use the same subject applications as used in our motivational study. Spring PetClinic is a demonstration application for the Java Spring framework. PetClinic is used regularly in performance research [3, 12, 21] as case study subject application. ZK is a web framework that assists developers in creating web GUIs. We selected ZK because of its maturity (i.e., over 22 thousand commits on GitHub) and its use of Hibernate.

6.2 Implementation

We implemented our approach in Java using the MOEA framework[5] for NSGA-II. We used SIGAR[6] to monitor the performance metrics that are used in our objectives. All test cases were executed on an Intel i7 3.6GHz quad-core processor with 16 GB RAM.

6.3 MOGA Parameters

Objectives

In our evaluation we focus on optimizing the CPU usage, memory usage and execution time. Since we express our objectives in the difference in percentage compared to the current configuration, our MOGA should search for configurations c that have minimized values for ΔCPU_c , ΔMEM_c and $\Delta EXECTIME_c$. Because our approach is multi-objective, it will search for a configuration that optimizes all these objectives together. As aggregation function for calculating ΔCPU_c , ΔMEM_c and $\Delta EXECTIME_c$ we evaluated *mean()* and *median()*.

Workload

We selected the PetClinic and ZK unit test cases that use ORM as described in Section 5.2. This resulted in 9 (out of 13) test cases that use ORM for PetClinic and 54 (out of 55) that use ORM for ZK.

Configuration Options

We used the configuration options that are described in Table 3.

[4] http://www.eclipse.org/jdt/

[5] http://moeaframework.org/

[6] http://sigar.hyperic.com/

Stopping Rule

To terminate the MOGA, we run our experiments with two different stopping rules, using a t-test and the mutual dominance rate. The goal of the stopping rules is to stop the algorithm when no progress is being made in finding a better configuration.

T-test [26]: We use a t-test on each objective of all configurations generated in two consecutive generations g_i and g_j. For example, assume the solutions found in g_i are configurations a_1, a_2 and a_3 in Table 5. Likewise, b_1, b_2 and b_3 are found in g_j. In our example in Table 5, we have two objectives (CPU and MEM). Hence, we run two t-tests to compare the configurations: one on the values of ΔCPU and one on the values of ΔMEM for all configurations in both generations. When the t-tests for all objectives show insignificant differences ($p > 0.05$) for two consecutive generations, we stop the MOGA.

Mutual Dominance Rate (MDR) [16]: The mutual dominance rate is an indicator of the progress that is made by the algorithm. We introduce the number of configurations in set A that are dominated by at least one configuration in set B as dom(A,B). We define set A as the set of configurations that are found during the previous generation of the MOGA. B is the set of configurations that are found during the current generation of the MOGA. Table 5 contains two sets A and B, for which *dom()* can be calculated as follows:

$$b_1 \preceq a_2, b_1 \preceq a_3 \rightarrow dom(A, B) = \{a_2, a_3\}$$
$$a_1 \preceq b_2, a_1 \preceq b_3 \rightarrow dom(B, A) = \{b_2, b_3\}$$

Using *dom()*, the MDR is defined as:

$$MDR = \frac{|dom(A, B)|}{|A|} - \frac{|dom(B, A)|}{|B|}$$
$$MDR = \frac{2}{3} - \frac{2}{3} = 0$$

When MDR is 0, no progress is being made as both the previous set and current set of found configurations contain the same number of dominating configurations.

When MDR is -1, the solutions are actually becoming worse as the configurations in the previous set were better than in the current set (C).

$$MDR = \frac{|dom(A, C)|}{|C|} - \frac{|dom(C, A)|}{|A|}$$
$$MDR = \frac{0}{3} - \frac{3}{3} = -1$$

When MDR is 1, we know that the algorithm is progressing as there are more configurations in the current set of found configurations that are better than the configurations in the previous set. We can see that set D is better than A because:

$$MDR = \frac{|dom(A, D)|}{|A|} - \frac{|dom(D, A)|}{|D|}$$
$$MDR = \frac{3}{3} - \frac{0}{3} = 1$$

Because of the randomness in the MOGA, consecutive MDR may have alternative signs. Hence, when the MDR is close to zero, the MOGA can terminate.

Table 5: Example set A, B, C and D

	Set A		Set B		
	ΔCPU	ΔMEM		ΔCPU	ΔMEM
a_1	50	50	b_1	50	50
a_2	20	20	b_2	40	40
a_3	10	10	b_3	10	10

	Set C		Set D		
c_1	5	5	d_1	100	100
c_2	5	5	d_2	100	100
c_3	5	5	d_3	100	100

In total, we evaluate the following 4 stopping criteria (all conditions must hold for 2 consecutive generations):

1. $STOP_t$: t-test, $p > 0.05$
2. $STOP_{0.5}$: $-0.5 < MDR < 0.5$
3. $STOP_{0.25}$: $-0.25 < MDR < 0.25$
4. $STOP_{0.1}$: $-0.1 < MDR < 0.1$

Combining the possible aggregation functions and stopping rules, we inspect a total of 8 combinations of aggregation functions and stopping rules (2 aggregation functions each evaluated with 4 stopping rules) for both studied applications during our case study. In the next section, we present our case study.

7. CASE STUDY

In this section, we discuss our case study. In particular, we focus on:

RQ1 How close are the configurations that are found by our approach to the optimal configurations? (*closeness*)

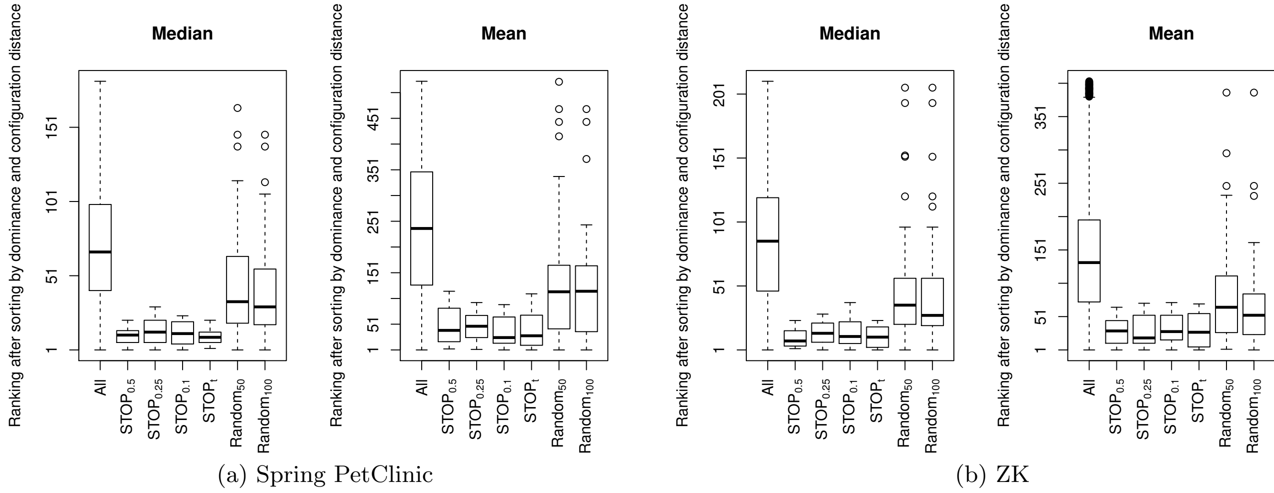
RQ2 What is the number of configurations that we need to inspect before our approach stops? (*speed*)

For both research questions, we explain the motivation behind it, our approach and our findings.

7.1 RQ1: Closeness

Motivation: The goal of our approach is to find the optimal configurations. However, limitations such as the required execution time for gathering the performance evaluation of a configuration require us to define a stopping rule. In this section, we evaluate how close the configurations found by our approach are to the optimal configurations. In addition, we evaluate how the aggregate function and stopping rule affect this closeness.

Approach: We rank all possible configurations for each subject application using non-dominated sorting, using the performance metrics that we collected during the execution of the workload (i.e., all test cases). After that, we calculate the number of configuration options that each configuration differs from the default configuration. We use this *distance*



(a) Spring PetClinic (b) ZK

Figure 6: Distribution of the ranks of all possible configurations and configurations found by our approach (rank 1 contains the optimal configuration, non-dominated and distance ranking)

to rank the configurations within a non-dominated rank. To illustrate this, Table 6 shows the top 5 ranks for PetClinic using *mean()* as an aggregate function, $STOP_{0.25}$ as the stopping rule and 00000000000 as the default configuration. Because randomness is involved in the execution of a genetic algorithm, we execute our approach 100 times for all 8 investigated combinations of aggregation functions and stopping rules to minimize the effects of variation.

To get a baseline to compare the closeness of our approach with, we implement random selection. During the execution of our approach, we need to inspect a number of configurations n. We expect that our approach finds higher-ranked configurations as compared to when we randomly pick n configurations and select the optimal one. We perform random picking 100 times to minimize the impact of outliers. Hence, we repeat the following 100 times:

1. We select 50 ($Random_{50}$) and 100 ($Random_{100}$) configurations randomly from all possible configurations (without replacement)
2. We rank the configurations based on dominance and configuration distance and we select the configuration with the highest rank and the smallest distance

Figure 6 depicts the distribution of ranks of all possible configurations, our found solutions and the solutions that are found during our random experiments, after ranking them by dominance and configuration distance as explained.

Findings: Our approach clearly outperforms random selection for all combinations of parameters. Figure 6 shows that the rank of the worst configuration that is selected by our approach is in all cases equal to or higher (i.e., closer to 1) than the median rank of the randomly-selected configurations.

> *Case study result 1: Our approach finds configurations that are much closer to the optimal configuration than when repeatedly selecting random configurations.*

The second observation that we make in Figure 6 is that for all combinations of aggregation functions and stopping rules, our approach finds configurations that are within the top ranked 25% of all possible configurations, i.e., the bottom of the boxplot. The boxplots in Figure 6 show that *median()* is the best performing aggregation function.

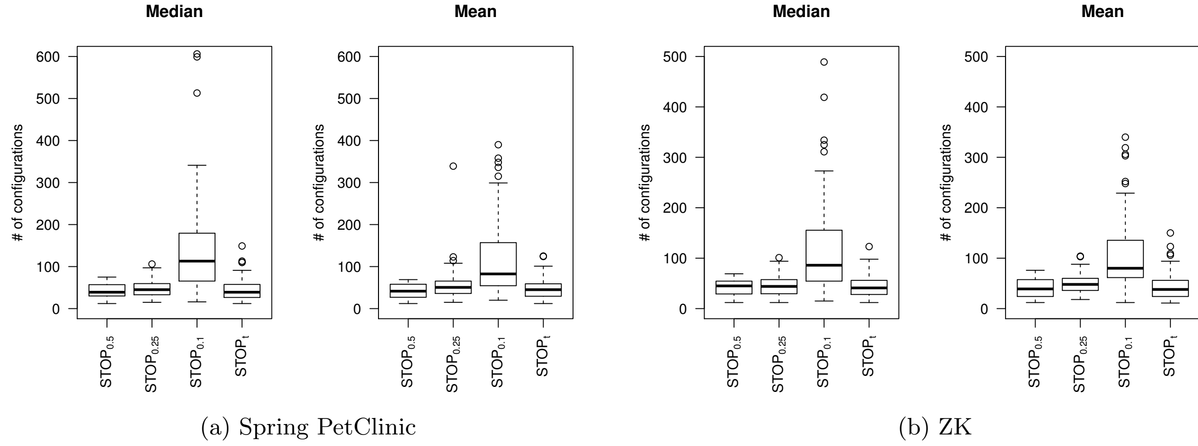
> *Case study result 2: Our approach finds configurations that are within the top ranked 25% of all possible configurations when using all combinations of aggregation functions and stopping rules in both subject applications.*

Table 6: Top 5 ranks for PetClinic ($mean$, $STOP_{0.25}$)

Rank	Non-dominated rank	Distance	Configurations
0	0	3	00000011100 00000100110
1	0	4	00010101010 00001011001
2	0	5	10001110010 10001110001 10011001010 10001111000
3	0	6	10001111010 10010111001 00010111101 10001110101 01100111001
4	0	7	00110111011 00110011111

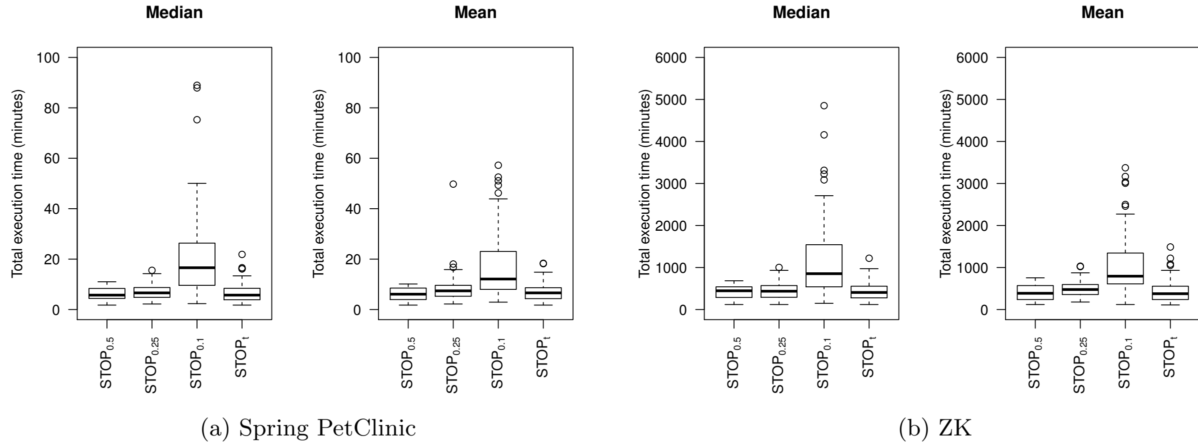
7.2 RQ2: Speed

Motivation: The speed with which our approach finds an optimal solution, i.e., the number of configurations it has to inspect, defines the practical applicability of our approach. Because the workload has to be executed (preferably multiple times to remove variation) in order to evaluate the performance of a configuration, the speed of our approach is dependent on the number of configurations that must be inspected.



(a) Spring PetClinic (b) ZK

Figure 7: Number of configurations generated to find an optimal configuration



(a) Spring PetClinic (b) ZK

Figure 8: Total execution time of our approach

Approach: To evaluate the speed with which our approach finds a solution, we monitor the number of new configurations that are generated before the algorithm terminates.

Findings: Because our approach requires the execution of the workload for every newly generated configuration, the number of new configurations generated directly influences the time in which our approach can find an optimal configuration. The execution time of the workload (i.e., all test cases executed sequentially) for PetClinic is 0.3 seconds and for ZK 28.1 seconds using the default configuration. To remove variation, we execute every workload 30 times. Hence, analyzing a new configuration for PetClinic takes 9 seconds and for ZK 843 seconds. To illustrate, analyzing all possible configurations using the subset of performance-related configuration options in Table 3 takes 5 hours for PetClinic and almost 20 days for ZK. When using a large number of configuration options, exhaustive analysis of all possible configurations is no longer feasible.

Figure 7 shows the distribution of the number of new configurations that are generated by our approach in order to find an optimal configuration for all combinations of aggregation functions and stopping rules. Figure 8 shows the distribution of the execution time of our approach. When using $STOP_{0.5}$, $STOP_{0.25}$ or $STOP_t$ as the stopping rule, the number of analyzed configurations causes our approach to run for 360-450 seconds for PetClinic and for 9-12 hours for ZK to find an optimal configuration. This allows our approach to be run before the release of an application to find an optimal configuration.

> *Case study result 3: The execution time of our approach allows it to be executed before a new release to find an optimal configuration.*

We observe that $STOP_{0.5}$, $STOP_{0.25}$ and $STOP_t$ generate approximately the same number of configurations (40-50) before the algorithm terminates. For $STOP_{0.1}$, this number is much higher, while we observed in the previous section that the closeness does not increase. The large number of configurations generated shows that $STOP_{0.1}$ is too strict as a stopping rule.

8. THREATS TO VALIDITY

In this section, we present the threats to the validity of our work.

8.1 Internal Validity

In this paper we proposed an approach to identify the optimal ORM configuration for an application. We considered all ORM configuration options as binary values. However, some of the configuration options are non-binary values. To keep the total number of configurations low, we made these non-binary options binary by allowing only their low and high values in our motivational study and case study evaluation. This allowed us to exhaustively execute all possible configurations when evaluating the configurations that are found by our approach.

For the same reason, we selected only a subset of ORM configurations. However, the set of configuration options that are used is independent from our approach and as such, the set of configuration options can easily be adapted or extended.

During our case study we used unit test cases to evaluate the performance impact of an ORM configuration change. The advantage of using unit test cases is that they are usually readily available and repeatable in an automated fashion. In addition, the unit test cases allow us to find performance issues at the unit level. The disadvantage is that they are unlikely to be representative of a realistic workload. Our approach is agnostic to the used workload and defined objectives, making it flexible to adapt. We believe there are no limitations on the workload, other than that it should not be random, as randomness makes a fair comparison of monitored metrics impossible.

We evaluated only two different types of stopping rules (i.e. t-test and MDR). While algorithm termination is a widely researched topic in MOGA research [4,16,26], these two types are used regularly.

8.2 External Validity

In our case study we studied two open source projects. The evaluation of our approach may not generalize to other projects. In addition, ORM is widely adapted by enterprise applications [20]; while our studied projects are open source projects. Our findings may not generalize to enterprise applications. In future work, we plan to evaluate our approach on a large-scale industrial project.

One programming language (Java) and one ORM framework (Hibernate) is covered in our case study. Although the studied language and ORM framework are widely used in practice, the findings may be different for other programming languages and other frameworks. More case studies with more projects, especially enterprise applications, other programming languages and also other ORM frameworks, would complement our work.

9. RELATED WORK

We now describe prior research that is related to this paper. We focus on the research that aims to optimize configurations of large software systems. We discuss two types of prior research towards optimizing configurations: 1) Understanding the impact of configuration and 2) Proactive configuration optimization.

9.1 Understanding the Impact of Configuration Options

Large software systems often provide a large number of configuration options [2]. All too often, practitioners are not aware of the impact of configuration options. However, a thorough understanding of the options is essential for optimizing configurations. Prior research proposes techniques that assist in understanding the impact of configuration options.

In existing work, performance models are built that use configuration options to predict performance metrics (e.g. response time, resource utilization). Practitioners can better understand the impact of configuration options by measuring the impact of these options on the performance metrics in the model. Siegmund et al. [22] build linear regression models to understand the performance impact of configuration options. To address the challenge of having a large number of configuration options, Siegmund et al. [23] leverage forward and backward feature selection techniques to reduce the number of configuration options in the model. Guo et al. [10] leverage non-linear regression models to model system performance. Their approach automatically identifies the configuration options that have the largest impact on performance and builds models with such configuration options. As a follow-up work, Zhang et al. [34] leverage Fourier transformations on performance counters to build performance prediction models.

Statistical methodologies are often leveraged for understanding the impact of configuration options. However, to quantify the impact of each configuration option, one would need to evaluate the performance of using every possible combination of configuration options. To reduce such effort, Debnath et al. [6] assume that the impact of configuration options on performance is monotonic and there only exist single and two-factor interactions among configuration options on performance. With this assumption, Debnath et al. leverages *Plackett and Burman* statistical design methodology [25] to rank the impact of configuration options on system performance.

Prior research can assist practitioners in understanding the impact of configuration options on performance. However, even with such knowledge, practitioners may still need help in choosing the optimal configurations. Moreover, all too often, the optimal configurations depend on the specific workload. Therefore, instead of understanding the impact of configuration options on performance, this paper focuses on automatically suggesting configurations to achieve optimal performance.

9.2 Proactive Configuration Optimization

To suggest better configurations for large software systems, prior research proposes techniques that proactively optimizes the configurations of a system. These techniques do not target any particular configuration issues in the system.

Configuration optimization with optimizing algorithms. The most widely used approach to optimize configurations is through the use of an optimizing algorithm, such as hill-climbing. Duan et al. [8] proposes a framework that assists in leveraging such optimizing algorithms to find optimal configurations. The framework leverages adaptive sampling to select the experiments to evaluate performance of configurations with and performs on-line experiments in the production environment with near zero performance overhead. Xi et al. [29] optimize the configuration for the application server using sampling and a smart hill-climbing algorithm, which selects important samples for the experiment

to search the optimal configuration. Lengauer et al. [15] proposes an approach that identifies the optimal configuration for the JVM garbage collector. Their approach leverages the ParamILS algorithm which performs local search iteratively. Wang et al. [27] propose an approach to optimize the configuration for Hadoop based on a hill-climbing algorithm.

To address the challenge of having a large search space for the optimal configurations, Osogami et al. [17] propose approaches that perform adaptive search to find a configuration that is better than the default configuration. Their approach only considers the configurations with minimal changes while searching for the better configuration. In addition, their approach aims to reduce the total time required for searching the best configuration by reducing the evaluation time for each configuration. A follow up work by Osogami et al. [18] improves their prior approach [17] by guessing the performance of configurations based on similarities between configurations. Instead of reducing the time of evaluating configuration performance, Thonangi et al. [24] reduces the candidates of optimal configurations by selecting a sample of configurations that most likely have the optimal performance.

In this paper, we also leverage an optimization algorithm, i.e., a multi-objective genetic algorithm, to find the optimal configurations for performance. Above mentioned techniques optimize the configurations of a system based on single objective i.e., the execution time. In contrast, our work focuses on optimizing multiple objectives at once.

Configuration optimization based on performance models. Section 9.1 presents prior research that build performance models in order to understand the impact of configurations. Such models are further leveraged to optimize configurations. Zheng et al. [35] propose an approach for optimizing configurations by traversing a configuration option dependency graph based on performance models. Diao et al. [7] monitor the CPU and memory utilization for web servers. Diao et al. model CPU and memory utilization and leverage such a model to achieve optimal configuration for an application.

Configuration optimization based on user experience. Popular large software systems often have a large user base. Experiences of choosing configuration options provide valuable information and can be generalized as guidelines of optimizing configurations of the system. Zheng et al. [36] observe that different users of a system may have the same optimal configurations and each user can have multiple near-optimal configurations. Therefore, Zheng et al. propose an approach that leverages existing configuration from different users and applies the configuration accordingly on the new software installation.

10. CONCLUSION

Object-relational mapping (ORM) provides a mapping between the higher level object-oriented model and the lower level relational model of a database management system. ORM frameworks offer a variety of configuration options that allow the user to configure how this mapping is performed in an optimal fashion.

In our motivational study we observe that ORM configurations have a large impact on the performance of an application.

Motivated by this observation, we propose an approach for automatically optimizing the performance configurations of ORM frameworks using a multi-objective genetic algorithm (MOGA). Using a MOGA allows us to optimize the performance based on multiple, possibly conflicting, objectives. In summary, these are the most important results of this paper:

- Two randomly selected configurations can lead to a large difference in execution time
- The default configuration as supplied by the ORM framework performed in 89-96% of the analyzed test cases significantly slower than the optimal configuration for that case
- Our approach can find a near-optimal configuration in a time that makes it feasible to find such a configuration before the new release of an application

11. REFERENCES

[1] Using effect size - or why the p value is not enough. http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3444174/.

[2] W. A. Babich. *Software configuration management: coordination for team productivity.* Addison-Wesley Reading, 1986.

[3] 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*, pages 1001–1012. ACM, 2014.

[4] K. Deb. *Multi-objective optimization using evolutionary algorithms*, volume 16. John Wiley & Sons, 2001.

[5] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *Evolutionary Computation, IEEE Transactions on*, 6(2):182–197, 2002.

[6] B. K. Debnath, D. J. Lilja, and M. F. Mokbel. Sard: A statistical approach for ranking database tuning parameters. In *Data Engineering Workshop, 2008. ICDEW 2008. IEEE 24th International Conference on*, pages 11–18. IEEE, 2008.

[7] Y. Diao, J. L. Hellerstein, S. Parekh, and J. P. Bigus. Managing web server performance with autotune agents. *IBM Systems Journal*, 42(1):136–149, 2003.

[8] S. Duan, V. Thummala, and S. Babu. Tuning database configuration parameters with ituned. *Proceedings of the VLDB Endowment*, 2(1):1246–1257, 2009.

[9] K. Eaton. How one second could cost amazon $1.6 billion in sales. http://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales (last visited: Sep 17 2015).

[10] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wasowski. Variability-aware performance prediction: A statistical learning approach. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 301–311. IEEE, 2013.

[11] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle. Paramils: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36(1):267–306, 2009.

[12] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. Automatic identification of load testing problems. In *IEEE International Conference on Software Maintenance (ICSM)*, pages 307–316. IEEE, 2008.

[13] Java tools and technologies landscape for 2014. http://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-for-2014/ (last visited: Oct 7 2015).

[14] V. B. Kampenes, T. Dybå, J. E. Hannay, and D. I. K. Sjøberg. A systematic review of effect size in software engineering experiments. *Information and Software Technology*, 49(11-12):1073–1086, Nov 2007.

[15] P. Lengauer and H. Mössenböck. The taming of the shrew: Increasing performance by automatic parameter tuning for java garbage collectors. In *Proceedings of the 5th ACM/SPEC international conference on Performance engineering*, pages 111–122. ACM, 2014.

[16] L. Martí, J. García, A. Berlanga, and J. M. Molina. An approach to stopping criteria for multi-objective optimization evolutionary algorithms: the mgbm criterion. In *IEEE Congress on Evolutionary Computation (CEC)*, pages 1263–1270. IEEE, 2009.

[17] T. Osogami and T. Itoko. Finding probably better system configurations quickly. In *ACM SIGMETRICS Performance Evaluation Review*, volume 34, pages 264–275. ACM, 2006.

[18] T. Osogami and S. Kato. Optimizing system configurations quickly by guessing at the performance. In *ACM SIGMETRICS Performance Evaluation Review*, volume 35, pages 145–156. ACM, 2007.

[19] Red Hat Middleware, LLC. Hibernate manual: Configuration. https://docs.jboss.org/hibernate/orm/3.3/reference/en/html/session-configuration.html (last visited: Sep 24 2015).

[20] A. R. Seddighi. *Spring Persistence with Hibernate: Build Robust and Reliable Persistence Solutions for Your Enterprise Java Application*. Packt Publishing Ltd, 2009.

[21] V. Sharma and S. Anwer. Performance antipatterns: Detection and evaluation of their effects in the cloud. In *Services Computing (SCC), 2014 IEEE International Conference on*, pages 758–765, June 2014.

[22] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner. Performance influence models for highly configurable systems. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, 2015.

[23] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake. Predicting performance via automated feature-interaction detection. In *Proceedings of the 34th International Conference on Software Engineering*, pages 167–177. IEEE Press, 2012.

[24] R. Thonangi, V. Thummala, and S. Babu. Finding good configurations in high-dimensional spaces: Doing more with less. In *Modeling, Analysis and Simulation of Computers and Telecommunication Systems, 2008. MASCOTS 2008. IEEE International Symposium on*, pages 1–10. IEEE, 2008.

[25] J. Tyssedal. Plackett–burman designs. *Encyclopedia of Statistics in Quality and Reliability*, 2008.

[26] T. Wagner, H. Trautmann, and L. Martí. A taxonomy of online stopping criteria for multi-objective evolutionary algorithms. In *Evolutionary Multi-Criterion Optimization*, pages 16–30. Springer, 2011.

[27] K. Wang, X. Lin, and W. Tang. PredatorâĂŤan experience guided configuration optimizer for hadoop mapreduce. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, pages 419–426. IEEE, 2012.

[28] A. Wool. A quantitative study of firewall configuration errors. *Computer*, 37(6):62–67, June 2004.

[29] B. Xi, Z. Liu, M. Raghavachari, C. H. Xia, and L. Zhang. A smart hill-climbing algorithm for application server configuration. In *Proceedings of the 13th international conference on World Wide Web*, pages 287–296. ACM, 2004.

[30] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 244–259. ACM, 2013.

[31] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 159–172. ACM, 2011.

[32] M. Yonkovit. The cost of not properly managing your databases. https://www.percona.com/blog/2015/04/06/cost-not-properly-managing-databases/ (last visited: Sep 17 2015).

[33] S. Zhang and M. D. Ernst. Automated diagnosis of software configuration errors. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 312–321. IEEE Press, 2013.

[34] Y. Zhang, J. Guo, E. Blais, and K. Czarnecki. Performance prediction of configurable software systems by fourier learning. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2015.

[35] W. Zheng, R. Bianchini, and T. D. Nguyen. Automatic configuration of internet services. *ACM SIGOPS Operating Systems Review*, 41(3):219–229, 2007.

[36] W. Zheng, R. Bianchini, and T. D. Nguyen. Massconf: automatic configuration tuning by leveraging user community information. In *ICPE*, pages 283–288, 2011.

View publication stats