CacheOptimizer: Helping Developers Configure Caching Frameworks for Hibernate-Based Database-Centric Web Applications

Tse-Hsun Chen¹, Weiyi Shang², Ahmed E. Hassan¹
Mohamed Nasser³, Parminder Flora³
Queen’s University¹, Concordia University², BlackBerry³, Canada
{tsehsun, ahmed}@cs.queensu.ca¹, shang@encs.concordia.ca²

1. INTRODUCTION

Web applications are widely used by millions of users worldwide. Thus, any performance problems in such applications can often cost billions of dollars. For example, a report published in 2012 shows that a one-second page load slowdown of the Amazon web applications can cost an average of 1.6 billion dollars in sales each year [25]. The complexity and scale of modern database-centric web applications complicate things further. As much as 88% of developers find their applications’ performance is deeply impacted by their reliance on databases [29].

Application-level caching frameworks, such as Ehcache [54] and Memcached [38], are commonly used nowadays to speed up database accesses in large-scale web applications. Unlike traditional lower-level caches (e.g., hardware or web proxies) [27, 18, 34], these application-level caching frameworks require developers to instruct them about what to cache; otherwise these frameworks are not able to provide any benefit to the application.

Deciding what should be cached can be a very difficult and time-consuming task for developers, which requires in-depth knowledge of the applications and workload. For example, to decide that the results of a query should be cached, developers must first know that the query will be frequently executed, and that the fetched data is rarely modified. Furthermore, since caching frameworks are highly integrated with the application, these frameworks are configured in a very granular fashion — with cache API calls that are scattered throughout the code. Hence, developers must manually examine and decide on hundreds of caching decisions in their application. Even worse, a recent study finds that most database-related code is undocumented [37], which makes manual configuration even harder.

Developers must continuously revisit their cache configuration as the workload of their application changes [22]. Outdated cache configurations may not provide as much performance improvement, and they might even lead to performance degradation. However, identifying workload changes is difficult in practice for large applications [53, 61]. Even knowing the workload changes, developers still need to spend great effort to understand the new workload and manually re-configure the caching framework.

In this paper, we propose CacheOptimizer, a lightweight approach that automatically helps developers decide what should be cached (and also automatically places the cache configuration code) in web applications that are implemented using Hibernate in order to optimize the configuration of...
caching frameworks. Using CacheOptimizer, developers can better manage the cost of their database accesses—greatly improving application performance [6][11][13][15][18].

CacheOptimizer first recovers the workload of a web application by mining the server access logs. Such logs are typically readily-available even for large-scale applications that are deployed in production environments. CacheOptimizer further analyzes the source code statically to identify the database accesses that are associated with the recovered workloads. To identify detail information about the recovered database accesses, such as the types of the access and the accessed data, CacheOptimizer leverages static taint analysis [30] to map the input variables of the web requests to the exact database accesses. Combining the recovered workload and the corresponding database accesses, CacheOptimizer models the workload, the database accesses, and the possible cache configurations as a coloured Petri net. By analyzing the Petri net, CacheOptimizer is able to determine an optimal cache configuration (i.e., given a workload, which objects or queries should be cached by the caching frameworks). Finally, CacheOptimizer automatically adds the appropriate configuration calls to the caching framework API into the source code of the application.

We have implemented our approach as a prototype tool and evaluated it on three representative open-source database-centric web applications (Pet Clinic [11], Cloud Store [20], and OpenMRS [11]) that are based on Hibernate [21]. The choice of Hibernate is due to it being one of the most used Java platforms for database-centric applications in practice today [58]. However, our general idea of automatically configuring a caching framework should be extensible to other database abstraction technologies. We find that after applying CacheOptimizer to configure the caching frameworks on the three studied applications, we can improve the throughput of the entire application by 27–138%.

The main contributions of this paper are:

1. We propose an approach, called CacheOptimizer, which helps developers in automatically optimizing the configuration of caching frameworks for Hibernate-based web applications. CacheOptimizer does not require modification to existing applications for recovering the workload, and does not introduce extra performance overhead.

2. We find that the default cache configuration may not enable any cache or may lead to sub-optimal performance, which shows that developers are often unaware of the optimal cache configuration.

3. Compared to having no cache (NoCache), the default cache configurations (DefaultCache), and enabling all caches (CacheAll), CacheOptimizer provides a better throughput improvement at a lower memory cost.

Paper Organization. The rest of the paper is organized as follows. Section 2 first discusses related work to our paper. Section 3 introduces background knowledge for common caching frameworks. Section 4 describes the design details of CacheOptimizer. Section 5 evaluates the benefits and costs of CacheOptimizer. Section 6 discusses threats to validity of our study. Finally, Section 7 concludes the paper.

2. RELATED WORK AND BACKGROUND

In this section, we discuss related work to CacheOptimizer. We focus on three closely related areas: software engineering research on software configuration, optimizing the performance of database-centric applications, and caching frameworks.

2.1 Software Configuration

Improving Software Configurations. Software configurations are essential for the proper and optimal operation of software applications. Several prior software engineering studies have proposed approaches to analyze the configurations of software applications. For example, Rabkin et al. [17] use static analysis to extract the configuration options of an application, and infer the types of these configurations. Xu et al. [2] conduct an empirical study on the configuration parameters in four open-source applications in order to help developers design the appropriate amount of configurability for their application. Liu et al. [57] focus on configuring client-side browser caches for mobile devices.

Detecting and Fixing Software Configuration Problems. Rabkin et al. [16] use data flow analysis to detect configuration-related functional errors. Zhang et al. [59] propose a tool to identify the root causes of configuration errors. In another work, Zhang et al. [60] propose an approach that helps developers configure an application such that the application’s behaviour does not change as the application evolves. Chen et al. [10] propose an analysis framework to automatically tune configurations to reduce energy consumption for web applications. Xiong et al. [56] automatically generate fixes for configuration errors using a constraint-based approach.

Prior research on software configuration illustrates that optimizing configurations is a challenging task. In this paper, we propose CacheOptimizer, which particularly focuses on helping developers optimize the cache configurations to improve the performance of large-scale web applications.

2.2 Improving Application Performance by Reducing the Overhead of Database Accesses

Most studies in literature propose frameworks to reduce the overhead of database accesses by batching [28], or re-ordering [6] database accesses. Ramachandra et al. [48] propose a framework for pre-fetching data from the database management system (DBMS) in batches in order to reduce database access overheads. Similarly, Cheung et al. [16] propose a framework for delaying database accesses as late as possible, and sending database access requests only when the data is needed in the application. Chavan et al. [9] propose a framework for sending queries asynchronously in order to improve application performance.

Several proposed frameworks improve application performance by analyzing the source code. In our prior work [12][14], we propose a static analysis framework to detect and rank database access performance anti-patterns. Developers can address these anti-patterns based on their priorities. Cheung et al. [17] leverage static analysis and code synthesis to automatically generate optimal SQL queries according to post-conditions and loop invariants. Grechanik et al. [31] propose a framework that combines both static and dynamic analysis to prevent database deadlocks. Chaudhuri et al. [8] use instrumented database access information
to find database-related performance problems in the code.

Compared to prior studies, we do not propose a new framework. Instead, CacheOptimizer helps developers optimize the configuration of the frameworks (in particular caching frameworks) that are already in use in practice today. In-depth knowledge of a software application is needed for software developers to optimally configure such frameworks.

2.3 Caching Frameworks

There are many prior studies on cache algorithms and frameworks. Many cache algorithms such as least recently used (LRU) [34], and most recently used (MRU) [18] are widely used in practice for scheduling lower-level caches. For example, such algorithms are used to improve the performance of web applications by caching web pages through proxies [7, 24]. Most of these caching algorithms operate in an unsupervised dumb fashion, i.e., these low-level caching algorithms do not require any application-level knowledge to operate.

Many modern applications generate dynamic content, which may be highly variable and large in size, based on data in the DBMS. Therefore, many low-level cache frameworks are becoming less effective. Many recent caching frameworks cache database access at the application level [27, 10]. When using these application-level caching frameworks, developers have full control of what should be cached in an application. However, to leverage these caching frameworks effectively, they must be configured properly.

Unlike most prior studies, CacheOptimizer does not try to manage cache scheduling. Instead, CacheOptimizer is designed to help developers optimize the configuration of application-level caching frameworks, which must be configured correctly for developers to fully leverage their benefits.

3. HIBERNATE AND CACHING MECHANISMS

3.1 Hibernate

CacheOptimizer automatically configures the caching framework for Hibernate-based web applications [21]. Hibernate is one of the most popular Java frameworks for abstracting database operations. Hibernate abstracts database accesses as object calls in Java instead of using SQL or JDBC directly. Hibernate is very popular among developers, because it helps reduce the amount of boilerplate code and development time [36]. For instance, a recent survey shows that among the 2,164 surveyed Java developers, 67.5% use Hibernate [58] instead of other database abstraction frameworks (including JDBC). Figure 1 shows an example of using Hibernate to abstract database accesses in Java. In this example, annotations (e.g., @Entity, @Table, and @Column) are added to User.java to specify the mapping between tables in a relational database and objects in Java. Based on such annotations, Hibernate automatically transforms user records to user objects and vice versa, and automatically translates the manipulation of the user object to the corresponding SQL queries. Hibernate is often used along caching frameworks like Ehcache [54]. These application-level caching frameworks aim to improve the performance of database-centric applications by reducing the number of database accesses.

3.2 Hibernate Caching Mechanism

Most caching frameworks act like an in-memory key-value store. When using Hibernate, these caching frameworks would store the database entity objects (objects that have corresponding records in the DBMS) in memory and assign each object a unique ID (i.e., the primary key). There are two types of caches in Hibernate:

- **Object cache.** As shown in workflow 1 (Figure 1), if the requested user object is not in the cache layer, the object will be fetched from the DBMS. Then, the user object will be stored in the cache layer and can be accessed as a key-value pair using its id (e.g., \{id: 1, User obj\}). If the object is updated, the cached data would be evicted to prevent a stale read (Workflow 2).

To cache database entity objects, developers must add an annotation @Cachable at the class declaration (as shown in User.java in Figure 1). Then, all database entity object retrieved by ID (e.g., retrieved using findUserById()) would be cached. These annotations configure the underlying caching frameworks.

- **Query cache.** The cache mechanism for query cache is slightly different from object cache. For example, the cached data for a select all query on the user table (Workflow 3) would look as follows:

  \[
  \{\text{select * from User} \rightarrow \{\text{id: 1, id: 2}\} \}
  \]

  The cache layer stores the ids of the objects (i.e., id 1 and 2) that are retrieved by the query, and uses the ids to find the cached objects (the corresponding User obj). Thus, the object cache must be enabled to use a query cache. When a user object is updated (workflow 4), the query cache needs to retrieve the updated object from the DBMS to prevent a stale read. Thus, if the queried entity objects are frequently modified, using a query cache may not be beneficial, and may even hinder performance [26].

To cache query results, developers must call a method like cache() before executing the query (Main.java in Figure 1). Such method is used to configure the underlying caching frameworks.

Adding caches incorrectly can introduce overhead to the application. Caching a frequently modified object or query will cause the caching framework to constantly evict and
Figure 2: A working example of CacheOptimizer. The + sign in front of the @Cachable line indicates that the caching configuration is added by CacheOptimizer.

renew the cache, which not only causes cache renewal overhead but may also result in executing extra SQL queries. Therefore, blindly adding caches without understanding the workload may lead to performance degradation 51.

4. CACHE OPTIMIZER

CacheOptimizer optimizes the configuration of caches that are associated with database accesses that occur for a given workload. Hence, our approach needs to recover the workload of an application then to identify which database access occurs within that particular workload. In the following subsections, we explain each step of the inner workings of CacheOptimizer in detail using a working example. The input of the working example shown in Figure 2 consists of two parts: 1) source code of the application and 2) web access logs. Figure 3 shows an overview of CacheOptimizer.

4.1 Recovering Control and Data Flow Graphs

We first need to understand the calling and data flow relationships among methods, and determine which application-level methods are impacted by database caching (i.e., which methods eventually lead to a database access). We therefore extract the call and data flow graphs of the application by parsing the source code of the application using the Eclipse JDT. We opt to parse the source code instead of analyzing the binary since we need to locate the Hibernate annotations in the source code – such annotations are lost after compiling the source code to Java bytecode. We mark all Hibernate methods that access the DBMS (e.g., `query.execute()`) in the call and data flow graphs. Such methods are easy to identify since they are implemented in the same class (i.e., in the `EntityManager` and the `Query` class of Hibernate). Once such methods are marked, we are able to uncover all the application-level methods that are likely to be impacted by optimizing the database cache. In our working example, after generating the call and data flow graphs, and identifying the Hibernate database access methods, we would know that the method `getUserByld()` contains one database access, and the parameter is passed in through a web request.

4.2 Linking Logs to Application-Level Methods

We recover the workload of the application by mining its web access logs. We leverage web access logs because of the following reasons. First, web access logs are typically readily available without needing additional instrumentation since many database-centric applications rely on RESTful web service (based on HTTP web requests) to accept requests from users 49. For example, large companies like IBM, Oracle, Facebook and Twitter all provide RESTful API1. Second, unlike application logs, web access logs have a universal structure (the format of all log lines are the same) 55. Hence, compared to application logs, web access logs are easier to analyze and do not usually change as an application evolves 50.

Web access logs may contain information such as the requestor’s IP, timestamp, time taken to process the request, requested method (e.g. GET), and status of the response. An example web access log may look like:

```
127.0.0.1 [05/Aug/2015:10:38:38 -0400] 1202 “GET /user/1 HTTP/1.1” 200
```

This web access log shows that a request is sent from the local host at August 05, 2015 to get the information of the user whose ID is 1. The status of the response is 200, and the application took 1,202 milliseconds to respond to the request.

In order to know which application-level methods will be executed for each web request, we use static analysis to match the web access logs to application-level methods. CacheOptimizer parses the standard RESTful Web Services (JAX-RS) specifications in order to find the handler method for each web request 42. An example of JAX-RS code is shown below:

```
@RequestMapping ( value = ”/user/[id]”, method = GET )
public User getUserByld (int id) {
    return findUserByld(id);
}
```

In this example, based on the JAX-RS annotations, we know that all GET requests with the URL of form “/user/[id]” will be handled by the `getUserByld` method.

For every line of web access log, CacheOptimizer looks for the corresponding method that handles that web request. After analyzing all the lines of web access logs, CacheOptimizer generates a list of methods (and their frequencies) that are executed during the run of the application.

In our working example, we map every line of web access log to a corresponding web request handling method, i.e., `getUserByld` method.

4.3 Database Access Workload Recovery

We want to determine which database accesses are executed for the workload. Since application-level cache highly depends on the details of the database accesses, we need to recover the types of the database access (e.g., a query versus a select/insert/update/delete of a database entity object by id) and the data that is associated with the database access (e.g., accessed tables and parameters). Such detailed information of database accesses helps us in determining the optimal cache configurations. We first link each web access log to its request-handler method in the code (as described in Section 4.2). Therefore, for each workload, we know the list of request-handler-methods that are executed (i.e., entry points into the application). Then, we conduct a call graph and static flow-insensitive interprocedural taint analysis on

1http://www.programmableweb.com/apis/directory
each web-request-handler method, using the generated call and data flow graphs (as described in Section 1).

Our algorithm for recovering the database access workload is shown in Algorithm 1. For each web-request-handler method, we identify all possible database accesses by traversing all paths in the call graph, and recording the type of the database access. After recovering the database access, we traverse the data flow graph of each web-request-handler method to track the usage of the parameters that are passed in through the web requests. We want to see if the parameters are used for retrieving/modifying the data in the DBMS. Such information helps us better calculate the optimized cache configuration. For example, we would be able to count the number of times a database entity object is retrieved (e.g., according to the id that is specified in the web requests), or how many times a query is executed (e.g., according to the search term that is specified in the web request). For POST, PUT, and DELETE requests, we track the URL (e.g., POST /newUser/1) to which the request is sent, which usually specifies which object the request is updating. If there is no parameter specified, then we assume that the request may modify any of the objects to be conservative on our advice on enabling the cache.

In our working example, we recover a list of database accesses. All of the accesses read data from the User table. In five of the accesses, the parameter is 1 and in one of the accesses, the parameter is 2.

### 4.4 Identifying Possible Caching Locations

After our static analysis step, we recover the location of all the database access methods in the code, and the mapping between Java classes and tables in the DBMS. Namely, we obtain all potential locations for adding calls to the cache configuration APIs. Thus, if a query needs to be cached, we can easily find the methods in the code that execute the query. If we need to add object caches, we can easily find the class that maps to the object’s corresponding table in the DBMS. In our example, we identify that the class User is a possible location to place an object cache. Our static analysis step is very fast (23–77 seconds on a machine with 16G RAM and an i5 2.3GHz CPU) for our studied applications (see Table 1), and is only required when deploying a new release. Thus, the execution time has minimal impact.

We use flow-insensitive static analysis approaches to identify possible caching locations, because it is extremely difficult to recover precise dynamic code execution paths without introducing additional overhead to the application (e.g., using instrumentation). During our static analysis step, if we choose to assign different probabilities to code branches, we may under-count or over-count reads and writes to the DBMS. Under-counting reads may result in failing to cache frequently read objects, which has little or no negative performance impact (i.e., the same as not adding a cache). However, under-counting writes may result in caching frequently modified objects and thus has significant negative effects on performance. In contrast, we choose a conservative approach by considering all possible code execution paths (over-counting) to avoid under-counting reads and writes. We may over-count reads and writes to the DBMS, but over-counting reads has minimal performance impact, since in such cases we would only place cache configuring APIs on objects that are rarely read from the DBMS; over-counting writes means that we may miss some objects that should have been cached, but will not affect the system performance (the same as adding no cache). Hence, our conservative choice by intentionally considering all possible code execution paths (over-counting) ensures that the caching suggestions would not have negative performance impact after placing the suggested caches. Note that there may be some memory costs when turning on the cache (i.e., use more memory), and in RQ2 we evaluate the gain of our approach when considering such costs.

### Algorithm 1: Our algorithm for recovering database accesses.

**Input:** CG, DG, Mthd /* call graph, data flow graph, the request handler method */

**Output:** AccessInfo, Params /* accessed DB tables and DB func type (query or key-value lookup) and parameter of the request */

1. AccessInfo ← ∅; Params ← ∅;
2. /* Traverse the call graph from Mthd */
3. foreach path ∈ CG.findAllPathFrom(Mthd) do
4.   foreach call ∈ path do
5.     if DBCall(call) then
6.       AccessInfo ← AccessInfo ∪ (getAccessedTable(call), getMthdType(call));
end
7. end
8. end
9. end
10. /* Track the usage of the input params */
11. foreach param ∈ Mthd.getParams() do
12.   foreach path ∈ DG.findAllPathFrom(param) do
13.     foreach node ∈ path do
14.       node ← pointToAnalysis(node)
15.       if usedInDBAccessCall(node) then
16.         Params ← Params ∪ (dbAccessCall, node)
end
17. end
18. end
19. end
20. end

---

**Figure 3: Overview of CacheOptimizer.**

![Flowchart of CacheOptimizer](image-url)
1) P1

2) P2

3) P3

4) P4

5) P5

Figure 4: An example of modeling potential cache benefits using a coloured Petri net. A red token represents a read to a specific database entity object (e.g., findUserById(1)), and a blue token represents write to a specific database entity object (updateUserById(1)).

4.5 Evaluating Potential Cache Benefits Using Coloured Petri Net

After linking the logs to handler methods and recovering the database accesses, CacheOptimizer then calculates the potential benefits of placing a cache on each database access call. We use Petri nets [33], a mathematical modeling language for distributed applications, to model the activity of caches such as cache renewal and invalidation. Petri nets allow us to model the interdependencies, so the reached caching decisions are global optimal, instead of focusing on top cache accesses (greedy). Petri nets model the transition of states in an application, and a net contains places, transitions, and arcs. Places represent conditions in the model, transitions represent events, and arcs represent the flow relations among places. Formally, a Petri net $N$ can be defined as:

$$N = (P, T, A)$$

where $P$ is the set of places, $T$ is the set of transitions, and $A$ is the set of arcs. Places may contain tokens, which represent the execution of the net. Any distributions of the tokens in the places of a net represent a set of configurations. A limitation of Petri nets is that there is no distinction between tokens. However, to use Petri nets to evaluate potential cache benefits, we need to model different data types (e.g., a Hibernate query versus an entity lookup by id) and values (e.g., query parameter). Thus, we use an extension of Petri nets, called coloured Petri net (CPN) [33]. In a CPN, tokens can have different values, and the values are represented using colours. Formally, a CPN can be defined as:

$$CPN = (P, T, A, \Sigma, C, N, E, G, I)$$

where $P$, $T$, and $A$ are the same as in Petri nets. $\Sigma$ represents the set of all possible colours (all possible tokens), $C$ maps $P$ to colours in $\Sigma$ (e.g., specify the types of tokens that can be in a place), and $N$ is a node function that maps $A$ into $(P \times T) \cup (T \times P)$. $E$ is the arc expression function, $G$ is the guard function that maps each transition into guard expressions (e.g., boolean), and finally $I$ represents an initialization function that maps each place to a multi-set of token colours.

In our CPN (shown in Figure 4), we define $P$ to be the states of the data in the cache, $P3$ is a repository that stores the total number of database accesses, $P4$ stores the total number of cache hits, and $P5$ stores the number of invalidated caches. $P2$ is an intermediate place for determining whether the data would be cached or invalidated.

We define $T$ to be all database accesses that are recovered from the logs. We define $\Sigma$ to distinguish the type of the database access call (e.g., read/write using ids or queries), and the parameters used for the access (obtained using Algorithm 1). Thus, our $C$ defines that $P4$ can only have colours of database access calls that are reads, and $P1$, $P2$, $P3$, and $P5$ may contain all colours in $\Sigma$. The transition function on $T1$ always forwards the tokens in the initial place $P1$ to $P2$ and $P3$. There are two guard functions on $T2$, where one allows a token to be moved to $P4$ if there are two or more tokens of the same colour in $P2$ (i.e., multiple reads to the same data, so a cache hit), and another guard function makes sure that if there is a write in $P2$, all the same write tokens and the corresponding read tokens are moved to $P5$ (e.g., the cache is invalidated).

In our example (Figure 4), we let red tokens represent the database access call findUserById(1), and blue tokens represent updateUserById(1). In (1), there are two red tokens, and $T1$ is triggered, so the two red tokens are stored in $P2$ and $P3$. Since there are two red tokens in $P2$, $T2$ is triggered, and moves one red token to $P4$ (a cache hit). The resulting CPN is shown in (2). When a blue token appears in $P1$, $T1$ is triggered and moves the blue token to both $P2$ and $P3$. Since there is a blue token in $P2$, $T2$ is triggered, and we move both the red and blue token to $P5$ (cache invalidation). The final resulting Petri net is shown in (3). Note that $T2$ acts slightly different for tokens that represent query calls. When an object is updated, the query cache needs to retrieve the updated object from the DBMS to prevent a stale read. Thus, to model the behaviour, $T2$ would be triggered to move the query token to $P5$ from $P2$ if we see any token that represents a modification to the query table.

We use the recovered database accesses of the workload to execute the CPN. For all tokens that represent the database access to the same data (e.g., a read and write to user by id 1), we examine their total counts in $P3$ and $P4$ to calculate the miss ratio (MR) of the cache. MR can be calculated as one minus the total number of cache hits in $P4$ divided by the total number of calls in $P3$. We choose MR because it is used in many prior studies to evaluate the effectiveness of caching (e.g., [21, 33, 62]). If MR is too high, caching the data would not give any benefit. For example, if a table is constantly updated, then data in that table should not be cached. Thus, we define a threshold to decide whether a database access call should be cached. In our CPN, if MR is smaller than 35%, then we place the cache configuration code for the corresponding query (query cache) or table (object cache). Since object cache must be turned on to utilize query cache, we enable query cache only if the MR of the object cache is under the threshold. Such that, there would not exist conflicting decisions for object and query cache. We choose 35% to be more conservative on enabling caches. We also vary MR to investigate the impact of MR.
4.6 Configuring the Caching Frameworks

CacheOptimizer automatically adds the appropriate calls to the cache configuration API. Since the locations that require adding cache configuration APIs may be scattered across the code, CacheOptimizer helps developers reduce manual efforts by automatically adding these APIs to the appropriate locations. For example, if the query that is executed by the request “/user?query=Peter” should be cached, CacheOptimizer would automatically call the caching framework’s API to cache the executed query in the corresponding handler method searchUserByName. In our example shown in Figure 2, the miss ratio of caching objects in the User class is 0.33, which is smaller than our threshold 0.35. CacheOptimizer automatically adds the @Cacheable annotation to the source code to enable cache for the User class.

5. EVALUATION

In this section, we present the evaluation of CacheOptimizer. We first discuss the applications that we use for our evaluation. Then we focus on two research questions: 1) what is the performance improvement after using CacheOptimizer; and 2) what is the gain of CacheOptimizer when considering the cost of such caches.

Experimental Setup. We evaluate CacheOptimizer on three open-source web applications: Pet Clinic [44], Cloud Store [29], and OpenMRS [41]. Table 1 shows the detailed information of these three applications. All three applications use Hibernate as the underlying framework to access database, and use MySQL as the DBMS. We use Tomcat as our web server, and use Ehcache as our underlying caching framework. Pet Clinic, which is developed by Spring [32], aims to provide a simple yet realistic design of a web application. Cloud Store is a web-based e-commerce application, which is developed mainly for performance testing and benchmarking. Cloud Store follows the TPC-W performance benchmark standard [1]. Finally, OpenMRS is large-scale open-source medical record application that is used worldwide. OpenMRS supports both web-based interfaces and RESTful services.

We use one machine each for the DBMS (8G RAM, Xeon 2.67GHz CPU), web server (16G RAM, Intel i5 2.3GHz), and JMeter load driver (12G RAM, Intel Quad 2.67GHz). The three machines are all connected on the same network. We use performance test suites to exercise these applications when evaluating CacheOptimizer. Performance test suites aim to mimic the real-life usage of the application and ensure that all of the common features are covered during the test [4]. Thus, for our evaluation, performance test suites are a more appropriate and logical choice over using functional tests. We use developer written tests for Pet Clinic [23], and work with BlackBerry developers on creating the test cases for the other applications. For Cloud Store, we create test cases to cover searching, browsing, adding items to shopping carts, and checking out. For OpenMRS, we use its RESTful APIs to create test cases that are composed of searching (by patient, concept, encounter, and observation etc), and editing/adding/retrieving patient information. We also add randomness to our test cases to better simulate real-world workloads. For example, we add randomness to ensure that some customers may checkout, and some may not. We use, for our performance tests, the MySQL backup files that are provided by Cloud Store and OpenMRS developers. The backup file for Cloud Store contains data for over 5K patients and 500K observations. The backup file for Cloud Store contains about 300K customer data and 10K items.

RQ1: What is the performance improvement after using CacheOptimizer?

Motivation. In this RQ, we want to examine how well the performance of the studied database-centric web applications can be improved when using CacheOptimizer to configure the caching framework.

Approach. We run the three studied applications using the performance test suites under four different sets of cache configurations: 1) without any cache configuration (NoCache), 2) with default cache configuration (DefaultCache), cache configurations that are already in the code, which indicates what developers think should be cached, 3) with enabling all possible caches (CacheAll), and 4) with configurations that are added by CacheOptimizer. We compare the performance of the applications when configured using these four different sets of cache configurations. We work with performance testing experts from BlackBerry to ensure that our evaluation steps are appropriate, accurate, and realistic. We use throughput to measure the performance. The throughput is measured by calculating the number of requests per second throughout the performance test. A higher throughput shows the effectiveness of the cache configuration, as more requests can be processed within the same period of time.

Table 2 shows the performance improvement of the applications under four sets of configurations. We use NoCache as a baseline, and calculate the throughput improvement after applying CacheOptimizer, CacheAll, and DefaultCache. The default cache configuration of Pet Clinic does not enable any cache. Therefore, we only show the performance improvement of DefaultCache for Cloud Store.

Table 1: Statistics of the studied applications.

<table>
<thead>
<tr>
<th>Application</th>
<th>Total lines of code</th>
<th>Number of Java files</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pet Clinic</td>
<td>3.8K</td>
<td>51</td>
</tr>
<tr>
<td>Cloud Store</td>
<td>35K</td>
<td>193</td>
</tr>
<tr>
<td>OpenMRS</td>
<td>3.8M</td>
<td>1,890</td>
</tr>
</tbody>
</table>

Table 2: Performance improvement (throughput) against NoCache after applying different cache configurations.

<table>
<thead>
<tr>
<th>Application</th>
<th>NoCache</th>
<th>CacheOptimizer</th>
<th>CacheAll</th>
<th>DefaultCache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pet Clinic</td>
<td>98.7</td>
<td>125.1 (+27%)</td>
<td>108.4 (+10%)</td>
<td>—</td>
</tr>
<tr>
<td>Cloud Store</td>
<td>110.7</td>
<td>263.4 (+138%)</td>
<td>249.3 (+125%)</td>
<td>114.7 (+4%)</td>
</tr>
<tr>
<td>OpenMRS</td>
<td>21.3</td>
<td>30.8 (+45%)</td>
<td>25.5 (+20%)</td>
<td>27.7 (+30%)</td>
</tr>
</tbody>
</table>
Table 3: Total number of possible places to add cache in the code, and the number of location that are enabled by CacheOptimizer and that exist in the DefaultCache.

<table>
<thead>
<tr>
<th></th>
<th>Total</th>
<th>CacheOptimizer</th>
<th>DefaultCache</th>
<th>CacheOptimizer</th>
<th>DefaultCache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pet Clinic</td>
<td>112</td>
<td>3 (2.7%)</td>
<td>0</td>
<td>112</td>
<td>0</td>
</tr>
<tr>
<td>Cloud Store</td>
<td>229</td>
<td>2 (6%)</td>
<td>0</td>
<td>229</td>
<td>0</td>
</tr>
<tr>
<td>OpenMRS</td>
<td>112</td>
<td>16 (14%)</td>
<td>0</td>
<td>122</td>
<td>1 (4%)</td>
</tr>
</tbody>
</table>

Figure 5: Number of handled requests overtime (cumulative).

and OpenMRS. Using CacheOptimizer, we see a throughput improvement of 27%, 138% and 45% for Pet Clinic, Cloud Store and OpenMRS, respectively. The throughput improvement of applying CacheOptimizer is always higher than that of DefaultCache and CacheAll for all the studied applications. Figure 5 further shows the cumulative throughput overtime. We can see that for the three studied applications, the throughput is about the same at the beginning regardless of us adding cache or not. However, as more requests are received, the benefit of caching becomes more significant. The reason may be that initially when the test starts, the data is not present in the cache. CacheOptimizer is able to discover the more temporal localities (reuse of data) in the workload and help developers configure the application-level cache more optimally. Therefore, as more requests are processed, frequently accessed data is then cached, which significantly reduces the overhead of future accesses. We see a trend that the longer the test runs, the more benefit we get from adding cache configuration code using CacheOptimizer. We also observe that the performance of Cloud Store with DefaultCache is close to the performance with no cache. Such an observation shows in some instances developers do not have a good knowledge of optimizing cache configuration in their own application.

CacheOptimizer enables a small number of caches to improve performance. CacheOptimizer can help developers change cache configurations quickly without manually investigating a large number of possible cache locations. Table 3 shows the total number of possible locations to place calls to object and query cache APIs in the studied applications. We also show the number of CacheOptimizer enabled caches, and the number of DefaultCache enabled caches. CacheOptimizer suggests adding object cache configuration APIs to a fraction (6–55%) of the total number of possible cache locations. In OpenMRS and Cloud Store, where there are more Hibernate queries, CacheOptimizer is able to improve performance by enabling 0.9% and 38% of all the possible caches, respectively. For the object cache of Cloud Store, CacheOptimizer even suggests enabling a smaller number of caches than DefaultCache. For large applications like OpenMRS with 112 possible object caches and 229 possible query caches, manually identifying the optimized cache configuration is time-consuming and may not even be possible.

Discussion. In our evaluation of CacheOptimizer, we observe a larger improvement in Cloud Store. After a manual investigation, we find that CacheOptimizer caches the query results that contain large binary data, e.g., pictures. Since the sizes of pictures are often larger, caching them significantly reduces the network transfer time, and thus results in a large performance improvement. We see less improvement when using DefaultCache, because most database access calls are done through queries (like workflow 3 in Figure 1), while the default cache configurations of Cloud Store are mostly for object cache (Table 3). Thus, enabling only object caches does not help improve performance. In OpenMRS, both CacheOptimizer and DefaultCache cache some database entity objects that are not often changed. However, CacheOptimizer is able to identify more object caches and queries that should be cached to further improve performance. We also see that the overhead of CacheAll causes OpenMRS to run slower when compared to DefaultCache. In Pet Clinic, we find that caching the owner information significantly improves the performance of searches. Moreover, since the number of vets in the clinic is often unchanged, caching the vet information also speeds up the application.

Adding cache configuration code, as suggested by CacheOptimizer, improves throughput by 27–138%, which is higher than using the default cache configuration and enabling all possible caches. The sub-optimal performance of DefaultCache shows that developers have limited knowledge of adding cache configuration.

RQ2: What is the gain of CacheOptimizer when considering the cost of such caches?

Motivation. In the previous RQ, we see that CacheOptimizer helps improve application throughput significantly. However, caching may also bring some memory overhead to the application, since we need to store cached objects in
the memory. As a result, in this RQ, we want to evaluate CacheOptimizer-suggested cache configuration when considering both the cost (increase in memory usage) and the benefit (improvement in throughput).

Approach. In order to evaluate CacheOptimizer when considering both benefit and cost, we define the gain of applying a configuration as:

\[ \text{Gain}(c) = \text{Benefit}(c) - \text{Cost}(c), \]  

where \( c \) is the cache configuration, \( \text{Gain}(c) \) is the gain of applying \( c \), while \( \text{Benefit}(c) \) and \( \text{Cost}(c) \) measure the benefit and the cost, respectively, of applying \( c \). In our case study, we measure the throughput improvement in order to quantify the benefit of caching, and we measure the memory overhead in order to quantify the cost of caching. We use the throughput and memory usage when no cache is added to the application as a baseline. Thus, \( \text{Benefit}(c) \) and \( \text{Cost}(c) \) are defined as follows:

\[ \text{Benefit}(c) = \text{TP}(c) - \text{TP}(\text{no cache}), \]  

\[ \text{Cost}(c) = \text{MemUsage}(c) - \text{MemUsage}(\text{no cache}), \]  

where \( \text{TP}(c) \) is the average number of processed requests per second with cache configuration \( c \), and \( \text{MemUsage}(c) \) is the average memory usage with cache configuration \( c \).

Since the throughput improvement and the memory overhead are not in the same scale, the calculated gain by Equation 1 may be biased. Therefore, we linearly transform both \( \text{Benefit}(c) \) and \( \text{Cost}(c) \) into the same scale by applying min-max normalization, which is defined as follows:

\[ x' = \frac{(x - x_{\text{min}})}{(x_{\text{max}} - x_{\text{min}})}, \]  

where \( x \) and \( x' \) are the values of the metric before and after normalization, respectively; while \( x_{\text{max}} \) and \( x_{\text{min}} \) are the maximum and the minimum values of the metric, respectively. We note that if one wants to compare the gain of applying multiple configurations, the maximum and the minimum values of the metric are calculated by considering all the values of the metrics across the different configurations, including having no cache. For example, if one would like to compare the gain of applying CacheOptimizer and CacheAll, throughput\text{max} is the maximum throughput of applying CacheOptimizer, CacheAll, and NoCache.

To evaluate CacheOptimizer, in this RQ, we compare the gain of applying CacheOptimizer, CacheAll, and DefaultCache against NoCache. The larger the gain, the better the cache configuration. If the gain is larger than 0, the cache configuration is better than using NoCache. In order to understand the gain of leveraging cache configuration throughout the performance tests, we split each performance test into different periods. Since a performance test with different cache configurations runs for a different length of time (see Figure 4), we split each test by each thousand of completed requests. For each period, we calculate the gain of applying CacheOptimizer, CacheAll, and DefaultCache.

We study whether there is a statistically significant difference in gain, between applying CacheOptimizer and CacheAll, and between applying CacheOptimizer and DefaultCache. To do this we use the Mann-Whitney U test \[39\] on the gains, as the gains may be highly skewed. Since the Mann-Whitney U test is a non-parametric test, it does not have any assumptions on the distribution. A p-value smaller than 0.05 indicates that the difference is statistically significant. We also calculate the effect sizes in order to quantify the differences in gain between applying CacheOptimizer and CacheAll, and between applying CacheOptimizer and DefaultCache. Unlike the Mann-Whitney U test, which only tells us whether the difference between the two distributions is statistically significant, the effect size quantifies the difference between the two distributions. Since reporting only the statistical significance may lead to erroneous results (i.e., if the sample size is very large, the p-value are likely to be small even if the difference is trivial) \[35\], we use Cliff’s d to quantify the effect size \[19\]. Cliff’s d is a non-parametric effect size measure, which does not have any assumption of the underlying distribution. Cliff’s d is defined as:

\[ \text{Cliff’s d} = \frac{\#(x_i > x_j) - \#(x_i < x_j)}{m \times n}, \]  

\[ \text{effect size} = \begin{cases} \text{trivial} & \text{if Cliff’s d} < 0.147 \\ \text{small} & \text{if } 0.147 \leq \text{Cliff’s d} < 0.33 \\ \text{medium} & \text{if } 0.33 \leq \text{Cliff’s d} < 0.474 \\ \text{large} & \text{if } 0.474 \leq \text{Cliff’s d} \end{cases}, \]  

where \# is defined the number of times, and the two distributions are of the size \( m \) and \( n \) with items \( x_i \) and \( x_j \), respectively. We use the following thresholds for Cliff’s d \[19\]:

<table>
<thead>
<tr>
<th></th>
<th>\text{Cliff’s d} \text{ &gt; 0.474}</th>
<th>\text{Cliff’s d} \text{ &gt; 0.33}</th>
<th>\text{Cliff’s d} \text{ &gt; 0.147}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{Pet Clinic}</td>
<td>\text{&gt; 0.61 (large)}</td>
<td>\text{&gt; 0.55 (medium)}</td>
<td>\text{&gt; 0.47 (large)}</td>
</tr>
<tr>
<td>\text{Cloud Store}</td>
<td>\text{&gt; 0.61 (large)}</td>
<td>\text{&gt; 0.55 (medium)}</td>
<td>\text{&gt; 0.47 (large)}</td>
</tr>
<tr>
<td>\text{OpenMRS}</td>
<td>\text{&gt; 0.61 (large)}</td>
<td>\text{&gt; 0.55 (medium)}</td>
<td>\text{&gt; 0.47 (large)}</td>
</tr>
</tbody>
</table>

Table 4: Comparing the gain of the application under three different configurations: CacheOptimizer, CacheAll, and DefaultCache

<table>
<thead>
<tr>
<th></th>
<th>\text{gain(CacheOptimizer)} \text{ &gt; gain(CacheAll)}</th>
<th>\text{gain(CacheOptimizer)} \text{ &gt; gain(DefaultCache)}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{Pet Clinic}</td>
<td>\text{&lt;&lt; 0.001}</td>
<td>\text{0.81 (large)}</td>
</tr>
<tr>
<td>\text{Cloud Store}</td>
<td>\text{0.01}</td>
<td>\text{0.32 (small)}</td>
</tr>
<tr>
<td>\text{OpenMRS}</td>
<td>\text{&lt;&lt; 0.001}</td>
<td>\text{0.61 (large)}</td>
</tr>
</tbody>
</table>

Results. CacheOptimizer outperforms DefaultCache and CacheAll when considering the cost of cache. Table 4 shows the result of our Mann-Whitney U test and Cliff’s d value when comparing the gain of applying CacheOptimizer with that of CacheAll and DefaultCache. We find that in all three studied applications, the gain of CacheOptimizer is better than the gain of CacheAll and DefaultCache (statistically significant). We also find that the effect sizes of comparing CacheOptimizer with CacheAll on gain are large for Pet Clinic (0.81) and OpenMRS (0.95). The only exception is Cloud Store, where the Cliff’s d value indicates that the effect of gain is small (0.32) when comparing CacheOptimizer with CacheAll. On the other hand, when compared to DefaultCache, CacheOptimizer has a large effect size for both Cloud Store and OpenMRS.

Discussion. We investigate the memory overhead of applying CacheOptimizer, CacheAll, and DefaultCache. We use the Mann-Whitney U test and measure effect sizes using Cliff’s d to compare the memory usage between applying CacheOptimizer and the memory usage of having no cache, CacheAll, and DefaultCache, respectively. The memory usage of applying CacheOptimizer and having no cache is statistically indistinguishable for Pet Clinic and OpenMRS, while for Cloud Store, applying CacheOptimizer has statistically significantly more memory usage than having no cache with a large effect size (0.78). This may explain why we see larger throughput improvement in Cloud Store. For OpenMRS, the memory usage of applying CacheOptimizer and
When considering both the benefit (throughput improvement) and cost (memory overhead), the gain of applying CacheOptimizer is statistically significantly higher than CacheAll and DefaultCache.

6. THREATS TO VALIDITY

External Validity. We only evaluated CacheOptimizer on three applications, so our findings may not generalize to other applications. We choose the studied applications with various sizes across different domains to improve the generalizability. However, evaluating CacheOptimizer on other applications would further show the generalizability of our approach. We implement CacheOptimizer specifically for Hibernate-based web applications. However, the approach in CacheOptimizer should be applicable to applications using different object-relational mapping frameworks or other database abstraction technologies. For example, our approach for recovering the database accesses from logs may also be used by non-Hibernate-based applications. With minor modifications (e.g., changes needed to the definitions of the tokens and transition functions in the coloured Petri net), CacheOptimizer can be leveraged to improve cache configurations of other applications.

Construct Validity. The performance benefits of caching highly depend on the workloads. Thus, we use performance tests to evaluate CacheOptimizer. It is possible that the workload from the performance tests may not be representative enough for field workloads. However, CacheOptimizer does not depend on a particular workload, nor do we have any assumption on the workload when conducting our experiments. CacheOptimizer is able to analyze any given workload and find the optimal cache configuration for different workloads. If the workload changes greatly and the cache configuration is no longer optimal. CacheOptimizer can save developers’ time and effort by automatically finding a new optimal cache configuration. For example, developers can feed their field workloads on a weekly or monthly basis, and CacheOptimizer would help developers optimize the configuration of their caching frameworks. To maximize the benefit of caching, our approach aims to “overfit” the cache configurations to a particular workload. Thus, similar to other caching algorithms or techniques, our approach will not work if the workload does not contain any repetitive reads from the DBMS.

Our approach for recovering the database access. Prior research leverages control flow graphs to recover the executed code paths using logs [62]. We do not leverage control flow graphs to recover the database accesses from web access logs for two reasons. First, as a basic design principle of RESTful web services, typically one web-request-handling method maps to one or very few database accesses [32, 19]. Second, although leveraging control flows may give us richer information about each request, it is impossible to know which branch would be executed based on web access logs. Heuristics may be used to calculate the possibility of taking different code paths. However, placing the cache incorrectly can even cause performance degradation. Thus, to be conservative when enabling caching and to ensure that CacheOptimizer would always help improve performance, we consider all possible database access calls. Our overestimation ensures that CacheOptimizer would not cache data that has a high likelihood of being frequently modified, so the CacheOptimizer added cache configurations should not negatively impact the performance. Future research should consider the use of control flow information for optimizing the cache configurations.

Cache concurrency level. There are different cache concurrency levels, such as read-only and read/write. In this paper we only consider the default level, which is read/write. Read/write cache concurrency strategy is a safer choice if the application needs to update cached data. However, considering other cache concurrency levels may further improve performance. For example, read-only caches may perform better than read/write cache if the cached data is never changed. Future research should add cache concurrency level information to CacheOptimizer when trying to optimize cache configuration.

Distributed cache environment. Cache scheduling is a challenging problem in a distributed environment due to cache concurrency management. Most caching frameworks provide different algorithms or mechanisms to handle such issues. Since the goal of CacheOptimizer is to instruct these caching frameworks on what to cache, we rely on the underlying caching frameworks for cache concurrency management. However, the benefit of using CacheOptimizer may not be as pronounced in a distributed environment.

7. CONCLUSION

Modern large-scale database-centric web applications often leverage different application-level caching frameworks, such as Ehcache and Memcached, to improve performance. However, these caching frameworks are different from traditional lower-level caching frameworks, because developers need to instruct these application-level caching frameworks about what to cache. Otherwise these caching frameworks are not able to provide any benefit. In this paper, we propose CacheOptimizer, an automated lightweight approach that determines what should be cached in order to utilize such application-level caching frameworks for Hibernate-based web applications. CacheOptimizer combines static analysis of source code and logs to recover the database accesses, and uses a coloured Petri net to model the most effective caching configuration for a workload. Finally, CacheOptimizer automatically updates the code with the appropriate cache configuration code. We evaluate CacheOptimizer on three open source applications (Pet Clinic, Cloud Store, and OpenMRS). We find that CacheOptimizer improves the throughput of the entire application by 27–138% (compared to DefaultCache and CacheAll), and the increased memory usage is smaller than the applications’ default cache configuration and turning on all caches. The sub-optimal performance of the default cache configurations highlights the need for automated techniques to assist developers in optimizing the cache configuration of database-centric applications.
8. REFERENCES


[32] IBM. Restful web services: The basics. http:


