View-Centric Performance Optimization for Database-Backed Web Applications

Junwen Yang[1], Cong Yan[2], Chengcheng Wan[1], Shan Lu[1], Alvin Cheung[2]
[1]*University of Chicago*, junwen, cwan, shanlu@uchicago.edu [2]*University of Washington*, congy, akcheung@cs.washington.edu

***Abstract*—Web developers face the stringent task of designing informative web pages while keeping the page-load time low. This task has become increasingly challenging as most web contents are now generated by processing ever-growing amount of user data stored in back-end databases. It is difficult for developers to understand the cost of generating every web-page element, not to mention explore and pick the web design with the best trade-off between performance and functionality. In this paper, we present Panorama, a view-centric and database-aware development environment for web developers. Using database-aware program analysis and novel IDE design, Panorama provides developers with intuitive information about the cost and the performance-enhancing opportunities behind every HTML element, as well as suggesting various global code refactorings that enable developers to easily explore a wide spectrum of performance and functionality trade-offs.**

I. Introduction

A. Motivation

High-quality web applications need to provide both good functionality — informative and well-displayed web-page content, and good performance — lower than 2 seconds of page load time [1], with every second's delay causing 11% fewer page views, a 16% decrease in customer satisfaction, and 7% loss in conversions [2]. These functionality and performance requirements are increasingly difficult to satisfy *simultaneously*, as modern web-page content often requires processing a huge amount of user data to generate and the amount of user data often increases by 10× per year [3]. Indeed, real-world web developers often don't understand the amount of data processing required to render their web pages, which results in numerous design changes to address performance issues [4]. Because of this, tools that can help developers understand performance implications of their web-page design and explore designs with different performance-functionality trade-offs can greatly improve the web development process.

To better understand the challenges faced by web developers, consider Tracks [5], a popular task-management application constructed using the Ruby on Rails framework. Tracks has a `todos/index` page displaying all the to-do's for a user. At one point, users complained that this page was unreasonably slow even though very few to-do's were displayed [6]. After some debugging, developers turned their attention to a sidebar on this page, which displayed all the projects the user had been involved in. As shown in Figure 1, the view

http://hyperloop.cs.uchicago.edu/panorama

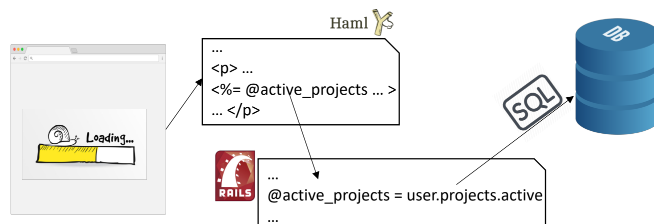

Fig. 1: Performance understanding challenge

file `sidebar.html.erb`, which produces this sidebar, renders these projects based on a Ruby variable `@active_projects` embedded in the HTML file; this variable is computed in a controller Ruby file `todos_controller.rb` through a seemingly straightforward assignment `@active_projects = user.projects.active`. It turns out that this code actually retrieves objects stored in a database, and is translated into a SQL query by the Rails framework at run time. It was the long time taken for the database to run this query that resulted in the poor performance observed by users. After realizing this, developers decided to remove the sidebar , and users can now see the main content of this page, the to-do items, much quicker.

This performance-functionality trade-off represents one of the many challenges that web developers face:

Understanding the performance cost behind every rendered web-page element is challenging for developers and requires cross-stack knowledge as the above example shows. As shown in Figure 1, modern web pages commonly contain dynamically generated contents. Consequently, the cost of a web-page element includes not only browser rendering time, but also web server computation time and backend database server processing time. As web applications are often constructed using the Model, View, Controller [7] architecture, it is difficult for users to manually search through multiple files across application modules to identify code that leads to performance issues, and it is even more difficult for developers to reason about what database queries could be issued as their application executes and how much data would be retrieved from database for rendering.

Existing profiling tools are insufficient in aiding with this regard. Some [8] only account for the client-side cost of every HTML tag, but do not account for the server-side cost; others [9] report database query cost but do not attribute queries to specific HTML tag, and hence cannot provide direct

guidance to web-page design. Furthermore, all these profiling tools rely on workloads provided by developers, and therefore cannot help predict performance problems that manifest from "real-world" workloads.

Exploring the performance-functionality trade-offs among different web-page designs is also challenging for developers, requiring cross-module and cross-language code refactoring. For example, to remove a web-page element, it is insufficient to just remove the corresponding HTML tag from the view file. Developers also need to check which variables are referred to by that HTML tag, which controller code snippets generated those variables, and whether removing those code snippets altogether will affect other parts of the application. In addition, there are also other web-page design alternatives with different performance-functionality tradeoffs. Unfortunately, most require global code restructuring and are difficult to carry out without tool support. Worse yet, recent work on detecting and fixing database-related inefficiencies in web applications only focuses on inefficient ORM-API usage, unnecessary data retrieval, and redundant queries [10]–[12], and is completely oblivious to web-page designs. In short, existing techniques do not consider performance-enhancing opportunities that require web-page design changes (which we refer to as *view changes*), and hence cannot help developers explore the performance-functionality trade-off space.

Indeed, empirical studies [4] have found that about a quarter of real-world web application performance problems are solved by developers through view changes, like pagination and view content removal. These changes often bring much more performance improvement than the view-preserving ones (8.79× vs 2.16× on average) but involve more changes (2.9 files vs. 1.4 files). They are difficult to perform manually and having good tooling support is hence crucial.

B. Contributions

In this work, we present a framework called Panorama that provides a view-centric and database-aware analysis for web developers to understand and optimize their database-backed web applications. Panorama currently targets applications written using the Ruby on Rails framework, and makes three major contributions as illustrated in Figure 2.

Panorama provides a view-centric estimator that helps developers understand the data-processing cost behind every HTML tag. Panorama both dynamically monitors database query performance using the test workload, statically estimates data processing complexity independent of any specific workload, and carefully attributes the cost to every HTML tag through its cross-stack dependency analysis. The details will be presented in Section IV.

Panorama provides a view-aware performance optimizer that helps developers carry out view-changing code refactoring to improve performance. Panorama suggests a variety of refactorings that (1) change the manner of content rendering (i.e., pagination or asynchronous loading); or (2) change the accuracy of the rendered contents (i.e., approximation); or (3) remove certain web-page contents from rendered contents.

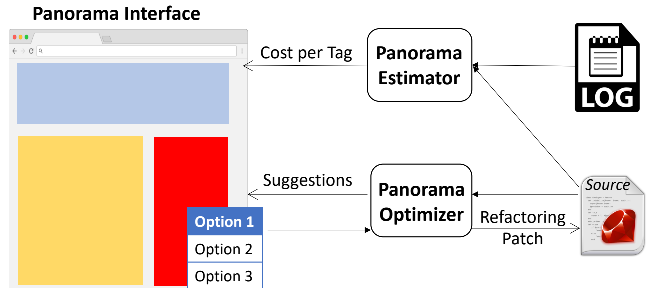

Fig. 2: Panorama overview

Through static program analysis, Panorama not only identifies opportunities for applying such refactoring, but also automatically suggests patches that complete such refactoring, often involving modifications to multiple files in model, view, and controller components. We present the details in Section V.

Panorama provides a unique interface for developers to effectively exploring different web-page designs with different performance-functionality trade-offs. Instead of separately presenting profiling information and refactoring suggestions, Panorama integrates them in the web browser—while testing a page of their web applications, the data processing cost for each HTML tag is presented as a heat map in the browser. Developers can right click on each HTML tag to see the different view-changing options for performance enhancement; they can choose any option and immediately see an updated web page with an updated heat map in the browser, with all code refactoring automatically done by Panorama in an accompanying Ruby editor.

We evaluated Panorama on 12 popular open-source Ruby on Rails applications. Panorama statically identifies 149 performance-enhancing opportunities through view changes. We randomly sampled 15 view changes suggested by Panorama and found that by applying the patches automatically generated by Panorama, these 15 view changes speed up end-to-end page load time by 4.5× on average (38× maximum), using database workloads that are similar to those used in real-world deployments. We believe the benefits will increase with even larger workloads. Furthermore, we conducted a thorough user study with 100 participants from Amazon Mechanical Turk. The study shows that web pages with these view changes are considered as similar or better than the original web pages in most cases, with more users preferring the design suggested by Panorama than the original ones. This user study result, as well as the fact that these optimizations save computation resources on web servers and database servers, justify the need for developers to explore the performance-functionality trade-off space in web application design, with Panorama being a first step towards that goal.

II. Background

Rails applications are structured based on the model-view-controller (MVC) architecture. We illustrate this in Figure 3. When a client requests for a URL such as `http://foo.com/projects/index/1` ①, a *controller action* "`projects/index`" is triggered. This action takes in

the parameters from the request (e.g., "1" in the URL as `params[:id]`) and interacts with the DBMS by calling the ActiveRecord API implemented by the Rails framework ②. Rails translates the function calls into SQL queries ③, whose results ④ are serialized into *model* objects (e.g., the `Project` model) and returned to the controller ⑤. Then, the returned objects are passed to the *view* ⑥ in order to generate a webpage ⑦ to send back to users ⑧. Each model is derived from `ActiveRecord`, and is mapped to a database table by Rails. A view file (ends with `.erb` or `.haml`) usually involves multiple languages including html, JavaScript, and ruby. The ruby code can dynamically generate the content of html elements or decide which element to show.

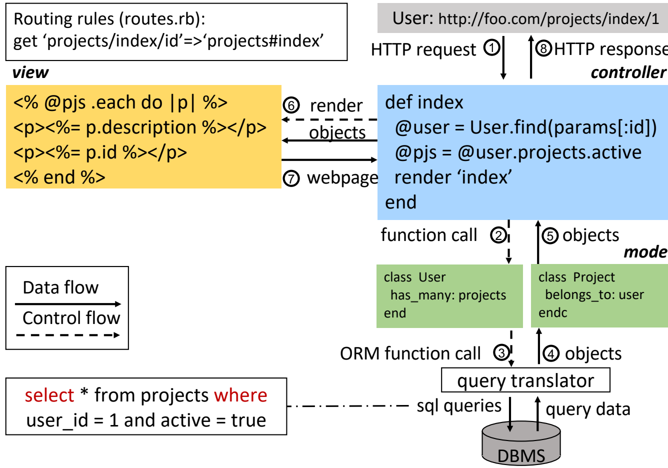

Fig. 3: Rails application architecture based on the MVC pattern

III. Static Analysis Framework

Panorama leverages a database-aware static analysis framework for Rails applications that we briefly describe below.

A. Action dependency graph

Panorama's static analysis centers around the action-dependency graph (ADG) that is constructed for each controller action. Figure 4 shows an example of ADG.

An ADG is a database-aware extension of the traditional program-dependence graph (PDG) [13]. Every node n in the ADG represents an intermediate representation (IR) statement in the corresponding action's JRuby [14] IR. Every edge e represents either control dependency or data dependency. Edges shown in Figure 4 all represent data dependencies.

In contrast to the PDG, every node in the ADG that issues a SQL query is associated with a query tag in ADG, such as node ① and node ② in Figure 4. Information about SQL queries that are issued and the tables they referenced are determined by analyzing `ActiveRecord` function calls and recorded in the ADG.

Since view files may also contain Ruby code to process or render data, they are also analyzed during the ADG construction. Specifically, for every action, like `user/show` in Figure 4, its corresponding view file is identified based on an explicit `render` statement or implicit file-name matching

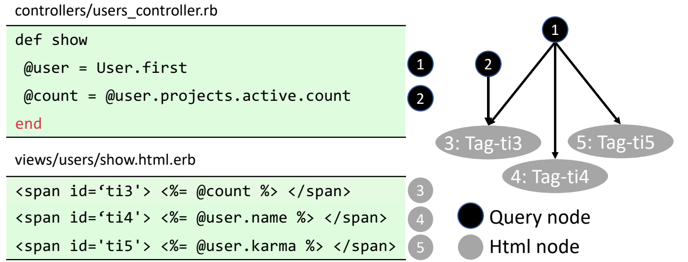

Fig. 4: Excerpt of an Action Dependency Graph

(as in Figure 4). The corresponding view file is then parsed, with all Ruby code embedded inside `<% ...%>` extracted and inlined as part of the ADG, like the three statements inside `show.html.erb` and the ADG shown in Figure 4.

B. Annotating the view component

In order for Panorama to attribute performance data correctly to each HTML tag, Panorama pre-processes every view file in the input web application to assign every HTML tag a unique ID. That is, for every tag <tag> that does not already have an ID, Panorama turns it into `<tag id = ti>`, where `ti` is a unique ID, as shown in the <span> tags of Figure 4. The current prototype of Panorama does not handle HTML tags that are programmatically generated by JavaScript code.

As mentioned, the view file has Ruby code embedded within it. For every node in ADG whose source code is in a view file, Panorama identifies its inner-most surrounding HTML tag and associates it with the corresponding tag ID. Panorama also checks whether its corresponding content is rendered or not by analyzing the HTML, and assigns an `is_rendered` property accordingly. This information will help Panorama attribute data processing cost to each HTML element and identify alternative view designs, as we will explain in later sections.

IV. Panorama View-Centric Cost Estimator

There are two key tasks in Panorama's cost estimation. First, given an HTML tag, Panorama determines which database queries are executed to generate the data that is rendered through that tag (we refer to them as contributing queries). Second, for each HTML tag, Panorama measures the data processing cost needed to render it.

While a web page's load time consists of client side rendering time, network communication between client and server, computation time on the server, and database query cost, Panorama's estimator currently focuses on database query cost, as query time often contributes a significant portion of the page load time. This is particularly true as the data size increases, and large query results lead to even more computation and rendering time. Query time/complexity is also difficult for developers to estimate, particularly given the ORM abstraction. As future work, we will incorporate other profiling tools to measure the performance of the client code [8], network, and server computation as part of Panorama.

A. Identifying contributing queries

Panorama identifies contributing queries for an HTML tag by statically analyzing control and data dependencies in the ADG. Given an HTML tag in a view file, Panorama first identifies all ADG nodes N that contain the tag's unique ID — these nodes contain the Ruby code embedded in the HTML tag. Then, Panorama traces backward along the ADG edges to identify all query nodes that any node in N has control or data dependence upon. All these queries, each identified by its ADG node ID and Ruby source code location, are considered as contributing queries of this HTML tag. For example, in Figure 4, tracing dependency edges backward from node ③, which corresponds to HTML tag with id `ti1`, will identify two contributing queries, nodes ① and ②.

Panorama further conducts forward dependency checking in the ADG to see how many other HTML tags each query node contributes to. This number can be used as a weight in computing the data processing cost of an HTML tag — if a query result is used to generate k HTML tags (e.g., the query node ① contributes to three HTML tags in Figure 4), we could attribute $1/k$ of the query cost to each HTML tag if the web developers choose so (while by default Panorama attributes the complete query cost to each tag).

B. Cost analysis

Panorama offers two modes of cost estimation with or without relying on testing workload.

1) Dynamic profiling: If a testing workload is available, Panorama will measure the cost of each contributing query during a testing run. However, using the query execution log from the backend database engine as the testing workload does not work — the database engine has no knowledge about frontend Ruby and HTML code, and hence does not allow Panorama to connect the statement in the web application that issues the query and the HTML tag uses the query result.

Panorama instead conducts its profiling through a hook API provided by Rails infrastructure, `ActiveRecordQueryTrace`. This API allows its hooked code to be called before and after issuing each SQL query. Using this mechanism, Panorama logs the amount of time of each query and the line of source code that issues this query during the testing run, and attributes the time to the corresponding HTML tags using contributing query analysis as discussed above.

2) Static estimation: Since a bottleneck-exposing workload may not be available during in-house testing, Panorama also uses static analysis to estimate the potential data-processing cost (in terms of its data complexity) to render each HTML tag. For ease of estimation, Panorama assumes that all tables in the database have the same size D. Then, for each contributing query, Panorama estimates its complexity (i.e., how its execution time might increase with D) by considering: (1) the number of times this query might be issued, and (2) time taken to execute one query instance.

To estimate the first factor, Panorama analyzes loops. If the query Q is not contained in any loop or is only contained by a loop whose iteration number does not increase with D, which we refer to as a *bounded loop*, Panorama then considers Q to be executed for a constant number of times. Otherwise, Panorama considers Q to be executed for D^k times, with k being the number of *unbounded* loops containing Q. To identify the *unbounded* loops, Panorama analyzes the bound variable of all loops that contain Q. If the loop iterates through a set of records returned by an *unbounded* database query, Panorama considers the loop to be unbounded. Specifically, in Rails, a query is *unbounded* in all but the following three cases: (1) it always returns a single value, like a `SUM` query; (2) it always returns a single record by selecting on primary key; (3) it always returns a bounded number of records using the `LIMIT` keyword.

To estimate the second factor, Panorama first identifies all the query operators inside the query Q. For example, for the query node ② in Figure 4, Panorama would identify three query operators from the `@user.issues.active.count` statement: a `SELECT` to get `issues`, another `SELECT` to get `active`, and finally a `count` operator. For most operators, we estimate its execution complexity to be $O(D)$. There are a few exceptions: we consider the complexity of a `JOIN` operator to be $O(D^2)$, and the complexity of an operator that explicitly uses index, such as `find` and `find_by_id`, to be constant.

Putting these two factors together gives Panorama the complexity estimation for one contributing query Q. For example, the estimated complexity of the query node ② in Figure 4 is O(D^3). If it is enclosed in an unbounded loop, its complexity would increase to O(D^4).

Panorama could choose to deliver the above performance information using either a detailed text description or a numeric score. The current prototype uses the latter: it uses the highest complexity among all contributing queries as the complexity score of an HTML tag. For example, in Figure 4, the complexity score of HTML node ③ is 3, based on the O(D^3) complexity estimated for query node ②, and the complexity scores of HTML node ④ and ⑤ are both 1, based on the O(D) complexity estimated for query node ①.

Of course, this is just a best-effort estimation from Panorama. There are several potential sources of inaccuracy that can be improved by future work. For example, some database tables may be much larger than others, which we do not consider; the database can also lower the query complexity than our estimation due to query optimization.

V. PANORAMA VIEW-AWARE OPTIMIZATION

Panorama suggests three categories of view-changing code refactoring to improve page-load time:

1) Display the same contents in a different style, such as pagination and asynchronous loading.
2) Display the same contents but with a different accuracy.
3) Remove a subset of contents from display.

These code refactorings can be applied for different types of HTML tags, and complement each other.

Panorama code refactoring works independently from Panorama cost estimator. As we will see in Section VI,

views/products/index.html.erb

```
1  <% @products.each do |product| %>
2    <%= product.image %>
3  <% end %>
4 +<%= will_paginate @products %>
```

(a)

controllers/products_controller.rb

```
1    products = Product.all
2 + .paginate(:page =>
3 + params[:page],
4 + :per_page => 30)
```

(b)

Fig. 5: Refactoring so that the paginated page displays 30 items at a time rather than the full list

Panorama interface will contain both features and help developers make informed refactoring decisions.

A. Display-style change: pagination

Many web pages are designed to display *all* database records satisfying certain conditions. When the database size grows, such pages will take an increasingly more time to load, and eventually become unresponsive.

A widely used solution to this problem, called *pagination*, is to display only a fixed number of records in a page and allow users to navigate to other pages for more records.

Although pagination is widely used in practice, there are still many cases where it is not used — 14 out of 140 real-world performance issues sampled by a previous study [4] are due to lack of pagination — either because developers are unaware of pagination, or because they did not anticipate the data size will become a performance problem. Therefore, we design Panorama to automatically identify pagination opportunities and conduct corresponding refactoring for developers.

1) Identifying opportunities: To identify these opportunities, Panorama checks each loop in the program for: (1) whether the loop iterates through the result of an unbounded query; and (2) whether each loop iteration leads to some content being rendered in an HTML tag. If a loop passes both checks, the corresponding HTML tag will be reported as a pagination candidate.

For the first check, Panorama locates the array variable that a loop iterates through, like `@products` in the loop shown in the left column of Figure 5, and then checks the data-flow edges in ADG to determine whether this variable is produced by an unbounded database query, as defined in Section IV-B2. For example, the ADG would show that `@products` is the result of `Product.all` on Line 1 of in Figure 5b, and `Product.all` will be translated to an unbounded database query at run time.

For the second check, Panorama searches for an ADG node n_V that is associated with an HTML tag and an `is_rendered` property inside the loop body (how to compute `is_rendered` is introduced in Sec. III-B). If n_V is found, like Line 2 in Figure 5a, the HTML tag associated with n_V is identified as a pagination candidate.

2) Generating patches: To carry out the refactoring, Panorama performs two changes to the source code using the `will_paginate` library [15]. First, in the controller, Panorama adds a `.paginate` call right after the code statement where the to-be-rendered database records are retrieved, like Line 2, 3 and 4 in Figure 5b. The constant there, which is configurable and 30 by default, determines how many records will be shown on every page. Second, in view, Panorama adds a `<=% will_paginate @products %>` statement right after the loop that renders these database records, as illustrated in Line 5 in Figure 5a. The `will_paginate` call inserts a page navigation bar into the web page, allowing users to navigate to remaining records after seeing the records displayed on the current page.

B. Display-style change: asynch-loading

Asynchronous programming is widely used to support low-latency interactive software [16]–[18]. For web applications, when there is an HTML tag that takes much longer time to render than other tags on the same page, we can instead compute and render the slow tag asynchronously, allowing users to see other parts of the web page more quickly.

For example, Discourse is a forum application. Its `topics/show` page mainly lists all the `posts` that belong to a topic. At the bottom of that page after the listing of all the `posts`, a list of `suggested topics` that are related to this topic are displayed. In an issue [19], users complained that this page is slow to load no matter a topic contains many or few posts. Developers then realized that the query to retrieve suggested topics is hurting the page-load time. Making things worse, these suggested topics are not the main interests of this page and often are not seen by users, as they are placed below all the posts and require users to scroll down to the bottom of the page to see. Consequently, developers created a patch that defers the display of `suggested topics` until all other content on the page is displayed.

1) Identifying opportunities: Conceptually, every HTML tag can be computed and rendered asynchronously. We only need to pay attention to two issues.

First, only tags that are among the slowest on a web page are worthwhile for asynchronous loading. Otherwise, loading an originally fast tag asynchronously does not help shorten the page load time, the Panorama estimator (Section IV) already provides information to help developers make this decision, and hence we do not discuss this issue below.

Second, if too many HTML tags on a web page are rendered asynchronously, the user experience could be greatly degraded. Furthermore, if one HTML tag is rendered asynchronously, other HTML tags may better be rendered asynchronously too if they share a common contributing query. For example, in Figure 4, once we decide to load HTML tag ③ asynchronously, ④ and ⑤ will be loaded asynchronously too, as they share a common query ②. Panorama considers this issue in identifying opportunities for asynchronous loading, and we will describe how this is handled below.

2) Generating patches: Given an HTML tag e, making its content computed and rendered asynchronously requires multiple changes to the controller and view components of a web application, as illustrated in Figure 6: (1) creating a new view file that renders e only, separating e from other tags on the same web page that will still be synchronously loaded; (2) adding a new controller action to compute *all and only* the content needed by e and render the new view file created above, separated from the computation for other tags on the same web page that will still be carried out synchronously;

views/users/_iss_cnt.html.erb **(1)**

```
+<span> <%= @count %></span>
```

controllers/users_controller.rb **(2)**

```
+def iss_cnt
+ @count=@users.issues.active.count
+ render :partial => 'iss_cnt'
+end
```

views/users/show.html.erb **(3)**

```
-<span> <%= @count %> </span>
+<%= render_async _iss_cnt_path%>
```

config/routes.rb **(4)**

```
+get :iss_cnt,:controller=>:users
```

applications.html.erb **(5)**

```
+<% content_for  :render_async %>
```

Fig. 6: Refactoring for asynchronous loading

(3) replacing e in the original view file with an AJAX request and adding a new routing rule so that the AJAX request will invoke the new action in (2) which then renders the view in (1) asynchronously.

The first item is straight-forward to automate. Panorama simply moves the HTML tag e into a newly created view file, like `_iss_cnt.html.erb` (Figure 6(1)), where `iss_cnt` corresponds to the name of the new controller action that Panorama will generate.

The second item is implemented by Panorama in three steps. It first identifies all Ruby variables used by e, like `@count` in Figure 6(1), and then applies static backward slicing to find all code statements C that are used to compute those variables, like `@count = @user.issues.active.count` in Figure 6. Specifically, Panorama starts from all the ADG nodes associated with the specific HTML-tag ID, and traces backwards in the ADG to identify all nodes inside the corresponding controller that e has control or data dependence on.

Panorama next applies forward taint analysis to see if any statement $c \in C$ is used to compute any other HTML tag e'. If such an e' is found, there is a dilemma about whether to render e' asynchronously: rendering e' asynchronously could potentially cause many other HTML tags, which share common backward slicing fragments with e', to be rendered asynchronously, and violate the design principle discussed in Section V-B1; yet rendering e' synchronously incurs extra overhead as c now needs to be computed twice, once for e and once for e'. Hence, Panorama currently considers e as unsuitable for asynchronous loading if e' exists.

Panorama finally moves the slice identified earlier to a new controller action, like `iss_cnt` in Figure 6(2) (the deletion from the previous controller is not shown for simplicity), and add a rendering statement at the end of the action, like `render :partial => 'iss_cnt'` in Figure 6(2), to render the same content in the same format as the original web application using the newly created view file.

Panorama conducts the third item leveraging the `render_sync` library [20] to replace the original HTML tag with "`<%= render_async [action] _path %>`" (Figure 6(3)), where `render_async` is an API call that issues an AJAX request for the specified `action` using jQuery [21]. Panorama then adds a new rule into the routing file to connect the AJAX request with the action it just created. As shown in Figure 6(4), this new routing rule follows the template "`get :[action], :controller => :[home]`", where `action` is the name of new action name, and `home` is the controller holding the `action`.

views/users/show.html.erb

```
- <span> <%= @count %></span>
+ <span><%=@count>N?'More than (N-1)':@count%></span>
```

controllers/users_controller.rb

```
- @count = @user.issues.active.count
+ @count = @user.issues.active.limit(N).count
```

Fig. 7: Refactoring for approximation

C. Display-accuracy change: approximation

Approximation is a widely used approach to improving performance and saving server resources [22]. Past database research also proposed approximated queries [23]. However, many techniques require changes to database engines [24] and hence cannot be applied to web application refactoring. Panorama focuses on approximating aggregation queries whose results are displayed as numeric values on web pages, as such approximation can be simply conducted by refactoring Rails code and easily reasoned about by web viewers.

For example, Redmine [25] is a project collaboration application like GitHub. Its `user/index` page lists all the recent activities of a user, all projects a user is involved in (with pagination), as well as two counts showing how many issues are currently assigned to and have been reported by this user. Although these two numerical counts occupy tiny space on the web page, they can take more time, even more than 1 second, to render than the remaining page, when a user is involved in hundreds of or more issues. One way to keep the page responsive is to set an upper-bound to such a count, like 100, and only shows the count to be "*more than 100*" when it is too big — when a count is too big, users probably does not care about the exact number anyway.

1) Identifying opportunities: Panorama iterates through all aggregation, including maximum, minimum, average, and count, queries in the application. For each query, Panorama checks its corresponding ADG node's out-going data-flow edges to see if the query result is *only* used in HTML-tag rendering. If so, an approximation opportunity is identified for corresponding HTML tag(s). Note that, Panorama does not suggest approximating an aggregation query if its result affects an HTML tag through control dependency, as that type of approximation may cause program execution to take a different path and hence potentially leads to large deviation from original program behaviors.

2) Generating patches: An approximation refactoring includes two parts. On the controller side, Panorama appends a `limit(N)` clause to the end of the aggregation query identified above, with the constant N configured by web developers. On the view side, instead of directly displaying the numeric query result, changes are made depending on the aggregation query type. For a `count` query, Panorama inserts a conditional statement to check the aggregation result: if the result is smaller than N then the accurate numerical result is displayed, otherwise "more than $N-1$", as shown in Figure 7; for an average query, Panorama adds "`about`" before the numeric query result rendered in the HTML tag; for a maximum or

views/sidebar/index.html.erb

```
1 - <div id='sidebar'>
2 - <% @active_projects.each do |p| %>
3 -  <%= p.description %>
4 -   <%= p.id %>
5 -  <% end %>
6 - </div>
```

(a)

controllers/todos_controller.rb

```
1 def index
2   ...
3 -   @active_projects =
4 -   user.projects.active
5   ...
6 end
```

(b)

Fig. 8: code change for removing

minimum query, Panorama adds "`at least`" or "`at most`" before the numeric query result.

D. Display contents removal

Obviously, one can remove an HTML tag to speed up the page loading. This strategy is indeed used in practice, as the `Tracks` example discussed in Section I. Whether an HTML tag is worthwhile to display cannot be determined automatically. Instead, what Panorama can do is to make the removal easy and error-free, so that developers can easily try out different design options and eventually make an informed decision.

1) Identifying opportunities: Removing an HTML tag e does not guarantee to save page-loading time, because if the expensive computation needed by e is also needed by other HTML tags, removing e alone will not help performance much. The current prototype of Panorama only suggests removing an HTML tag e if its contributing query that is not fed to any other HTML tag. This way, removing e can guarantee to save some data-processing time. Of course, future work can relax this checking criterion.

2) Generating patches: Removing an HTML tag e from the web page again involves changes to both the view component and the controller component of a web application. On the view side, Panorama simply removes the specific HTML tag. On the controller side, Panorama again analyzes control-dependency and data-dependency graph to remove code that was used only to help generate e.

To do so, Panorama first identifies all the nodes in ADG that are associated with e's ID. Panorama deletes those nodes, removes all condition checking whose two branches now execute exactly the same code because of those node deletions, and then check if there are any other ADG nodes that become useless and should be deleted — a node is useless if it has no out-going data-dependency or control-dependency edges. Panorama repeats this process for several rounds until no more nodes are identified as useless.

We use the view file code snippet in Figure 8a as an example. The HTML tag shown here corresponds to the sidebar that lists all projects in the Tracks example discussed in Figure 1. Given this tag, Panorama first identifies the Ruby expression `@active_projects`, and then checks the ADG to see how `@active_projects` is computed in the controller (Figure 8b). Panorama also finds out that the `@active_projects` computed in Figure 8b is not used in anywhere else. Consequently, the content-removal change will simply delete the sidebar tag in the view file and the corresponding computation in the controller file, as shown in Figure 8.

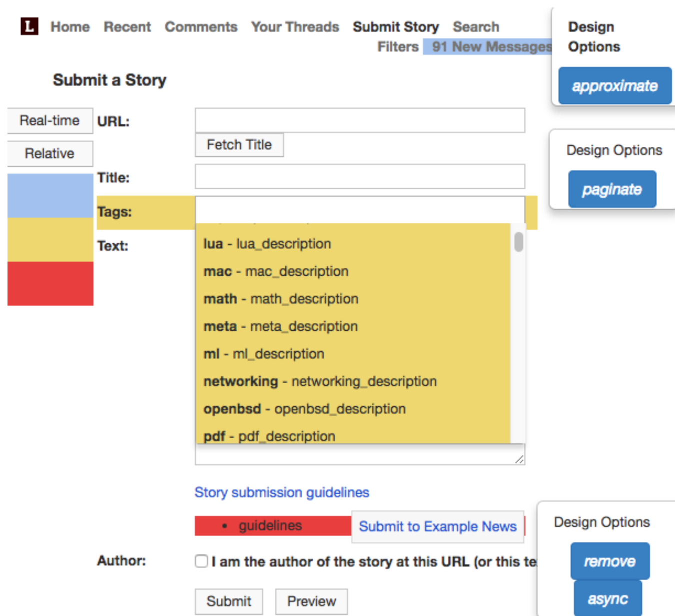

Fig. 9: An example of Panorama browser interface

VI. The Panorama Interface

Panorama comes with a new interface to present the cost estimation and optimization information, and help developers explore different web-page design options. We discuss the Panorama interface in this section.

A. Information display in browser

Showing view-centric cost estimation information. Panorama visualizes the performance information obtained by dynamic profiling or static estimation (Section IV) through a heat-map with the more costly HTML element having a more red-ish background, as illustrated in Figure 9.

To generate this heat map, Panorama reads the output of its view-centric cost estimation (Section IV) and creates a JavaScript file `interactive.js` that sets the background color of every HTML tag through "`$(tag-id).css(''background-color'', color);`", where `tag-id` is the unique HTML tag ID and `color` is computed based on the cost estimation for this HTML tag (Section IV). Web developers can choose to see different heat-maps with buttons on the web page, like "Real-time" (dynamic profiling results) and "Relative" (statically estimated results) in Figure 9. We set the color using the HSL color scheme, with more expensive tags rendered with smaller hue values (i.e., more red-ish) and cheaper tags with larger hue values (i.e., more blue-ish).

Showing view-aware optimization suggestions. `interactive.js` described above helps display not only data-processing cost but also alternative view-design options for various HTML tags. Users simply right click an HTML tag in the browser to get a list of design options, as shown in Figure 9. The implementation is straight-forward, given the unique ID of every HTML tag and the performance-enhancing

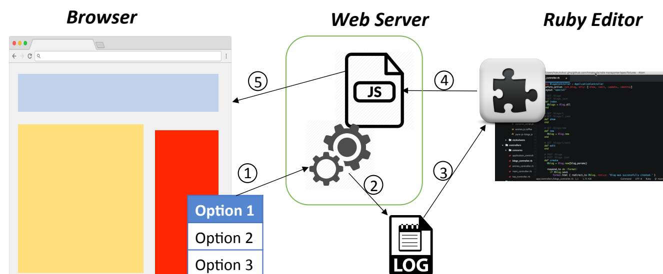

Fig. 10: Panorama interface implementation

opportunities identified through Panorama static analysis as discussed in Section V.

B. Design-space exploration in browser

To help developers explore different performance–functionality design trade-offs, Panorama further connects the browser-side information display and the Ruby editor side refactoring together: (1) developers first understand the data-processing cost of various HTML tags in the browser; (2) once developers choose an alternative design option for an HTML tag, the corresponding code refactoring will be automatically applied and displayed in the accompanying Ruby editor for developers to review; (3) once the source code is updated, the heat-map in the browser is updated accordingly. Developers can explore different design options, and eventually pick the best ones that suit their need.

To support this interface, Panorama carefully uses JavaScript, IDE plugin, and other mechanisms to help the communication between the browser and the Ruby editor, as illustrated in Figure 10.[1]

First, Panorama automatically instruments the web application under development to help communicate developers' design choices to the Ruby editor. Specifically, Panorama adds a controller `_PANO_handle_request` into the web application. Whenever developers click a design-option button, like one of those blue *paginate*, *async*, *approximate*, *remove* buttons in Figure 9, Panorama will send an HTTP request to invoke the `_PANO_handle_request` controller action (① in Figure 10), which then records the design-option type and the corresponding HTML tag ID into a web-server side file `request.log` (② in Figure 10).

In the editor, which we use RubyMine [26], Panorama uses a thread to monitor the `request.log` file. Whenever this file is changed, this monitoring thread will trigger the plugin to apply corresponding code refactoring in the IDE, with all the code changes generated using algorithms described in Section V (③ in Figure 10).

After the code change, the data-processing cost estimation will be updated automatically, which results in updates to a performance profile and corresponding updates to `interactive.js` with changed background-color settings (④ in Figure 10). The changes in `interactive.js` lead to an automated refresh in the browser with the updated heat-map

[1]The current prototype of Panorama assumes that the web-application under testing is deployed on the same machine as the Ruby editor.

TABLE I: Opportunities detected by Panorama in 12 apps

App	Ds	Lo	Gi	Re	Sp	Ro	Fu	Tr	Da	On	FF	OS	SUM
pagi.	1	6	1	10	2	20	5	9	1	6	3	3	69
approx.	1	1	0	7	0	5	1	3	0	23	0	0	43
removal	1	2	0	7	0	4	1	2	2	2	0	0	22
asynch	1	2	0	2	0	2	1	2	2	2	0	0	15
SUM	4	11	1	26	2	31	8	16	5	33	3	3	149

display, as we use the Ruby `react-rails-hot-loader` to enable automated refresh at every change in the Ruby source code or heat-map display code (⑤ in Figure 10).

VII. Evaluation

Our evaluation focuses on three research questions: **RQ1**: Can Panorama identify view-aware optimization opportunities from latest versions of popular web applications? **RQ2**: How much performance benefits can view-aware optimization provide? **RQ3**: Is the performance-functionality trade-off space exposed by Panorama worthwhile for developers to explore? **RQ4**: Does Panorama estimator estimate the per-tag data-processing cost accurately?

A. Methodology

Applications. We evaluate Panorama using a suite of 12 open-source Ruby on Rails applications, including top 2 most popular Ruby applications from 6 major categories of web applications on GitHub: Discourse (Ds) and Lobster (Lo) are forums; Gitlab (Gi) and Redmine (Re) are collaboration applications; Spree (Sp) and Ror_ecommerce (Ro) are E-commerce applications; Fulcrum (Fu) and Tracks (Tr) are Task-management applications; Diaspora (Da) and Onebody (On) are social network applications; OpenStreetmap (OS) and FallingFruit (FF) are map applications. They have all been actively developed for years, with hundreds to tens of hundreds of code commits.

Workload. Since we cannot obtain real-world user data, we use synthetic data generation scripts released by previous work that to populate the databases following real-world data distribution and statistics. Similar to [4], we use the number of records in a web application's main database table to describe the workload size. By default, we use a 20,000-record workload unless otherwise specified. To our best knowledge, **all** the database sizes used in our evaluation are similar or smaller than the sizes in real-world web applications.

Platform. We profile the Rails applications on AWS Cloud9 platform [27], which has 2.5GB RAM and a 8-core CPU.

B. RQ1: how many opportunities does Panorama identify?

As shown in Table I, Panorama can indeed identify many view-aware optimization opportunities. Specifically, Panorama static analysis identifies 149 performance-enhancing opportunities from the current versions of our benchmark applications. Every type of optimization opportunities is identified from at least 8 applications.

These 149 opportunities apply to 119 unique HTML tags. For 101 HTML tags, only one view-change suggestion is

TABLE II: Speed up of 15 view changes

	Pagination						Asynchronous		Approximation				Content Removal		
ID	Ro1	Tr1	Fu1	Re2	On1	Re1	Lo2	On5	Re2	On2	Tr2	On3	Lo1	Re3	On4
Server Time Speedup (X)	19.4	13.5	6.8	4.7	2.1	1.8	37.8	1.1	2.1	1.4	1.2	1.3	33	1.4	1.1
Server Time Speedup (X)	9.4	9.2	5.9	3.6	2.7	1.6	17.2	1.2	1.6	1.3	1.2	1	8.7	1.3	1.2

Every case is denoted by <application-short-name>-ID

TABLE III: Database sizes and page load time of 12 user-study cases

	Pagination			Asynchronous			Approximation			Content Removal		
ID	Re1	Ro1	Tr4	Re4	Tr3	Lo1	Re2	Tr1	Di1	Lo2	Tr5	Tr6
DB Size (k-record)	2	0.8	2	2	2	20	100	100	100	20	100	2
Base Page Load Time (s)	2	1.9	2.5	2	1.9	1.8	2.5	2.5	2.5	1.8	2.5	1.9
New Page Load Time (s)	0.5	0.4	1	0.5	0.4	0.3	1	1	1	0.3	1	0.4

3 red IDs are cases from existing issue-tracking systems; the other 9 cases are all in latest versions discovered by Panorama.
Every case is denoted in the same way as Table II, with 6 common cases.

made. For the remaining 18, Panorama suggests two or three changes. Particularly, there are 15 HTML tags where *removal* and *asynchronous* loading both apply. Overall, these four types well complement each other.

C. RQ2: how much performance benefits?

To quantitatively measure the performance benefits of these alternative view designs, we randomly sampled 15 optimization opportunities identified above, with 6, 2, 4, and 3 cases from Pagination, Asynchronous (loading), Approximation, and Content Removal respectively, in 6 different applications. For each application, before and after optimization, we run a Chrome-based crawler that visits links randomly for 2 hours and measure the average end-to-end-latency and server-cost of every action. We then compute speedup accordingly.

As shown in Table II, the performance benefits of these view changes are significant. By changing only one HTML tag, these 15 cases on average achieve $8.6\times$ speed up on the server side and $4.5\times$ speed up for end-to-end page load time. Among the four optimization types, *pagination*, *asynchronous* loading, and content *removal* have cases where the end-to-end page load time achieves about or more than $10\times$ speedup.

D. RQ3: are alternate view designs worthwhile?

We evaluate the quality of a web page from two aspects: (1) how much users like the performance and functionality of a web page; (2) how much resources are needed to generate the page on the server side.

All four types of view changes suggested by Panorama can help save server resources — *pagination*, *approximation*, and content *removal* all reduce tasks that need to be done by web and database servers; *asynchronous* loading provides more scheduling flexibility to servers.

Therefore, we believe an alternative web design is worthwhile for developers to explore, as long as users feel pages under this new design is *not worse* than the original one. To evaluate this, we conduct a thorough user study.

1) User study set-up: We recruited 100 participants on Amazon Mechanical Turk (Mturk). These participants are all more than 18 years old and living in the United States, with more than 95% MTurk Task Approval rate.

Our benchmark suite includes 12 web pages from 5 web applications. For each of these 12 baseline pages, Panorama automatically generates a new page with exactly one HTML tag changed. We refer to the original page as *Base* and the one optimized by Panorama as *New*. These 12 web pages cover all four types of view changes, with exactly 3 cases in each type. Furthermore, for every change type[2], we cover one case from on-line issue reports — these changes were already adopted by developers to fix performance problems in previous versions of web applications, and some cases discovered by Panorama in current versions of these applications. We also reuse cases from Table II as much as we can.

Since the performance advantage of *New* pages depends on the database size, to ease comparison, we populate the database for each benchmark so that the load-time difference between the *Base* version and the *New* version is exactly 1.5 seconds. The detail settings are shown in Table III.

Each participant is assigned 8 tasks. In each task, they are asked to click two links one by one, and then answer questions about (1) which page they think is faster ("Performance" in Table IV); (2) which page they think delivers more or better organized content ("Functionality" in Table IV); and (3) which page do they like more with everything considered ("Overall" in Table IV). These two links are the *Base* and *New* versions of one benchmark, with random ordering between them.

2) User study results: A summary of the user study results is shown in Table IV, and the questionnaire and raw data are available on Panorama webpage [28]. In this table, we show the percentage of users who think New is better minus those who think Base is better, which we refer to as the *net* benefit of the new design, for every type of refactoring and every question (Performance, Functionality, and Overall). Users are given the two pages in random order, and are not aware of the view-design difference between the two pages in advance.

A short answer to our research question is "Yes". In fact,

[2]Except for approximation, as we did not find performance issue reports in these 12 web applications that are solved by approximation.

TABLE IV: Net user perception enhancement by New design (% users who prefer New − % users who prefer Base)

	Approximation	Asynch	Paginate	Removal
Performance	23.50%	31.00%	45.00%	35.50%
Functionality	-7.50%	17.50%	-13.00%	5.50%
Overall	1.00%	18.50%	28.00%	10.50%

for all type of view-changing optimization, users think the New design is *not worse* than the Base design. Particularly, for *asynchronous* loading, *pagination*, and *removal*, the new designs clearly win more users than the baseline designs.

In terms of performance, the net win for the New design is clear. Many users indeed notice the 1.5-second difference in the page-load time. In 10 out of 12 cases, the New design has a net positive benefit on more than 30% of the participants.

In terms of functionality, the results are quite interesting. For *approximation* and *pagination*, many users did notice the content difference, leading to the Base design winning about 10% of users. However, for *asynchronous* loading and content *removal*, surprisingly, many users neither notice the content difference nor think New design delivers worse contents. It could be that removing contents made the page cleaner to some users. For example, after removing the sidebar in Tracks (Tr5 in Table III), some participants like it because "Adding the sidebar makes scrolling harder."

In terms of overall perception, the New design has a **net win**. Among the four types of optimizations, paginations and asynchronous loading are the most appealing to users, while approximation is the least appealing.

We also conducted another set of user study with another 100 participants, where we use an even larger (2-10×) database size and hence make the page-load time differences between New design and Base design even bigger (3 seconds). We do observe that more participants noticed the performance advantage of the New design. However, we also observe that the overall perception only goes up a little bit more for the New design. We skip the details for the space constraints.

E. RQ4: how accurate is the Panorama estimator?

We use the web page shown in Figure 9 as a case study. 15 HTML tags on this page render dynamically generated contents. With 200 database records, dynamic profiling shows that the story tag is the top performance bottleneck, followed by the guideline tag and then the message-count the cheapest. When the workload increases to 2000 and 20000 records, dynamic profiling shows that the guideline tag is the top bottleneck, followed by the story tag. These results match the performance gains we can get by optimizing these three tags: using 20000 records, asynchronously loading or removing the guideline text can reduce the page end-to-end load time by more than 1 second; paginating the story-tag can speed up the page load time by about 100 milliseconds; approximating the message count does not change page load time.

The static mode of Panorama estimator can indeed predict performance bottleneck without running the application — the guideline text gets the highest complexity score (5) among all 15 tags, followed by the story tag (4), and then the message-count tag (3), with the remaining 12 tags getting 0 points.

F. Threats to validity

Threats to the validity of our work could come from multiple sources. **Internal Validity**: The HTML tags that are dynamically generated through JavaScript can not be detected or analyzed by Panorama. **External Validity**: The 12 applications in our benchmark suite may not represent all real-world applications; The synthesized databases may not represent real-world workloads; The machine and network settings of our profiling may differ from real users setting; the 100 participants of our user-study from MTurk may not represent all real-world users. Overall, we have tried our best to conduct an unbiased study.

VIII. Related work

a) ***ORM performance problems****:* Much previous work aimed to identify specific performance problems of web applications built with ORM framework, like unnecessary data retrieval in applications [11], anti-patterns like the so-called "N+1" query problem [29], sub-optimal database physical design [30], computation that is more efficient to compute inside the database [31]–[35], and general database-aware data-flow optimization [12]. However, all prior work focuses on view-preserving optimization that does not change web page design. We instead show that by changing the view design, there are many performance enhancing opportunities and we build Panorama to interactively help developers make such design-performance trade-off decisions.

b) ***Detecting and fixing performance bugs****:* Plenty of research [36]–[43] aims to detect and fix performance problems of general purpose software, such as loop inefficiency, temporary object bloating, inefficient data structure, etc. Detecting and fixing performance bugs of cross-stack web applications built upon ORM frameworks requires different techniques.

IX. Conclusion

It is increasingly challenging to develop web applications that can deliver both good functionality and desired performance. We present Panorama, a tool that helps web developers explore the performance-functionality trade-off space in their web application design. The Panorama estimator provides developers with data-processing cost information for every HTML tag that renders dynamically generated data, while the Panorama optimizer identifies and automates view-changing refactoring that can greatly improve performance. The Panorama interface integrates estimator and optimizer together to enable effective web application design.

X. Acknowledgement

This work is supported in part by the NSF through grants CCF-1837120, CNS-1764039, 1563956, 1514256, IIS-1546543, 1546083, 1651489, OAC-1739419; DARPA award FA8750-16-2-0032; DOE award DE-SC0016260; the Intel-NSF CAPA center; gifts from Adobe, Google, Huawei, and CERES research center for Unstoppable Computing.

References

[1] F. F.-H. Nah, "A study on tolerable waiting time: how long are web users willing to wait?" *Behaviour & Information Technology*, pp. 153–163, 2004.

[2] "Speed is a killer," https://blog.kissmetrics.com/speed-is-a-killer/.

[3] "Startup = growth," http://paulgraham.com/growth.html.

[4] J. Yang, P. Subramaniam, S. Lu, C. Yan, and A. Cheung, "How not to structure your database-backed web applications: a study of performance bugs in the wild," in *ICSE*, 2018, pp. 800–810.

[5] "tracks, a task management application," https://github.com/TracksApp/tracks/.

[6] "tracks-870," https://github.com/TracksApp/tracks/issues/870/.

[7] T. Reenskaug and J. Coplien, "More deeply, the framework exists to separate the representation of information from user interaction," *The DCI Architecture: A New Vision of Object-Oriented Programming*, 2013.

[8] "chrome development tool," https://developers.google.com/web/tools/chrome-devtools/evaluate-performance/reference.

[9] "active record query trace," https://github.com/ruckus/active-record-query-trace.

[10] M. Selakovic and M. Pradel., "Performance issues and optimizations in javascript: an empirical study," in *ICSE*, 2016, pp. 61–72.

[11] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks." in *ICSE*, 2016, pp. 1148–1161.

[12] J. Yang, P. Subramaniam, S. Lu, C. Yan, and A. Cheung, "Powerstation: Automatically detecting and fixing inefficiencies of database-backed web applications in ide," in *FSE*, 2018.

[13] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[14] "Jruby," https://github.com/jruby/jruby.

[15] "will-paginate tool package," https://github.com/mislav/will-paginate.

[16] S. Okur, D. L. Hartveld, D. Dig, and A. v. Deursen, "A study and toolkit for asynchronous programming in c," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 1117–1127.

[17] S. Okur, D. Dig, Y. Lin *et al.*, "Study and refactoring of android asynchronous programming," 2015.

[18] Y. Lin, C. Radoi, and D. Dig, "Retrofitting concurrency for android applications through refactoring," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 341–352.

[19] "Discourse-4663 issue report," https://github.com/discourse/discourse/pull/4663.

[20] "asynchronous rendering library," https://github.com/renderedtext/render-async.

[21] "jquery," https://jquery.com/.

[22] A. Farrell and H. Hoffmann, "Meantime: Achieving both minimal energy and timeliness with approximate computing." in *USENIX*, 2016, pp. 421–435.

[31] A. Cheung, A. Solar-Lezama, and S. Madden, "Optimizing database-backed applications with query synthesis," in *PLDI*, 2013, pp. 3–14.

[23] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. S. Manku, C. Olston, J. Rosenstein, and R. Varma, "Query processing, approximation, and resource management in a data stream management system." in *CIDR*, 2003, pp. 245–256.

[24] Y. E. Ioannidis and V. Poosala, "Histogram-based approximation of set-valued query-answers," in *VLDB*, 1999, pp. 174–185.

[25] "redmine, a project management application," https://redmine.org/.

[26] "Rubymine, a popular ruby on rails ide," https://www.jetbrains.com/ruby/.

[27] "Aws cloud9," https://aws.amazon.com/cloud9/.

[28] "Panorama website," https://hyperloop-rails.github.io/panorama/.

[29] 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 *ICSE*, 2014, pp. 1001–1012.

[30] C. Yan, J. Yang, A. Cheung, and S. Lu, "Understanding database performance inefficiencies in real-world web applications," in *CIKM*, 2017.

[32] M. B. S. Ahmad and A. Cheung, "Automatically leveraging mapreduce frameworks for data-intensive applications," in *SIGMOD*, 2018, pp. 1205–1220.

[33] A. Cheung, S. Madden, O. Arden, and A. C. Myers, "Automatic partitioning of database applications," *PVLDB*, vol. 5, no. 11, pp. 1471–1482, 2012.

[34] A. Cheung, S. Madden, and A. Solar-Lezama, "Sloth: Being Lazy is a Virtue (when Issuing Database Queries)," in *SIGMOD*, 2014, pp. 931–942.

[35] M. B. S. Ahmad and A. Cheung, "Leveraging parallel data processing frameworks with verified lifting," in *Proceedings Fifth Workshop on Synthesis, SYNT@CAV 2016*, 2016, pp. 67–83.

[36] S. Zaman, B. Adams, and A. E. Hassan, "A qualitative study on performance bugs," in *MSR*, 2012, pp. 199–208.

[37] M. Jovic, A. Adamoli, and M. Hauswirth, "Catch me if you can: performance bug detection in the wild," in *SIGPLAN*, 2011, pp. 155–170.

[38] A. Nistor, L. Song, D. Marinov, and S. Lu, "Toddler: Detecting performance problems via similar memory-access patterns," in *ICSE*, 2013, pp. 562–571.

[39] L. Song and S. Lu, "Performance diagnosis for inefficient loops," in *ICSE*, 2017, pp. 370–380.

[40] J. Li, Y. Chen, H. Liu, S. Lu, Y. Zhang, H. S. Gunawi, X. Gu, X. Lu, and D. Li, "Pcatch: automatically detecting performance cascading bugs in cloud systems," in *EuroSys*, 2018, pp. 7:1–7:14.

[41] A. Nistor, P. Chang, C. Radoi, and S. Lu, "CARAMEL: detecting and fixing performance problems that have non-intrusive fixes," in *ICSE*, 2015, pp. 902–912.

[42] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in *PLDI*, 2012, pp. 77–88.

[43] B. Dufour, B. G. Ryder, and G. Sevitsky, "A scalable technique for characterizing the usage of temporaries in framework-intensive java applications," in *FSE*, 2008, pp. 59–70.