Shareable Benchmarking Reports with Enhanced Filters and Dynamic Statistics for BenchExec

Author: Dennis Simon

Supervisor: Prof. Dr. Dirk Beyer
Mentor: Dr. Philipp Wendler
Submission Date: 18. April 2021

Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 18. April 2021

.....

Dennis Simon

Abstract

BenchExec is a benchmarking tool developed at the Software and Computational Systems Lab of the Ludwig-Maximilians-Universität in Munich, Germany. One of its features is the automatic generation of a HTML table that presents the benchmarking results in an interactive and visual environment. The HTML table was recently rewritten in 2019 in the context of a bachelor thesis, with the performance and usability of the tables being greatly improved. This thesis showcases new sets of features that were built on top of this implementation and additionally showcases the integration of a rework of the filter functionality including the addition of a new global, more accessible user interface, the addition of multi-selection capabilities for status and category filters and components that allow a more granular filtering by task-ID. In order to improve user experience and to unlock shareability, navigation is now done via a "Hash Routing" approach, with most state, including filters, being serialized in the URL thus enabling the navigation history handling of the browser and allowing users to share state via a copied URL. To achieve a more consistent representation of the state of selected data, statistics that were previously statically included in the table are now asynchronously calculated using pooled web workers to reflect any changes in selected data that might occur by filtering. Lastly, the performance of the successful implementation is compared to the previous implementation. The changes described in this paper are already in use with some of them already being successfully extended further.

Contents

1: Introduction & Motivation

In order to analyse the efficiency and complexity of algorithms and to be able to make educated decisions when evaluating different approaches, developers and researchers often times use benchmarking as a basis of reasoning. Especially in high-performance computing communities the use of benchmarking as a means of performance evaluation is regularly used and standardized [3]. In order to gain reliable and representative insight into the performance of algorithms or tools a certain sample size is needed to account for minor deviations and build more precise aggregated representations of the expected performance.
The benchmarking runs usually produce their results in a list or tabular form. Especially for runs with a large sample size or sets of measures for different parts of the tested tool, these will become hard to comprehend and efficiently analyse.
BenchExec [3], a benchmarking tool developed by the Software Systems Lab of the LMU Munich provides an auxiliary tool called table-generator to assist its users with the consumption and comprehension of test results.
The table-generator tool embeds the benchmarking results produced by BenchExec into a pre-build HTML file. The application included in the HTML file exposes a number of different methods of visualization using graphs as well as a filterable tabular representation and statistical description of the dataset. The HTML tables last underwent a complete rework in 2019 in the context of a bachelor thesis [4].
BenchExec is used in competitions [2] [1] with the HTML tables being used as the main way to present results. Since the rework, users have requested a number of additional features[1] that would improve usefulness and usability of the HTML tables.
The ability of looking at different subsets of the full dataset is especially interesting as it gives additional detailed and contextual insights into the dataset, however, the capabilities of filtering were limited in functionality

[1] `https://github.com/sosy-lab/benchexec/issues?q=is%3Aissue+label%3A%22HTML+table%22+label%3Aenhancement+`

and usability. Additionally, the statistical representation of the selected dataset was statically embedded and did not react to changes in filtering. Consequently it might not always match the currently filtered selection of the dataset. The goal of this thesis is to analyse and discuss the capabilities and limitations of the current implementation of the filtering and the calculation of the statics table as well as the creation of a concept and the implementation of an improved version.

2: Existing Solution

The HTML tables are prebuilt React.js[1](subsequently React) applications that have been built and bundled using webpack[2] and enhanced with BenchExec result data using the *table-generator* tool. React uses a syntax extension to pure JavaScript called JSX[3].
In order to find ways to improve the functionality and user experience, we must first explore the currently existing solution and shed light on its proven good approaches to user experience and parts that have a negative impact on the efficiency or experience of users.
To build an initial list of tasks to perform with corresponding priorities and ultimately enabling us to build a timeline, it is beneficial to walk through each of the main parts of the application and reference its components with a list of reported issues[4].
Various terms that are commonly used in BenchExec and the HTML tables are also used in this paper. The relevant terms used in context of this paper are defined in the glossary[5] as follows:

run A single execution of a tool. It consists of the full command-line arguments (including input file) and the resource limits, and produces a result including measured values for the resource consumption.

task A combination of an input file and an expected result that defines a problem for a tool to solve.

tool A program that should be benchmarked with BenchExec.

Additionally, the definition of a **runset** is a *set of all run executions for a given tool*. These are usually represented in vertical blocks in the HTML table as can be seen in figure 2.2.

[1] A popular javascript framework by facebook `https://reactjs.com`

[2] `https://webpack.js.org/`

[3] `https://reactjs.org/docs/introducing-jsx.html`

[4] `https://github.com/sosy-lab/benchexec/issues?q=is%3Aissue+label%3A%22HTML+table%22+label%3Aenhancement+`

[5] `https://github.com/sosy-lab/benchexec/blob/master/doc/INDEX.md`

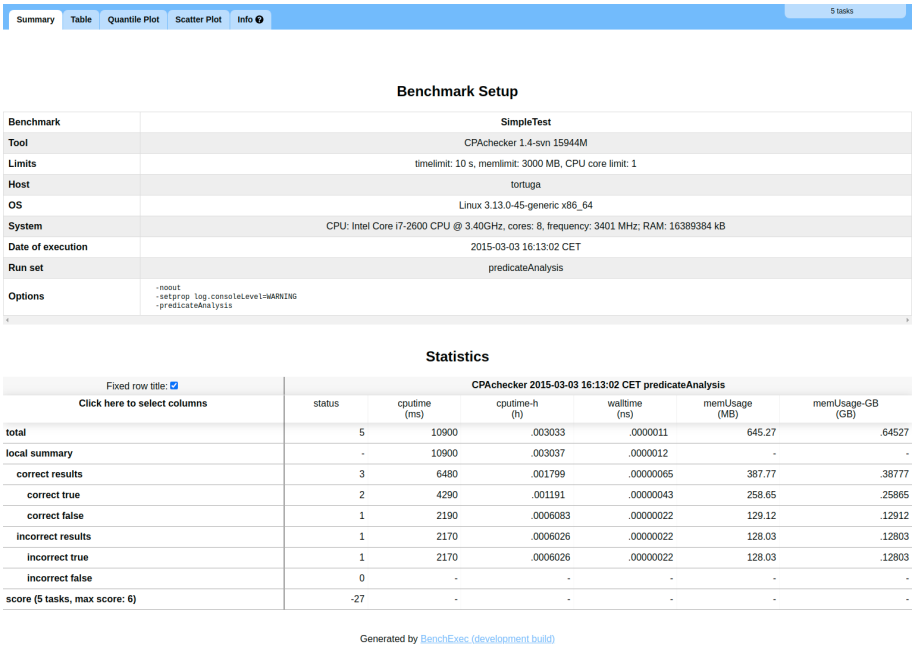

Figure 2.1: Summary page

2.1 Summary Tab

The summary tab is the main view of the application and consists of two bigger components: The setup overview, a static summary of the configurations of each individual runset and a table containing statistics giving a general overview over success and failure rates and aggregations over each measure. The cells in the statistics table on the bottom of figure 2.1 show the values of each run matching the criteria in the row headers aggregated as a sum by default, but will show advanced aggregations like **average**, **standard deviation** and **median** in form of a tooltip when the user hovers over the individual cells. This table also gives the user the option to hide or show individual columns in tables and plots.

2.2 Table Tab

The table tab displayed in figure 2.2 gives the user a complete overview over each run in each runset and the results associated with it.

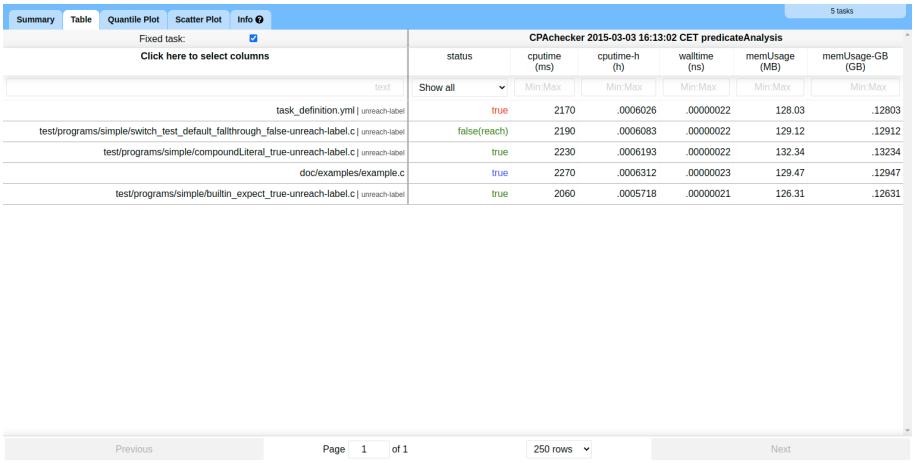

Figure 2.2: table tab

The row below the table headers is used to filter the dataset. A number of different input fields are used depending on what kind[6] of value is represented in each column:

Text For textual columns a text input search field is used to perform free text filtering.

Numeric A text input field is displayed that either accepts single values or value ranges in the form of `[<minimum value>]:[<maximum value>]`

Enumerable A drop-down selection field is displayed that lets the user choose between the distinct values of the column

Filters are applied globally and will have effects on the plot tabs as well

2.3 Quantile Plot Tab

The quantile plot tab gives a graphical representation of the dataset that is shown on the table tab presented in section 2.2. The user can quickly change between parameters and compare them between different runsets.

[6]The UI generally differentiates between *textual*, *numeric* and *enumerable* column values.

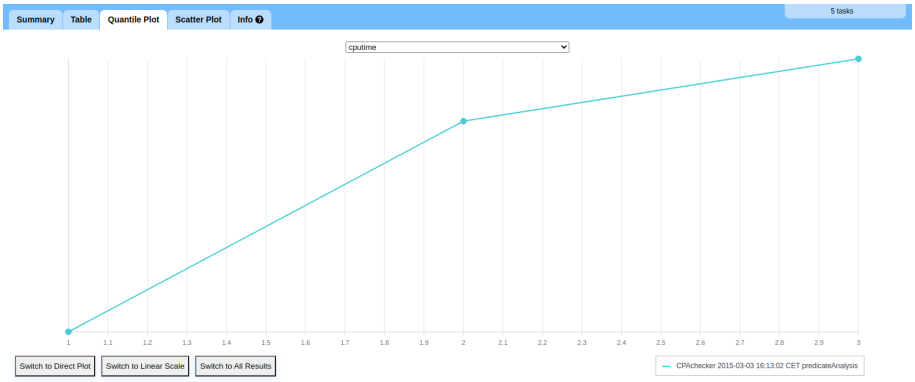

Figure 2.3: Quantile plot tab

Each runset gets represented using a plotted line in a different color with the x-axis representing the list of tasks and the y-axis representing the values of the selected parameter [4]. The user has a number of different settings to configure the graph to their needs:

- Switch between the quantile and direct plot mode.
- Switch between linear and logarithmic scaling.
- Switch between showing only tasks that are classified as *correct* or all tasks.

The data set being used is also subjected to filtering as introduced in section 2.2 on page 7.

2.4 Scatter Plot Tab

The scatter plot tab is similarly structured to the quantile plot tab introduced in section 2.3. Here the user can choose two properties from any runsets and compare them on the scatter plot to explore correlations. The values of the first chosen parameter get mapped to the x-axis and the second one to the y-axis. The intersections of the values of each parameter get rendered as points in the series.

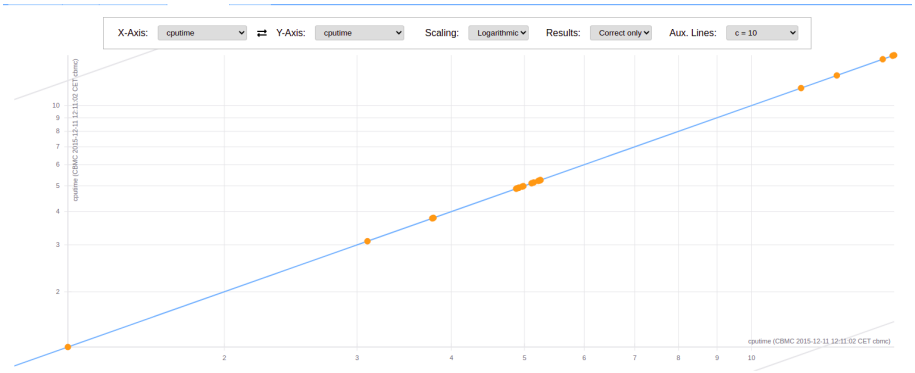

Figure 2.4: Scatter plot tab

The scatter plot tab allows for similar customization to settings as the quantile plot tab. Various drop-down fields give the user control over the following settings:

- Tweak the auxiliary lines.
- Switch between linear and logarithmic scaling.
- Switch between showing only tasks that are classified as *correct* or all tasks.

2.5 Issues with the Previous Implementation

The previous set of functionality and user interface already gave good control and some powerful visualization tools to the user. Most of these features had been recently implemented according to requests and feedback by the main user group of the tool [4]. Yet there were some parts of the current approach that could be improved to enhance user experience and to reduce efficiency blockers.

2.5.1 Navigation

The application follows a single page application (SPA) approach that results in the creation of a single HTML file with its contents being dynamically updated and rendered on user interaction [6]. This allows the HTML page to

be loaded locally via the `file://` protocol, without the need of a server to handle rendering and routing.
This effect became visible when one used the UI to navigate to a different page: The new page is loaded, but the URL did not change. While this results in a smooth experience and removes the need for a server to be included in bundling, the navigation to different pages only happens in code but did not actually touch the browsers navigation history. The browsers history object contains information about visited pages in the current session[7] and it is this object that gets used to resolve navigation targets when the user uses the "BACK" and "FORWARD" buttons in the browsers user interface.

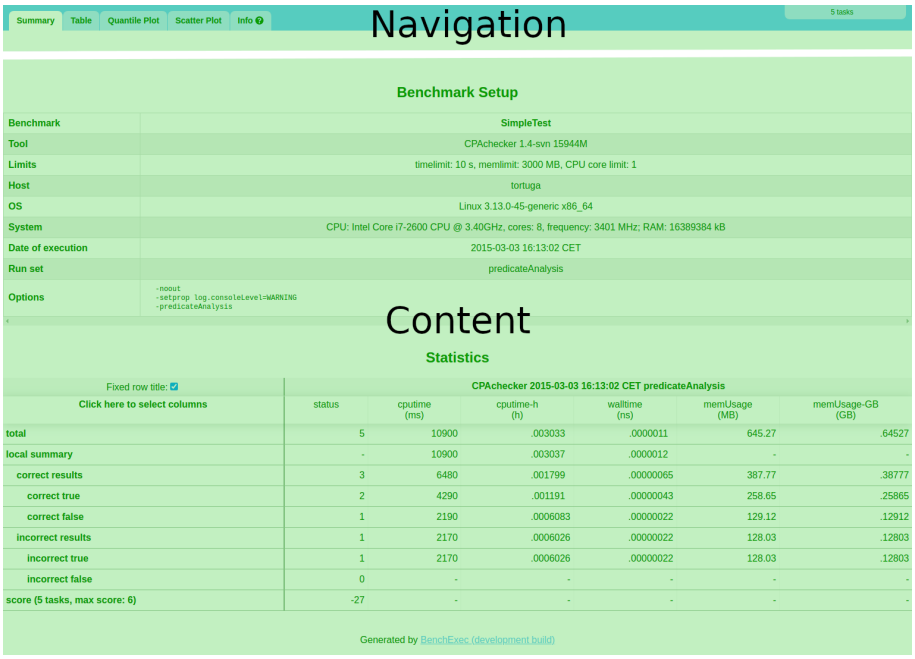

Figure 2.5: Top-level abstraction of the *HTML table*

The user interface of the application can be abstracted into two main components as seen in figure 2.5: The *navigation* and the *content* components. While the navigation component stays the same and is always visible from all pages, with each click on an item in the navigation menu the content in the *content* container will be replaced. Internally this approach was supported by the *react-tabs*[8] library and modeled (as the name of the library suggests) like a tab menu in classical graphical user interfaces.

[7]The *session* in this context refers to "[...] the pages visited in the tab or frame that the current page is loaded in." [5]

[8]`https://github.com/reactjs/react-tabs`

As the navigation in the application is handled purely app-internal and did not alter the history object, the BACK and FORWARD buttons of the browser became either non-functional or, in the worst case, might have navigated the user back to a previously visited page which resulted in a complete loss of any application state.

2.5.2 Filtering

Filtering the dataset was only possible from the table tab, yet any filters set in the respective section on the table tab have an effect the application globally, meaning that the filtering on the table tab also impacts the quantile and scatter plot. This can be confusing for the user as the separation of the components using tabs can give the impression that any actions performed will only be applied to the current scope (e.g the tab, the graph, etc.) as it is the case with the configuration of the plots as shown in figure 2.3 and figure 2.4. A more intuitive approach is to make the filtering functionality part of the global navigation UI (introduced in section 2.5.1) that is visible on every tab and therefore can get more easily identified as a globally acting functionality.
Additionally, while the filters affect the table tab and the plot tabs, the statistics table on the overview tab stayed unchanged, even though contextually the currently relevant data is now the filtered dataset. This is another slight inconsistency that might add to any existing confusion about the state of the dataset and the effects of filtering on it.
The actual logic for filtering the dataset with the user supplied parameters was handled completely by *React Table*[9]. This hid any implementation details and added a hard dependency on *React Table* which made it harder to move to any alternate solutions in the future. As *React Table* has stopped support[10] of the currently used version, this might be necessary.

2.5.3 Statistics

Right at the core of the *Summary* page, the statistics table intends to give the user detailed information about their runsets in the form of aggregated statical measures[11] that aim at providing a quick overview over the state of the whole dataset.

[9] *React Table* is a data-grid library that is used to render all tables in the application. https://github.com/tannerlinsley/react-table/tree/v6

[10] `https://github.com/tannerlinsley/react-table#version-6`

[11] Calculated and displayed are: min, max, average, median and standard deviation

These statistics were pre-calculated and prepared in BenchExec and then loaded as static data in the *HTML table*.
While this approach has clear positive effects, like reducing the performance impact to the *HTML table* by technically outsourcing the calculation of the statistics and thus mitigating the need to run these potentially expensive computations during the load of the page in the browser, there are downsides to this approach.
The issue with this implementation was that it did not react to changes in the filtered dataset. If a user chooses to filter data, then as this filtered data is used as a basis for operation in all other consuming components (e.g. plots), this change should also be reflected in the statistics. The expected behavior therefore is that if a filter is applied then the statistics will be calculated over this filtered dataset and ignore data that has been filtered out.

2.5.4 Shareability

As BenchExec is a benchmarking tool that gets regularly used in competitions [1][2], users might work in a team or at least want to share specific insights that were gained using filtering and/or specific configuration of the plots with other members or any other interested third party entities. By providing the option to specifically link to a certain plot this can be improved further. Previously there was no real way of doing this as any application state was lost when the page is deleted during refresh or closure of the browser and is not persisted anywhere.

3: New Features

While the discussion of issues in section 2.5 already hinted the high-level changes that happened in context of this thesis, there were some considerations to do regarding the order of implementation as well. Additionally, some changes required more effort and some additional planning to implement them seamlessly.

3.1 Roadmap & Challenges

With the outcome of the analysis of the previous state of the application in section 2.5, a list of changes was compiled which needed to be introduced in the system in order to improve it further.
This list could then be transformed into a roadmap that helped with optimizing the order of implementation and grouping tasks into logical sets to reduce context switching. To ensure that this process works, clear definitions of tasks and their dependencies are required. For example, if we would want to implement a new UI element for filtering, we would first need to ensure that the filtering logic is extracted into a global component or that the interface of the component responsible for handling filtering is at least globally accessible in a fashion that suits our needs and does not break best practice recommendations.

3.1.1 Building the Roadmap

In the following the list of tasks for the roadmap will be compiled. Each task will be described in a tabular form as follows:

Name A short descriptive name that will be used to reference this task throughout the document

Description A more detailed description of what should be archived with this task

Name	Description	Grouping	Dependencies
Hash routing	Add *React Router* to enable the usage of hash routing	navigation	None
Query params	Handle the setting and retrieval of query parameters in the URL	navigation	None

Table 3.1: Changes to navigation

Grouping One or more keywords that will be used to assign the task to logical groups that contain tasks of certain similarity. The top-level grouping that was done for sections 2.5.1, 2.5.2 and 2.5.4 are good first candidates for groupings that might be used here.

Dependencies A list of tasks that need to be performed before this task can be started. Will contain NONE if the task has no dependencies.

Navigation

The changes to navigation mainly involved the missing implementation of any altering of the *history* object as described in section 2.5.1.
In order to make the *history* object usable there are generally two options that could be used. As users navigate from page to page in a normal browser session the pages get added to the browsers history object [12]. Additionally, the history object also exposes an API that makes manipulation of the state of the sessions history possible via interaction using javascript [12]. Navigation was implemented with the help of hash routing which will be introduced in section 3.2.1. Additionally, changes to configuration are encoded in the query parameters of the URL. This will be introduced further in section 3.1.3.

There are various libraries that integrate with React that can do this for us. Two popular[1] libraries are *Reach Router* and *React Router*. Both solutions support building a navigation in a declarative manner out of the box[2]. As

[1] `https://www.githubcompare.com/reacttraining/react-router+reach/router`

[2] React router: Navigating in the example enables the back button of the frame: `https://reactrouter.com/web/example/basic`
Reach router: `https://reach.tech/router/api/navigate`

Name	Description	Grouping	Dependencies
Filter algorithm	Create a custom filter algorithm to be used by the application	filtering	None
Filter refactoring	Refactor all current usages of filters to use the new implementation	filtering	Filter algorithm
Filter multiselect	Add multiselect functionality for enumerable columns	filtering	Filter refactoring
Filter UI	Add global User Interface	filtering	filtering

Table 3.2: Changes to filtering

React Router does everything that we need[3] and is the more popular solution[4], it was chosen as the navigation backbone.
This boils down to the tasks shown in table 3.1.

3.1.2 Filtering

The changes required for the new implementation of filtering in the HTML tables of BenchExec's table-generator can be roughly divided into three subsections:

Algorithmic implementation The previous approach used *React Table*'s included filtering logic. In order to assist decoupling from *React Table* as a dependency and to make it easier to extend filtering functionality as well as being more compliant to Reacts recommended design of "moving state up" [7], a custom implementation of the filtering logic was implemented.

Functionality Previously, the filters did not have the ability to select multiple values for enumerable column types.

[3]Hash routing: `https://reactrouter.com/web/api/HashRouter`
Query parameters: `https://reactrouter.com/web/example/query-parameters`
[4]`https://www.githubcompare.com/reacttraining/react-router+reach/router`

Name	Description	Grouping	Dependencies
Location encoding	Encode the currently viewed tab in the URL	sharing	navigation
Hidden columns encoding	Encode hidden columns in the URL	sharing	navigation
Quantile plot config encoding	Encode deviations from default plot configuration in the URL	sharing	navigation
Scatter plot config encoding	Encode deviations from default plot configuration in the URL	sharing	navigation
Filter state encoding	Serialize all set filters in the url and deserialize them when loaded	sharing	navigation, filtering

Table 3.3: Changes to shareability

UI The filter UI was only visible on the table tab but had side-effects for other pages as well. A UI for filters that is accessible on every page was implemented

These sections are ordered in the order of implementation, as each section depends on the implementation of the previous one.
There are no external dependencies to these changes, as they do not access external logic but get used by other components. This results in the set of tasks as defined in table 3.2.

3.1.3 Shareability

Users had no direct way of sharing specific application states or views with other users. The only way to do so was to give other users a guide with steps to reproduce the same or a similar state.

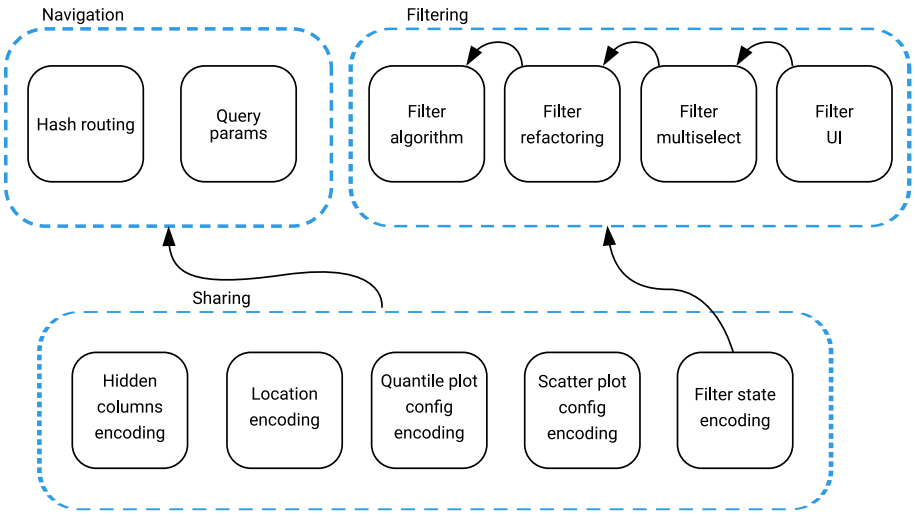

Figure 3.1: Dependency graph of proposed improvements

One set of new features is the ability to share specific views and configurations in a more intuitive and practical manner via the URL.
In the changes of this paper the following state is now encoded in the URL:

- Tab currently being viewed
- Columns which the user chose to hide from any visualizations (see section 2.1)
- Configuration for the quantile plot (see section 2.3)
- Configuration for the scatter plot (see figure 2.4)
- Filter state

In order to successfully implement these changes, the handling of location hashes and query parameters first needed to be implemented, resulting in the *navigation* grouping becoming a dependency of this group.
This leaves us with the list of tasks described in table 3.3

3.1.4 List of Improvements

The list of items in table 3.4 can be represented as a directed graph with the direction of the arrows modelling which other nodes are dependencies. $A \rightarrow B$ would be interpreted as "A depends on B" as can be seen in figure 3.1

Name	Description	Grouping	Dependencies
Hash routing	Add *React Router* to enable the usage of hash routing	navigation	None
Query params	Handle the setting and retrieval of query parameters in the URL	navigation	None
Filter algorithm	Create a custom filter algorithm to be used by the application	filtering	None
Filter refactoring	Refactor all current usages of filters to use the new implementation	filtering	Filter algorithm
Filter multiselect	Add multiselect functionality for enumerable columns	filtering	Filter refactoring
Filter UI	Add global User Interface	filtering	filtering
Location encoding	Encode the currently viewed tab in the URL	sharing	navigation
Hidden columns encoding	Encode hidden columns in the URL	sharing	navigation
Quantile plot config encoding	Encode deviations from default plot configuration in the URL	sharing	navigation
Scatter plot config encoding	Encode deviations from default plot configuration in the URL	sharing	navigation
Filter state encoding	Serialize all set filters in the url and deserialize them when loaded	sharing	navigation, filtering

Table 3.4: List of all improvements

3.2 Implemented new Features

With the list of tasks and their order of implementation that were defined in the last chapter, this chapter will now focus on the structure and ideas behind the implementation of each change as well as technical implementation details.

3.2.1 Hash Routing & Query Parameters

Hash routing uses the URLs *fragment portion* identified by the hash (#) symbol and interprets its content to resolve a routing target. The fragment portion is used to identify a secondary resource to a primary resource [8], thus setting a fragment portion still references the preceding document and changes to the portion will usually not result in a reload of the document. In the following the fragment portion of the URL will subsequently be called ***hash part***.
As already discussed in section 3.1.1, we will use *React Router* (subsequently now called "router") as a routing and navigation state handling framework to implement hash routing. The router is configured using react components in a manner as shown in the abstracted code from the HTML table displayed in figure 3.2.
The application (or a suitable component on a high level of hierarchy) is wrapped using a ***<Router>*** component.
The *<Router>* component should be the core of every application using *React Router* and handles how navigation events are being processed [11]. The *<Router>* component itself is a low-level interface which is implement by the more high-level *<BrowserRouter>*[5] and *<HashRouter>*[6] components. As we wanted to implement hash routing, the *<HashRouter>* was chosen for our purpose. As the configuration of the router is defined in a declarative manner as seen in figure 3.2, it is easy to configure and to maintain.
Implementing these components was pretty straight-forward as the previous approach using *react-tabs* follows a similar structure, with the links in the navigation section (refer figure 2.5) selecting the active tab and each page being wrapped in a separate component that gets configured to become active when the connected tab anchor has been clicked. Similar to *React Router* the whole application is wrapped in an overarching component. Because of the similarities in structure most components just need to be replaced with the corresponding counterpart, resulting in minimal refactoring work.

[5] `https://reactrouter.com/web/api/BrowserRouter`
[6] `https://reactrouter.com/web/api/HashRouter`

```
<HashRouter>
  <div className="overview">
    <div className="links">
      <Link to="/">Overview</Link>
      <Link to="/table">Table</Link>
      <Link to="/quantile">Quantile</Link>
      <Link to="/scatter">Scatter</Link>
      <Link to="/info">Info</Link>
    </div>
    <div className="route-container">
      <Switch>
        <Route exact path="/">
          <Summary />
        </Route>
        <Route path="/table">
          <Table />
        </Route>
        <Route path="/quantile">
          <QuantilePlot />
        </Route>
        <Route path="/scatter">
          <ScatterPlot />
        </Route>
        <Route path="/info">
          <Info />
        </Route>
      </Switch>
    </div>
  </div>
</HashRouter>;
```

Figure 3.2: *React Router* example JSX code from the HTML table (abstracted)

After the change the router was configured to provide the following routes:

Route	**Target**
/	Overview
/table	Table
/quantile	Quantile Plot
/scatter	Scatter Plot

Following this definition, a URL of `file://<path-to-html>/#/` would resolve the *<Overview>* component, while `file://<path-to-html>/#/table` would resolve the *<Table>* component.

The usage of the routers *<HashRouter>* automatically configures the *<Link>* components to append the configured path to the hash-portion of the URL and the *<Route>* components to read from it.
We do however also require the use of query parameters to assist shareability as discussed in section 3.1.3. *React Router* has no distinct way of handling this. In fact, it is even recommended by the router to use the standard browser API to access query parameters via `URLSearchParams`[7].
The usage of the standard API however has some limitations in our use-case. To be consistent with the common usage and positioning of query parameters in URLs, the URL would need to follow the usual schema like `<protocol>://<hostname>/<path>?<query-parameters>`.
This causes some issues. As we do all our navigation via the local filesystem, we need to store our route information in the hash-portion of the URL. As the hash-portion is "terminated by the end of the URI" [8], the query parameters, when positioned after the route in the hash-portion, will be interpreted as being part of the hash-portion even if it is prefixed by a "?" and thus will not be picked up by browser provided accessors for the search portion[8] of the URL.
In order to read and modify query parameters, custom getters and setters were implemented that accesses the parameters via string operations over the URL received by the global `location.href` property.

3.2.2 Persistance of Component Configurations in the URL

Adding the possibility of navigating via the URL as well as persisting information via query-parameters that were set up in the previous section now allows us to move configuration of certain elements or components of the application into the URL. This configuration will be read during the load of the page and the application state and components will be configured as defined in the URL.
Apart from the state representing the curently active tab that was introduced in form of routes in the last section, some components in these active tabs also have state that needs to be persisted in the URL.
The components that allow configuration via the URL namely are the *<QuantilePlot>* and the *<ScatterPlot>* components. All configuration options outlined in section 2.3 and figure 2.4 are persisted into the URL if they do not match default values and are read from the URL during navigation. The

[7] `https://reactrouter.com/web/example/query-parameters`
[8] The portion of the URL containing query parameters

state of these configuration items is in the form of typical query parameter key/value pairs like `<key>=<value>` separated by an ampersand (&) symbol.

The available query parameters that can be used to set the configuration of components are as follows:

Key	**Description**
hidden<runsetId>	Sets the column id of a runset to be hidden. *runsetId* is the index of the runset in question
selection	Sets the *selection* configuration for quantile plots
plot	Sets the *plot* configuration for quantile plots
scaling	Sets the *scaling* configuration for quantile and scatter plots
results	Sets the *results* configuration for quantile and scatter plots
toolX	Sets the tool to select values from for the X axis of scatter plots
columnX	Sets the column to select values from for the X axis of scatter plots
toolY	Sets the tool to select values from for the Y axis of scatter plots
columnY	Sets the column to select values from for the Y axis of scatter plots
line	Sets the aux. lines configuration in scatter plots

3.2.3 Filter Algorithm & Refactoring

The filter functionality is a central part of the tool and useful to pinpoint outlier values and focus the analytic capabilities of the application onto a certain subset of the dataset. As explained in section 3.1.2, filtering was only available from the table view. This was mainly caused by the hard dependency on *react-table* for filtering. The steps outlined in table 3.2 consequently removed this dependency and allow access to the filtering functionality from a globally accessible component that allows changes to the dataset from any tab of the application. This component will be introduced in detail in section 3.2.4.

The Filter Algorithm

Generally a filtering logic consists of two main inputs: the *dataset* and the *filters*. The purpose of the filter is to remove entries of the *dataset* that don't satisfy the constraints defined in *filters*. The dataset contains rows that represent one task. This task might be in more than one runsets. In this case, the row will contain information for all runsets that it belongs to. Each run will have one or more describing properties per runset. These are the same as described in section 2.2. An approach to filtering might be to sequentially iterate over each cell and then apply any existing constraint for this column onto them.
Let the amount of rows be n, the amount of total columns be m and the size of the filters be k.
The approach presented above will iterate over each cell of the table of size $n \cdot m$ and then needs to iterate over all filters to remove items for each constraint defined in the filters. This results in a runtime complexity of $\mathcal{O}(n \cdot m \cdot k)$ which, as this filtering will be done on the fly in the browser, is a suboptimal approach.
An approach that would allow us to remove one of the largest dimensions n or m would be more desireable.
To archive this we will create an approach using a reduced version of the filter list, called a *matcher*, and then implement a logic that will filter out any items breaking the constraints in the matcher out of the dataset in one iteration.
The idea behind the matcher is that the dataset can be represented as a structure that is grouped by and accessible by different properties (mainly the runset ID and the column ID are used for filtering), thus we can use these IDs to directly retrieve the constraints from the matcher object by direct access, mitigating the need to iterate over all cells. The selection of the columns in each row then is constant.
The global filter object will be added on top of the already existing filter component on the table tab. In order to ensure that both solutions are compatible, the matcher object will be constructed from the array of filter objects that are generated internally by react-table.
As the IDs of both the runset and the column are already present in these objects, reducing an array of filter objects into one is a simple task. For the special case of numerical range filters first introduced in section 2.2 we will introduce two new properties, `min` and `max`, to remove the need of parsing the raw range filter string on each iteration. A point to consider here is that the syntax of the range filters allows for shortcuts like `"123:"` or `":"`, translating into *"All values bigger than or equal to 123"* and *"All values"* respectively.

One of the requirements for the filter algorithm is the ability to handle multi-select filters[9]. This results in two different connections for filters depending on their grouping in the matcher object: All constraints within the same column of a run should be disjuncted to allow the multi-select behavior, all constraints across columns of the same run should be conjuncted.
The algorithm introduced looks as follows: ($a|| = b$ is short for $a = a||b$, $a\& = b$ is short for $a = a\&b$)

[9] `https://github.com/sosy-lab/benchexec/issues/481`

```
1: for row in dataset do
2:     if matcher.id exists then
3:         if row.id !matches matcher.id then
4:             remove row
5:         end if
6:     end if
7:     for runset in matcher do
8:         columnPass = false
9:         categoryPass = false
10:        statusPass = false
11:        for column in matcher[runset] do
12:            for filter of matcher[runset][column] do
13:                if filter.min exists or filter.max exists then
14:                    value = row[runset][column].value
15:                    columnPass|| = value ≥ min&value ≤ max
16:                else if isCategory(filter) then
17:                    value = row[runset][column].value
18:                    categoryPass|| = value == filter.value
19:                    columnPass = categoryPass&statusPass
20:                else if isStatus(filter) then
21:                    value = row[runset][column].value
22:                    statusPass|| = value == filter.value
23:                    columnPass = categoryPass&statusPass
24:                else
25:                    value = row[runset][column].value
26:                    columnPass|| = value == filter.value
27:                end if
28:                if columnPass == true then
29:                    break
30:                end if
31:            end for
32:            if columnPass == false then
33:                remove row
34:            end if
35:        end for
36:    end for
37: end for
```

Figure 3.3: New algorithm used for filtering

Using the approach in figure 3.3 we only iterate over the rows of the dataset and then over the set constraints of the filters. Columns will be accessed directly via the corresponding property of the row, resulting in a complexity of $\mathcal{O}(n \cdot k)$, removing the factor m. As factor m was one of the two biggest factors, this results in an improved performance in most cases.

Refactoring and Challenges

To make the new algorithm, which is running in a top-level component, and the old filter selectors that are embedded in the table component compatible, two refactorings were necessary. Firstly, the previous filter functionality that was handled by *react-table* was completely disabled. All filtering is now handled using the new algorithm, yet we still want to keep the functionality of setting filters via the column headers of the table.
The column headers of the table contain components that allow for the creation of filters as introduced in section 2.2. This is important as users that are used to applying filters using the table tab or just feel more comfortable doing it there can still use the same functionality even though, under the hood, a different algorithm is being used.
This leads to the second refactoring. The state of the dataset was kept in two different places. *React-table* kept a reference to the original, unfiltered dataset. When filtering in the table component occurs, the results of this filtering algorithm along with the applied filters were passed back to the *Overview* component which then replaced its internal *data* state object with the filtered result.
To ensure compatibility between the newly created algorithm and the table, the filters that get created by the column headers get intercepted and redirected to the new filter algorithm. The results of the execution of the algorithm will then be applied to the global *data* state object. Lastly, the *react-table* component will receive a reference to the currently filtered data as well as the currently set filters. As we have disabled the filtering functionality of the table component previously, the state of the global filtered data and the dataset that is handled internally in the table component are equal. This is a crucial requirement for the newly added global filter user interface.

3.2.4 Extending the User Interface

In order to show an exhaustive list of all available filters, a lot of UI space would be needed for more complex tables. This has two implications. First, we need a component that provides a lot of available space for the filters. Second, as the list of available filters grows with the complexity of the table

and might result in a big list of filters to chose from, it would provide better user experience to not show all options in the beginning but to let the user add them when needed.

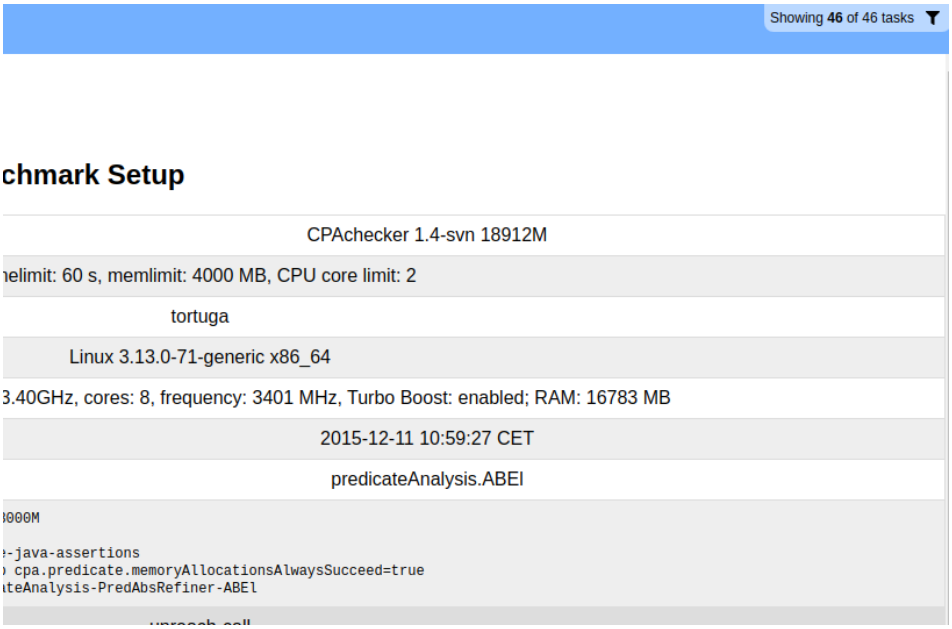

Figure 3.4: The new filter button

In the top right corner of the navigation UI is a component that displays the number of selected items in the dataset. This can be seen in figure 3.4. As filtering the dataset will have direct impact on this component and as it gives the user an idea on the size of the currently filtered data, the button to access the filters has been merged with it. To make this more obvious to the user, an icon hinting the filter functionality has been added. When the user clicks the button, a sidebar will slide in from the right. This sidebar gives the user an overview of all currently set filters and also allows the addition of filters when necessary.

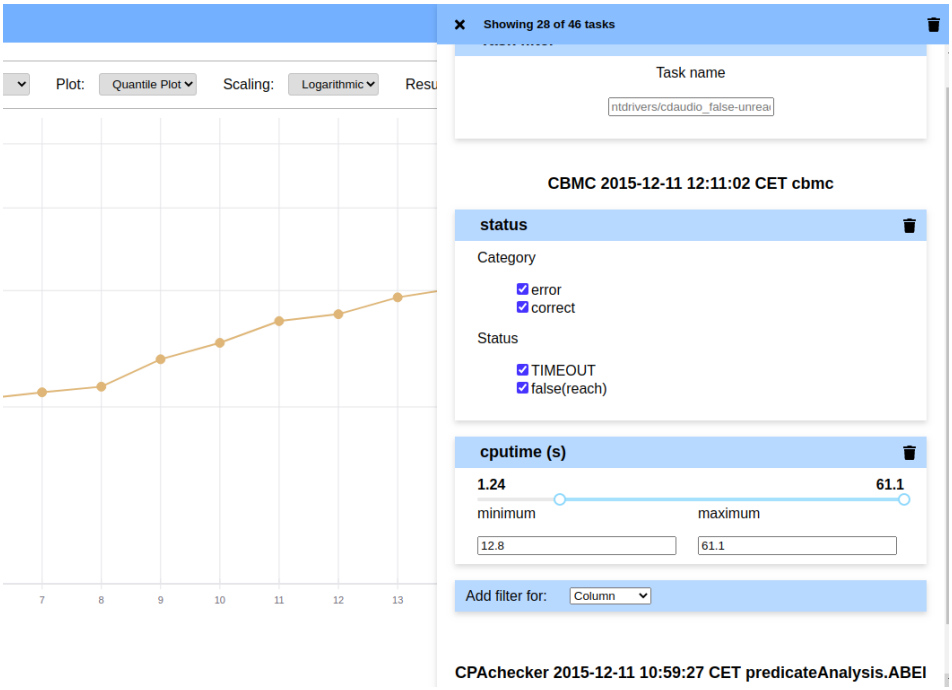

Figure 3.5: Opened filter component

The header at the top of the sidebar in figure 3.5 gives the user the overview of selected vs. available tasks in the same way that the toggling button provides. The user may slide the sidebar back out of view by pressing the **X** button on the left side of the header. With a click of the trashcan icon on the right side of the header all constraints can be removed and thus the filter will be reset to its initial state.

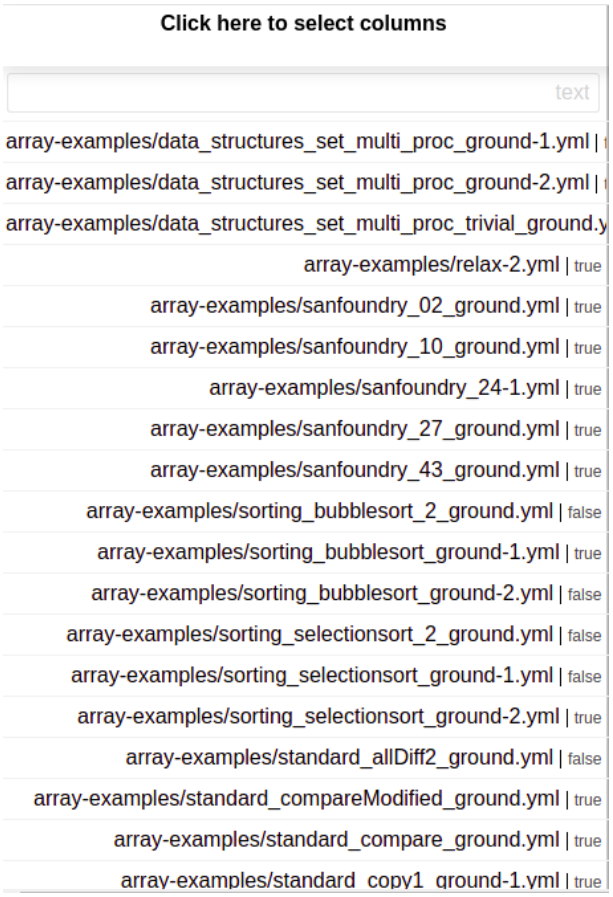

Figure 3.6: Task-ID filter in the table tab

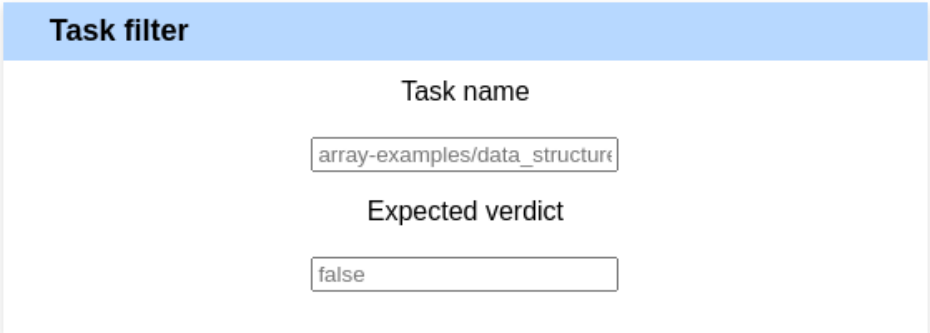

Figure 3.7: Task-ID filter in the filter component

Below the header the user will find an "ID-Filter", which will add a filter constraint for a task-ID (refer to section 2.2). This constraint will be applied

over all runsets, as the task-ID is static. With the new global filter component users are now able to specifically filter specific parts of the task-ID. The task-ID may be a construct of different properties, for example "task name" and "expected verdict". Previously the ID was only filterable as a whole, the user was unable to apply filters to different parts as seen in figure 3.6. This is now improved in the new filter component, as users are given the option of applying more granular filters on the task-ID by allowing the setting of different constraints on each different part of the task-ID as shown in figure 3.7.
Below the ID-Filter section, one section per runset is shown. These sections are identified by a header string that contains the name of the runset. In each of these sections the user can create a constraint for any of the runsets columns. These constraints are conjuncted and are applied to the individual runset.
When a user chooses to create a new constraint, a card component will be rendered as can be seen in figure 3.5. This card component can be seen as a logical and visible wrapper of an underlying constraint. The input fields in the card component will be chosen depending on the type of the data that gets represented in the column that this constraint is applied to.
The inputs can be one of:

- Checkboxes for enumerable values
- A slider with two handles and two numeric input fields to select a range for numeric values
- A text-box for text filtering of strings

Visual representations of these input types can be referenced in figure 3.8.
As we have implemented the new filtering algorithm the checkboxes can now allow the selection of multiple values, which has been a requested feature[10]. Like discussed in the previous section, these values will be disjuncted during evaluation of the constraints.
As numeric values are filtered by setting a specific interval of valid values, a slider with two handles has been implemented for numeric values to help with setting numeric filters more quickly. This presentation of the filter visually emphasizes the functionality of the filter itself and additionally also gives instant feedback to how the currently selected filter range relates to the total range of all values of the dataset. When the user moves the slider, the input fields will update with the representative values and vice-versa.

[10] `https://github.com/sosy-lab/benchexec/issues/481`

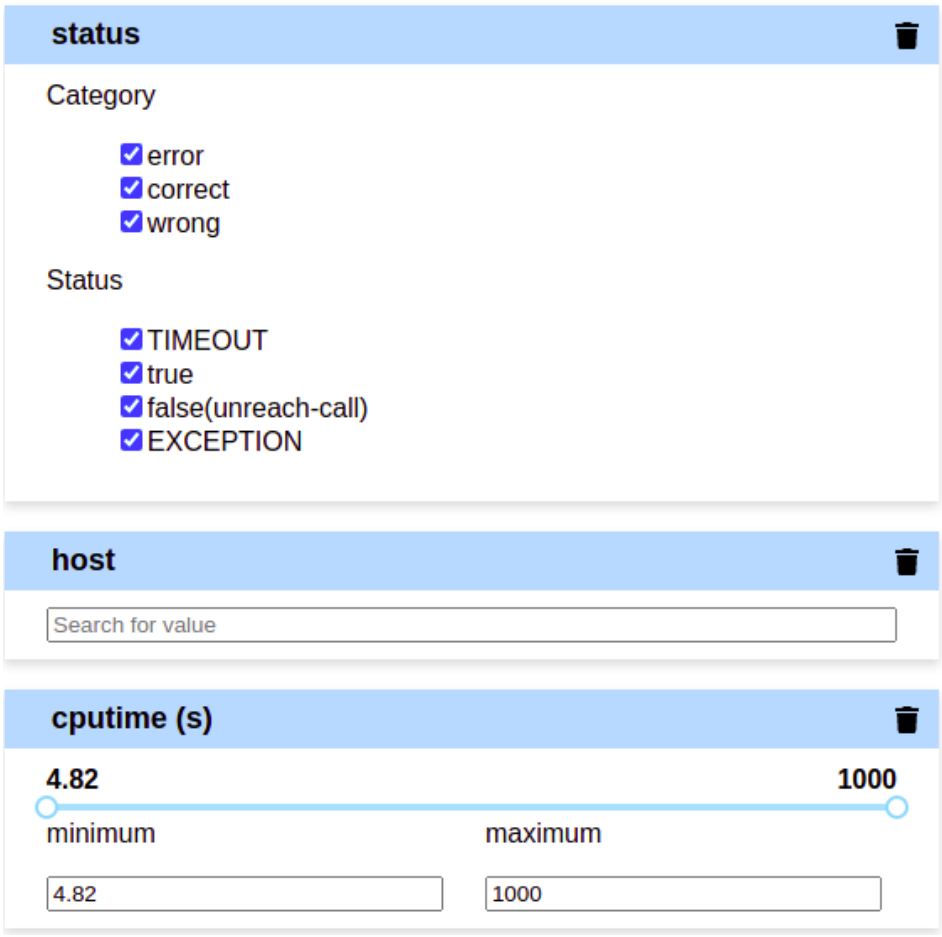

Figure 3.8: Input types for filters

String filters are represented by a text input field that allows the user to perform a text search over all strings of a certain column in the dataset.

By ensuring compatibility in the implementation step of the new filter algorithm in the previous section, the filter state of the table component and the filter state of the sidebar are synchronized.
The user can choose to remove a single constraint by pressing the trashcan icon on the top-right corner of each *FilterCard* or delete all set filters by pressing the trash can icon on the top right of the sidebar. Another merit of moving the state and implementation of the filter functionality up is that we now have the option to also serialize the filter state in the url to both allow users to share the state of a filter to third parties and to make keep track of set filters in the browser history and therefore tying an "undo" functionality

to the browsers back button.

3.2.5 Serialization of Filters

Following the core concept of moving as much state as possible into the URL as introduced in section 3.1.3, filters are now serialized in the URL. The serialization works by transforming the filter state into a declarative representation of the filters that is also human readable. The filters are represented using the following grammar:

```
[idFilter ","]runsetFilter {"," runsetFilter}
with
idFilter := "id(values(" value {"," value} "))"

runsetFilter := runsetId "(" columnFilter {"," columnFilter} ")"

columnFilter := columnId "*" name "*(" filter ")"

filter := valueFilter|statusColumnFilter

valueFilter := "value(" value ")"

statusColumnFilter := statusFilter|categoryFilter|
(statusFilter "," categoryFilter)

statusFilter := "status(" ("in(" value {"," value} ")"|
"notIn(" value {"," value} ")"|"empty()") ")"

categoryFilter := "category(" ("in(" value {"," value} ")"|
"notIn(" value {"," value} )|"empty()") ")"

value := ?  urlencoded terminal value ?

runsetId := ?  id of the runset to apply the filter to ?
```

To help minimizing the length of the serialized string while still being able to represent the values in a understandable way, *enumerable* value types are either represented using the keywords *in* or *notIn*. If more than half of all values are selected, *notIn* will be used to represent the filter, as the amount of non-selected items will be less. Otherwise, *in* will be used. For example, if a

table has three different status values *(true, false, error)* and only *true* is selected, it would result in `1(0*status*(statusFilter(in(true)))`, whereas if *error* and *false* are selected the serialization would be `1(0*status*(statusFilter(notIn(true)))`. The serialized string can represent all different column types by encoding their raw input values directly into the url as so: `1*cputime*(value(%3A1120)))`.

3.2.6 Statistic Calculations via Workers

Another side-effect to the extraction of filters into a top-level component of the application is that any other components can subscribe to receiving filtering results from it. This allowed for the addition of a feature that has been requested[11], which is the recalculation of the previously statically included statistics that are shown own the *Overview Page* in the *Statistics Table*. Statistics now automatically get recalculated whenever the dataset has been filtered. As we need to calculate these statistics additionally to the filtering over the whole dataset, calculation has been off-loaded to web workers[12]. JavaScript is executed in a single thread in the browser, the main thread. The main thread handles the execution of reflows, garbage collection, JavaScript execution and layout [9]. As JavaScript is run on the same thread that handles the painting of the page, any long running, blocking scripts may freeze up the page and restrict the rendering and interactivity of the UI. Web workers (subsequently named workers) are separate scripts that are each executed in separate threads [13]. By nature of the execution of workers in separate threads, communication with the workers and other workers or the main thread is done via messages [13].

As aggregations for the statistics are computed on a per-column level, we are able to split the filtered data set by column and then calculate the aggregations for each column in parallel. By splitting the calculation in smaller chunks we are also increasing performance, as multiple results are calculated simultaneously as well as reducing the load on each worker instance by minimizing the work it needs to do in one execution. As there is some overhead attached to the creation of workers, a pool of workers will be created at the first load of the application. Workers in this pool will receive jobs and return to an idle state after completing each job, making them re-usable. A job in this context is the calculation of statistics for a column of the filtered dataset and the job is considered complete once the calculation of statistics are done

[11] `https://github.com/sosy-lab/benchexec/issues/125`

[12] `https://developer.mozilla.org/de/docs/Web/API/Worker`

[13] `https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers#sending_messages_to_and_from_a_dedicated_worker`

```
1  // enqueue is an exposed method of the WorkerDirector which
       handles the creation,
2  // delegation, and processing of jobs
3  import { enqueue } from 'workerDirector';
4
5  // The property "name" is the identifier of the pool to use.
6  // The property "data" is the data that a worker should
       process
7  const job = enqueue({ name: 'stats', data });
8
9  // as jobs are wrapped in a Promise, we can suspend the
       current code block and await completion
10 const result = await job;
```

Figure 3.9: Example of the usage of the WorkerDirector module

and results have been received on the main thread. If no free workers are available when a job is received, it will be put into a queue and processed once a worker completes a job and becomes available again.
This method is implemented using a new module, the `WorkerDirector`, which acts as an orchestrator managing the queue of incoming batches of data and distributes them to a pool of workers as jobs. The `WorkerDirector` is set-up to be able to handle different pools containing different implementations of workers, which makes offloading other potentially expensive tasks in the future easy. Each scheduled Job that gets put into the queue is wrapped in a JavaScript `Promise` which enables us to use JavaScripts native asynchronous API to handle the asynchronous nature of such scheduled jobs as displayed in figure 3.9.

4: Evaluation

All of the changes and added features that are outlined in this paper involve significant changes to how data is handled in the whole application. While removing the coupling between dependencies gives a lot of freedom in terms of responding to change, we also add functionality that sits on top of the extracted code and we might have removed any internal optimizations of dependencies like *react-table* to reduce render times, as the components now no longer handle filtering and data changes internally. This chapter discusses the gained capabilities versus penalties in performance by comparing different versions of the application via benchmarks. The code to set-up, run and process the benchmarks is publicly available[1]. The tests are run using cypress[2], a frontend testing tool. As cypress runs the test in an actual browser, we are able to get reliable results as render times are included in the timings of each test.

4.1 Setup

The tables to be tested against were generated using two versions of the BenchExec tool, one before[3] the implementations discussed in this paper (referenced as *old* in the plots) and one after[4] all changes had been implemented (referenced as *new*). The data that is used by the TableGenerator is randomly generated for each benchmarking run. Each table has 15 columns and the number of rows will be increased in increments of 1,000 with runs being executed for numbers of rows spanning from 1,000 to 58,000. We execute the benchmarking task (for example testing the numeric filters) 10 times each and will store the raw results as well as aggregated results in a JSON file. These files are then transformed into csv files for further analysis.

[1] `https://github.com/DennisSimon/benchexec-benchmarks`
[2] `https://www.cypress.io/`
[3] Commit shorthash: 99470a
[4] Commit shorthash: 072d3da

4.2 Benchmarking of Filters

The changes to the filter logic have the biggest impact to the application overall as most components consume the resulting dataset. In the following we will observe the changes in performance for filtering:

- Numeric filters
- Enumerable filters using status and category fields
- Task-ID filters

The test is performed by first navigating to the *Table* tab and then enter a value to be filtered by. The time from the start of the input of the value to the UI components being updated with the new values will be used as timing data for the benchmarks. It is worth noting that as these timings include the render time as well, there might be further improvements to reducing render costs of the table in the case when filtered data is provided externally, this however is not investigated in this paper.

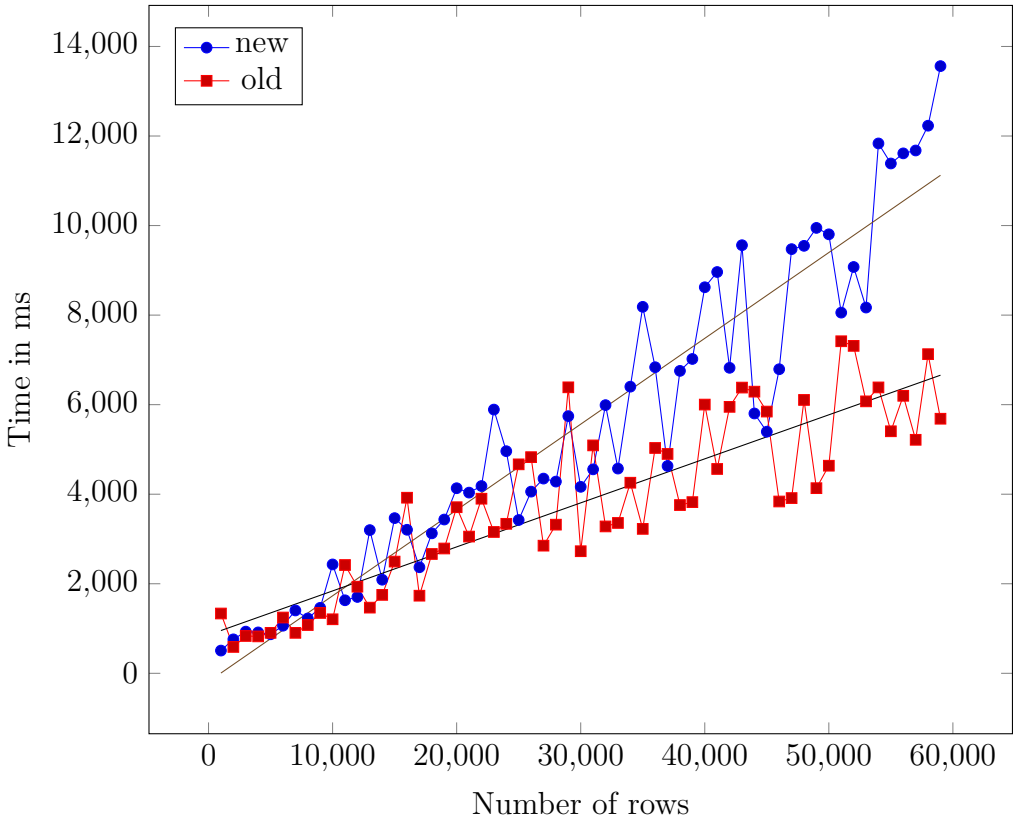

Figure 4.1: Average computation time of numeric filters per number of rows

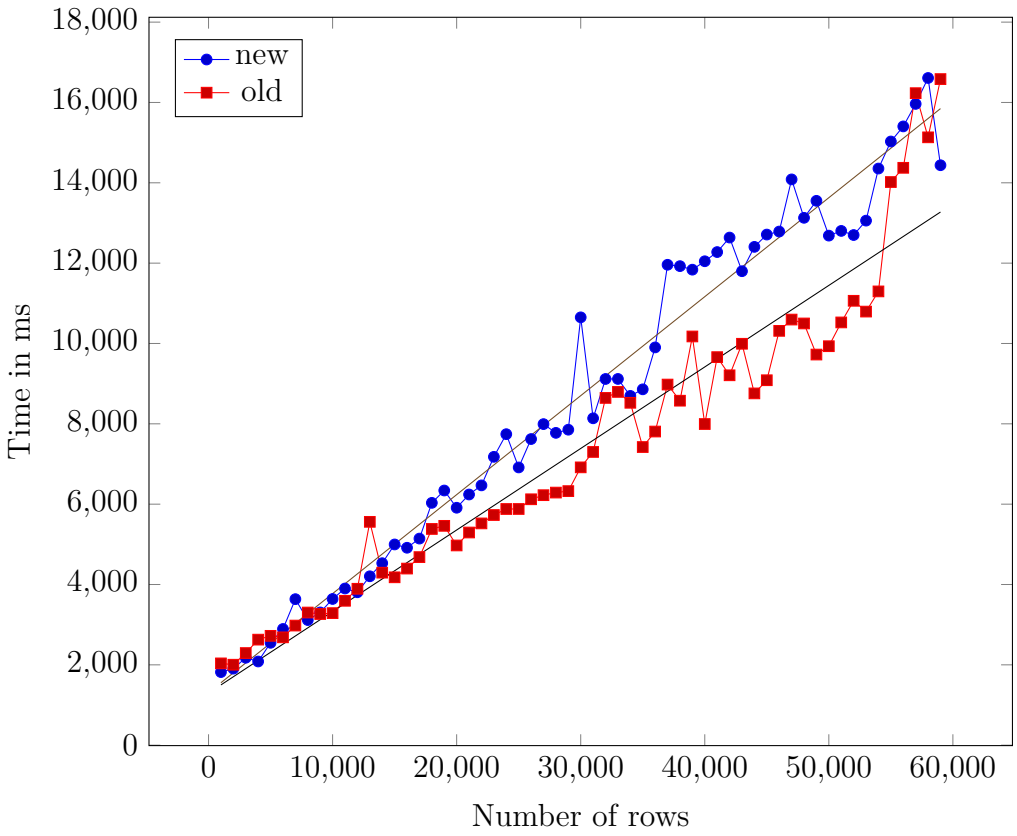

Figure 4.2: Average computation time of status filters per number of rows

As can be seen in figure 4.1, testing of numeric filters produced a rather noisy result which might be caused by the random generation of numeric values in the test tables. The regression lines are diverging from each other, resulting in performance penalties of around roughly 1.5 - 2x for the new filter implementation on large tables.

The performance of the status filters in figure 4.2 are closer to the original implementation with the divergence of the regression lines being a lot smaller than the numeric filters. The benchmark results also are less noisy, apart from a sudden spike in execution time for the old implementation starting at 55,000 rows. This spike has been observed over multiple benchmarking runs and has been consistent.

The execution of times of the task-ID filters shown in figure 4.3 are similar. The regression lines are only diverging by a small amount. The benchmarking data has as notable spike for the execution times of the old implementation

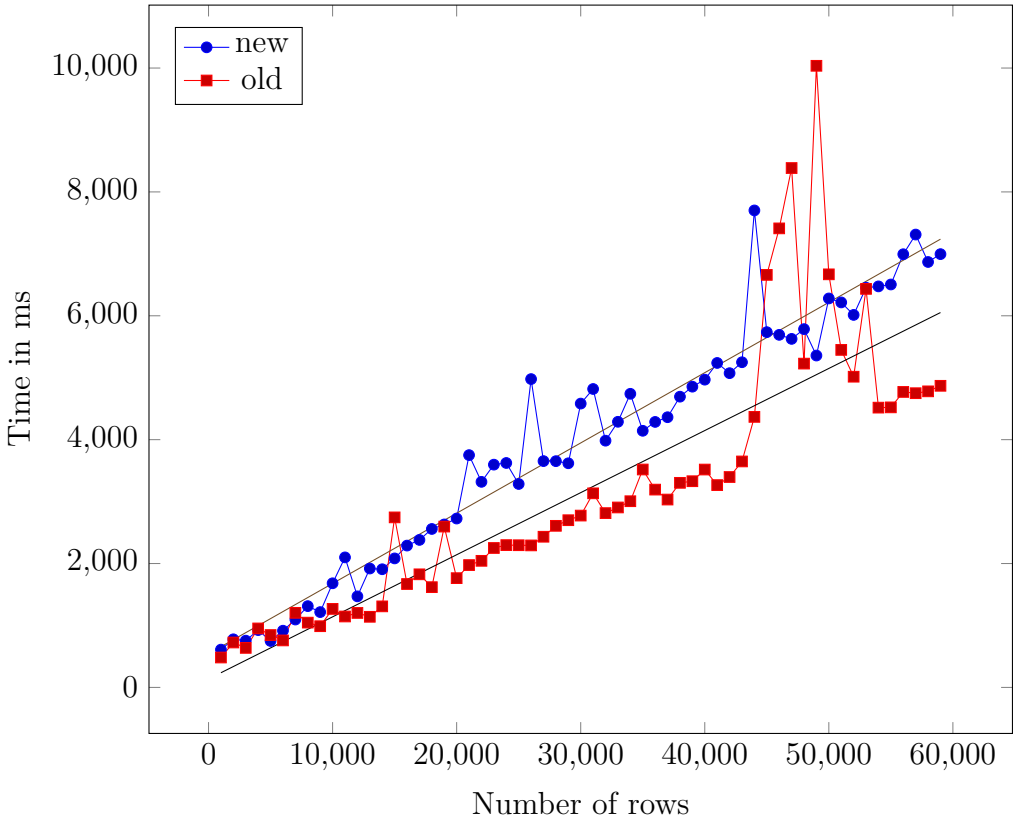

Figure 4.3: Average computation time of task-id filters per number of rows

between 44,000 and 54,000 number of rows. A similar spike is observable for the new implementation at 43,000 number of rows but it managed to normalize again at 44,000 number of rows.

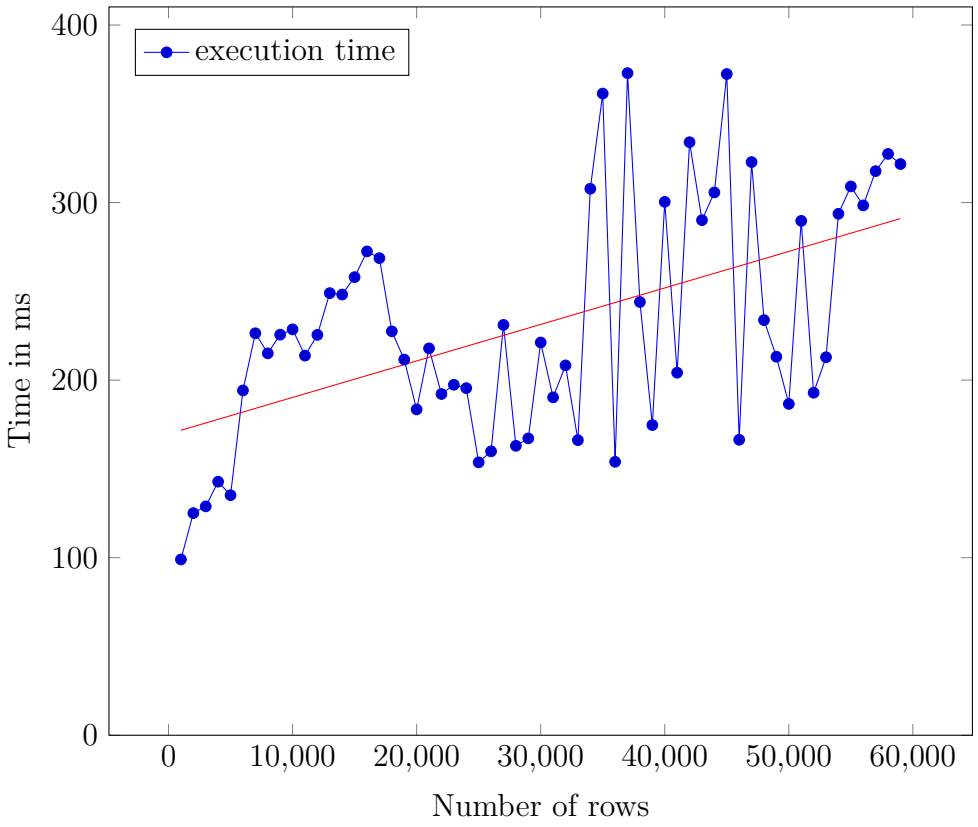

Figure 4.4: Average computation time of statistics

4.3 Benchmarking of Statistics Calculation

The calculation times for statistics in figure 4.4 vary between 99 and 372 milliseconds. As all results are in the sub-second area, the user experience should not be impacted negatively by this change [10].

4.4 Result Evaluation

When comparing all results discussed in the the previous sections we can see that the newly implemented filters have some performance penalty attached and get out-performed by the solution native to *react-table*.
The cause for this penalty likely is related to now missing internal optimization in *react-table*, forcing a complete re-render of the whole table on change.
The additional features that were introduced by the changes however outweigh the penalties, as the features greatly enhance the usability of the tool

and allow for further optimization and development of new features in the future.
As calculations of statistics are in the sub-second area and statistics are rendered below the fold on navigation to the overview tab by default, the results are acceptable.

5: Conclusion & Future Work

The goal of this thesis was to enhance the functionality and user experience of the BenchExec HTML tables and to provide changes to functionality in a modular, maintainable and extendable way. Some functionalities, like the handling of state in the URL, are already extended by others[1], showcasing that the goal of re-usability and extendability is already met. As the implementation process of the features described was an iterative one, many features are already actively being used by users. The feedback received was positive, with the serialization of state in the URL and the hash routing being mentioned the most.
Another big goal of the thesis was to extend the filters. These are now globally accessible in the application and provide new features and improvements like the redesign of the UI, multi-selection of enumerable values and the ability to retrieve and set filter configurations from the URL, adding value for distributed teams.
The work described in this paper adds a lot of new functionality and can also be used as the foundation for other improvements to come. As the handling of filtering, state management and statistics calculation is now done within the application, there can be many interesting additions to it. As seen in section 4.2 there are still topics to explore for further improvements, especially regarding the numeric filters.
For example the performance loss on the *Table* page is, as already discussed, likely due to the loss of internal optimization of *react-table* causing a lot of unnecessary render cycles which i could not completely mitigate yet. Providing a implementation with strong memoization strategies to reduce the number of renders could improve the perceived performance of the filters on the *Table* page by reducing raw render time.
Currently the worker pool for the statistics calculation is hard-coded to include 8 workers. This amount of workers might be too less or too much depending on the system and the amount of columns in the table. Providing a dynamic pool size depending on the dataset and the available resources of

[1] `https://github.com/sosy-lab/benchexec/pull/582`

the system running the *HTML table* could be an interesting topic for research to optimize the use of workers. Another angle to consider would also be the off-loading of other tasks, like filtering or other calculations, to workers and observing if a increase in performance could be achieved.
Lastly, the development of the HTML tables is not completed and, as software development is a iterative process, will likely see many additional changes, including changes to the features described in this paper, in the future.

Bibliography

[1] Dirk Beyer. "Advances in Automatic Software Verification: SV-COMP 2020". en. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Armin Biere and David Parker. Vol. 12079. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 347–367. ISBN: 978-3-030-45236-0 978-3-030-45237-7. DOI: 10.1007/978-3-030-45237-7_21. URL: http://link.springer.com/10.1007/978-3-030-45237-7_21 (visited on 2021-04-04).

[2] Dirk Beyer. "Software Verification: 10th Comparative Evaluation (SV-COMP 2021)". en. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Jan Friso Groote and Kim Guldstrand Larsen. Vol. 12652. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 401–422. ISBN: 978-3-030-72012-4 978-3-030-72013-1. DOI: 10.1007/978-3-030-72013-1_24. URL: http://link.springer.com/10.1007/978-3-030-72013-1_24 (visited on 2021-04-04).

[3] Dirk Beyer, Stefan Löwe, and Philipp Wendler. "Reliable Benchmarking: Requirements and Solutions". In: *International Journal on Software Tools for Technology Transfer (STTT)* 21.1 (2019), pp. 1–29. DOI: 10.1007/s10009-017-0469-y. URL: https://www.sosy-lab.org/research/benchmarking/.

[4] Laura Bschor. *Modern Architecture and Improved UI for Tables of \sc BenchExec*. Published: Bachelor's Thesis, LMU Munich, Software Systems Lab. 2019.

[5] Mozilla MDN contributors. *History - Web APIs | MDN*. URL: https://developer.mozilla.org/en-US/docs/Web/API/History (visited on 2021-01-09).

[6] Mozilla MDN contributors. *SPA (Single-page application) - MDN Web Docs Glossary: Definitions of Web-related terms | MDN*. URL: https:

//developer.mozilla.org/en-US/docs/Glossary/SPA (visited on 2021-01-09).

[7] GitHub contributors. *Lifting State Up – React*. en. Documentation. URL: https://reactjs.org/docs/lifting-state-up.html (visited on 2021-01-20).

[8] Larry Masinter, Tim Berners-Lee, and Roy T. Fielding. *Uniform Resource Identifier (URI): Generic Syntax*. en. URL: https://tools.ietf.org/html/rfc3986#section-3.5 (visited on 2021-01-22).

[9] Mozilla MDN contributors. *Main thread - Definition | MDN*. URL: https://developer.mozilla.org/en-US/docs/Glossary/Main_thread (visited on 2021-04-09).

[10] Jakob Nielsen. *Response Time Limits: Article by Jakob Nielsen*. en. URL: https://www.nngroup.com/articles/response-times-3-important-limits/ (visited on 2021-03-14).

[11] react-router GitHub contributors. *React Router - Router Component*. en. URL: https://reacttraining.com/react-router (visited on 2021-01-22).

[12] W3C Organisaton. *6.10 Session history and navigation — HTML 5*. URL: https://dev.w3.org/html5/pf-summary/history.html (visited on 2021-01-10).

[13] Web Hypertext Application Technology Working Group. *HTML Standard - Web Workers*. URL: https://html.spec.whatwg.org/#workers (visited on 2021-04-09).

List of Figures

List of Tables