Performance-Aware Design of Web Application Front-Ends

Dennis Westermann[1], Jens Happe[1], Petr Zdrahal[2], Martin Moser[2], and Ralf Reussner[3]

[1] SAP Research, Karlsruhe, Germany
{dennis.westermann,jens.happe}@sap.com
[2] SAP AG, Walldorf, Germany
{petr.zdrahal,martin.moser}@sap.com
[3] Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
reussner@kit.edu

Abstract. The responsiveness of web applications directly affects customer satisfaction and, as a consequence, business-critical metrics like revenue and conversion rates. However, building web applications with low response times is a challenging task. The heterogeneity of browsers and client devices as well as the complexity of today's web applications lead to high development and test efforts. Measuring front-end performance requires a deep understanding of measurement tools and techniques as well as a lot of manual effort. With our approach, developers and designers can assess front-end performance for different scenarios without measuring. We use prediction models derived by a series of automated, systematic experiments to give early feedback about the expected performance. Our approach predicts the front-end performance of real-world web applications with an average error of 11% across all major browsers.

1 Introduction

Recent industrial studies [4] show that the responsiveness of web applications directly affects customer satisfaction and, as a consequence, business-critical metrics like revenue and conversion rates. Guidelines on how to optimize front-end performance, such as those published in the books of Steve Souders [7,8], are very popular among web developers. Also, tools like WebPageTest [2] or YSlow [3] are more and more adopted to support the implementation of performance best practices and to help identifying performance problems. For the development of web-based enterprise applications, companies often rely on JavaScript libraries that provide a uniform appearance, as well as a set of UI elements and utility functions commonly used in this kind of applications. Besides the classical challenges addressed by the guidelines and tools mentioned before, UI developers and designers need to evaluate the impact of the design of a screen on front-end performance. This involves questions like „How many columns and rows can I add to a table of type X in my web application without violating performance requirements?“or „What is the impact of backend call Y on front-end performance?". Theoretically, these questions could also be answered with the existing performance measurement and analysis tools. However, practically the effort for applying measurement-based approaches to these kind of questions is too high, which hinders the flexible, performance-aware construction and evaluation of screen designs. Moreover, the development of a screen's design is usually conducted before the screen is actually implemented (e.g., using wireframe or mockup tools). As a consequence, early performance feedback (prior to implementation) is essential to drive the deployment of fast web applications [5].

In this paper, we present an approach that enables performance-aware design of web application front-ends. Our main contribution is a methodology that allows performance experts to efficiently derive prediction models for UI libraries. These models can, for example, be integrated in design tools in order to give early performance feedback to the large amount of designers and developers that use the library. To get the early feedback about the expected front-end performance of their design, developers neither need to implement the web application, nor do they need to conduct performance measurements.

The main challenge in deriving the performance prediction models is to deal with the huge design space that is spanned by a UI library. To overcome this challenge, we build on the results of our previous research on automated performance evaluation experiments [9,10] and propose an experiment-based prediction model construction process. In our case study, we evaluated the impact of different screen design alternatives on front-end performance for applications developed with the JavaScript library SAP UI5 [1]. Based on the experiment results, we derived a set of assumptions and heuristics and developed a prediction model that allows estimating the impact of screen designs on performance for three major browsers (Internet Explorer, Chrome, and Firefox). While the experiment results and the prediction model are specific for the SAP UI5 library, we also describe our systematic process that can be applied for the efficient construction of such prediction models in other scenarios.

We validate our approach by comparing predictions to measurements using screens of two real-world enterprise web applications. Both have been developed with the SAP UI5 library. The results show that we can predict the front-end performance for the screens of these applications with an average prediction error of 11% across all studied browsers.

2 Prediction Model

The performance prediction model introduced in this section quantifies the relationship between the construction of SAP UI5 based web application screens and the browser CPU time consumed by the screens in different browsers. Moreover, we outline a process that allows to derive such a model efficiently.

Based on the results of a set of upstream experiments, we define the following assumptions and heuristics:

- The browser CPU time is a stable metric to describe front-end performance costs of a web application. Moreover it abstracts from influences that are hard to control such as network latency.
- The browser CPU time consumed to process different UI element types is additive (i.e., there is no interdependency between control types).
- The browser CPU time differs significantly between different UI element types.
- The properties of complex control types can significantly contribute to the browser CPU time consumed to process a UI element.
- The placement of UI elements on a screen does not have a significant effect on CPU time (at least if the nesting level stays in a reasonable range).

Utilizing these assumptions and heuristics, we define a performance prediction model as well as a process to derive a concrete instance of this prediction model for SAP UI5.

If a screen S of a web application consists of the UI elements $e_1, ..., e_n$, we write: $S = e_1 \cdot ... \cdot e_n$ where $\cdot$ denotes the composition of UI elements (e.g. a screen that consists of tables, buttons, and text fields). Hence, when a UI developer creates a screen S, he evaluates $e_1 \cdot ... \cdot e_n$. We assume this composition as associative and commutative (i.e., the UI elements can be arbitrarily placed on the screen).

Furthermore, we define $\phi(S)$ as the front-end performance of screen S which is in our case expressed as the browser CPU time consumed to load the full screen. Following the additivity and placement assumptions, we state that the performance of the UI element composition is the sum of the performance values of the individual UI elements $(\phi(e_1), ..., \phi(e_n))$ and a constant offset (ϵ_S).

$$\phi(S) = \phi(e_1 \cdot ... \cdot e_n) + \epsilon_S = \phi(e_1) + ... + \phi(e_n) + \epsilon_S \tag{1}$$

The offset ϵ_S describes the browser CPU time consumed to load an empty screen. This includes for example the CPU time required to load the UI libraries and the CSS files (i.e. all components of a screen that are independent of a certain UI element).

Depending on its properties $p_1, \ldots, p_k$ (e.g., number of columns and rows of a table), a UI element e yields different front-end performance characteristics. We estimate the performance value of UI element e as

$$\phi_{type}(p_1, \ldots, p_k) \tag{2}$$

In order to derive an instance of such a prediction model for the SAP UI5 library and the three major browsers, we developed a systematic process. The process is implemented in a set of automatically executable experiments. Having this set of automatically executable experiments has the benefits that (i) the manual effort to create a model instance is limited to a minimum (i) the model instance can be easily updated for new browser or UI library versions and (iii) the procedure can be reused to derive model instances in similar setups.

In the following, we give a detailed description of the process and demonstrate how we implemented this process.

Deriving the Screen Offset (ϵ_S): As a first step, we determine the CPU time consumed by the browser to process the basic screen layout in which we place the different UI element types for our experiments. Therefore, we define and run an experiment that measures an empty screen. As a result we get the ϵ_S for the three browsers: $\epsilon_{S_{CH}} = 300ms \quad | \quad \epsilon_{S_{FF}} = 420ms \quad | \quad \epsilon_{S_{IE}} = 290ms$

Analyzing UI Element Types: To deal with the vast amount of UI element types, we group them in simple types and complex types. As simple types we define those UI elements with a performance cost per instance of less than 5 ms. For these elements, we do not conduct a detailed evaluation of the properties. Instead, we just determine a general fixed performance value for each instance of a UI element type that is considered as simple. Examples for such simple UI element types in our study are buttons, text views, or labels and the performance value per instance that we assigned to this group is 2 ms. Hence, we estimate the performance value of a simple UI element with the function: $\phi_{simple}() = 2 \times \#simpleUIelements$. That value is approximated based on a small set of experiments. We use the same value for all three browsers as we did not observe a significant difference between the browsers for processing these kind of UI elements.

The complex UI element types are those that significantly contribute to the browser CPU time when added to a screen. Examples for such UI element types are tables, service calls and row repeaters. For these UI element types, we run two experiment series. In the first series of experiments, we determine which property of the UI element significantly influences the browser CPU time. And in the second series, we derive $\phi_{type}(p_1, \ldots, p_k)$ for those properties that are considered as performance-relevant. To determine the performance-relevant properties in the first series of experiments, we apply standard statistical designs such as *One-at-a-Time* or *Plackett-Burman* designs in combination with statistical correlation analysis methods. The selection of the actual design is based on the size of the parameter space spanned by the number of UI element properties and their potential values. As an example, for the table UI element type, the number of columns ($\#cols$) and rows ($\#rows$) have been identified as the only performance-relevant properties. With the second series of experiments we aim at quantifying the relationship between the different manifestations of a table (combinations of $\#cols$ and $\#rows$), and the browser CPU time ($CPUtime$). If we set the possible value ranges for the two variables in this example to $\#rows : 1..20$ and $\#cols : 1..20$, we run into the curse of dimensionality and even for this small example it would take $20 * 20 = 400$ experiments to measure the complete space. In our setup, this would mean that we would have to measure one week to determine only this relationship for the three browsers. To reduce the number of required experiments we apply advanced statistical inference approaches [10] that automatically determine which experiments to execute in order to get an accurate prediction function. As a result we get a multi-dimensional regression function such as the one outlined below (derived for Firefox).

$$CPUtime_{FF} = 584 + 30 * max(0; \#cols - 5) - 33 * max(0; 5 - \#cols) \\ + 25 * max(0; \#rows - 5) - 29 * max(0; 5 - \#rows) \quad (3)$$

Deriving this function for a single browser takes approx. 2-8 hours depending on the complexity of the underlying function. Limiting the number of experiments by manually restricting the potential space is also a possible approach that can be sufficient for simple functions but implies a higher risk that important combinations have not been measured [10].

In order to predict any combination of table manifestations on a screen, we proceed as follows: We subtract the offset of a blank screen ($\epsilon_{S_{FF}} = 420$) from the function outlined in Equation 4 in order to remove this offset from the estimation. We use the resulting function as the implementation of $\phi_{type}(p_1, \ldots, p_k)$. Thus, we estimate the Firefox browser CPU time for the UI element type *table* with the following function:

$$\phi_{table}(\#cols, \#rows)_{FF} = 164{+}30{*}max(0; \#cols_i{-}5){-}33{*}max(0; 5{-}\#cols_i) \\ + 25 * max(0; \#rows_i - 5) - 29 * max(0; 5 - \#rows_i) \quad (4)$$

Construct Prediction Model Instance. Once all components of the prediction model instance are determined they can be composed according to Equation 1 in order to predict the browser CPU time for a screen S. For example:

$$\phi(S)_{FF} = \epsilon_S + \phi_{simple}()_{FF} \\ + \phi_{table}(\#cols, \#rows)_{FF} + \phi_{jsoncall}(datasize)_{FF} + ... \quad (5)$$

For our study, we derived a prediction model that contains most of the simple and complex UI element types used in enterprise applications built with SAP UI5.

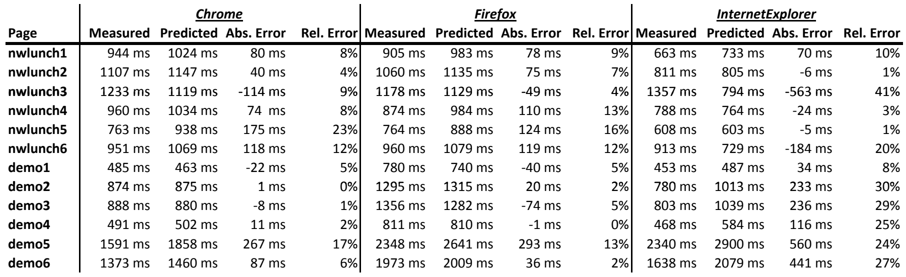
Validate Prediction Model. The constructed prediction model instance is an abstraction of the real behaviour that is based on assumptions, heuristics and statistical inference. Hence, it has to be validated that the estimated performance values sufficiently reflect the behaviour of the real screens. In the following section, we validate our prediction model as well as the prediction model instances that we derived for the SAP UI5 library and the three browsers.

3 Validation and Discussion

The goal of our validation is to judge prediction accuracy and thus the utility of our heuristics and the practicability of our approach. Therefore, we compare our predictions with actual performance measurements. We selected twelve real-world pages built with the SAP UI5 library. Six pages are taken from demo applications. These pages cover a broad spectrum of different manifestations of the two most important control types in business applications. The other six pages are taken from a real application called Networking Lunch which is a social enterprise application where people can search for other people interested in the same topic and setup a joint lunch meeting.

3.1 Results

In Figure 1(a) we show the results for the twelve validation screens. The average prediction error across all screens and browsers is 11%. For 72% of the predictions, the relative prediction error is less than 15% and there is only one real outlier with an error higher than 30%. The predictions for Chrome (8% average error) and Firefox (7% average error) have been better than those for Internet Explorer (18% average error). Between the two applications, we could not observe a general difference with respect to prediction accuracy (average error for both applications is 11%).



	Chrome				***Firefox***				***InternetExplorer***			
Page	**Measured**	**Predicted**	**Abs. Error**	**Rel. Error**	**Measured**	**Predicted**	**Abs. Error**	**Rel. Error**	**Measured**	**Predicted**	**Abs. Error**	**Rel. Error**
nwlunch1	944 ms	1024 ms	80 ms	8%	905 ms	983 ms	78 ms	9%	663 ms	733 ms	70 ms	10%
nwlunch2	1107 ms	1147 ms	40 ms	4%	1060 ms	1135 ms	75 ms	7%	811 ms	805 ms	-6 ms	1%
nwlunch3	1233 ms	1119 ms	-114 ms	9%	1178 ms	1129 ms	-49 ms	4%	1357 ms	794 ms	-563 ms	41%
nwlunch4	960 ms	1034 ms	74 ms	8%	874 ms	984 ms	110 ms	13%	788 ms	764 ms	-24 ms	3%
nwlunch5	763 ms	938 ms	175 ms	23%	764 ms	888 ms	124 ms	16%	608 ms	603 ms	-5 ms	1%
nwlunch6	951 ms	1069 ms	118 ms	12%	960 ms	1079 ms	119 ms	12%	913 ms	729 ms	-184 ms	20%
demo1	485 ms	463 ms	-22 ms	5%	780 ms	740 ms	-40 ms	5%	453 ms	487 ms	34 ms	8%
demo2	874 ms	875 ms	1 ms	0%	1295 ms	1315 ms	20 ms	2%	780 ms	1013 ms	233 ms	30%
demo3	888 ms	880 ms	-8 ms	1%	1356 ms	1282 ms	-74 ms	5%	803 ms	1039 ms	236 ms	29%
demo4	491 ms	502 ms	11 ms	2%	811 ms	810 ms	-1 ms	0%	468 ms	584 ms	116 ms	25%
demo5	1591 ms	1858 ms	267 ms	17%	2348 ms	2641 ms	293 ms	13%	2340 ms	2900 ms	560 ms	24%
demo6	1373 ms	1460 ms	87 ms	6%	1973 ms	2009 ms	36 ms	2%	1638 ms	2079 ms	441 ms	27%

(a) Validation results.

The highest error is 41% for screen 3 of the Networking Lunch application in Internet Explorer. Although this screen has nearly the same characteristics as screen 4 (for which the error is only 3%), we underestimate the browser CPU time by 563ms. We could not yet figure out the root cause of this difference. It is interesting that we did not observe such a large deviation for this screen in the other browsers. The weaknesses in the predictions for the Internet Explorer is also visible for the demo application screens. However, for these screens we overestimate the browser CPU time. This overestimation is most likely caused by the estimation function for the odata service calls as these contribute largely to the estimated overall CPU time for the screens. Hence, we have to run further experiments to improve the regression function for odata calls in Internet Explorer.

In general, the results demonstrate that our assumptions are valid and that the introduced abstractions and heuristics do not significantly compromise the prediction accuracy.

3.2 Threats to Validity

The results presented in Section 3.1 demonstrate that our approach can accurately predict the front-end performance of enterprise web applications. However, it is important to note the threats to validity of our approach in order to understand its applicability in practice. The main restrictions we currently see are:

Small Validation Set. The screens evaluated in Section 3.1 are only part of two web applications. However, both are very different in type and front-end performance. One represents a typical enterprise web application for processing data, the other a social enterprise application. Even though the predictions complied to measurement for the case studies presented in Section 3.1, a broader set of validation scenarios is required, to ensure its general applicability.

Single Library. In our industrial case study at SAP, developers of web applications usually use only the SAP UI5 library to build a web application front-end. The library encapsulates other common JavaScript libraries. In other development environments, especially non-enterprise web application development, it is often the case that multiple libraries are combined to develop the front-end code. Moreover, additional style definitions can affect front-end performance in standard web sites [7] which could have been neglected for the enterprise web applications developed with the SAP UI5 library and the corresponding predefined styles. However, the experiment-based evaluation process presented in this paper, as well as the experiment automation tooling [6] can be used to efficiently derive prediction models for other libraries.

Custom JavaScript Code. Our prediction focuses on the influence of UI elements and service calls on front-end performance. This is a reasonable assumption for typical enterprise applications. However, developers often add custom JavaScript code to process data, to create new controls or to change configuration. This custom code will add to the browser CPU time and thus to front-end performance. While such custom code played only a minor role in the case studies presented in Section 3.1, it may have huge effects on front-end performance in other cases. However, our goal is to give early feedback on front-end performance, thus, we cannot consider such effects in our prediction.

Effort. The efforts necessary to implement the approach, i.e., to create and maintain the prediction models are a crucial factor for the practical applicability of the approach. These tasks should be performed by a small team of UI library and performance experts. Our experiment automation tooling [6] supports and guides the team in the course of the prediction model construction process which limits the efforts to a minimum. The decision if a software vendor wants to invest the efforts in constructing a prediction model for his libraries depends on the number of designers and developers that can benefit from the feedback provided by the models.

4 Conclusions

In this paper, we presented an approach that shifts performance evaluation efforts to a small team of UI library and performance experts. We introduced a methodology that enables the expert team to efficiently derive prediction models for UI libraries used by the development groups. The bulk of developers and designers in an organisation benefit from the model by getting early performance feedback that is, for example, integrated in design tools. The feedback allows designers and developers to evaluate the front-end performance of web applications prior to implementation. They can assess different design alternatives and chose the one with the best trade off between performance and user experience (which does not necessarily have to be a trade off).

We applied the approach at SAP by creating a prediction model for the SAP UI5 library and validated the accuracy of the model by comparing the predictions to measurements of real web application screens. We integrated the derived prediction model in an easy-to-use tool that is used by SAP UI5 developers to easily evaluate the performance of their screen designs and by performance trainers to raise the performance-awareness in developer training sessions.

In our future work, we are going to derive prediction models for web application screens that run on mobile devices. Moreover, we plan to investigate other popular JavaScript libraries.

References

1. Sap ui5: Ui development toolkit for html5, `http://scn.sap.com/community/developer-center/front-end` (last visited March 2013)
2. Webpagetest, `http://www.webpagetest.org/` (last visited March 2013)
3. Yslow, `http://developer.yahoo.com/yslow/` (last visited March 2013)
4. Bixby, J.: Web performance today, `http://www.webperformancetoday.com/2010/07/01/the-best-graphs-of-velocity/` (last visited March 2013)
5. Brad Frost. Performance as design (2013), `http://bradfrostweb.com/blog/post/performance-as-design/` (last visited March 2013)
6. sopeco.org. Software performance cockpit, sopeco (2013), `http://sopeco.org` (last visited March 2013)
7. Souders, S.: High Performance Web Sites: 14 Steps to Faster-Loading Web Sites. O'Reilly (2007)
8. Souders, S.: Even Faster Web Sites: Performance Best Practices for Web Developers. O'Reilly (2009)
9. Westermann, D., Happe, J., Hauck, M., Heupel, C.: The Performance Cockpit Approach: A Framework for Systematic Performance Evaluations. In: 36th EUROMICRO SEAA Conf., pp. 31–38. IEEE CS (2010)
10. Westermann, D., Happe, J., Krebs, R., Farahbod, R.: Automated inference of goal-oriented performance prediction functions. In: 27th IEEE/ACM Int. Conf. on Automated Software Engineering, ASE 2012, pp. 190–199. ACM, New York (2012)