Layout as a Service (LaaS): A Service Platform for Self-Optimizing Web Layouts

Markku Laine(✉), Ai Nakajima, Niraj Dayama, and Antti Oulasvirta

Aalto University, Helsinki, Finland
{markku.laine,ai.nakajima,niraj.dayama,antti.oulasvirta}@aalto.fi

Abstract. To personalize a web page, case-specific rules or templates must be specified that define the visuospatial layout of elements as well as device-specific adaptation rules for an individual. This approach scales poorly. We present *LaaS*, a service platform for self-optimizing web layouts to improve their usability at individual, group, and population levels. No hand-coded rules or templates are needed, as *LaaS* uses combinatorial optimization to generate web layouts for stated design objectives. This allows personalization to be controlled via intuitive objectives that affect the full web layout. We present an extensible architecture and solutions for (1) layout generation using integer programming, (2) data abstractions to mediate between browsers and layout generators, and (3) page restructuring. Moreover, we show how *LaaS* can be easily deployed as part of existing web pages. Results demonstrate that our approach can produce usable personalized web layouts in diverse scenarios.

Keywords: Self-adaptive web interfaces · Web-based interaction · Web personalization · Web layouts · Web service architectures

1 Introduction

Designing a *web layout* is laborious and challenging: Given elements can be laid out in many different ways, yet content and functionality need to appear interesting and presented in an appealing and accessible way. However, the "one design fits all" approach is inherently suboptimal from the usability point-of-view. For any individual user, a page designed for a larger population will always compromise the particular interests, expectations, and capabilities. Previous work suggests that layout personalization could bring significant per-user improvements in usability and experience and could relieve designers and developers from manual work. However, while there are computational methods and architectures for web personalization, no viable solution has been proposed how to adapt *full web layouts* to individuals without manually precoded rules or templates.

This paper contributes a novel service architecture design and computations for *objective-level web layout personalization.* That is, layouts are adapted by reference to desired effects on end-users: "I want this page to be improved for [design objective]". In objective-level control, the full layout of a page, including elements and their positions and sizes will be *generated* given the user's data. No rules or templates are needed. This extends web personalization from content-level personalization to consider full layouts. In this paper, we explore selection time and visual saliency as two common objectives in layout optimization [5].

In the rest of this paper, we present *LaaS*, an architecture and computations for *self-optimization of web layouts*. Our cloud-based service architecture allows offloading computation effort to the cloud. The computational tasks of selecting and layouting elements on a page are NP-hard problems and not solvable in a browser for realistic problem instances. We make two further technical contributions. First, we extend combinatorial optimization based approaches [5] to permit adapting layouts to a wide variety of users, pages, and devices with no predefined rules or templates. Changes to existing codebase are minimal (1 line per page). This is practically out of reach of rule-based approaches, which scale up poorly. Second, we present a data abstraction for the visuospatial design of the page, which allows the optimizer to be agnostic of the underlying web technologies (here: HTML, CSS, and DOM). The architecture is easy to deploy and fully controllable by the service owners, who may want different outcomes on different pages and must trust that the outcomes produced are of high quality. The process is practically invisible to end-users. Moreover, thanks to the separation of a design task from the generator, proprietary machine learning methods can be incorporated to *LaaS*. We demonstrate the system with a clickstream-based generation of personalized web news portals, a realistic and challenging case with needs for variability in web layouts.

2 Related Work

LaaS focuses on the *grid layout*, a common design principle for organizing graphical UIs, available in many design tools, UI toolkits, and layout managers. Visual flow and motor selection are two important goals in their design [6, 9]. After determining the visuospatial organization of a web layout, it must be implemented, typically using standard web technologies. Adaptation rules, such as for Responsive Web Design, must be added, which is often done by hand. Designers spend considerable time with repetitive tasks related to UI layouts. So far, no architecture has been proposed for web layout adaptation that is able to adapt the full layout for an individual without predefined rules or templates.

Web personalization has been a topic of interested since the 1990s. A number of machine learning and data mining methods have been proposed for modeling page contents and web usage patterns in order to drive the *recommendation* and *selection* of contents. Research on techniques for *presenting content* has focused on the *ordering*, *emphasis*, and *scaling* of contents [2], as well on message framing and use of colors. However, no method has been proposed that could handle any and all of these on the web.

Service architectures for adaptive web layouts have been either rule-based or template-based. For example, AERO is a template-based framework for web layout synthesis [10]. The approach is based on a suite of templates specified in HTML and CSS, of which one is selected in accordance with a customizable scoring function. An issue with both rule and template-based approaches is that they rely on decisions at design-time. In general, existing approaches are not well-suited for handling continuous and unanticipated changes.

Combinatorial optimization has been studied as a method for the GUI design [5]. Early research mostly used rules and design heuristics to generate layouts that adhere to known design guidelines and more recently data-driven approaches. *Model-based approaches*, on the other hand, use *white-box* (first principles) models that provide a theory-driven and transparent approach to layout generation. In the generative process, layout quality is measured against some model or design heuristic. When heuristics are used as objective functions, however, optimization systems scale up poorly due to the large number of rules. *Predictive models of user performance and experience* have been proposed to address this issue [5]. We found only one application of prediction-based methods for web layouts. In Familiariser [9], a visual search model was fit to a user's site visitation history and used in a browser-side optimizer to re-layout a page to make elements quicker to find. In *ability-based optimization*, UI designs are generated by taking into account motor or cognitive impairments of an individual, which are represented as parameters in predictive models [8]. For an overview of design objectives that can be modeled using predictive models, see [5].

The layout problem is recognized in operations research as an NP-hard problem, and our design problem is an instance of it. In combinatorial geometry, grid layout has been studied in the context of 2D bin packing, rectangular packing, and the guillotine cuts problem [5]. Generation of multiple, varied, near-optimal solutions has been discussed. An elementary version of the grid design problem has been previously proposed [4]. However, they merely attempted to find the most densely packed solution by squeezing elements closely together.

3 LaaS: Architecture and Computations

This section presents Layout as a Service (LaaS), a service platform architecture and computations that enable *objective-level web layout personalization.* The distributed system architecture consists of a set of loosely-coupled client-side and server-side components that communicate with each other over HTTPS using a REST API, as depicted in Fig. 1. The client-side components are dynamically loaded to the end-user's browser during the initial phase of a page load; Layout Parser and Layout Adapter are executed *before* the web page is displayed to the user, whereas Event Logger collects user behavior data *while* the user interacts with it. The server-side components are run *on demand* (e.g., daily, weekly) in the cloud; Design Task Generator generates a design task specification for the latest version of each layout, whereas Layout Generator optimizes them accordingly.

In the architecture, expensive computations, especially layout generation, are executed on the server side. The adaptation of layout elements, on the other hand, takes place in the browser to permit adaptation of both server-side and client-side rendered pages. The architecture is also designed in a modular way

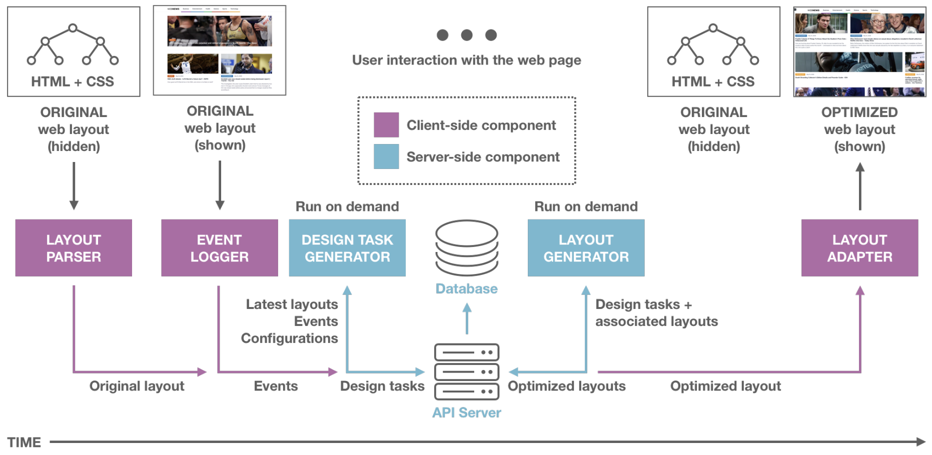

Fig. 1. *LaaS* architecture, including core components and interactions between them.

such that it is possible to plug in machine learning components that help in user modeling and/or design task generation.

The following subsections describe the above-mentioned *LaaS* core components and their function in greater detail.

3.1 Layout Parser

In order to reproduce a web layout, Layout Parser *automatically* (1) assigns a unique identifier for each element, (2) parses the web page structure and styles, (3) detects and labels key elements, (4) precomputes permissible shapes for key elements, and finally (5) creates a user interface technology independent representation (JSON) of the *original layout*. This parsing process needs to be done *only once* per selected layout optimization level (individual, group, population).

3.2 Event Logger

Event Logger is responsible of capturing user interactions on a web page and sending the data to API Server. The collected data includes, among others: event type, layout identifier, client identifier, page URL, and other event-specific data (e.g., clicked element identifier, link target, and timestamp). Our current implementation records clicks and visits on web pages. Support for new event types and metrics (e.g., document scroll and time spent on the page) can be easily added by extending *LaaS* event logging capabilities.

3.3 Design Task Generator

To support adaptation on demand for any given target, as well as to support controllability, *LaaS* separates the design task from the generator. Design Task

Generator (DTG) creates a specification of *design task*, which serves as a communication vehicle between the designer (here: DTG) and Layout Generator. It allows the designer to specify (1) various design objectives and constraints on a web layout to be generated and (2) compute per-element importance values based on collected user behavior data. We currently use click frequencies to obtain element importance values. However, the architecture is flexible enough to support other, more advanced computational methods, such as machine learning.

3.4 Layout Generator

We formulate a mixed integer linear programming model (MIP) in order to reorganize web pages as grid layouts. We chose MIP to achieve a balance between computational performance and solution quality. Our MIP formulation (1) ensures well-formed layouts that are rectangular and well-aligned and (2) optimizes them for stated design objectives, in our case selection time and visual saliency. Linearity of our model ensures better performance and enables use of suitable MIP callbacks [3]. Further, our MIP model has size depending solely on the number of elements involved, i.e., the size of our MIP model is independent of the canvas size. The MIP model works in three phases:

Phase 1: Layout Sanctity. We ensure a non-overlapping, non-overflowing grid where elements are placed within permissible size limits and in permissible locations. We use continuous decision variables to represent the location of all four edges of every individual element. Continuous decision variables avoid pixel-level discretization, which is important for the performance of the solver. To prevent overlapping elements, we use an approach introduced by Hart and Yi-Hsin [4]. Hence, the core MIP formulation developed in this phase provides non-overflowing, non-overlapping solutions with element sizes within limits.

Phase 2: Alignment. This phase computes and restricts the number of independent grid lines required to represent the selected candidate solution. It ensures that the resulting layouts are well-aligned and aesthetically acceptable. To represent overall alignment objectively, we define notional Cartesian grid lines on all pixels of the canvas. If any two (or more) elements have any of their edges aligned with each other, those elements share the single grid line for those edges. So, the *total number of grid lines actually utilised* in any feasible solution is a direct indicator of the overall alignment within that solution. The objective of minimizing the total number of grid lines achieves the design intent of well-aligned solutions.

Phase 3: Functional Layout. The functional placement formulation determines the placement and sizing of relevant elements to ensure that the layout has high usability. We have picked and implemented the following two design objectives, but any other objectives that can be efficiently represented in the MIP could be included (for previous work, see [5]).

– *Selection Time.* We use Fitts' law to compute the time required to reach a specific element on the screen. Fitts' law is widely used in model-based optimization as an objective function [1]. Selection time (ST) is a function of target distance (D) and size (W): $ST = a + b\log_2(D/W + 1)$. In our case, we assume that the user starts scanning the screen from the top-left corner. So, we use a linear function of the distance from the screen corner as a substitute approximation. If the design task instance prioritizes selection time, the optimizer attempts to minimize the predicted time required for important elements. The most obvious effect is that very important elements may become larger and placed closer to the top-left corner of the web page.

– *Visual Saliency.* Saliency refers to how attention-grabbing an element is given the rest of the page [7]. In our case, we compute saliency as the relative size of the element. While area would be a proxy for the saliency, this is further complicated by the requirement that the permissible areas of the element must be picked from within a fixed number of permissible shapes only. So, we pick the most salient size from the permissible shape set, if provided, and use that size for laying out elements. Similarly, color and other qualities could be implemented.

3.5 Layout Adapter

Restructuring hierarchical web pages is particularly challenging because even the smallest change to the DOM structure can break the web page's visual appearance, functionality, or both. Thus, Layout Adapter is designed so that it can reposition and resize web layout elements without changing the original tree structure. Once an *optimized layout* for the web page becomes available in the cloud, it can be fetched and applied *before* the web page is shown to the user.

3.6 Deployment

Enabling *LaaS* on a website can be done in just two steps. First, the service owner registers a website to obtain an embeddable `<script>` tag with a tracking identifier. Second, the obtained script is added on those web pages of the website, whose usability needs to be improved. Injecting the `<script>` tag into the web pages can be done either *manually* with minimal source code changes or *automatically* via our proxy server installation.

We also offer *LaaS* Control Panel for service owners to manage various *LaaS* related settings on their website, such as service status, design objectives, target elements, and data collection events.

4 Results

Fig. 2 shows example outputs for WebNews, a custom news aggregator website hosted on our server. The original design shown in Fig. 2(a) has one template-based multi-column grid layout, to which in the initial design news articles from six different categories are allocated. Normally, adapting the *full layout* of a page like this would require predefined templates or a (very) large number of rules.

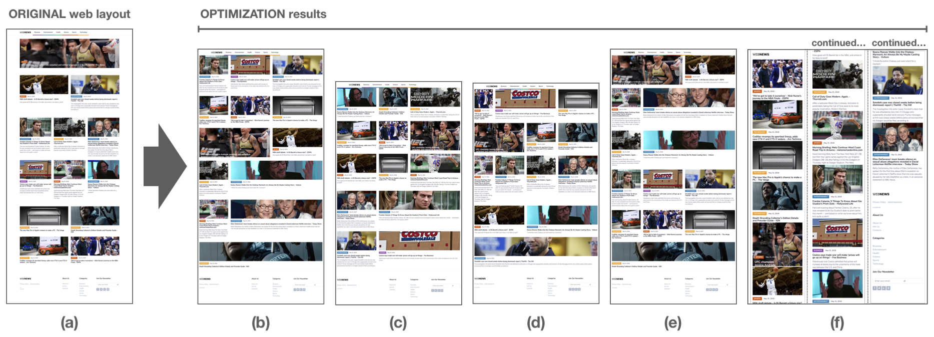

Fig. 2. Results for a web news page: (a) Original web layout with multiple content cards; (b) Optimized to improve selection time of a single card (Costco); (c) Optimized for visual saliency of the same card; (d) Optimized for both selection time and saliency of the same card; (e) Optimized for both and with more complex interest distribution (all sports and business cards); (f) Optimized for mobile device screen width.

We divide the example results into two classes: demonstrator results done with simpler scenarios and two more realistic cases. Fig. 2(b-d) shows adaptation results for different combinations of the two design objectives, using a single card as the illustrative example. The produced layouts are well-formed and the element-of-interest (Costco) behaves as desired: it is moved to a closer position for selection in (b), more visually salient in (c), and both combined in (d). Fig. 2(e) shows a more complex example, where a bimodal interest distribution (sports and business categories) is accounted for. Fig. 2(f) shows the page adapted for a mobile screen. All layouts are properly formed: there are no holes and no overlapping elements. The layouts adhere to proper, well-aligned grids. This would be very laborious to achieve with a rule or template-based approach.

4.1 Discussion

There are two predominant methods for rendering layouts on the web: server-side rendering and client-side rendering. While client-side rendering has gained popularity over the past few years, a myriad of web applications (incl. WebNews), frameworks (e.g., WordPress), and libraries use or support server-side rendering, including React and Vue.js. The *LaaS* architecture is designed to work with both server-side and client-side rendered layouts; however, our research efforts have almost exclusively focused on the former up until this point.

Quality of Experience (QoE) [11] describes, from a holistic perspective, how well a service such as a website satisfies its users' expectations. While *LaaS* can produce usable personalized web layouts, we acknowledge that its use may unfavorably impact other QoE factors, such as page load times and aesthetics. However, according to our informal testing these effects are small and can be mitigated with the use of known techniques, such as caching and image re-cropping.

5 Conclusion

We presented first steps toward *objective-level* control of personalization, including an architecture and involved computations. We believe that at least within the scope of grid-based web layouts, this goal is within reach. The examples we showed would be very laborious to achieve with a rule-based approach. This result warrants more research on this approach. *LaaS* provides an extensible architecture concept for future work to build on. It supports, by design, easy deployment on many present-day pages and integration with widely used machine learning methods for user modeling and recommendations. The core MIP solutions, on the other hand, can be extended with other design objectives.

References

1. Bailly, G., Oulasvirta, A., Kötzing, T., Hoppe, S.: MenuOptimizer: Interactive optimization of menu systems. In: Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology. p. 331–342. UIST '13, ACM (2013). https://doi.org/10.1145/2501988.2502024
2. Bunt, A., Carenini, G., Conati, C.: Adaptive Content Presentation for the Web, pp. 409–432. Springer (2007). https://doi.org/10.1007/978-3-540-72079-9_13
3. Castillo, I., Westerlund, J., Emet, S., Westerlund, T.: Optimization of block layout design problems with unequal areas: A comparison of MILP and MINLP optimization methods. Computers & Chemical Engineering **30**(1), 54–69 (2005). https://doi.org/10.1016/j.compchemeng.2005.07.012
4. Hart, S.M., Yi-Hsin, L.: The application of integer linear programming to the implementation of a graphical user interface: a new rectangular packing problem. Applied Mathematical Modelling **19**(4), 244–254 (1995)
5. Oulasvirta, A., Dayama, N.R., Shiripour, M., John, M., Karrenbauer, A.: Combinatorial optimization of graphical user interface designs. Proceedings of the IEEE **108**(3), 434–464 (2020). https://doi.org/10.1109/JPROC.2020.2969687
6. Pang, X., Cao, Y., Lau, R.W.H., Chan, A.B.: Directing user attention via visual flow on web designs. ACM Trans. Graph. **35**(6) (2016). https://doi.org/10.1145/2980179.2982422
7. Rosenholtz, R., Li, Y., Mansfield, J., Jin, Z.: Feature congestion: A measure of display clutter. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. p. 761–770. CHI '05, ACM (2005). https://doi.org/10.1145/1054972.1055078
8. Sarcar, S., Jokinen, J.P.P., Oulasvirta, A., Wang, Z., Silpasuwanchai, C., Ren, X.: Ability-based optimization of touchscreen interactions. IEEE Pervasive Computing **17**(1), 15–26 (2018). https://doi.org/10.1109/MPRV.2018.011591058
9. Todi, K., Jokinen, J., Luyten, K., Oulasvirta, A.: Individualising graphical layouts with predictive visual search models. ACM Trans. Interact. Intell. Syst. **10**(1) (2019). https://doi.org/10.1145/3241381
10. Vernica, R., Venkata, N.D.: AERO: An extensible framework for adaptive web layout synthesis. In: Proceedings of the 2015 ACM Symposium on Document Engineering. p. 187–190. DocEng '15, ACM (2015). https://doi.org/10.1145/2682571.2797084
11. Wechsung, I., De Moor, K.: Quality of Experience Versus User Experience, pp. 35–54. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-02681-7_3