

Designing a High-Performance Mobile Cloud Web Browser

Ryan H. Choi^{*}
Software R&D Center
Samsung Electronics
ryan.choi@samsung.com

Youngil Choi
Software R&D Center
Samsung Electronics
duddlf.choi@samsung.com

ABSTRACT

A mobile cloud web browser is a web browser that enables mobile devices with constrained resources to support complex web pages by performing most of resource demanding operations on a cloud web server. In this paper, we present a design of a mobile web cloud browser with efficient data structure.

Categories and Subject Descriptors

D.2.11 [Software Engineering]: Software Architectures;
J.0 [Computer Applications]: General

Keywords

Mobile cloud web browser; DOM; Succinct

1. INTRODUCTION

The widespread of Internet-connected mobile devices provide users a web experience anywhere at any time. While mobile devices are rapidly advancing, mobile-specific constraints such as CPU, memory, battery and network bandwidth impose the performance limits on mobile devices. To support these devices, web servers often provide mobile-specific web pages, i.e., web servers deliver rich HTML/CSS/JS pages to desktop clients, while minimum, often text-based, web pages are delivered to mobile devices. This is an attempt to reduce the size and complexity of web pages—a simplified mobile web page results in a smaller and simpler DOM¹ that is less demanding to mobile devices.

In this paper, we propose a design of a cloud web browser that enables mobile devices with constrained CPU and memory resources to support complex web pages. Unlike existing works [2, 3, 1], where the goal is to improve the performance by parallelism, our cloud web browser is specific to mobile devices. The idea of our cloud web browser is to perform

^{*}Corresponding author

¹<http://www.w3.org/TR/dom/>

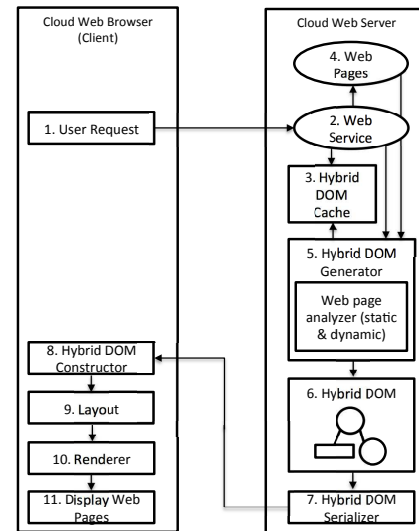


Figure 1: Cloud web browser and server interaction

complex operations such as DOM construction, CSS selector matching, and layout on a cloud web server, and when a web page is requested, a memory-efficient, pre-built DOM is delivered instead of an HTML file. Then, when a client receives the DOM, the client restores it and performs simple rendering operations to display the web page.

The challenge is to reduce the DOM construction and maintenance costs in terms of CPU and memory usage, respectively, without sacrificing the performance and features of HTML/CSS/JS. In addition, HTML and JS semantics must be respected. DOM is the main data structure that represents a web page in a web browser. By traversing DOM, web contents, relationships between elements, styles, and layout positions are calculated. Furthermore, DOM is the data structure that JS accesses to modify the contents of a web page. DOM is mandatory, but it is costly to build and maintain. This is because DOM construction is CPU intensive [3], and also the size of DOM is typically 5–10 times bigger than the corresponding HTML page. Moreover, CSS selector matching and JS are also CPU intensive—DOM construction/CSS selector matching/JS take more than 50% of the total web browser processing time [1]. Furthermore, reducing the network bandwidth when sending web pages is another challenge.

2. CLOUD WEB BROWSER

Figure 1 shows an overview of how a client web browser and cloud web server interact. When a web page is re-

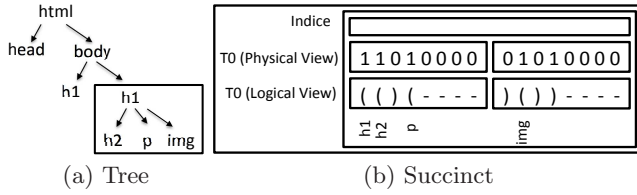


Figure 2: Hybrid-DOM

quested, our cloud web server builds a smaller, partially/fully-built, and memory-efficient DOM called hybrid-DOM that represents the requested HTML page. Then, the cloud web server performs CSS selector matching to decorate the hybrid-DOM with CSS values, and runs JS on behalf of clients before it is sent to the client. Upon receiving the hybrid-DOM, the client simply restores it, and executes client-specific JS and CSS. The restoration of hybrid-DOM is much less resource-demanding than constructing DOM and executing JS and CSS sequentially. Finally, the hybrid-DOM is sent to the client's renderer, and displayed on the web browser. It is important to emphasize that hybrid-DOM is 100% compatible to the traditional DOM API, therefore other existing components such as layout and rendering engines do not need any modification.

3. HYBRID-DOM

We now provide an overview of hybrid-DOM. Figure 2 shows an example of hybrid-DOM. It shares the same purposes and functionalities as the traditional DOM except it is memory-optimized. A hybrid-DOM consists of two parts: a tree part and succinct part. A tree part represents traditional DOM nodes, while a succinct part represents the nodes in balanced parentheses encoding (BPE) scheme [5]. In Figure 2(a), the subtree enclosed in a rectangle denotes a succinct part, and its details are shown in Figure 2(b). To decide which part of DOM is represented by which representation, our cloud web server analyzes node update frequency—the nodes frequently updated are represented by tree parts, and the nodes that are hardly updated are represented by succinct parts.

In succinct part, opening and closing parentheses are used to represent a start and end tag of a node, respectively. This sequence of parentheses is encoded in a sequence of bits, where 1 and 0 are used for each opening and closing parenthesis, respectively. An ancestor-descendant relationship is represented by nested enclosing parentheses. A parent-child relationship is represented by direct enclosing parentheses. Tree traversal and node navigation are performed by adding and subtracting a sequence of 1s found in $T0$ blocks. The value obtained at the end of adding/subtracting 1 indicates the difference between opening and closing parentheses, which in turn represents the level difference in the HTML tree. By calculating the difference, ancestor (or descendant) of a node can be found. To improve the node navigation performance, indices are built on top of $T0$. Indices summarize the number of bits in a block to efficiently count bits. To support updates, empty spaces are reserved in each $T0$ block, and these are marked as “-”.

The advantage of hybrid-DOM is that, it requires much less memory footprint while supporting all navigational operations required by CSS/JS engines. It is also network optimized in a way that it is easily transferable to a client, as succinct parts are already encoded in binary and seri-

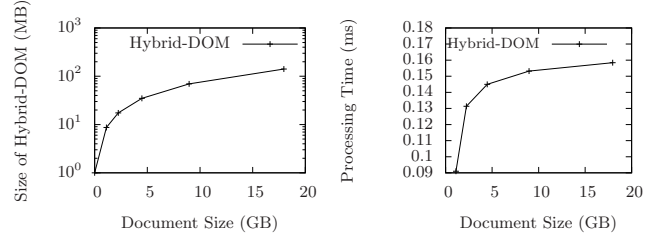


Figure 3: Size of Hybrid-DOM

Figure 4: Processing Time

alized. A client restores hybrid-DOM by simply reading the binary stream and directly copying it to its memory. Note that, this is different from traditional binary compression/decompression schemes in a way that, the succinct part does not require to decompress data in order to read it.

4. PRELIMINARY RESULTS

We now present experiment results. To test hybrid-DOM, XMark [4] was used to generate 5 documents of size 1.12, 2.24, 4.48, 8.97 and 17.96 GB, and converted to hybrid-DOM. 10% of random nodes are kept as trees. We then selected a set of 50,000 random nodes and ran *getParent()*, as this operation is the most demanding operation.

Figure 3 shows the size of hybrid-DOM. As document size increases, the size of hybrid-DOM linearly increases, since the size of $T0$ is linearly proportional to the number of nodes in a document. The experiment show that hybrid-DOM is several orders of magnitude smaller than the DOM in text format. Figure 4 shows the average processing time for the documents when running *getParent()*. The sublinear pattern is due to efficient use of indices while counting bits.

5. CONCLUSION

In this paper, we presented our ongoing work, a cloud web browser with memory-efficient DOM that enables mobile clients with constrained resources to display complex web pages. Preliminary experiments show that complex and large documents can be represented by hybrid-DOM and navigated efficiently. At present, we are working on optimizing hybrid-DOM indices with better dynamic web page support.

As future works, we plan to investigate parallelism on hybrid-DOM. We also plan to develop an efficient web page analyzer. Finally, we plan to finish the development of our cloud web browser.

6. REFERENCES

- [1] C. Cascaval, S. Fowler, P. Montesinos-Ortego, W. Piekarski, M. Reshadi, B. Robotmili, M. Weber, and V. Bhavsar. Zoomm: a parallel web browser engine for multicore mobile devices. In *PPoPP*, pages 271–280. ACM, 2013.
- [2] C. G. Jones, R. Liu, L. Meyerovich, K. Asanovic, and R. Bodik. Parallelizing the web browser. In *HotPar*, pages 7–7. USENIX Association, 2009.
- [3] L. A. Meyerovich and R. Bodik. Fast and parallel webpage layout. In *WWW*, pages 711–720. ACM, 2010.
- [4] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. Xmark: A benchmark for xml data management. In *VLDB*, pages 974–985. Morgan Kaufmann, 2002.
- [5] R. K. Wong, F. Lam, and W. M. Shui. Querying and maintaining a compact xml storage. In *WWW*, pages 1073–1082. ACM, 2007.