A Parallel Approach to Mobile Web Browsing

Kiho Kim, Hoon-Mo Yang, Cheong-Ghil Kim, Shin-Dug Kim

Yonsei university, Department of Computer Science
School of Engineering, C532, Shinchon-dong, 134, Seoul 120-749, Republic of Korea
heavyarms3@gmail.com, {hmyang,sdkim}@yonsei.ac.kr,
cgkim@nsu.ac.kr

Abstract. This paper present a parallel approach about mobile web browsing, especially layout and paint parts. Web browser is one of the most frequently used applications in mobile devices and performance of web browser is an important factor affecting mobile device user experience. From our previous research, we found that layout and paint takes significant portion of web browser execution time and has similar execution characteristics. In this paper, we propose parallel render tree traversal algorithm for layout and paint parts in web browser: creating thread for sub-tree traversal processing. Moreover, to validate proposed Algorithm, we design a simple simulation implementing parallel tree traversal with web page render tree. The experiment results show that execution time is reduced average 28% in dual-core, 32% in quad-core compare to single-thread execution in paint simulation. In layout simulation, average 38% in dual-core, 57% in quad-core execution time is reduced.

Keywords: mobile web browser, parallel algorithm, multi-core, tree traversal.

1 Introduction

Web browser is a software application for retrieving, presenting, and traversing information resources on the World Wide Web. With the wide spread of networking infra structures, internet usage increases extremely and web browser becomes one of most frequently used applications. Many software companies are struggling to achieve more market share in web browser marketplace and web browser performance is one of the most important factors on this browser war. Recently, many smart-phone makers released new smart-phones or tablet-PCs, installed multi-core processor. These smart-phones are designed with dual-core chips now, but as time goes by, quad-core and many-core chips will be used in smart-phones same as in desktop systems[2]. Thus, with the trend towards the multi-core processors in mobile processors, we focus on parallelizing web browser for high performance web browsing. To utilize extra cores, we exploit parallelism using multi-thread library.

In this paper, we propose a parallel render tree traversal algorithm: thread creation for sub-tree traversal for mobile web browsing layout and paint functions. To validate our proposed algorithm, we design a simple simulation environment that has similar processing pattern to layout and paint functions and do experiment. According to our simulation results, the execution time is reduced by average 28% in dual-core, 32% in

J.Y. Zhang et al. (Eds.): MobiCASE 2011, LNICST 95, pp. 338–344, 2012.

quad-core for paint simulation. In layout simulation, execution time is reduced by average 38% in dual-core, 57% in quad-core. If we create more threads or sub-tree traversal, we can get better performance and the more cores are used, the better performance we can achieve. The rest of paper is organized as follows: Section 2 presents the background work that we've done before this research and other related works. Section 3 shows our parallel algorithm for mobile web browsing. Section 4 describes validation of our parallel algorithms. In Section 5, performance analysis of this parallel approach is provided. Finally, we conclude in Section 6.

2 Background

We choose target web browser engine: WebKit[3], and analyzed it. WebKit is an open source web browser engine and used for many desktops and mobile web browsers such as Apple Safari and mobile Safari browser[4], Google Chrome browser[5], and so on. Android web browser is also based on WebKit. The basic work flow of web browser is shown in Figure 1: load HTML, download resources, scan and parse documents, generate its corresponding document object model (DOM) tree and render tree[6, 7], and layout and paint render tree. After the initial page load, scripts respond to events generated by user input and server messages, typically modifying DOM, causing page re-layout and re-paint. We also did WebKit performance profiling over several hardware platforms and custom benchmarks for web pages. We categorized WebKit into several major functions: HTML parsing, CSS parsing, CSS style update, Javascript Processing, layout, and paint. Then we measure the execution time of each major function in real system using WebKit function and Linux system call modifying WebKit source code. Figure 2 shows our profiling results.

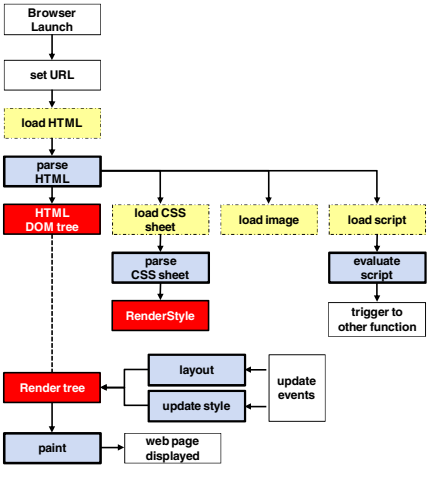

Fig. 1. Basic work flow of web browser

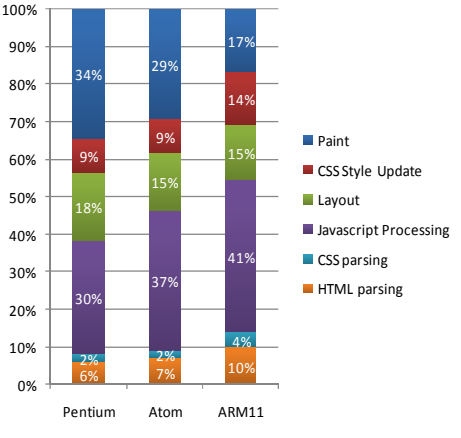

Fig. 2. Major function execution time ratio

There are few studies about web browser performance [8], parallel web browser [9], [10], and mobile browsers[11]. In [8], author also evaluates performance of web browser in various platforms. In [9], author introduces brief idea that how to parallelize each part of web browser: front-end, page layout, and scripting. In [10], author introduces new algorithms for CSS selector matching, layout solving, and font rendering. In [11], author introduces mobile web paradigms, mobile web rendering engines, various mobile browsers, and future of mobile web.

Based on our previous research and other related issue about performance of web browser, we propose a new parallel algorithm for layout and paint functions. In the following sections, details of algorithm will be described.

3 Proposed Algorithm

3.1 Task to Parallelize

To parallelize web browser, we need to find suitable sub-tasks to parallelize and we focus on layout and paint functions. First, the overall execution time to process layout and paint functions takes average around 40% of the entire execution time when measuring the performance under Pentium Dual Core 1.6GHz processors. Since these two functions take a significant portion of execution time, we can get more performance improvement when parallelizing these two. Moreover, layout and paint functions have similar execution characteristics, as the render tree traversal shown in Figure 3. Layout and paint functions recursively visit render tree nodes and perform their own tasks at each node. This means layout and paint functions just work for itself alone, not interacting with other data structures or functions. In other words, we don't need to worry about typical synchronization problem in multi-thread programming within these two functions.

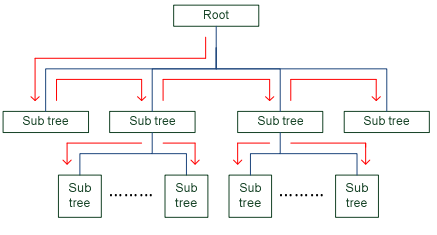

Fig. 3. Render tree traversal

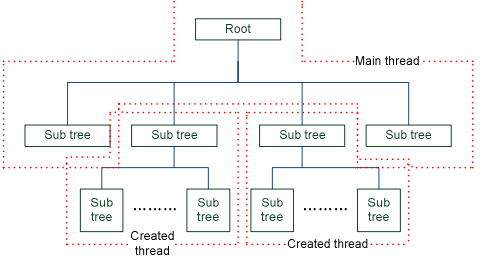

Fig. 4. Sub tree traversal with created thread

3.2 Structure of Render Tree

A web page is parsed into a tree of nodes, called the document object model (DOM). Document, elements, and text will can be nodes of this tree. Render tree is very similar to the DOM, where it is a tree of objects and has style information of elements derived from CSS. Web page is composed of a group of boxes and some boxes belong to other boxes. Boxes are represented as tags in html document, elements in DOM, and render tree nodes in a render tree. Since a big box contains several small boxes, overall structure of render tree is formed as unbalanced tree as shown in Figure 5. In this web

page, selected two big boxes take most of nodes in render tree and these two boxes also contain several small boxes. The box positioned in below forms bigger sub-tree than the above one since it contains more small boxes than above one.

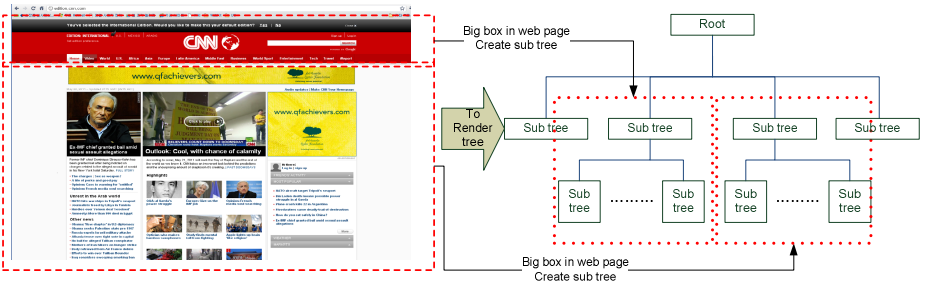

Fig. 5. Overall structure of render tree

3.3 How to Parallelize

We propose a way to create multi-threads for processing sub-tree traversal as shown in Figure 4 using POSIX thread library[12]. For example, when web browser paints web page on screen, browser will draws one box at a time but if we create threads for sub-tree processing, we can draw several boxes at a time. However, there is a problem that render trees are formed as unbalanced trees as we mentioned above and that will cause an unbalanced load balancing problem among threads. To solve this load unbalancing problem, the number of nodes in each sub-tree is used. While a thread visits nodes in the render tree, if the number of nodes in this sub-tree is greater than factor(named as Thread Factor: TF), as another threshold, and has siblings, the thread creates new threads and created-threads will handle this sub-tree.

4 Simulation for Algorithm Validation

To validate the effect of this algorithm we design a simulator implementing our parallel render tree traversal algorithm. When simulator visits each node, it performs assigned tasks. For the input of simulation, we use WebKit DumpRenderTree tool to get the render tree of real web page. WebKit DumpRenderTree prints render tree of web page to the console and we convert it into text file to use it as the input of simulation. To make simulation more similar to real web browser, we differentiate each node's task. For example, image sizes of web pages are very various and displaying big sized image will need more calculation than small sized image. We also found that leaf nodes of render tree are nodes of real images or texts, and inside nodes are wrapper box block. Therefore, we assume that leaf nodes will consume more processing power than inside nodes when browser does painting. So we use each node's type and size to reflect the characteristics of web browsing, since DumpRenderTree gives each node's type and size also. However, layout function doesn't need task differentiation. Layout function performs decision function of each node's position and size. Therefore, job done at each node is not quite different unlike

paint function. So, we use same iteration number at each node for layout function simulation. The number of iterations is named as Layout Factor (LF).

For the task that will be performed at each node, we use arbitrary memory allocation, integer and string calculations and execute them repetitively. For the differentiation of each node's task, the number of iterations at each node is used. Calculation method of iteration number is shown as follows:

Table 1. Iteration number

Node type	***Inside Node***	***Leaf Node***
The number of Iterations	Inner Node Factor (INF)	x size*y size*Leaf Node Factor (LNF)

5 Experiment Result

We choose 20 web pages to simulate and get Render tree of them. Most of pages have 1000~1500 render tree nodes. Experiments were performed in Intel i7 2600 processors that drop the clock speed to 1.6GHz and 4GB memory, running ubuntu 10.0.4. We use gettimeofday() function to get elapsed time of executing program. We measure execution time of simulator and compare the single thread version with multi thread version. We use dual-core to quad-core, and using Intel Hyper-Threading(HT)[13] technique to see the correlation between number of core and performance improvement.

Table 2. Factor for experiment

Factor	***high***	***mid***	***Low***
INF	10	10	10
LNF	4	3	2
LF	1500	3000	4500

Fig. 6. Reduction of execution time in paint simulation

Figure 6 shows reduction of execution time in paint and layout experiments. On average, reduction of execution time is 28% in dual-core, 32% in quad-core in case of TF is 10 in paint. As shown in figure, if TF is smaller, execution time reduction is bigger. In dual-core, TF 10's performance is average 7% better than TF 30's. Same trend is shown in quad-core also. Moreover, quad-core's performance improvement is better than dual-core's performance improvement. When TF is 10, average 13% better performance improvement we can get in quad-core compare to dual-core. However, HT doesn't give impact because we use just integer calculation, string calculation and memory allocation in this experiment, there is not enough functional units to allocate. In addition, INF and LNF don't give significant impact to performance. Execution time reduction of high, mid, and low is almost similar. On average, reduction of execution time is 38% in dual-core, 57% in quad-core in case of TF is 10 in layout. Compare with paint experiment, we can get better performance in layout experiment. Since layout experiment's per node iteration is same, so load balancing among threads is much better than paint. Except that, overall trend of experiment result is similar to paint simulation. LF also doesn't give significant impact to performance.

6 Conclusion

In this paper, we proposed a parallel approach about mobile web browsing, especially layout and paint parts. We parallelized layout and paint parts by implementing parallel render tree traversal algorithm. Moreover, to validate this algorithm, we design a simple simulation environment that has similar processing pattern to layout and paint functions. The experiment results show that execution time is reduced average 28% in dual-core, 32% in quad-core for paint simulation. In layout simulation, execution time is reduced average 38% in dual-core, 57% in quad-core. By using this parallel algorithm, we can utilize multi-core processor in mobile devices for most frequently used application and offer better user experience.

Acknowledgments. This research is was supported by the MKE(The Ministry of Knowledge Economy), Korea and Microsoft Research, under IT/SW Creative research program supervised by the NIPA(National IT Industry Promotion Agency) (NIPA-2010-C1810-1002-0023

References

1. Gartner Press Releases, http://www.garthner.com/it/page.jsp?id=1689814
2. Strategy Analytics Press Releases, http://www.strategyanalytics.com/default.aspx?mod=pressreleaseviewer&a0=4998
3. The WebKit Open Source Project, http://webkit.org
4. APPLE Safari, http://www.apple.com/safari/
5. Google Chrome, http://www.google.com/chrome/intl/en/make/features.html

6. W3C Document Object Model, http://www.w3.org/DOM/
7. WebCore Rendering 1-The Basics, http://www.webkit.org/blog/114/webcore-rendering-i-the-basics/
8. Meyerovich, L.: Rethinking Browser Performance. Login 34(4), 14–20 (2009)
9. Jones, C.G., Liu, R., Meyerovich, L., Asanovic, K., Bodik, R.: Parallelizing the Web Browser. In: HotPar 2009 Proceedings of the First USENIX Conference on Hot Topics in Parallelism (2009)
10. Meyerovich, L., Bodik, R.: Fast and Parallel Webpage layout. In: WWW 2010 Proceedings of the 19th International Conference on World Wide Web (2010)
11. Hernadez, E.A.: War of the Mobile Browsers. IEEE Pervasive Computing 8, 82–85 (2009)
12. POSIX Threads Programming, http://computing.llnl.gov/tutorials/pthreads
13. Intel® Hyper-Threading Technology, http://www.intel.com/technology/platform-technology/hyper-threading/index.htm