International Journal of Advances in Computer Science & Its Applications– IJCSIA
Volume 5: Issue 1 [ISSN : 2250-3765]
Publication Date : 30 April, 2015

Reducing Loading Time of Smart TV's web Application

[Cheolhee Lee | Taeho Hwang | Jaemin Jung | Youjip Won]

***Abstract*— Smart TVs use Webkit as a web browser engine to provide contents. Webkit exploits web standard resources, such as HTML, CSS, JavaScript, and images, to run applications. When an application is started, Webkit loads resources to the memory and creates DOM(Document Object Model) tree and render tree; this process takes a long time. Generally, the web browsers on PCs or mobile devices sometimes change DOM tree and render tree by downloading modified web resources from the web server. However, smart TV's applications do not change DOM tree and render tree because they use web resources stored in a disk. If DOM tree and render tree can be stored and reused, it is possible to reduce application's loading time. This paper proposes FastIO technique which selectively adds persistency to dynamically allocated memory. It reduces overall application loading time by removing the process of loading resources from storage, parsing the HTML documents, and creating DOM tree and render tree. When we compared the application loading times, the result showed that the web browser with FastIO is 44.8x faster than the legacy web browser in an Ramdisk environments, respectively.**

***Keywords*— Web Platform, Smart TV Application, Webkit, Parsing, DOM Tree and Render Tree, Persistency**

I. Introduction

Smart TVs provide a variety of web contents, such as web surfing, VOD watching, social network, and games. Generally, a smart TV uses a web platform as execution environment of applications [1]. With recent improvements in the web technology, web platforms provide web apps that have similar functions as native apps [2].

Webkit [3] is an open source framework that runs web applications on the web platform. When an application is started, Webkit performs the process of loading web resources, such as HTML, XML, CSS, JavaScript, and images, to memory and converting these resources into a tree data structure that is composed of nodes. There are two types of tree data structures: DOM(Document Object Model) tree and render tree [4]. The nodes of a tree contain information on each element of the web page, including its location, size, color, and images. However, whenever an application is started, Webkit repeats the process of loading resources and creating tree structures. This is a CPU intensive task which takes a long time. Therefore, if DOM tree and render tree can be stored and reused, it is possible to reduce the loading time of applications.

Cheolhee Lee, Taeho Hwang, Jaemin Jung, Youjip Won
Hanyang University
Seoul, Korea

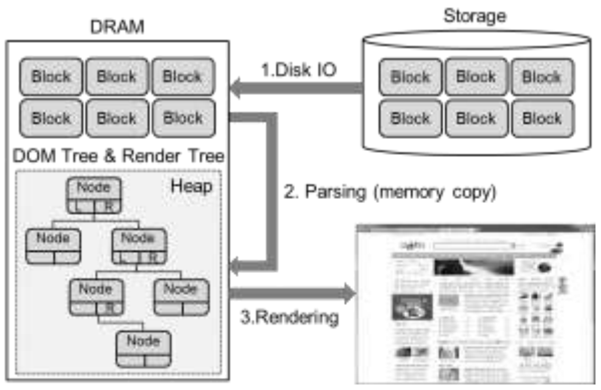

Figure 1. Loading an HTML Document

This paper proposes *FastIO* technique which selectively adds persistency to dynamically allocated memory. When the technique is applied to Webkit, it removes the process of creating DOM tree and render tree. FastIO proposes the concept of persistent heap, called an *object*, to support persistency to the memory area. An object is a set of linked list of consecutive pages with a name. Each object is mapped to a file through mmap() system call. A file that is mapped to an object is called an *object file*. Objects are non-volatile because they are stored in object files through page cache. Therefore, when an application is restarted, the data is reused by mapping the object file to process address space through mmap() system call. In FastIO, a naming system is designed to manage each object's name and mapping location in order to reuse the object. Through this naming system, it is possible to find out an object's address by its name, and the library manages objects through this address. A special object called *metadata object* is created to retain naming system information. FastIO provides an interface which is capable of dynamically allocating and releasing objects in byte units. This interface is designed based on glibc's malloc() mechanism.

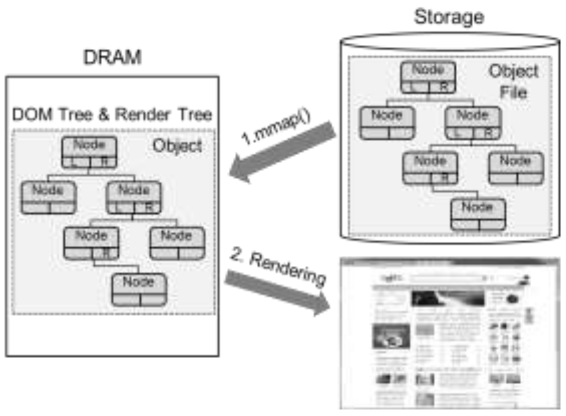

Figure 2. Web Browser with FastIO

II. Related Work

Recently, many researches are being conducted on improving the performance of web browsers. One among the researches parallelizes web browser processes to minimize a web browser's resource requesting time. It divides web browser processes into main-thread, which includes the process of requesting and parsing HTML documents, and sub-thread, which includes the process of calling sub resources [5]. In addition, to improve energy efficiency and browsing speed of a web browser, one of the researches separates the web browsing process into parsing, layout, and JavaScript compiling sub-processes. An optimal number of threads is determined for each sub-process and the sub-processes are executed in parallel [6]. ZOOMM [7] is a parallel web browser engine optimized for multi-core mobile devices. It divides subsystem into the resource manager, DOM engine, rendering engine, JavaScript engine, and user interface. Each subsystem is designed to operate in parallel. Smart caching [8] is a method that utilizes the fact that style information does not change unless DOM tree's element changes. It removes the process of style formatting and layout by caching and restoring the DOM tree element's style information and layout operation calculation.

Persistence [9] is a concept proposed by Atkinson in 1981. It means 'data required by the system must be maintained until its necessity disappears'. SoftPM [10] ensures data's persistency through a persistent area called the container. If the root node is given persistency, it provides persistency to all nodes connected to it by bringing them into a container. SoftPM automatically provides persistency to the memory data connected to the root node. NVRAM [11] is a non-volatile memory that can substitute HDDs or SSDs as a storage. It can also substitute DRAM as a main memory. Mnemosyne [12] and NV-Heaps [13] are examples of researches on providing data persistency using NVRAM. They provide persistency to address place in order to optionally provide persistency to data and reuse it without deserialization process.

III. Overview

Currently, smart TVs use a web platform as an environment to run applications. Web platforms began to gain attention when the need for an environment, which can run applications on various types of devices, arose. The web can be operated on any device with a web browser which gives it an advantage over applications. For this reason, the ultimate goal of web platforms is to provide an environment where applications can be developed with web standards [14], such as HTML, CSS, and JavaScript, as in Chrome OS or WebOS, and can be run without access to the Internet.

A. *Web Page Loading in Webkit*

Web platforms use Webkit as a web browser engine to run applications. Webkit is an open source framework which offers necessary functions to create a web browser and run web applications. It can be modified by users and it renders various web contents on user's devices.

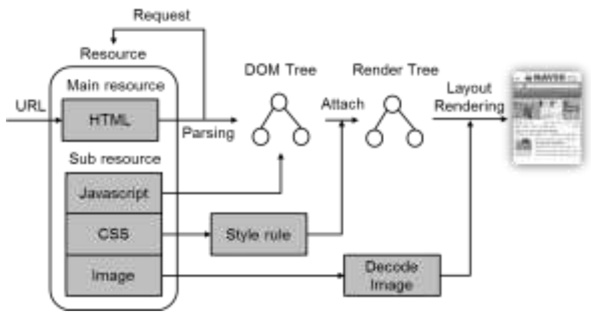

Figure 3. Loading a Web Page in Webkit

Figure 3 shows steps required to display a web page on the screen using Webkit. First, Webkit loads HTML document into a memory from web server or storage through URL, and performs tokenizing and parsing process for the HTML file. In the parsing step, HTML parser converts the document into HTML elements, such as HTML tags and contents, and creates a tree data structure called DOM (Document Object Model) tree. At this point, a render tree is also created as a corresponding tree to DOM tree. HTML pages include information on the page layout, CSS of the page, existence of JavaScript, size of images, and the number of images. Webkit additionally requests HTML, CSS, JavaScript, and images, which are loaded into and calculated at the storage. In case of CSS, style information is extracted through parsing process, just as with HTML, and the style rule of the web page is created. JavaScript updates DOM tree, requests resources, and modifies style information through JavaScript engine. Through these steps, Webkit processes resources necessary to render a webpage and performs layout and style information updates. It decides where on the render tree to allocate which elements. When the composition of a render tree is finished, the tree is used for rendering a web page. The page is displayed using information contained in nodes of the render tree.

B. *Problem Statement*

Generally, a web browser receives necessary resources from a web server when it accesses a web page. Since the resources from web server change frequently, all steps necessary to display a web page need to be performed with every access. However, smart TV applications use the data stored in storage and the loaded resources do not change every time an application is accessed. Therefore, the process of parsing stored resources into tree data structure is not always necessary when an application is run in a smart TV.

TABLE I. COMPARISON BETWEEN RESOURCE LOADING TIME AND WEB PAGE LOADING TIME

Web Site	Resource Loading	Web Page Loading
Naver	0.241 sec	1.044 sec
Facebook	0.082 sec	0.531 sec
Youtube	0.271 sec	1.115 sec
Daum	0.335 sec	1.041 sec
Google	0.127 sec	0.998 sec

Table 1 shows resource loading time and web page loading time for five websites. Resource loading time includes the time it takes to load the resources in memory and to create DOM tree and render tree. Web page loading time is the total time consumed to load each web page . On average, resource loading time accounted for about 21.6% of web page loading time. By reusing DOM tree and render tree, we can remove the parsing process and shorten the loading time of an application.

IV. Design

This section explains FastIO technique which provides persistency to dynamically allocated memory and reuses the tree data structure.

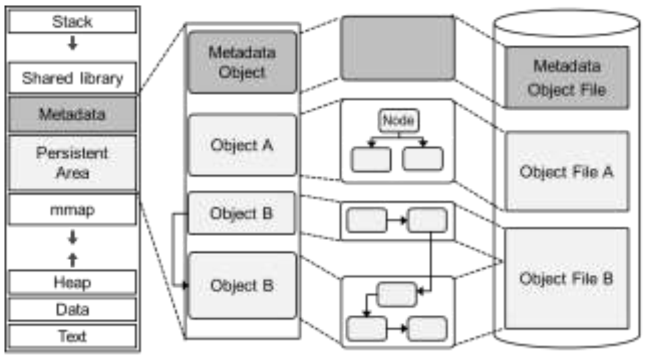

Figure 4. Basic Concept of FastIO

FastIO proposes the concept of persistent heap, called an object, to support persistency to the memory area. An object is an instance that has a name and is a linked list of consecutive pages. It is allocated in the process address space through mmap() system call and it is an area mapped with the file. The file that is mapped to an object is called an object file. Therefore, the data in object area is stored in the object file. When an application is restarted, Fast IO technique reuses data within the object by mapping the object file to process address space. Moreover, FastIO provides an interface which allocates and releases memory in the object in byte units. The allocated data in byte unit is called a node. Metadata exists to manage the names and mapping information of objects. It is stored in a special object called a metadata object.

A. *Pointer Validation Problem*

To reuse data structure in an object, the object file should be mapped to process address space through mmap() system call. However, if allocated mapping location of the object changes in the process address space, there occurs the problem of invalid pointer address value which represents incorrect connection between the nodes. There are two ways to solve this problem. The first is to use pointer swizzling to find a node in the object. The second is to ensure pointer validation by always mapping the object to the same address space. Since the first method incurs overhead from using pointer swizzling process to access the pointer of nodes, we used the second method and designed objects to be mapped to the same address whenever an application is run.

B. *Collision between Shared Library and Object*

When an application is started, the application's process address space is created. At this time, a shared library used by the application, such as libc, is loaded to the mmap area. In Linux, shared library is allocated to the address space called mmap_base. mmap_base is an initial address of mmap area. It is a random value (-1MB ~ 1MB) away from 0X7701000. Therefore, whenever an application is started, mmap_base changes which also changes the mapping location of shared library. This could result in a conflict between the object and shared library. For example, let us assume that an object was allocated to 0X3000 when an application was first started. When the application is restarted, a shared library is already loaded at 0X3000 which means the object cannot be loaded at that address (Figure 5).

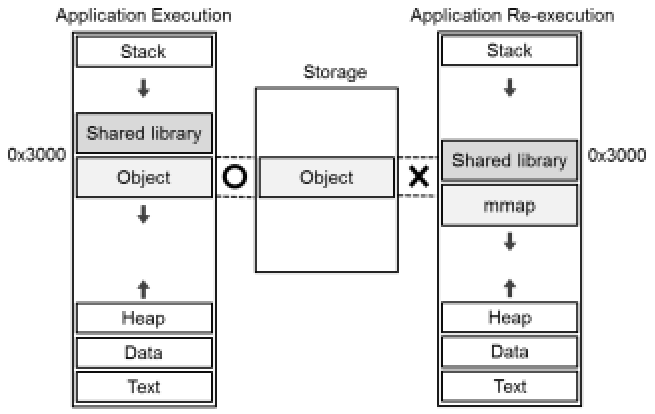

Figure 5. Example of Collision between a Shared Library and an Object

FastIO reserves an area for shared library to prevent a collision with objects. It sets the lowest address where shared library can be located as P_base, and allocates objects from this area. P_base is calculated as follows.

$$P\_base = 0Xb771000 - (1MB + Shared\ Library\ Size)$$

FastIO sets the range where shared library can be located as shared library area and reserves it for unused shared library so that objects are not created in the area.

C. *Namespace for Object Management*

Object files are accessible through the namespace of file system. However, the namespace does not contain detailed information on where to allocate specific objects. Therefore, to reuse the data structure in an object, we designed the namespace for object management. With the new design, it is possible to find the address of a specific object with a name, and the application can reuse the data in object through this namespace information.

D. *Byte Unit Allocation/Free in Object*

To dynamically allocate and release the data in byte units in an application, we call on malloc() / free() function. This function call allocates data in the heap area. To allocate memory in byte units, called a node, to a persistent area called an object, we designed the allocation scheme based on the memory allocation mechanism of glibc. *malloc_state* exists in an object's first address space. It manages an object's unused area, called free chunk, by size. If memory allocation in byte uinit is requested with the size and the name of object, the namespace is searched for the object. Then, it searches free chunks in object and allocate memory according to the requested size. However, if the request cannot be allocated due to lack of empty space in an object, object expands object's address space in order to allocate the request. In the case of heap, address space increases continuously when there is insufficient memory to satisfy the allocation request. However, we designed objects to increase discontinuously because continuous address space may be used for other purpose.

V. Experiment

This section explains the performance evaluation of FastIO technique. Performance was measured in experiments. The experiment compares the speed of Webkit-based web browser with FastIO to the web browser without FastIO. The experiment was conducted under AMD Phenom X4 925 Processor, 4GB DDR3 DRAM, SSD (Samsung 840 Pro 256GB) environment. An Ramdisk was used for the storage for experiment environment. Webkit-based QT-4.6.4 web browser was used for the experiments.

Generally, A web browser downloads HTML documents through URL and parses the HTML document. At this time, the browser requests resources from the web server. On the other hand, smart TV's application load resources from the storage instead of a web server. So, we stores resources on the storage and use file path instead of URL to load resources from the storage When QT web browser is run. QT web browser parse HTML Document to extract rendering information of web page. At this time, the QT web browser requests sub resources from storage and creates DOM Tree and render tree. It displays a web page on the screen using the rendering information of DOM tree and render tree. However, when the web browser is closed or changed, the DOM tree and render tree of the web page are removed from the memory. When the web browser is restarted, the process of loading HTML document from the storage and parsing HTML document and creating DOM tree and render tree is repeated. We applied FastIO to the QT web browser to add persistency to DOM tree, render tree, and sub resources of in-memory structure. This removes the process of parsing HTML document and creating DOM tree and render tree when the web browser is restarted.

We measured the time it takes for the web browser to load HTML document into memory, parse the document, load sub resources into memory, and create DOM and render trees. Figure 6 is the result measured on the Ramdisk for 5 web pages. It shows the resource loading time of legacy web browser and a web browser with FastIO. Mainresource Loading refers to the time spent on loading the HTML document into the memory. Subresource Loading refers to the time spent on loading CSS, JavaScript, images, and other sub resources into memory and converting them into in-memory structure. Parsing refers to the time spent on parsing the HTML document and creating DOM and render trees. FastIO refers to the time spent on loading to the memory from the files to reuse the DOM tree, render tree, and in-memory structure of sub resources. When FastIO was applied to QT web browser, the speed of loading a web page improved by 44.8x in an Ramdisk SSD environment,

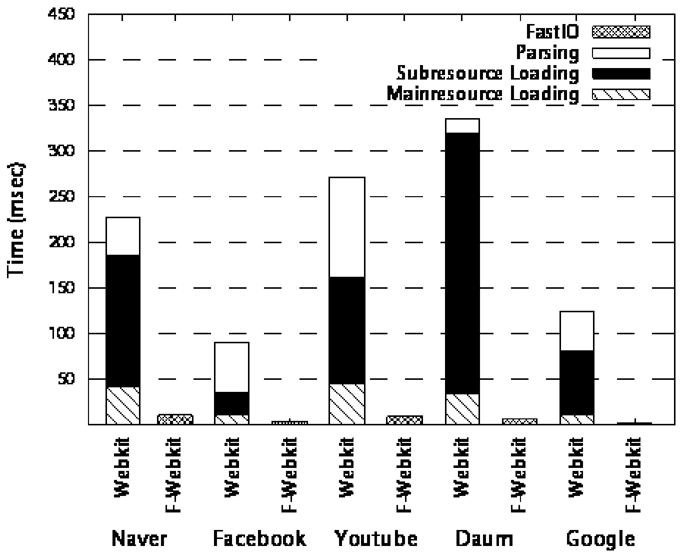

Figure 6. Loading Times on Legacy Web Browser and FastIO Browser

VI. Conclusion

This paper proposes FastIO technique to improve application's loading speed in a smart TV environment. FastIO is a technique that adds persistency to dynamically allocated memory. By applying FastIO to Webkit, it adds persistency to DOM tree, render tree, and resources of in-memory structure. These resources with persistency are reused every time an application is started, removing the process of creating DOM tree and render tree. the result showed that the web browser with FastIO is 7.9x faster than the legacy web browser in an Ramdisk environments. In addition to smart TVs, FastIO can also be implemented on various applications and devices that perform parsing process (deserialization) in using HTML document to improve their performance.

Acknowledgment

This work is supported by IT R&D program MKE/KEIT (No. 10041608, Embedded System Software for New-memory based Smart Device)

References

[1] S.Y. Lee, S.T. Park and J.W. Hong, "Advanced Smart TV 2.0 System and Service based on HTML5", Journal of Convergence Information Technology, vol.8, no.14, pp.172-180, September 2013.

[2] A. Charland, and B. Leroux. "Mobile application development: web vs. native." Communications of the ACM, vol.54, no.5, pp.49-53, 2011.

[3] Webkit Open Source Project, http://www.webkit.org/.

[4] K. Kim, H.M.Yang, C.G. Kim, and S.D. Kim, "A Parallel Approach to Mobile Web Browsing", In Mobile Computing, Applications, and Services, vol.95 pp.338-344, 2012.

[5] D. Kim, C. Lee, and WW. Ro, "Parallelized sub-resource loading for web rendering engine", Journal of system Architecture, vol.59, no.9, pp.785-793, 2013.

[6] C.G. Jones, R. Liu, L. Meyerovich, K. Asanovic and R. Bodik, "Parallelizing the web browser", In Proceedings of the First USENIX Workshop on Hot Topics in Parallelism, April 2009.

[7] C. Cascaval, S. Fowler, P. montesinos-ortego, W. Piekarski, M. Reshadi, B.Robatmili, M.Weber, and V.Bhavsar, "ZOOMM: a parallel web browser engine for multicore mobile devices", ACM SIGPLAN Notices, vol.48, no.8, pp.271-280, 2013.

[8] K. Zhang, L. Wang, A. Pan, and B.B. Zhu. Smart caching for web browsers. In Proceeding of the 19th international conference on World wide web, Raleigh, USA, April 2010.

[9] A. Dearle, J. Rosenberg, F. Henskens, F. Vaughan, and K. Maciunas, "An examination of operating system support for persistent object systems", In System Sciences, 1992. Proceedings of the Twenty-Fifth Hawaii International Conference on, Kauai, Hawaii, January, 1992

[10] J. Guerra, L. Marmol, D. Campello, C. Crespo, R. Rangaswami, and J. Wei, "Software Persistent Memory" In Proceding of the USENIX ATC, Boston, MA, June. 2012.

[11] B.C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-change technology and the future of main memory", IEEE Micro, vol.30, no.1, pp.143, January, 2010.

[12] H. Volos, and M. Swift, "Mnemosyne: Lightweight Persistent Memory", In Proceeding of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), California, USA, March 2011.

[13] J. Coburn, A. Caulfield, A. Akel, L. Grupp, R. Gupta, R. Jhala, and S. Swanson. "Nv-heaps: Making Persistent Objects Fast and Safe with next-generation, non-volatile Memories" In Proceeding of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) ,California, USA, March, 2011.

[14] W3C Standards, http://www.w3.org/standards/.

About Author (s): Cheolhee Lee

ChoelHee Lee is a MS Student at Dept. of Computer and Software Engineering, Hanyang University. He is member of Embedded Software System Laboratory at Hanyang University. He is interested in Operating system and Embedded system software.

About Author (s): Taeho Hwang

Taeho Hwang is a Ph.D. student at Dept. of Electrical and Computer Engineering., Hanyang University. He is a member of Embedded Software Systems Laboratory at Hanyang University. His research interests include Operating Systems with byte-addressable non-volatile memory.

About Author (s): Jaemin Jung

Jaemin Jung is a Ph.D. student at Embedded Software System Laboratory at Department of electronics and computer engineering, Hanyang University. He did his BS and MS in Department of electronics and computer engineering, Hanyang University, Seoul, Korean in 2007 and 2009, respectively. His research interests include Operating systems, byte-addressable non-volatile memory and file system.

About Author (s): Youjip Won

He is currently Professor at Division of Electrical and Computer Engineering, Hanyang University, Seoul Korea. He is leading Embedded Software System Lab. He did his BS and MS in Dept. of Computer Science, Seoul National University, Seoul, Korea in 1990 and 1992, respectively. He received his Ph. D in Computer Science from University of Minnesota in 1997. Before joining Hanyang University in 1999, he worked at Intel Corp. as Server Performance Analyst.