FastIO: Eliminating the Deserialization Overhead in Launching Web Applications

Jaemin Jung¹, Youjip Won ²

¹² Department of Computer Engineering, Hananyang University, Seoul, Korea
{jmjung,wywon}@hanyang.ac.kr

Abstract—Web applications are launched by Webkit engine. Dominant fraction of launch latency is spent on serializing the resource file. Webkit engine loads the HTML format based resources from the storage device, and converts them to memory representation creating DOM (Document Object Model) tree and render tree. The overhead of this deserialization is excessive and the modern web applications suffers from tens of a second in application launch latency. We develop a new software layer FastIO. FastIO maps a file system region to memory and uses this memory region to persistently store the tree representation, e.g., DOM tree and render tree, of the resources directly to the storage device. The FastIO layer eliminates the overhead of serializing and deserializing the HTML based resources; parsing the documents, and creating DOM tree and render tree. We implement FastIO layer in commodity PC and adopted FastIO on the Webkit engine. With FastIO, the speed of launching an application increases by 44.8x, 7.9x, and 2.9x when the resource files reside in ramdisk, SSD, and eMMC, respectively.

Keywords – Persistent heap, web application, Webkit, deserialization

I. INTRODUCTION

Native application platforms consist of Operating System (OS), e.g., linux, solaris, windows, etc, and ISA (Instruction Set Architecture), e.g., x86, ARM, etc. However, the native application platforms have disadvantages in portability. To reduce development costs in diverse of hardware and OS, JVM (Java Virtual Machine) [23] and web platforms are deployed in these days. These application platforms support to launch applications independently to hardware and OS. Java applications are compiled into an intermediate code called bytecode. Since JVM translates the bytecodes into the machine code in runtime, Java applications can be executed in any environments that JVM is running on. The application platform for web applications is called a web platform. The web platform also has an advantage in portability since web applications can be launched in various web engines, e.g., Webkit [5] and Gecko [2]. Most web platforms are based on an open source framework called Webkit. When an application is launched, Webkit loads XML, HTML, CSS, JavaScript [4], and images into memory. Then, it converts these resources into tree data structures called DOM (Document Object Model) tree and render tree [19].

Mobile Operating Systems, e.g., Android [1], webOS [6], and ChromeOS [14], adopt these machine-independent application platforms. Android provide Java-based application platforms, e.g., dalvik [13] or ART (Android Runtime). WebOS and ChromeOS provide Webkit-based application platforms. However, compared to native application platform, these kind of application platforms have disadvantages in aspects of performance since they require additional overhead of translation or tree construction [9]. While Java applications are required to translate bytecode, the overheads can be minimized by pre-compiling, Just-In-Time compiler, etc. However, time consuming tree construction is always required for launching web applications. When the application is finished, Webkit discards the DOM tree and render tree. Whenever an application is started, Webkit repeats the process of loading resources and creating DOM tree and render tree. This deserialization overhead is CPU-intensive and becomes more significant in mobile or embedded devices.

In this work, we develop FastIO technique and adopt it on Webkit to eliminate the deserialization overhead in launching web applications. FastIO is aligned to the persistent heap approaches [28, 11, 15, 16]. Existing persistent heap techniques have difficulties in deploying them since they are designed for NVRAM devices that have disadvantage in cost and size at current. However, FastIO uses flash devices to make memory data persistent. For flash devices,
existing persistent heap mechanisms induces significant overheads due to excessive page invalidation and bytes-leading block IO. In FastIO, we eliminated the overhead for ensuring consistency which is inessential for eliminating the deserialization overhead. FastIO is implemented in user-level library and does not require any modification of the kernel. In addition, we adopted FastIO for Webkit to measure the performance improvement by eliminating the deserialization overhead.

II. RELATED WORK

The techniques to improve the performance of web browsers have been around for decades. Smart caching [29] observed that style data and layout data can be reused for revisiting the same web page. It eliminates the redundant computations in style formatting and layout calculation by caching style and layout data. A number of works propose to improve the performance of web browsers by exploiting parallelism. Separating the works of web browsing into several threads is proposed to improve the energy efficiency and the speed of web browsing [17]. It executes multiple threads in parallel for parsing, layout calculation, and JavaScript handling. A technique to divide the render tree into sub-trees and exploit parallelism for each sub-tree is proposed to improve the loading speed of a web browser [19]. Kim et al. proposes to divide the process of web page loading in to a main-thread and sub-threads [18]. The main-thread requests and parses HTML documents and the sub-threads request sub resources. ZOOMM [8] is a parallel web browser engine optimized for mobile devices equipped with multi-core. It provides subsystems which operate in parallel for resource manager, DOM engine, rendering engine, JavaScript engine, and user interface.

This work focuses on the memory persistence to eliminate deserialization overheads by reusing memory data. The memory persistence is also known as orthogonal persistence which was proposed by Atkinson [12]. Orthogonal persistence means that every memory data as well as file data is ensured to be maintained against process termination and system crashes. The development of non-volatile memory technologies, e.g., FRAM, PCM, RRAM, and STT-MRAM, is providing opportunities to support persistence on the memory [27, 20]. The key of providing consistency is ensuring the ordering of memory writes. The memory writes can be reordered due to out-of-order execution and write-back cache policy of the processor. Techniques to provide a persistent heap described above also handles consistent problem by guaranteeing the write order with cache-related instructions. Bhandari et.al. compared various software based ordering guarantee mechanisms in the x86 architecture [7]. Persistent Memory Block Driver [10] is a block device abstraction for the NVRAM and provides several operations for ordering guarantee. WSP [25] suggested using capacitor to ensure persistence and consistency for the whole system including caches and registers in the processor as well as memory.

To resolve the cost and capacity issues of current non-volatile memory technologies, alternative hybrid approaches are proposed. NV-DIMM [26], and teraDIMM [3] are hardware techniques that provide DIMM interface to use NAND flash memory as byte-addressable persistent memory.

III. BACKGROUND

A. Launching Web Application in Webkit –

Webkit is an open source framework which provides facilities to create a web browser and execute web applications. In Webkit, the process of loading web pages and launching web applications are similar except that web page loading requires a network connection. Figure 1 shows process of loading web pages in Webkit. After finding the web server by URL, a HTML document which is main resource is downloaded into the memory or the storage. The HTML document contain information of the page layout, style of the page, existence of JavaScript, size of images, and the number of images. To extract these information, Webkit performs parsing the HTML file. The parser in Webkit tokenizes the HTML document into HTML elements, such as HTML tags and contents, and creates two tree data
structures called a DOM (Document Object Model) tree and a render tree, respectively. Then, sub resources including CSS, JavaScript, and images are additionally requested to the web server and downloaded into the memory or the storage. JavaScript engine updates DOM tree, requests resources, and modifies style information with the downloaded JavaScript. Similar to HTML, CSS is parsed and the style information is extracted from CSS. Finally, Webkit displays the web page using the tree structures.

![Figure 1. Process of loading web page in Webkit](image)

### B. Problem Assessment –

<table>
<thead>
<tr>
<th>Web Site</th>
<th>Resource Loading (sec)</th>
<th>Web Page Loading (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Facebook</td>
<td>0.082</td>
<td>0.531</td>
</tr>
<tr>
<td>Youtube</td>
<td>0.271</td>
<td>1.115</td>
</tr>
<tr>
<td>Google</td>
<td>0.127</td>
<td>0.988</td>
</tr>
<tr>
<td>Naver</td>
<td>0.241</td>
<td>1.044</td>
</tr>
<tr>
<td>Daum</td>
<td>0.335</td>
<td>1.041</td>
</tr>
</tbody>
</table>

In loading web pages and launching web applications, the process of constructing tree structures from the resources, e.g., HTML, CSS, and JavaScript, are required. Table 1 shows resource loading time and web page loading time for five websites. The time is measured where the resources reside in the local storage. Resource loading time is the time to load the resources into memory and to create DOM tree and render tree. Web page loading time is the overall time to display the web page. On average, resource loading time takes about 21.6% of web page loading time. This implies that the overhead of constructing tree structures are significant and imposes significant delay in launching web application as well as loading web pages. In this work, we call the overhead of constructing tree structures from resources, e.g., HTML, CSS, JavaScript, etc, as deserialization overhead.

In web page loading, each accesses for web pages require downloading web pages. Although some resources can be cached in the local storage, network delay is unavoidable to ensure that the web pages are up-to-date. Moreover, web pages tends to be updated frequently. However, web applications are rarely updated compared to web pages. Nevertheless, constructing tree structures are performed every time when web applications are launched. The reason is that the tree structures cannot be reused since they locate in the volatile memory which serves for the heap of the process address space. The deserialization overhead can be also found in loading various kinds of files, e.g., spreadsheet, word processor, etc, which are formatted in markup languages, e.g., XML.

Persistent heap mechanisms [28, 11, 15, 16] provide opportunity to eliminate the deserialization overhead by reusing the memory data. However, they are designed for NVRAM (Non-Volatile RAM) devices. As commercial NVRAM devices have small size and high cost, it obstructs the deployment of utilizing the persistent heap layer. In this work, we propose providing the persistent heap layer with flash devices to encourage an immediate deployment. Existing persistent heap techniques make an effort on providing failure-safe atomicity to ensure the consistency of internal data structures. However, we do not need to concern about the consistency and durability of the tree structure since they can be reconstructed from resource files against the crash. Moreover, ordered word writes to flash imposes excessively high...
overhead since each word is written in block unit and each word update for the same block invalidates the page immoderately.

IV. DESIGN

A. Overview –

This paper proposes FastIO technique which preserve memory data after the process is terminated. Figure 2 illustrates the overall structure of FastIO. FastIO reserves virtual address region in the process address space. The region of FastIO consists of persistent area and metadata. The persistent area consists of objects. Each object is a persistent heap and has its own name. Object can be created, deleted, and mapped by the interfaces of FastIO. Dynamic memory allocation is handled on the object which is specified as a parameter of the dynamic allocation interface. To extent the object in the process address space without relocation, FastIO support that an object can resides in several discontinuous pieces of pages. The metadata object is a special purpose object in FastIO. Hence, the metadata object is treated as an object internally. To make the object persistent, FastIO stores each object into the object file. Similarly, metadata is written into the metadata object file. FastIO uses the mmap() system call to map object files and a metadata object file.

The metadata region maintains names and mapping information. Since FastIO provides a programming model to reuse the persistent data with a name, FastIO stores names of objects in the metadata. The mapping information is maintained in metadata region to make objects be always mapped to the same address space. The object files are mapped by mmap() system call. When an application is restarted, the virtual addresses which persistent data is located at is changed after the termination of process. If the address changes, the pointer values stored in the objects lose validity. To prevent this FastIO keeps the address to map each object. The details are discussed in later.

FastIO provides primitives to reuse memory data after the process termination. Figure 3 shows the differences of Webkit behavior when FastIO is not adopted and adopted. Without FastIO, resources are load into the page cache and then DOM tree and render tree is constructed (Figure 3a). These processes are alway required since the tree structures...
are not retained. However, we can reuse the tree structures with FastIO (Figure 3b). By reusing the memory data, it is possible to eliminate the deserialization overhead.

B. Design Principles –

In this work, we develop FastIO providing a persistent heap layer to eliminate deserialization overheads in loading files written in markup languages, e.g., XML, and launching web applications. In developing FastIO, we make efforts on satisfying three design principles: reusability of memory data, flash-based persistence without significant overhead, and dynamic allocation/deallocation capability. Reusing memory data is essential to eliminate deserialization overhead.

To provide reusability of memory data, FastIO provides a persistent heap layer. We define interfaces and programming model to support reusability of memory data. FastIO uses flash devices to store memory data to resolve cost and size issues in using NVRAM technologies, e.g., STT-MRAM, PCM, etc. Typically, writes to the persistent memory are serialized to guarantee write order. Even in NVRAM which is byte-addressable, ordering guarantee overhead is significant since it limits parallelism and cache efficiency. The overhead of ordering guarantee becomes more significant for flash devices. The flash devices are accessed in page unit and erased in block unit. In using flash devices as persistent memory, every writes for few bytes incur block write. Moreover, consecutive writes to the same page induces excessive invalidation due to out-of-place constraint of flash I/O. Fortunately, we do not need to concern consistency in exploiting persistence to eliminate deserialization overhead. Once the deserialized data is stored in the persistent memory, it is not changed until the originated data, e.g., web applications, is updated. Thus, we may assume the persistent data is read-only after creation. Inconsistent data is remained in the persistent heap for crashes during creation of persistent data. However, it can be simply handled in our case since persistent memory structures can be reconstructed from original files.

FastIO should provide dynamic allocation/deallocation capability. In launching web applications, the nodes of tree structures are dynamically allocated by malloc(). FastIO provides malloc()-like interfaces to provides dynamic persistent memory allocation/deallocation.

C. Interfaces –

FastIO provides eight interfaces for reuse and dynamic allocation/deallocation: p_create(), p_delete(), p_map(), p_unmap(), p_malloc(), p_free(), set_prime_node(), and get_prime_node(). The brief description of these interfaces are as follows.

- void p_reserve() initializes the process address space to use FastIO primitives.
- int p_create(char* name) creates an object and an object file. The name of the new object is determined by the name parameter. Initial size of the object file is 4KB. The mapping and the name information is registered in the metadata of FastIO. After then, it performs mapping the new object into the process address space.
- int p_delete(char* name) deletes an object and an object file specified by the name parameter. The mapping and the name information is unregistered in the metadata of FastIO.
- int p_map(char* name) maps an object specified by the name parameter into the process address space.
- int p_unmap(char* name) unmaps an object specified by the name parameter into the process address space.
- void* p_malloc(char* name, int size) allocates persistent memory from the name object. The size of memory allocation is specified by the size. It returns the allocated address as legacy malloc() does.
- void p_free(char* name, void* addr) frees persistent memory specified by the parameters.
- void set_prime_node(char* name, void* addr) sets the address of prime node of the object specified by the name parameter. The address is stored in the metadata region of FastIO.
- void* get_prime_node(char* name) returns the address of the prime node which is set by the user.

The p_reserve() is a function that should be invoked before using FastIO. The p_reserve() prevents address conflicts between FastIO region and legacy file mapping, e.g., mmap(). The details of the address conflict in FastIO will be discussed in later.
p_create(), p_delete(), p_map(), and p_unmap() return success or fail values to handle error condition. The dynamic allocation/deallocation interfaces of FastIO is similar to the legacy malloc()/free() which is provided by glibc, except that FastIO requires an additional name parameter. FastIO defines a prime node which is a special node among the dynamically allocated memory. The prime node is one of the dynamically allocated memory which make it possible to visit every dynamically allocated structures. For example, prime nodes of tree structure, linked-list, and array are root node, list header, and the start address, respectively. The prime node should be set by the user explicitly for reuse. FastIO provides set_prime_node() and get_prime_node() interfaces to set and get the address of the prime node.

When the size of an object is insufficient to handle p_malloc() request, the size of the object and the object file are increased. Then, modified object information, e.g., address region, is affected into the metadata of FastIO. Increasing the size of an object is implemented by p_extend() internal function. The p_extend() function increases the size of the object file and maps the file to process address space. To determine the address of the expanded portion, p_extend() allocates the address region from the persistent area in the process address space and updates the mapping information of the object which resides in the metadata of the FastIO.

![Figure 4. Example source code for FastIO](image_url)

![Figure 5. Process of p_create() interface in FastIO](image_url)

**D. Programming Model**

FastIO provides a simple programming model in exploiting the persistence of the memory. Figure 4 shows programming example of FastIO. At first, p_reserve() is invoked in order to prevent address conflict. The p_create("list") is used to create an object named "list". When the object named "list" exists, it is required to map the object explicitly by p_map("list"). To reuse data structures in the object, the prime node is obtained by get_prime_node("list"). The application uses p_malloc("list", size) to allocate persistent memory from the "list" object. The prime node is set by set_prime_node("list", node) since the head of the list is changed. After finishing the use of the object, p_unmap("list") is called to release this object in process address space.

**V. IMPLEMENTATION**

We develop FastIO on Linux 2.6.35 for x86-32bit architecture. FastIO provides the persistent heap primitives in the user library level. As FastIO does not requires modification in kernel, FastIO can be easily ported to UNIX systems without significant modifications. In this section, we explain the implementation issues in detail.

**A. Persistent Support with Flash Devicest**

As FastIO provides persistence with the flash devices, it is required to map the pages of flash into the process address space. For this purpose, FastIO uses mmap() system call. Because the flash devices cannot be accessed without loading into the memory, the kernel load the data from the flash device to the page cache. Then, the page cache is mapped to the process address space and every read/write on the memory is performed on the page cache. This incurs inconsistency.
FastIO: Eliminating the Deserialization Overhead in Launching Web Applications

between the page cache and the flash devices. To drain the updated data, msync() system call can be used. Since FastIO is based on the flash devices, synchronizing memory structures smaller than a page size is not required. Moreover, completely constructed in-memory structure of FastIO are read-only for our use case of reusing in-memory structure of web applications. Our approach in providing a persistent heap for flash devices can reduce significant overhead of ensuring consistency.

Figure 5 illustrates the process of handling p_create() call. For comprehension, the metadata structure of FastIO is represented in conceptually. The detailed structure of metadata will be discussed in next section. FastIO is provided as a library. When p_create("Object B") is called, the name “Object B” is search on the metadata of FastIO. If not found, object file named “Object B” is created on the directory which is reserved by FastIO. Then, register the “Object B” in the metadata of FastIO. After then, mmap() is invoked with the specified address in the FastIO library. Finally, the mapping is established for the “Object B”.

B. Metadata Structure –

Figure 6 shows the metadata structure of FastIO. The metadata is stored in the metadata object file and mapped into the metadata region (figure 2). As the metadata object is also an object in FastIO, the metadata object has also the prime node. The p_superblock is the prime node of the metadata object. The p_superblock maintains two pointers for p_ns_entry structures: the head of list and the pointer to the hash table. Each p_ns_entry can be founded by both the list and the hash table. The hash table uses the name as a key. When searching an object with a name, the hash table is used to avoid sequential list search. The p_ns_entry is allocated for each object and consists of four field: a name, in-use count, and two pointers. The name is used to search object by its name. The in-use count contains the number of processes that maintain the mapping to the object. It is used to prevent the object which is used for other processes from being deleted by p_delete(). The two pointers indicate p_vm_area structures. The p_vm_area represents a single
contiguous address region for the object. Each p_vm_area contains the start address, size, and offset in the file. The start address and the size determines the region to map object into the process address space. The persistent data for the region is maintained in the file and the offset of file to load is contained in the p_vm_area. As the object can reside in discontinuously in FastIO, p_vm_area structures for the same object are organized with the linked list structure. The two pointers in p_ns_entry indicates the first and last p_vm_area} structures which organize the object.

C. Ensuring Pointer Validity –

The p_map() interface performs mapping an object file into the process address space by mmap() system call. Since FastIO stores in-memory data structures into the object file, it is possible to reuse in-memory data after process termination. To support reusing data properly, it is required to ensure the validity of pointer variables to reuse them. The pointer variable becomes invalid when the address of data designated by the pointer is changed. Figure 7 illustrates the situation that the pointer becomes invalid. After re-launching the application, the pointer variable can indicate a wrong address when the address that object are mapped is changed. Although the address is given to the mmap() by the parameter, the address is ignored if the mapping is impossible on that address. There are two ways to solve this problem. The first is to use pointer swizzling [24] to find a node in the object. The second is to ensure pointer validity by always mapping the object to the same address space. Because of the overhead in the pointer swizzling, we use the second method and designed objects to be mapped to the same address.

D. Preventing Address Conflicts –

To ensure the validity of pointers, FastIO always maps the object into the fixed address. Thus, it is required to ensure that any objects of FastIO does not conflict with other memory regions to locate them properly. Since FastIO uses mmap() system call to map object files, the object mapping on the specified address may fail if the address is occupied by others. There can be two reason of address conflicts: shared libraries and file mapping. FastIO resolves the address conflicts by reserving the address space.

When an application is executed, shared libraries, e.g., libc, are mapped into the process address space. In Linux, shared libraries are also mapped by the mmap(). The mmap() system call initially search a proper virtual address region starting from mmap_base which is defined in the kernel. The value of mmap_base is randomly determined when a process is create. The problem for this random value of mmap_base is that newly created object of FastIO can conflict with shared libraries in other process contexts.

Figure 8 shows the example of address conflict by shared libraries. An object was located in 0x3000 when an application was executed at first. However, a shared library is loaded at 0X3000 when the application is restarted after termination. In this case, the object cannot be loaded at that address at which the object has been located before. To avoid address conflicts, FastIO always locates new objects starting from the fixed location defined by P_base in FastIO. The size of the shared libraries vary among the applications and the value of mmap_base vary among process contexts, it is possible to avoid address conflict by carefully determining the P_base. Fortunately, the variation of mmap_base is...
FastIO: Eliminating the Deserialization Overhead in Launching Web Applications

only 2MB centered at 0xB7701000 in Linux. In our current implementation, we assume that the only a few shared libraries are used. However, it is trivial to adjust the value of P_base.

Figure 9 illustrates the process address space of an application that is initialized for FastIO. To prevent the address conflicts between shared libraries and objects, FastIO reserves the unused region between the end of the shared libraries and the P_base. The reservation process is performed by p_reserve(). By parsing “/proc/pid/maps” files, FastIO extracts the end of the shared libraries. Then, empty address spaces between shared libraries and P_base are reserved by calling mmap() system call. After this reservation, it is ensured that objects are not mapped to this area.

Another kind of address conflicts can occur due to the file mapping. In FastIO, objects are always mapped to the same address to ensure the validity of pointers. However, if the addresses for the object are used for file mapping in advance of object mapping, an address conflict occurs. To prevent this, p_reserve() searches every objects and reserves address spaces of objects. Figure 10 shows the reserved areas of FastIO in the process address space. Note that this reservation does not map existing objects into the process address space. To ensure that the region of existing objects are not allocated for other purposes, the p_reserve() performs dummy mapping to reserve the region which can be used for objects of FastIO. Therefore, it is required to discard the reserved mapping before mapping an object when p_map() is called.

E. Dynamic Memory Allocation/Deallocation –

The legacy dynamic allocation/deallocation is performed in the heap area by malloc()/free() which are provided by the libc. FastIO provides familiar interfaces for dynamically allocating and deallocating persistent memory: p_malloc() and p_free(), respectively. The dynamic allocator of FastIO is almost similar to the libc dynamic allocator. The malloc_state structure maintains free lists of memory chunk. The allocation is performed by the searching the free list. Different from the libc, malloc_state is stored at the beginning of each object in FastIO. This is because FastIO allocates and deallocates persistent memory from each object and the information of free chunks need to be persistent.

When p_malloc() is called with the size and the name, the corresponding object is searched. Then, it searches free chunks in the found object and allocates memory in the object. When the request cannot be allocated due to lack of free space in an object, the object need to be expanded. In the case of the legacy volatile heap, the boundary of the heap increases continuously by the brk() system call. However, the object cannot be enlarged continuously in the process address space if the addresses to enlarge are already occupied by other file or object mapping. To support discontinuous expansion of objects, p_extent() internal function is used in FastIO. The p_extent() function extend the size of the object file and perform mapping the extended area of the file by mmap() system call. hen, the extended pages are inserted into the list of free chunks.
F. Adopting FastIO on Webkit –

To load web pages, a web browser starts from downloading and parsing HTML document. Then, DOM tree and render tree are constructed and web page is drawn on the screen according to the DOM tree and the render tree. However, if the browser is terminated or it accesses another page, the DOM tree and the render tree of the page are discarded from the memory. When web browser is restarted, HTML document is loaded into memory from the storage and DOM tree and render tree are reconstructed. Launching web applications is almost similar except that they do not require a network connection and are always loaded from the local storage.

We adopted FastIO on the QT-4.6.4 web browser to eliminating the deserialization overhead. The QT-4.6.4 web browser is based on Webkit. Figure 12 is a flow chart of web browser with FastIO. First, existence of object for URL is tested. If there is object, object file is loaded into memory and DOM tree, render tree and in-memory structure sub resources are reused. But, if there is not object, web browser loads resources and DOM tree and render tree are created. The process of loading resources and creating DOM tree and render tree is eliminated when accessing the same URL again by storing DOM tree, render tree, and in-memory structures of sub-resources with FastIO.

VI. EXPERIMENT AND RESULT

A. Experimental Setup –

To measure the performance improvement of adopting FastIO, we apply FastIO on Webkit-based QT-4.6.4 web browser. In addition, we measured the performance of key-value stores adopting FastIO to measure the performance when FastIO is used for read/write workloads. Current implementation of adopting FastIO on the Webkit is achieved to store in-memory structures of resources into the object of FastIO. The experiment was performed under AMD Phenom X4 925 Processor with 4GB DDR3 DRAM. We measure the resource loading time in Webkit-based web browser. For flash storage, we used a commercial SATA3 SSD (Samsung 840 Pro 256GB) and eMMC 4.41 based flash device. To measure the improvement of FastIO in the faster storage device, we used ramdisk.

B. Eliminating the Deserialization Overheads –

We measure the improvement of eliminating deserialization overhead in the web browser. In this experiment, we compared the loading time of web page resources in the unmodified web browser and the FastIO-adopted web browser. Although our interest is the improvement of launching web applications, the process of loading web pages and launching web applications are almost similar except for the network delay. To eliminate the network delay in web page loading, we store every resources on the local storage. In addition, we use a file path instead of URL.

![Figure 13. Resource loading times on legacy Webkit and Webkit with FastIO](image-url)
FastIO: Eliminating the Deserialization Overhead in Launching Web Applications

Figure 13 shows the resource loading time of an original web browser and a web browser with FastIO, which are depicted as Webkit and F-Webkit, respectively. We measure the performance on five web pages when resources reside in SSD, ramdisk and eMMC. For Webkit, the loading time is defined by loading resources from the storage and constructing in-memory structure including DOM tree and render tree. For F-Webkit, the loading time is defined by mapping an object and loading every pages of the object from the storage. Main resource loading refers to the time spent on loading the HTML document into the memory. Sub resource loading refers to the time spent on loading CSS, JavaScript, images, and others into memory and converting them into in-memory data structure. Parsing refers to the time spent on parsing the HTML document and creating a DOM tree and a render tree. FastIO refers to the time spent on loading to memory from the files to reuse the DOM tree, render tree, and in-memory structure of sub resources.

In case of legacy web browser, most of time is spent for loading main and sub resources. In youtube and facebook web pages, many portion of time is possessed by constructing DOM tree and render tree. In other web pages, parsing time is not very significant. However, the performance improvement of adopting FastIO is impressive. The loading speed is higher than that of legacy web browser on ramdisk, SSD, and eMMC by average 44.8x, 7.9x, 2.9x, respectively. As we described in Table 1, the resource loading time occupied 21.6% of web page loading on average. By applying the improvement of resource loading with FastIO, the overall web page loading time on average can be improved by 21.1%, 18.9%, and 14.15% on ramdisk, SSD, and eMMC, respectively.

C. I/O Overheads in FastIO –

Table 2 shows the number of files and the size of total resources required to loading web pages in Webkit and F-Webkit. In F-Webkit, every resources for a web page are stored as a single object file. However, the size of an object increase since in-memory data structures maintain uncompressed data for images. The increased size can be a problem when the storage size is not sufficient. The resource size of F-Webkit implies that the amount of size are eventually required in the memory to load the web page or launch the web application. As the size of storages is larger than that of memory by order of magnitude in most devices, we assume that the storage is capable of storing the resource for FastIO. In addition, the original resource files can be deleted after creating in-memory structures completely. Although FastIO does not ensure consistency during creation of in-memory structures, it does not have consistency issues after creation since FastIO assumes that the in-memory structures are read-only.

As the size of the deserialized data maintained by FastIO is larger than that of original resource files, F-Webkit requires more I/Os than original Webkit. Figure 14 shows the I/O times of Webkit and F-Webkit in loading web pages. As expected, F-Webkit consumes more time for I/O compared to Webkit and it is dependent to the performance of the underlying devices, e.g., ramdisk, SSD, and eMMC. For slow storage devices such as HDD, FastIO may not have any advantages in loading web pages or launching web applications. However, FastIO improves the overall loading time significantly for flash devices despite of the I/O overhead (Figure 13). For more fast storage devices, the improvement is more significant as shown in the result of ramdisk.
D. FastIO as a General Persistent Heap Layer –

In designing FastIO, we focus on eliminating the deserialization overhead in launching web applications. In this case, the persistent memory regions are read-only. However, FastIO can be still used for read/write workloads by adopting msync() properly. To measure the performance when the FastIO is used for read/write workloads, we implement two kinds of key-value stores adopting the FastIO: B-tree based and hash based key-value stores. We compare the performance of the key-value stores with Berkeley DB (BDB) on a SSD device. Insert, lookup, and delete operations are performed for 1 million pairs of key-values. The size of key and value are 16 byte and 32 byte respectively. In implementing key-value stores, we use msync() to providing consistency.

Although msync() based synchronizing can induce significant overhead in flash-based FastIO, the performance is improved compared to BDB. The key-value store based on B-tree is 1.9x-2.7x faster than BDB (Figure 15a). The improvement is more significant for hash-based key-value store. The performance is 2.4x-12.8x faster than BDB (Figure 15b). This result suggests that FastIO can be adopted as a general-purpose persistent heap layer.

VII. CONCLUSION

This paper proposes FastIO technique to eliminate the deserialization overhead in launching web applications. FastIO is a technique that provides a persistent heap layer to reusing the memory in the process address space. We adopted FastIO into the Webkit to support reusing in-memory structures including DOM tree and render tree and remove the process of deserialization. FastIO has storage overheads since the size of in-memory structures stored in the storage increases compared to original resource file. However, FastIO improves the performance of loading web pages or launching web application significantly for the flash devices. By adopting FastIO on the Webkit, we improve the resource loading time by 7.9x and 2.9x for SSD and eMMC, respectively. Since most mobile devices use flash based storage, FastIO can provide improvement in launching web applications. As the performance of the storage devices are improving, we believe that the improvement of adopting FastIO will be more significant in the future. Our work contributes in providing a persistent heap layer for flash devices without significant overhead and adopting it to eliminate the deserialization overhead in launching web applications.

VIII. ACKNOWLEDGEMENTS

This work is sponsored by IT R&D program MKE/KEIT (No. 10041608, Embedded system Software for New-memory based Smart Device), ITRC program of MSIP /IITP (No. IITP-2015-H8501-15-1006), and ICT R&D program of MSIP/IITP (No. 12221-14-1005, Software Platform for ICT Equipment’s).

REFERENCE

FastIO: Eliminating the Deserialization Overhead in Launching Web Applications


